What is an algorithm?

An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.
Euclid’s Algorithm

Problem: Find \( \gcd(m,n) \), the greatest common divisor of two nonnegative, not both zero integers \( m \) and \( n \)

Examples: \( \gcd(60,24) = 12 \), \( \gcd(60,0) = 60 \), \( \gcd(0,0) = ? \)

Euclid’s algorithm is based on repeated application of equality

\[
\gcd(m,n) = \gcd(n, m \mod n)
\]

until the second number becomes 0, which makes the problem trivial.

Example: \( \gcd(60,24) = \gcd(24,12) = \gcd(12,0) = 12 \)
Two descriptions of Euclid’s algorithm

Step 1  If \( n = 0 \), return \( m \) and stop; otherwise go to Step 2

Step 2  Divide \( m \) by \( n \) and assign the value for the remainder to \( r \)

Step 3  Assign the value of \( n \) to \( m \) and the value of \( r \) to \( n \). Go to Step 1.

while \( n \neq 0 \) do
    \[ r \leftarrow m \mod n \]
    \[ m \leftarrow n \]
    \[ n \leftarrow r \]
return \( m \)
Other methods for computing \( \gcd(m,n) \)

**Consecutive integer checking algorithm**

**Step 1** Assign the value of \( \min\{m,n\} \) to \( t \)

**Step 2** Divide \( m \) by \( t \). If the remainder is 0, go to Step 3; otherwise, go to Step 4

**Step 3** Divide \( n \) by \( t \). If the remainder is 0, return \( t \) and stop; otherwise, go to Step 4

**Step 4** Decrease \( t \) by 1 and go to Step 2
Other methods for gcd($m,n$) [cont.]

Middle-school procedure

Step 1  Find the prime factorization of $m$
Step 2  Find the prime factorization of $n$
Step 3  Find all the common prime factors
Step 4  Compute the product of all the common prime factors and return it as gcd($m,n$)

Is this an algorithm?
Sieve of Eratosthenes

Algorithm for generating consecutive primes not exceeding any given integer \( n > 1 \).

Input: Integer \( n \geq 2 \)
Output: List of primes less than or equal to \( n \)

for \( p \leftarrow 2 \) to \( n \) do
  \( A[p] \leftarrow p \)

for \( p \leftarrow 2 \) to \( \lfloor \sqrt{n} \rfloor \) do
  if \( A[p] \neq 0 \) // \( p \) hasn’t been previously eliminated from the list
    \( j \leftarrow p \cdot p \) // all smaller multiples \( 2p, \ldots, (p-1)p \) are eliminated
    while \( j \leq n \) do
      \( A[j] \leftarrow 0 \) // mark element as eliminated
      \( j \leftarrow j + p \) // \( p \cdot (p+1), p \cdot (p+2), \ldots \)
//copy the remaining elements of A to array L of the primes

i ← 0

for p ← 2 to n do
  if A[p] ≠ 0
    L[i] ← A[p]
    i ← i + 1

return L
As an example, consider the application of the algorithm to finding the list of primes not exceeding \( n = 25 \):

\[
\begin{array}{cccccccccccccccc}
2 & 3 & 5 & 7 & 9 & 11 & 13 & 15 & 17 & 19 & 21 & 23 & 25 \\
2 & 3 & 5 & 7 & 11 & 13 & 17 & 19 & 23 & 25 \\
2 & 3 & 5 & 7 & 11 & 13 & 17 & 19 & 23
\end{array}
\]

For this example, no more passes are needed because they would eliminate numbers already eliminated on previous iterations of the algorithm. The remaining numbers on the list are the consecutive primes less than or equal to 25.
Why study algorithms?

- Theoretical importance
  - the core of computer science

- Practical importance
  - A practitioner’s toolkit of known algorithms
  - Framework for designing and analyzing algorithms for new problems
Two main issues related to algorithms

- How to design algorithms
- How to analyze algorithm efficiency
Design and Analysis of Algorithm

Algorithm design and analysis process:

1. Understand the problem
2. Decide on:
   - computational means,
   - exact vs. approximate solving,
   - algorithm design technique
3. Design an algorithm
4. Prove correctness
5. Analyze the algorithm
6. Code the algorithm
Algorithm design techniques/strategies

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs

- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and bound
Analysis of algorithms

How good is the algorithm?
- time efficiency
- space efficiency

Does there exist a better algorithm?
- lower bounds
- optimality
Important problem types

- **sorting**
  - *Two properties: stable and in-place*

- **searching**

- **string processing**

- **graph problems**
  - *Traveling salesman problem, graph-coloring problem*

- **combinatorial problems**
  - *Finding a combinatorial object that satisfies certain constraints*

- **geometric problems**
  - *Closest-pair problem, convex-hull problem*

- **numerical problems**
  - *Example: solving equations, evaluating functions, computing definite integrals and so on*
Fundamental data structures

- list
  - array
  - linked list
  - string
- stack
  \[ LIFO \]
- queue
  \[ FIFO \]
- priority queue
  \[ \text{HEAP} \]
- Graph
  \[ G = \langle V, E \rangle \]
  \[ \text{Representation:} \]
  \[ \text{Adjacency matrix} \]
  \[ \text{Adjacency list} \]
- Tree (connected acyclic graph)
- set and dictionary
**FIGURE 1.3** Array of $n$ elements.

**FIGURE 1.4** Singly linked list of $n$ elements.

**FIGURE 1.5** Doubly linked list of $n$ elements.
FIGURE 1.6 (a) Undirected graph. (b) Digraph.

FIGURE 1.7 (a) Adjacency matrix and (b) adjacency lists of the graph in Figure 1.6a.
FIGURE 1.8 (a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.

FIGURE 1.9 Graph that is not connected.
FIGURE 1.10 (a) Tree. (b) Forest.

FIGURE 1.11 (a) Free tree. (b) Its transformation into a rooted tree.
FIGURE 1.12 (a) Binary tree. (b) Binary search tree.

FIGURE 1.13 Standard implementation of the binary search tree in Figure 1.12b.
Chapter 2: Getting Started

Chapter 2 overview

Goals

- Start using frameworks for describing and analyzing algorithms.
- Examine two algorithms for sorting: insertion sort and merge sort.
- See how to describe algorithms in pseudocode.
- Begin using asymptotic notation to express running-time analysis.
- Learn the technique of “divide and conquer” in the context of merge sort.

Insertion sort

The sorting problem

Input: A sequence of $n$ numbers $(a_1, a_2, \ldots, a_n)$.
Output: A permutation (reordering) $(a'_1, a'_2, \ldots, a'_n)$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

The sequences are typically stored in arrays.
We also refer to the numbers as keys. Along with each key may be additional information, known as satellite data. [You might want to clarify that “satellite data” does not necessarily come from a satellite.]

We will see several ways to solve the sorting problem. Each way will be expressed as an algorithm: a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

Expressing algorithms

We express algorithms in whatever way is the clearest and most concise. English is sometimes the best way.
Chapter 2: Getting Started

- Pseudocode is similar to C, C++, Pascal, and Java. If you know any of these languages, you should be able to understand pseudocode.
- Pseudocode is designed for *expressing algorithms to humans*. Software engineering issues of data abstraction, modularity, and error handling are often ignored.
- We sometimes embed English statements into pseudocode. Therefore, unlike for “real” programming languages, we cannot create a compiler that translates pseudocode to machine code.

**Insertion sort**

A good algorithm for sorting a small number of elements.

It works the way you might sort a hand of playing cards:

- Start with an empty left hand and the cards face down on the table.
- Then remove one card at a time from the table, and insert it into the correct position in the left hand.
- To find the correct position for a card, compare it with each of the cards already in the hand, from right to left.
- At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.
Insertion Sort

**Example**

```
Example

<table>
<thead>
<tr>
<th>j</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>j</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
```

**cost times**

- $c_1 \ n$
- $c_2 \ n - 1$
- $c_3 \ 0 \ n - 1$
- $c_4 \ n - 1$
- $c_5 \ \sum_{j=2}^{n} t_j$
- $c_6 \ \sum_{j=2}^{n} (t_j - 1)$
- $c_7 \ \sum_{j=2}^{n} (t_j - 1)$
- $c_8 \ n - 1$

**Algorithm**

**INSERTION-SORT** $(A, n)$

for $j = 2$ to $n$

\[ \text{key} = A[j] \]


$\ i = j - 1$

while $i > 0$ and $A[i] > \text{key}$

\[ A[i + 1] = A[i] \]

\[ i = i - 1 \]

\[ A[i + 1] = \text{key} \]
Insertion Sort

Correctness

We often use a loop invariant to help us understand why an algorithm gives the correct answer. Here’s the loop invariant for INSERTION-SORT:

**Loop invariant:** At the start of each iteration of the “outer” for loop—the loop indexed by \( j \)—the subarray \( A[1..j-1] \) consists of the elements originally in \( A[1..j-1] \) but in sorted order.

To use a loop invariant to prove correctness, we must show three things about it:

**Initialization:** It is true prior to the first iteration of the loop.

**Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.

**Termination:** When the loop terminates, the invariant—usually along with the reason that the loop terminated—gives us a useful property that helps show that the algorithm is correct.

Using loop invariants is like mathematical induction:

- To prove that a property holds, you prove a base case and an inductive step.
- Showing that the invariant holds before the first iteration is like the base case.
- Showing that the invariant holds from iteration to iteration is like the inductive step.
- The termination part differs from the usual use of mathematical induction, in which the inductive step is used infinitely. We stop the “induction” when the loop terminates.
- We can show the three parts in any order.
Insertion Sort

For insertion sort

Initialization: Just before the first iteration, \( j = 2 \). The subarray \( A[1 \ldots j - 1] \) is the single element \( A[1] \), which is the element originally in \( A[1] \), and it is trivially sorted.

Maintenance: To be precise, we would need to state and prove a loop invariant for the “inner” while loop. Rather than getting bogged down in another loop invariant, we instead note that the body of the inner while loop works by moving \( A[j - 1], A[j - 2], A[j - 3] \), and so on, by one position to the right until the proper position for key (which has the value that started out in \( A[j] \)) is found. At that point, the value of key is placed into this position.

Termination: The outer for loop ends when \( j > n \), which occurs when \( j = n + 1 \). Therefore, \( j - 1 = n \). Plugging \( n \) in for \( j - 1 \) in the loop invariant, the subarray \( A[1 \ldots n] \) consists of the elements originally in \( A[1 \ldots n] \) but in sorted order. In other words, the entire array is sorted.
Pseudocode conventions

[Covering most, but not all, here. See book pages 20–22 for all conventions.]

- Indentation indicates block structure. Saves space and writing time.

- Looping constructs are like in C, C++, Pascal, and Java. We assume that the loop variable in a for loop is still defined when the loop exits (unlike in Pascal).

- // indicates that the remainder of the line is a comment.

- Variables are local, unless otherwise specified.

- We often use objects, which have attributes. For an attribute attr of object x, we write x.attr. (This notation matches x.attr in Java and is equivalent to x->attr in C++. Attributes can cascade, so that if x.y is an object and this object has attribute attr, then x.y.attr indicates this object’s attribute. That is, x.y.attr is implicitly parenthesized as (x.y).attr.

- Objects are treated as references, like in Java. If x and y denote objects, then the assignment y = x makes x and y reference the same object. It does not cause attributes of one object to be copied to another.

- Parameters are passed by value, as in Java and C (and the default mechanism in Pascal and C++). When an object is passed by value, it is actually a reference (or pointer) that is passed; changes to the reference itself are not seen by the caller, but changes to the object’s attributes are.

- The boolean operators “and” and “or” are short-circuiting: if after evaluating the left-hand operand, we know the result of the expression, then we don’t evaluate the right-hand operand. (If x is FALSE in “x and y” then we don’t evaluate y. If x is TRUE in “x or y” then we don’t evaluate y.)
Analyzing algorithms

We want to predict the resources that the algorithm requires. Usually, running time. In order to predict resource requirements, we need a computational model.

Random-access machine (RAM) model

• Instructions are executed one after another. No concurrent operations.
• It’s too tedious to define each of the instructions and their associated time costs.
• Instead, we recognize that we’ll use instructions commonly found in real computers:
  • Arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling. Also, shift left/shift right (good for multiplying/dividing by $2^k$).
  • Data movement: load, store, copy.
  • Control: conditional/unconditional branch, subroutine call and return.
• Each of these instructions takes a constant amount of time.

The RAM model uses integer and floating-point types.

• We don’t worry about precision, although it is crucial in certain numerical applications.
• There is a limit on the word size: when working with inputs of size $n$, assume that integers are represented by $c \lg n$ bits for some constant $c \geq 1$. ($\lg n$ is a very frequently used shorthand for $\log_2 n$.)
  • $c \geq 1 \Rightarrow$ we can hold the value of $n \Rightarrow$ we can index the individual elements.
  • $c$ is a constant $\Rightarrow$ the word size cannot grow arbitrarily.

How do we analyze an algorithm’s running time?

The time taken by an algorithm depends on the input.

• Sorting 1000 numbers takes longer than sorting 3 numbers.
• A given sorting algorithm may even take differing amounts of time on two inputs of the same size.
• For example, we’ll see that insertion sort takes less time to sort $n$ elements when they are already sorted than when they are in reverse sorted order.
Analysis of Algorithms

**Input size**

Depends on the problem being studied.

- Usually, the number of items in the input. Like the size $n$ of the array being sorted.
- But could be something else. If multiplying two integers, could be the total number of bits in the two integers.
- Could be described by more than one number. For example, graph algorithm running times are usually expressed in terms of the number of vertices and the number of edges in the input graph.

**Running time**

On a particular input, it is the number of primitive operations (steps) executed.

- Want to define steps to be machine-independent.
- Figure that each line of pseudocode requires a constant amount of time.
- One line may take a different amount of time than another, but each execution of line $i$ takes the same amount of time $c_i$.
- This is assuming that the line consists only of primitive operations.
  - If the line is a subroutine call, then the actual call takes constant time, but the execution of the subroutine being called might not.
  - If the line specifies operations other than primitive ones, then it might take more than constant time. Example: “sort the points by $x$-coordinate.”
Analysis of insertion sort

[Now add statement costs and number of times executed to INSERTION-SORT pseudocode.]

- Assume that the $i$th line takes time $c_i$, which is a constant. (Since the third line is a comment, it takes no time.)
- For $j = 2, 3, \ldots, n$, let $t_j$ be the number of times that the while loop test is executed for that value of $j$.
- Note that when a for or while loop exits in the usual way—due to the test in the loop header—the test is executed one time more than the loop body.

The running time of the algorithm is

$$\sum_{\text{all statements}} \text{(cost of statement)} \cdot \text{(number of times statement is executed)}.$$

Let $T(n) =$ running time of INSERTION-SORT.

$$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$

$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1).$$

The running time depends on the values of $t_j$. These vary according to the input.
ININSERTION-SORT(A, n)

for j = 2 to n
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key

Best case
The array is already sorted.

• Always find that A[i] ≤ key upon the first time the while loop test is run (when i = j - 1).
• All t_j are 1.
• Running time is
  \[ T(n) = c_1n + c_2(n - 1) + c_4(n - 1) + c_5(n - 1) + c_8(n - 1) \]
  \[ = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \]
• Can express \( T(n) \) as \( an + b \) for constants \( a \) and \( b \) (that depend on the statement costs \( c_i \)) \( \Rightarrow T(n) \) is a linear function of \( n \).
Analysis of Algorithms

**Worst case**
The array is in reverse sorted order.

- Always find that $A[i] > key$ in while loop test.
- Have to compare $key$ with all elements to the left of the $j$th position $\Rightarrow$ compare with $j - 1$ elements.
- Since the while loop exits because $i$ reaches 0, there’s one additional test after the $j - 1$ tests $\Rightarrow t_j = j$.

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j \text{ and } \sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j - 1).$$

- $\sum_{j=1}^{n} j$ is known as an *arithmetic series*, and equation (A.1) shows that it equals
  $$\frac{n(n + 1)}{2}.$$ 

- Since $\sum_{j=2}^{n} j = \left( \sum_{j=1}^{n} j \right) - 1$, it equals $\frac{n(n + 1)}{2} - 1$.

```
INSERTION-SORT(A, n)
    for j = 2 to n
        key = A[j]
        // Insert A[j] into the sorted sequence A[1..j - 1].
        i = j - 1
        while i > 0 and A[i] > key
            A[i + 1] = A[i]
            i = i - 1
        A[i + 1] = key
```

<table>
<thead>
<tr>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
<td>$n$</td>
</tr>
<tr>
<td>$c_2$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>$c_3$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>$c_4$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>$c_5$</td>
<td>$\sum_{j=2}^{n} t_j$</td>
</tr>
<tr>
<td>$c_6$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
</tr>
<tr>
<td>$c_7$</td>
<td>$\sum_{j=2}^{n} (j - 1)$</td>
</tr>
<tr>
<td>$c_8$</td>
<td>$n - 1$</td>
</tr>
</tbody>
</table>
Analysis of Algorithms

- Letting $k = j - 1$, we see that $\sum_{j=2}^{n}(j - 1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$.

- Running time is

$$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \left( \frac{n(n + 1)}{2} - 1 \right)$$

$$+ c_6 \left( \frac{n(n - 1)}{2} \right) + c_7 \left( \frac{n(n - 1)}{2} \right) + c_8 (n - 1)$$

$$= \left( \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left( c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n$$

$$- (c_2 + c_4 + c_5 + c_8).$$

- Can express $T(n)$ as $an^2 + bn + c$ for constants $a, b, c$ (that again depend on statement costs) $\Rightarrow$ $T(n)$ is a quadratic function of $n$. 

Analysis of Algorithms

Worst-case and average-case analysis

We usually concentrate on finding the **worst-case running time**: the longest running time for *any* input of size $n$.

**Reasons**

- The worst-case running time gives a guaranteed upper bound on the running time for any input.
- For some algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for absent items may be frequent.
- Why not analyze the average case? Because it’s often about as bad as the worst case.

**Example:** Suppose that we randomly choose $n$ numbers as the input to insertion sort.

On average, the key in $A[j]$ is less than half the elements in $A[1 \ldots j - 1]$ and it’s greater than the other half.

$\Rightarrow$ On average, the **while** loop has to look halfway through the sorted subarray $A[1 \ldots j - 1]$ to decide where to drop $key$.

$\Rightarrow$ $t_f \approx j/2$.

Although the average-case running time is approximately half of the worst-case running time, it’s still a quadratic function of $n$. 

Order of growth

Another abstraction to ease analysis and focus on the important features. Look only at the leading term of the formula for running time.

- Drop lower-order terms.
- Ignore the constant coefficient in the leading term.

Example: For insertion sort, we already abstracted away the actual statement costs to conclude that the worst-case running time is \( an^2 + bn + c \). Drop lower-order terms \( \Rightarrow an^2 \).
Ignore constant coefficient \( \Rightarrow n^2 \).

But we cannot say that the worst-case running time \( T(n) \) equals \( n^2 \). It grows like \( n^2 \). But it doesn’t equal \( n^2 \).

We say that the running time is \( \Theta(n^2) \) to capture the notion that the order of growth is \( n^2 \).

We usually consider one algorithm to be more efficient than another if its worst-case running time has a smaller order of growth.
Designing algorithms

There are many ways to design algorithms. For example, insertion sort is *incremental*: having sorted $A[1 \ldots j - 1]$, place $A[j]$ correctly, so that $A[1 \ldots j]$ is sorted.

**Divide and conquer**

Another common approach.

**Divide** the problem into a number of subproblems that are smaller instances of the same problem.

**Conquer** the subproblems by solving them recursively.

*Base case:* If the subproblems are small enough, just solve them by brute force.

**Combine** the subproblem solutions to give a solution to the original problem.
Merge sort

A sorting algorithm based on divide and conquer. Its worst-case running time has a lower order of growth than insertion sort.

Because we are dealing with subproblems, we state each subproblem as sorting a subarray $A[p..r]$. Initially, $p = 1$ and $r = n$, but these values change as we recurse through subproblems.

To sort $A[p..r]$:

**Divide** by splitting into two subarrays $A[p..q]$ and $A[q+1..r]$, where $q$ is the halfway point of $A[p..r]$.

**Conquer** by recursively sorting the two subarrays $A[p..q]$ and $A[q+1..r]$.

**Combine** by merging the two sorted subarrays $A[p..q]$ and $A[q+1..r]$ to produce a single sorted subarray $A[p..r]$. To accomplish this step, we’ll define a procedure $\text{MERGE}(A, p, q, r)$.

The recursion bottoms out when the subarray has just 1 element, so that it’s trivially sorted.

$\text{MERGE-SORT}(A, p, r)$

```
if $p < r$ // check for base case
    $q = \lfloor (p + r)/2 \rfloor$ // divide
    $\text{MERGE-SORT}(A, p, q)$ // conquer
    $\text{MERGE-SORT}(A, q + 1, r)$ // conquer
    $\text{MERGE}(A, p, q, r)$ // combine
```

*Initial call:* $\text{MERGE-SORT}(A, 1, n)$
Example

Bottom-up view for $n = 8$: [Heavy lines demarcate subarrays used in subproblems.]

```
sorted array
1 2 2 3 4 5 6 7

merge
2 4 5 7 1 2 3 6

merge
2 5 4 7 1 3 2 6

merge
5 2 4 7 1 3 2 6

initial array
1 2 3 4 5 6 7 8
```
Designing Algorithms

Bottom-up view for $n = 11$:

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

merge

```

```
| 1 | 2 | 4 | 4 | 6 | 7 | 2 | 3 | 5 | 6 | 7 |

merge

```

```
| 2 | 4 | 7 | 1 | 4 | 6 | 3 | 5 | 7 | 2 | 6 |

merge

```

```
| 4 | 7 | 2 | 1 | 6 | 4 | 3 | 7 | 5 | 2 | 6 |

merge

```

```
| 4 | 7 | 2 | 6 | 1 | 4 | 7 | 3 | 5 | 2 | 6 |

merge

```

```
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

initial array

```

[Here, at the next-to-last level of recursion, some of the subproblems have only 1 element. The recursion bottoms out on these single-element subproblems.]
Merging

What remains is the \texttt{MERGE} procedure.

\textbf{Input:} Array $A$ and indices $p, q, r$ such that

- $p \leq q < r$.
- Subarray $A[p..q]$ is sorted and subarray $A[q + 1..r]$ is sorted. By the restrictions on $p, q, r$, neither subarray is empty.

\textbf{Output:} The two subarrays are merged into a single sorted subarray in $A[p..r]$.

We implement it so that it takes $\Theta(n)$ time, where $n = r - p + 1 = \text{the number of elements being merged.}$
Designing Algorithms

Pseudocode

MERGE(A, p, q, r)

\[ n_1 = q - p + 1 \]
\[ n_2 = r - q \]

let \( L[1 \ldots n_1 + 1] \) and \( R[1 \ldots n_2 + 1] \) be new arrays

for \( i = 1 \) to \( n_1 \)
    \( L[i] = A[p + i - 1] \)

for \( j = 1 \) to \( n_2 \)
    \( R[j] = A[q + j] \)

\( L[n_1 + 1] = \infty \)
\( R[n_2 + 1] = \infty \)

\( i = 1 \)
\( j = 1 \)

for \( k = p \) to \( r \)
    if \( L[i] \leq R[j] \)
        \( A[k] = L[i] \)
        \( i = i + 1 \)
    else \( A[k] = R[j] \)
        \( j = j + 1 \)
Example:

A call of Merge(A,9,12,16)
Designing Algorithms

Running time

The first two **for** loops take $\Theta(n_1 + n_2) = \Theta(n)$ time. The last **for** loop makes $n$ iterations, each taking constant time, for $\Theta(n)$ time. Total time: $\Theta(n)$.

Analyzing divide-and-conquer algorithms

Use a *recurrence equation* (more commonly, a *recurrence*) to describe the running time of a divide-and-conquer algorithm.

Let $T(n)$ = running time on a problem of size $n$.

- If the problem size is small enough (say, $n \leq c$ for some constant $c$), we have a base case. The brute-force solution takes constant time: $\Theta(1)$.
- Otherwise, suppose that we divide into $a$ subproblems, each $1/b$ the size of the original. (In merge sort, $a = b = 2$.)
- Let the time to divide a size-$n$ problem be $D(n)$.
- Have $a$ subproblems to solve, each of size $n/b \Rightarrow$ each subproblem takes $T(n/b)$ time to solve $\Rightarrow$ we spend $aT(n/b)$ time solving subproblems.
- Let the time to combine solutions be $C(n)$.
- We get the recurrence

$$T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq c, \\
 aT(n/b) + D(n) + C(n) & \text{otherwise}.
\end{cases}$$
Designing Algorithms

Analyzing merge sort

For simplicity, assume that $n$ is a power of 2 \( \Rightarrow \) each divide step yields two sub-problems, both of size exactly $n/2$.

The base case occurs when $n = 1$.

When $n \geq 2$, time for merge sort steps:

**Divide:** Just compute $q$ as the average of $p$ and $r$ \( \Rightarrow \) $D(n) = \Theta(1)$.

**Conquer:** Recursively solve 2 subproblems, each of size $n/2$ \( \Rightarrow \) $2T(n/2)$.

**Combine:** **Merge** on an $n$-element subarray takes $\Theta(n)$ time \( \Rightarrow \) $C(n) = \Theta(n)$.

Since $D(n) = \Theta(1)$ and $C(n) = \Theta(n)$, summed together they give a function that is linear in $n$: $\Theta(n)$ \( \Rightarrow \) recurrence for merge sort running time is

$$T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
2T(n/2) + \Theta(n) & \text{if } n > 1.
\end{cases}$$

Solving the merge-sort recurrence

By the master theorem in Chapter 4, we can show that this recurrence has the solution $T(n) = \Theta(n \lg n)$. [Reminder: \( \lg n \) stands for \( \log_2 n \).]

Compared to insertion sort (\( \Theta(n^2) \) worst-case time), merge sort is faster. Trading a factor of $n$ for a factor of $\lg n$ is a good deal.
On small inputs, insertion sort may be faster. But for large enough inputs, merge sort will always be faster, because its running time grows more slowly than insertion sort’s.
We can understand how to solve the merge-sort recurrence without the master theorem.

- Let $c$ be a constant that describes the running time for the base case and also is the time per array element for the divide and conquer steps. [Of course, we cannot necessarily use the same constant for both. It’s not worth going into this detail at this point.]
- We rewrite the recurrence as
  \[
  T(n) = \begin{cases} 
  c & \text{if } n = 1, \\
  2T(n/2) + cn & \text{if } n > 1.
  \end{cases}
  \]
- Draw a recursion tree, which shows successive expansions of the recurrence.
- For the original problem, we have a cost of $cn$, plus the two subproblems, each costing $T(n/2)$:
  \[
  \begin{array}{c}
  \text{cn} \\
  \text{---} \\
  \text{T(n/2)} \quad \text{T(n/2)}
  \end{array}
  \]
- For each of the size-$n/2$ subproblems, we have a cost of $cn/2$, plus two subproblems, each costing $T(n/4)$:
  \[
  \begin{array}{c}
  \text{cn} \\
  \text{---} \\
  \text{cn/2} \quad \text{cn/2} \\
  \text{---} \\
  \text{T(n/4)} \quad \text{T(n/4)} \quad \text{T(n/4)} \quad \text{T(n/4)}
  \end{array}
  \]
- Continue expanding until the problem sizes get down to 1:
Designing Algorithms

- Each level has cost $cn$.
  - The top level has cost $cn$.
  - The next level down has 2 subproblems, each contributing cost $cn/2$.
  - The next level has 4 subproblems, each contributing cost $cn/4$.
  - Each time we go down one level, the number of subproblems doubles but the cost per subproblem halves $\Rightarrow$ cost per level stays the same.
- There are $\lg n + 1$ levels (height is $\lg n$).
  - Use induction.
  - Base case: $n = 1 \Rightarrow 1$ level, and $\lg 1 + 1 = 0 + 1 = 1$.
  - Inductive hypothesis is that a tree for a problem size of $2^i$ has $\lg 2^i + 1 = i + 1$ levels.
  - Because we assume that the problem size is a power of 2, the next problem size up after $2^i$ is $2^{i+1}$.
  - A tree for a problem size of $2^{i+1}$ has one more level than the size-$2^i$ tree $\Rightarrow i + 2$ levels.
  - Since $\lg 2^{i+1} + 1 = i + 2$, we’re done with the inductive argument.
- Total cost is sum of costs at each level. Have $\lg n + 1$ levels, each costing $cn$ $\Rightarrow$ total cost is $cn \lg n + cn$.
- Ignore low-order term of $cn$ and constant coefficient $c$ $\Rightarrow \Theta(n \lg n)$.
Chapter 3
Chapter 3: Growth of Functions

Chapter 3 overview

• A way to describe behavior of functions *in the limit*. We’re studying asymptotic efficiency.
• Describe growth of functions.
• Focus on what’s important by abstracting away low-order terms and constant factors.
• How we indicate running times of algorithms.
• A way to compare “sizes” of functions:
  \[ O \approx \leq \]
  \[ \Omega \approx \geq \]
  \[ \Theta \approx = \]
  \[ o \approx < \]
  \[ \omega \approx > \]
Orders of Growth

- $C(n)$: the cost (e.g., number of steps) required by an algorithm on an input of size $n$.

- **Efficiency**: how cost grows with the size $n$ of a given problem instance.

- **Order of growth**: the functional form of $C(n)$ up to a constant multiple as $n$ goes to infinity.
# Orders of Growth

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log_2 n$</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.3</td>
<td>10</td>
<td>3.3*10</td>
<td>10²</td>
<td>10³</td>
</tr>
<tr>
<td>$10^2$</td>
<td>6.6</td>
<td>$10^2$</td>
<td>6.6*10²</td>
<td>10⁴</td>
<td>10⁶</td>
</tr>
<tr>
<td>$10^3$</td>
<td>10</td>
<td>$10^3$</td>
<td>1.0*10⁴</td>
<td>10⁶</td>
<td>10⁹</td>
</tr>
<tr>
<td>$10^4$</td>
<td>13</td>
<td>$10^4$</td>
<td>1.3*10⁵</td>
<td>10⁸</td>
<td>10¹²</td>
</tr>
<tr>
<td>$10^5$</td>
<td>17</td>
<td>$10^5$</td>
<td>1.7*10⁶</td>
<td>10¹⁰</td>
<td>10¹⁵</td>
</tr>
<tr>
<td>$10^6$</td>
<td>20</td>
<td>$10^6$</td>
<td>2.0*10⁷</td>
<td>10¹²</td>
<td>10¹⁸</td>
</tr>
</tbody>
</table>
## Orders of Growth

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log_2 n$</th>
<th>$n$</th>
<th>$n\log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.3</td>
<td>10</td>
<td>3.3$\times$10</td>
<td>10$^2$</td>
<td>10$^3$</td>
<td>~10$^3$</td>
<td>~3.6$\times$10$^6$</td>
</tr>
<tr>
<td>10$^2$</td>
<td>6.6</td>
<td>10$^2$</td>
<td>6.6$\times$10$^2$</td>
<td>10$^4$</td>
<td>10$^6$</td>
<td>~1.3$\times$10$^{30}$</td>
<td>~9$\times$10$^{157}$</td>
</tr>
<tr>
<td>10$^3$</td>
<td>10</td>
<td>10$^3$</td>
<td>1.0$\times$10$^4$</td>
<td>10$^6$</td>
<td>10$^9$</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10$^4$</td>
<td>13</td>
<td>10$^4$</td>
<td>1.3$\times$10$^5$</td>
<td>10$^8$</td>
<td>10$^{12}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10$^5$</td>
<td>17</td>
<td>10$^5$</td>
<td>1.7$\times$10$^6$</td>
<td>10$^{10}$</td>
<td>10$^{15}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10$^6$</td>
<td>20</td>
<td>10$^6$</td>
<td>2.0$\times$10$^7$</td>
<td>10$^{12}$</td>
<td>10$^{18}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Efficient
Kinds of Efficiency

• Need to decide which instance of a given size to use as the representative for that class:

  • *Worst Case*
  • *Best Case*
  • *Average Case*  
    (over all possible instances)
Asymptotic Notation

• Definition: given an asymptotically non-negative fn. $g(n)$,

$$\Theta(g(n)) = \{f(n): \exists (c_1, c_2, n_0) > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0\}$$

• Translation:
  – An asymptotically non-negative function $f(n)$ belongs to the set $\Theta(g(n))$ if and only if there exist positive constants $c_1$ and $c_2$ such that $f(n)$ can be “sandwiched” between $c_1 g(n)$ and $c_2 g(n)$, scaled versions of $g(n)$, for sufficiently large $n$. 
Asymptotic Notation

Definitions: given an asymptotically non-negative $f(n)$, $g(n)$,

$\Theta(g(n)) = \{ f(n) : \exists (c_1, c_2, n_0) > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0 \}$

$O(g(n)) = \{ f(n) : \exists (c, n_0) > 0 \text{ such that } 0 \leq f(n) \leq c g(n) \forall n \geq n_0 \}$

$\Omega(g(n)) = \{ f(n) : \exists (c, n_0) > 0 \text{ such that } 0 \leq c g(n) \leq f(n) \forall n \geq n_0 \}$
Notational Convention

• Unfortunately,  
  \[ f(n) = \Theta(g(n)) \] means \( f(n) \in \Theta(g(n)) \)
Asymptotic Notation

\[ f(n) = \mathcal{O}(g(n)) \]

\[ f(n) = \Omega(g(n)) \]

\[ f(n) = \Theta(g(n)) \]
Cases vs. Orders

• Do not confuse worst-case with $O()$
Distinguishing $O$ from $\Theta$

• Can you draw a function that is in $O(n^2)$ but not $\Theta(n^2)$?
Distinguishing $O$ from $\Theta$

• Can you draw a function that is in $O(n^2)$ but not $\Theta(n^2)$?
Example

• Show that \( \frac{1}{2} n^2 - 3n \in \Theta(n^2) \)

• Must find positive constants \( c_1, c_2, n_0 \) such that

\[
c_1 n^2 \leq \frac{1}{2} n^2 - 3n \leq c_2 n^2 \quad \forall n \geq n_0
\]

• Divide through by \( n^2 \) to get

\[
c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2 \quad \forall n \geq n_0
\]

• Hint: consider one side of the inequality at a time:
  – for \( n_0 = 7 \) we have \( 0 < c_1 \leq \frac{1}{14} \)
  – for \( n_0 = 7 \) we can chose \( c_2 \geq \frac{1}{2} \)

• This proof is constructive
Another Example

Show that $6n^3 \notin \Theta(n^2)$

- Use proof by contradiction: assume that $6n^3 \in \Theta(n^2)$
- Suppose positive constants $c_2, n_0$ exist such that $6n^3 \leq c_2 n^2 \forall n \geq n_0$
- But this implies that $n \leq \frac{c_2}{6} \forall n \geq n_0$
- i.e., $n$ is bounded by $\frac{c_2}{6}$, a constant
- But $n$ is unbounded; hence we have a contradiction.
- Thus, our assumption was false; hence we have shown that $6n^3 \notin \Theta(n^2)$
Example

$2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

Examples of functions in $O(n^2)$:

$n^2$

$n^2 + n$

$n^2 + 1000n$

$1000n^2 + 1000n$

Also,

$n$

$n/1000$

$n^{1.99999}$

$n^2 / \lg \lg \lg n$
Example

\( \sqrt{n} = \Omega(lg \, n) \), with \( c = 1 \) and \( n_0 = 16 \).

Examples of functions in \( \Omega(n^2) \): 

- \( n^2 \)
- \( n^2 + n \)
- \( n^2 - n \)
- \( 1000n^2 + 1000n \)
- \( 1000n^2 - 1000n \)

Also,

- \( n^3 \)
- \( n^{2.00001} \)
- \( n^2 \, lg \, lg \, lg \, n \)
- \( 2^{2^n} \)
Other proof types?

- Induction
- Deduction
- Others!
Duality

\[ f(n) \in \Omega(g(n)) \text{ if and only if } g(n) \in O(f(n)) \]
The Limit Rule

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \begin{cases} \in R^+ & \text{then } f(n) \in O(g(n)) \land g(n) \in O(f(n)) \\ = 0 & \text{then } f(n) \in O(g(n)) \land g(n) \not\in O(f(n)) \\ = +\infty & \text{then } f(n) \not\in O(g(n)) \land g(n) \in O(f(n)) \end{cases}$$
The Limit Rule

\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} \begin{cases} \in R^+ & \text{then } f(n) \in O(g(n)) \land g(n) \in O(f(n)) \\ = 0 & \text{then } f(n) \in O(g(n)) \land g(n) \not\in O(f(n)) \\ = +\infty & \text{then } f(n) \not\in O(g(n)) \land g(n) \in O(f(n)) \end{cases} \]
Example
Example
Useful Identity: L’Hopital’s Rule

Applicable when:
• \( \lim f(n) = \lim g(n) = 0 \)
• \( \lim f(n) = \lim g(n) = \infty \)

As \( n \to \infty \)
Review: Log Identities
Review: Log Identities
Review: More Logarithms
Review: More Logarithms
Multiplication
Classic Multiplication

American Style

\[
\begin{array}{c}
5001 \\
\times \ 
502 \\
10002 \\
0 \\
+ 25005 \\
\hline
2510502 \\
\end{array}
\]

English Style

\[
\begin{array}{c}
5001 \\
\times \ 502 \\
25005 \\
0 \\
+ 10002 \\
\hline
2510502 \\
\end{array}
\]

O(n²) for 2 n-digit numbers
Theorem

\[ f(n) = \Theta(g(n)) \text{ if and only if } f = O(g(n)) \text{ and } f = \Omega(g(n)). \]

Leading constants and low-order terms don’t matter.

Asymptotic notation in equations

When on right-hand side

\( O(n^2) \) stands for some anonymous function in the set \( O(n^2) \).

\[ 2n^2 + 3n + 1 = 2n^2 + \Theta(n) \] means \[ 2n^2 + 3n + 1 = 2n^2 + f(n) \text{ for some } f(n) \in \Theta(n). \] In particular, \( f(n) = 3n + 1 \).

By the way, we interpret \( \# \) of anonymous functions as \( \# \) of times the asymptotic notation appears:

\[
\sum_{i=1}^{n} O(i) \quad \text{OK: 1 anonymous function}
\]

\[ O(1) + O(2) + \cdots + O(n) \quad \text{not OK: } n \text{ hidden constants} \]

\[ \Rightarrow \text{no clean interpretation} \]
Growth of Functions

*When on left-hand side*

No matter how the anonymous functions are chosen on the left-hand side, there is a way to choose the anonymous functions on the right-hand side to make the equation valid.

Interpret $2n^2 + \Theta(n) = \Theta(n^2)$ as meaning for all functions $f(n) \in \Theta(n)$, there exists a function $g(n) \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n)$.

Can chain together:

\[
2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2).
\]

Interpretation:

- First equation: There exists $f(n) \in \Theta(n)$ such that $2n^2 + 3n + 1 = 2n^2 + f(n)$.
- Second equation: For all $g(n) \in \Theta(n)$ (such as the $f(n)$ used to make the first equation hold), there exists $h(n) \in \Theta(n^2)$ such that $2n^2 + g(n) = h(n)$. 
Growth of Functions

**o-notation**

\[ o(g(n)) = \{ f(n) : \text{for all constants } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0 \} . \]

Another view, probably easier to use: \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0. \)

\[ n^{1.9999} = o(n^2) \]
\[ n^2 / \lg n = o(n^2) \]
\[ n^2 \neq o(n^2) \text{ (just like } 2 \neq 2) \]
\[ n^2 / 1000 \neq o(n^2) \]

**ω-notation**

\[ \omega(g(n)) = \{ f(n) : \text{for all constants } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0 \} . \]

Another view, again, probably easier to use: \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty. \)

\[ n^{2.0001} = \omega(n^2) \]
\[ n^2 \lg n = \omega(n^2) \]
\[ n^2 \neq \omega(n^2) \]
Growth of Functions

Comparisons of functions

Relational properties:

Transitivity:
\[ f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \implies f(n) = \Theta(h(n)). \]
Same for \( O, \Omega, o, \) and \( \omega. \)

Reflexivity:
\[ f(n) = \Theta(f(n)). \]
Same for \( O \) and \( \Omega. \)

Symmetry:
\[ f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)). \]

Transpose symmetry:
\[ f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)). \]
\[ f(n) = o(g(n)) \text{ if and only if } g(n) = \omega(f(n)). \]

Comparisons:

- \( f(n) \) is asymptotically smaller than \( g(n) \) if \( f(n) = o(g(n)). \)
- \( f(n) \) is asymptotically larger than \( g(n) \) if \( f(n) = \omega(g(n)). \)

No trichotomy. Although intuitively, we can liken \( O \) to \( \leq, \Omega \) to \( \geq, \) etc., unlike real numbers, where \( a < b, a = b, \) or \( a > b, \) we might not be able to compare functions.

Example: \( n^{1+\sin n} \) and \( n, \) since \( 1 + \sin n \) oscillates between 0 and 2.
Growth of Functions

Standard notations and common functions

Monotonicity

- \( f(n) \) is **monotonically increasing** if \( m \leq n \Rightarrow f(m) \leq f(n) \).
- \( f(n) \) is **monotonically decreasing** if \( m \geq n \Rightarrow f(m) \geq f(n) \).
- \( f(n) \) is **strictly increasing** if \( m < n \Rightarrow f(m) < f(n) \).
- \( f(n) \) is **strictly decreasing** if \( m > n \Rightarrow f(m) > f(n) \).

Exponentials

Useful identities:

\[
\begin{align*}
a^{-1} &= \frac{1}{a}, \\
(a^m)^n &= a^{mn}, \\
a^m a^n &= a^{m+n}.
\end{align*}
\]

Can relate rates of growth of polynomials and exponentials: for all real constants \( a \) and \( b \) such that \( a > 1 \),

\[
\lim_{n \to \infty} \frac{n^b}{a^n} = 0,
\]

which implies that \( n^b = o(a^n) \).

A surprisingly useful inequality: for all real \( x \),

\[
e^x \geq 1 + x.
\]

As \( x \) gets closer to 0, \( e^x \) gets closer to \( 1 + x \).
Growth of Functions

Logarithms

Notations:
\[ \lg n = \log_2 n \quad \text{(binary logarithm)}, \]
\[ \ln n = \log_e n \quad \text{(natural logarithm)}, \]
\[ \lg^k n = (\lg n)^k \quad \text{(exponentiation)}, \]
\[ \lg \lg n = \lg(\lg n) \quad \text{(composition)}. \]

In the expression \( \log_b a \):

- If we hold \( b \) constant, then the expression is strictly increasing as \( a \) increases.
- If we hold \( a \) constant, then the expression is strictly decreasing as \( b \) increases.
Growth of Functions

Useful identities for all real $a > 0, b > 0, c > 0$, and $n$, and where logarithm bases are not 1:

$$a = b^{\log_b a}$$
$$\log_c (ab) = \log_c a + \log_c b$$
$$\log_b a^n = n \log_b a$$
$$\log_b a = \frac{\log_c a}{\log_c b}$$
$$\log_b (1/a) = -\log_b a$$
$$\log_b a = \frac{1}{\log_a b}$$
$$a^{\log_b c} = c^{\log_b a}$$

Changing the base of a logarithm from one constant to another only changes the value by a constant factor, so we usually don’t worry about logarithm bases in asymptotic notation. Convention is to use $\lg$ within asymptotic notation, unless the base actually matters.
Growth of Functions

Just as polynomials grow more slowly than exponentials, logarithms grow more slowly than polynomials. In \( \lim_{n \to \infty} \frac{n^b}{a^n} = 0 \), substitute \( \lg n \) for \( n \) and \( 2^a \) for \( a \):

\[
\lim_{n \to \infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n \to \infty} \frac{\lg^b n}{n^a} = 0,
\]

implying that \( \lg^b n = o(n^a) \).

Factorials

\( n! = 1 \cdot 2 \cdot 3 \cdot n \). Special case: \( 0! = 1 \).
Can use Stirling’s approximation,

\[
n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right),
\]

to derive that \( \lg(n!) = \Theta(n \lg n) \).
Chapter 4 overview

Recall the divide-and-conquer paradigm, which we used for merge sort:

**Divide** the problem into a number of subproblems that are smaller instances of the same problem.

**Conquer** the subproblems by solving them recursively.

  **Base case:** If the subproblems are small enough, just solve them by brute force.

**Combine** the subproblem solutions to give a solution to the original problem.

We look at two more algorithms based on divide-and-conquer.
Divide and Conquer

Analyzing divide-and-conquer algorithms

Use a recurrence to characterize the running time of a divide-and-conquer algorithm. Solving the recurrence gives us the asymptotic running time.

A recurrence is a function is defined in terms of

- one or more base cases, and
- itself, with smaller arguments.

**Examples**

- \( T(n) = \begin{cases} 
  1 & \text{if } n = 1, \\
  T(n - 1) + 1 & \text{if } n > 1. 
\end{cases} \)

  Solution: \( T(n) = n \).

- \( T(n) = \begin{cases} 
  1 & \text{if } n = 1, \\
  2T(n/2) + n & \text{if } n \geq 1. 
\end{cases} \)

  Solution: \( T(n) = n \log n + n \).

- \( T(n) = \begin{cases} 
  0 & \text{if } n = 2, \\
  T(\sqrt{n}) + 1 & \text{if } n > 2. 
\end{cases} \)

  Solution: \( T(n) = \log \log n \).
Divide and Conquer

- Floors and ceilings  (Normally, they can be ignored.)
- Exact vs. asymptotic functions
- Boundary conditions

In algorithm analysis, we usually express both the recurrence and its solution using asymptotic notation.

- Example: $T(n) = 2T(n/2) + \Theta(n)$, with solution $T(n) = \Theta(n \log n)$.
- The boundary conditions are usually expressed as “$T(n) = O(1)$ for sufficiently small $n$. “
- When we desire an exact, rather than an asymptotic, solution, we need to deal with boundary conditions.
- In practice, we just use asymptotics most of the time, and we ignore boundary conditions.
Divide and Conquer

Methods for solving recurrences:

• Substitution Method: Guess a bound and use Mathematical induction to prove.

• Master Method: Memorize three cases and use them to solve recurrences of the form:
  \[ T(n) = a \ T(n/b) + f(n) \]

• Iteration Method: However, it is too easy to make an error in parenthesization, and that recursion trees give a better intuitive idea than iterating the recurrence of how the recurrence progresses.

• Recursion-tree Method: Converts the recurrence into a tree whose nodes represent the cost incurred at various levels of recursion. We use techniques for bounding summations to solve the recurrence.
The Maximum-subarray problem

Consider investing in stock market. You can buy one unit of stock only one time and then sell it at a later date, buying and selling after close of the trading day. Suppose you have a “Cristal Ball” to see the price of the stock in the future. What is your strategy?
The Maximum-subarray problem

Should you always buy at the lowest or sell at the highest? Consider:

Can solve by brute force: check all \( \binom{n}{2} = \Theta(n^2) \) subarrays. Can organize the computation so that each subarray \( A[i..j] \) takes \( O(1) \) time, given that you’ve computed \( A[i..j-1] \), so that the brute-force solution takes \( \Theta(n^2) \) time.

We need to find the nonempty, contiguous subarray of array \( A \) whose values have the largest sum.
The Maximum-subarray problem

Solving by divide-and-conquer

Use divide-and-conquer to solve in $O(n \lg n)$ time.

[Maximum subarray might not be unique, though its value is, so we speak of a maximum subarray, rather than the maximum subarray.]

Subproblem: Find a maximum subarray of $A[low..high]$. In original call, $low = 1$, $high = n$.

Three possibilities for the location of the maximum-subarray with respect to the midpoint (the divide point):
The Maximum-subarray problem

**Divide** the subarray into two subarrays of as equal size as possible. Find the midpoint $mid$ of the subarrays, and consider the subarrays $A[low..mid]$ and $A[mid + 1..high]$.

**Conquer** by finding a maximum subarrays of $A[low..mid]$ and $A[mid + 1..high]$.

**Combine** by finding a maximum subarray that crosses the midpoint, and using the best solution out of the three (the subarray crossing the midpoint and the two solutions found in the conquer step).

This strategy works because any subarray must either lie entirely on one side of the midpoint or cross the midpoint.
The Maximum-subarray problem

Finding the maximum subarray that crosses the midpoint

Not a smaller instance of the original problem: has the added restriction that the subarray must cross the midpoint.

Again, could use brute force. If size of $A[low..high]$ is $n$, would have $n/2$ choices for left endpoint and $n/2$ choices right endpoint, so would have $\Theta(n^2)$ combinations altogether.

Can solve in linear time.

- Any subarray crossing the midpoint $A[mid]$ is made of two subarrays $A[i..mid]$ and $A[mid + 1..j]$, where $low \leq i \leq mid$ and $mid < j \leq high$.
- Find maximum subarrays of the form $A[i..mid]$ and $A[mid + 1..j]$ and then combine them.

Procedure to take array $A$ and indices $low$, $mid$, $high$ and return a tuple giving indices of maximum subarray that crosses the midpoint, along with the sum in this maximum subarray:
The Maximum-subarray problem

**Find-Max-Crossing-Subarray** \((A, \text{low, mid, high})\)

// Find a maximum subarray of the form \(A[i \ldots mid]\).

\[
\text{left-sum} = -\infty \\
\text{sum} = 0 \\
\text{for } i = \text{mid} \text{ downto low} \\
\quad \text{sum} = \text{sum} + A[i] \\
\quad \text{if } \text{sum} > \text{left-sum} \\
\quad \quad \text{left-sum} = \text{sum} \\
\quad \quad \text{max-left} = i
\]

// Find a maximum subarray of the form \(A[mid + 1 \ldots j]\).

\[
\text{right-sum} = -\infty \\
\text{sum} = 0 \\
\text{for } j = \text{mid} + 1 \text{ to high} \\
\quad \text{sum} = \text{sum} + A[j] \\
\quad \text{if } \text{sum} > \text{right-sum} \\
\quad \quad \text{right-sum} = \text{sum} \\
\quad \quad \text{max-right} = j
\]

// Return the indices and the sum of the two subarrays.

**return** \((\text{max-left}, \text{max-right}, \text{left-sum} + \text{right-sum})\)

**Time:** The two loops together consider each index in the range \(\text{low, \ldots, high}\) exactly once, and each iteration takes \(\Theta(1)\) time \(\Rightarrow\) procedure takes \(\Theta(n)\) time.
The Maximum-subarray problem

Divide-and-conquer procedure for the maximum-subarray problem

FIND-MAXIMUM-SUBARRAY (A, low, high)

    if high == low
        return (low, high, A[low])             // base case: only one element
    else mid = [(low + high)/2]
        (left-low, left-high, left-sum) =
            FIND-MAXIMUM-SUBARRAY (A, low, mid)
        (right-low, right-high, right-sum) =
            FIND-MAXIMUM-SUBARRAY (A, mid + 1, high)
        (cross-low, cross-high, cross-sum) =
            FIND-MAX-CROSSING-SUBARRAY (A, low, mid, high)
        if left-sum \geq \text{right-sum} \text{ and } left-sum \geq cross-sum
            return (left-low, left-high, left-sum)
        elseif right-sum \geq left-sum \text{ and } right-sum \geq cross-sum
            return (right-low, right-high, right-sum)
        else return (cross-low, cross-high, cross-sum)

Initial call: FIND-MAXIMUM-SUBARRAY (A, 1, n)

• Divide by computing mid.
• Conquer by the two recursive calls to FIND-MAXIMUM-SUBARRAY.
• Combine by calling FIND-MAX-CROSSING-SUBARRAY and then determining which of the three results gives the maximum sum.
• Base case is when the subarray has only 1 element.
The Maximum-subarray problem

**Analysis**

*Simplifying assumption:* Original problem size is a power of 2, so that all subproblem sizes are integer. [We made the same simplifying assumption when we analyzed merge sort.]

Let $T(n)$ denote the running time of `FIND-MAXIMUM-SUBARRAY` on a subarray of $n$ elements.

**Base case:** Occurs when `high` equals `low`, so that $n = 1$. The procedure just returns $T(n) = \Theta(1)$.

**Recursive case:** Occurs when $n > 1$.

- Dividing takes $\Theta(1)$ time.
- Conquering solves two subproblems, each on a subarray of $n/2$ elements. Takes $T(n/2)$ time for each subproblem $\Rightarrow 2T(n/2)$ time for conquering.
- Combining consists of calling `FIND-MAX-CROSSING-SUBARRAY`, which takes $\Theta(n)$ time, and a constant number of constant-time tests $\Rightarrow \Theta(n) + \Theta(1)$ time for combining.
The Maximum-subarray problem

Recurrence for recursive case becomes
\[ T(n) = \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \]
\[ = 2T(n/2) + \Theta(n) \quad \text{(absorb } \Theta(1) \text{ terms into } \Theta(n)) . \]

The recurrence for all cases:
\[ T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
2T(n/2) + \Theta(n) & \text{if } n > 1 . 
\end{cases} \]

Same recurrence as for merge sort. Can use the master method to show that it has solution \( T(n) = \Theta(n \lg n) . \)

Thus, with divide-and-conquer, we have developed a \( \Theta(n \lg n) \)-time solution. Better than the \( \Theta(n^2) \)-time brute-force solution.

[Can actually solve this problem in \( \Theta(n) \) time. See Exercise 4.1-5.]
Strassen’s algorithm for matrix multiplication

**Input:** Two $n \times n$ (square) matrices, $A = (a_{ij})$ and $B = (b_{ij})$.

**Output:** $n \times n$ matrix $C = (c_{ij})$, where $C = A \cdot B$, i.e.,

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

for $i, j = 1, 2, \ldots, n$.

Need to compute $n^2$ entries of $C$. Each entry is the sum of $n$ values.

**Obvious method**

**SQUARE-MAT-MULT** $(A, B, n)$

let $C$ be a new $n \times n$ matrix

for $i = 1$ to $n$

for $j = 1$ to $n$

$\quad c_{ij} = 0$

for $k = 1$ to $n$

$\quad c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$

return $C$

**Analysis:** Three nested loops, each iterates $n$ times, and innermost loop body takes constant time $\Rightarrow \Theta(n^3)$. 
Strassen’s algorithm for matrix multiplication

Is \( \Theta(n^3) \) the best we can do? Can we multiply matrices in \( o(n^3) \) time?

Seems like any algorithm to multiply matrices must take \( \Omega(n^3) \) time:

- Must compute \( n^2 \) entries.
- Each entry is the sum of \( n \) terms.

But with Strassen’s method, we can multiply matrices in \( o(n^3) \) time.

- Strassen’s algorithm runs in \( \Theta(n^{\lg 7}) \) time.
- \( 2.80 \leq \lg 7 \leq 2.81 \).
- Hence, runs in \( O(n^{2.81}) \) time.
Strassen’s algorithm for matrix multiplication

Simple divide-and-conquer method

As with the other divide-and-conquer algorithms, assume that \( n \) is a power of 2. Partition each of \( A, B, C \) into four \( n/2 \times n/2 \) matrices:

\[
A = \begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}, \quad
B = \begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}, \quad
C = \begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix}.
\]

Rewrite \( C = A \cdot B \) as

\[
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix} = \begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix} \cdot \begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix},
\]

giving the four equations

\[
\begin{align*}
C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \\
C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \\
C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \\
C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}.
\end{align*}
\]

Each of these equations multiplies two \( n/2 \times n/2 \) matrices and then adds their \( n/2 \times n/2 \) products.
Strassen’s algorithm for matrix multiplication

```
REC-MAT-MULT(A, B, n)
    let C be a new n x n matrix
    if n == 1
        c_{11} = a_{11} \cdot b_{11}
    else partition A, B, and C into n/2 x n/2 submatrices
        C_{11} = REC-MAT-MULT(A_{11}, B_{11}, n/2) + REC-MAT-MULT(A_{12}, B_{21}, n/2)
        C_{12} = REC-MAT-MULT(A_{11}, B_{12}, n/2) + REC-MAT-MULT(A_{12}, B_{22}, n/2)
        C_{21} = REC-MAT-MULT(A_{21}, B_{11}, n/2) + REC-MAT-MULT(A_{22}, B_{21}, n/2)
        C_{22} = REC-MAT-MULT(A_{21}, B_{12}, n/2) + REC-MAT-MULT(A_{22}, B_{22}, n/2)
    return C
```

Note: We can partition matrices without copying entries by instead using index calculations. It would take only constant time, instead of $O(n^2)$ time. However, the asymptotic analysis won’t change when we use either technique.
Strassen’s algorithm for matrix multiplication

Analysis
Let $T(n)$ be the time to multiply two $n \times n$ matrices.

Base case: $n = 1$. Perform one scalar multiplication: $\Theta(1)$.

Recursive case: $n > 1$.

- Dividing takes $\Theta(1)$ time, using index calculations. [Otherwise, $\Theta(n^2)$ time.]
- Conquering makes 8 recursive calls, each multiplying $n/2 \times n/2$ matrices $\Rightarrow 8T(n/2)$.
- Combining takes $\Theta(n^2)$ time to add $n/2 \times n/2$ matrices four times. [Doesn’t even matter asymptotically whether we use index calculations or copy: would be $\Theta(n^2)$ either way.]

Recurrence is

$$T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
8T(n/2) + \Theta(n^2) & \text{if } n > 1.
\end{cases}$$

Can use master method to show that it has solution $T(n) = \Theta(n^3)$. Asymptotically, no better than the obvious method.
Strassen’s algorithm for matrix multiplication

*Constant factors and recurrences:* When setting up recurrences, can absorb constant factors into asymptotic notation, but cannot absorb a constant number of subproblems. Although we absorb the 4 additions of \( \frac{n}{2} \times \frac{n}{2} \) matrices into the \( \Theta(n^2) \) time, we cannot lose the 8 in front of the \( T(n/2) \) term. If we absorb the constant number of subproblems, then the recursion tree would not be “bushy” and would instead just be a linear chain.
Strassen’s algorithm for matrix multiplication

Strassen’s method

Idea: Make the recursion tree less bushy. Perform only 7 recursive multiplications of $n/2 \times n/2$ matrices, rather than 8. Will cost several additions of $n/2 \times n/2$ matrices, but just a constant number more $\Rightarrow$ can still absorb the constant factor for matrix additions into the $\Theta(n^2)$ term.

The algorithm:

1. As in the recursive method, partition each of the matrices into four $n/2 \times n/2$ submatrices. Time: $\Theta(1)$.
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$. Each is $n/2 \times n/2$ and is the sum or difference of two matrices created in previous step. Time: $\Theta(n^2)$ to create all 10 matrices.
3. Recursively compute 7 matrix products $P_1, P_2, \ldots, P_7$, each $n/2 \times n/2$.
4. Compute $n/2 \times n/2$ submatrices of $C$ by adding and subtracting various combinations of the $P_i$. Time: $\Theta(n^2)$.

Analysis

Recurrence will be

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

By the master method, solution is $T(n) = \Theta(n^{\log_2 7})$. 

20
Details

Step 2: Create the 10 matrices

\[ S_1 = B_{12} - B_{22} , \]
\[ S_2 = A_{11} + A_{12} , \]
\[ S_3 = A_{21} + A_{22} , \]
\[ S_4 = B_{21} - B_{11} , \]
\[ S_5 = A_{11} + A_{22} , \]
\[ S_6 = B_{11} + B_{22} , \]
\[ S_7 = A_{12} - A_{22} , \]
\[ S_8 = B_{21} + B_{22} , \]
\[ S_9 = A_{11} - A_{21} , \]
\[ S_{10} = B_{11} + B_{12} . \]

Add or subtract \( n/2 \times n/2 \) matrices 10 times \( \Rightarrow \) time is \( \Theta \left(n^2\right) \).
Strassen’s algorithm for matrix multiplication

Step 3: Create the 7 matrices

\[ P_1 = A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22}, \]
\[ P_2 = S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22}, \]
\[ P_3 = S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11}, \]
\[ P_4 = A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11}, \]
\[ P_5 = S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}, \]
\[ P_6 = S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}, \]
\[ P_7 = S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}. \]

The only multiplications needed are in the middle column; right-hand column just shows the products in terms of the original submatrices of \( A \) and \( B \).

Step 4: Add and subtract the \( P_i \) to construct submatrices of \( C \):

\[ C_{11} = P_5 + P_4 - P_2 + P_6, \]
\[ C_{12} = P_1 + P_2, \]
\[ C_{21} = P_3 + P_4, \]
\[ C_{22} = P_5 + P_1 - P_3 - P_7. \]
Strassen’s algorithm for matrix multiplication

To see how these computations work, expand each right-hand side, replacing each $P_i$ with the sub-matrices of A and B that form it, and cancel terms:

\[
A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
- A_{22} \cdot B_{11} + A_{22} \cdot B_{21} + A_{12} \cdot B_{22}\\n- A_{11} \cdot B_{22} - A_{22} \cdot B_{21} + A_{12} \cdot B_{22} + A_{12} \cdot B_{21} + A_{12} \cdot B_{21}
\]

\[
A_{11} \cdot B_{11} + A_{12} \cdot B_{22}\\nA_{11} \cdot B_{12} + A_{12} \cdot B_{22}\\nA_{21} \cdot B_{11} + A_{22} \cdot B_{11} + A_{22} \cdot B_{21} \]

\[
A_{21} \cdot B_{11} + A_{22} \cdot B_{21} + A_{21} \cdot B_{12} + A_{22} \cdot B_{22} + A_{21} \cdot B_{12}
\]
Strassen’s algorithm for matrix multiplication

Theoretical and practical notes

Strassen’s algorithm was the first to beat $\Theta(n^3)$ time, but it’s not the asymptotically fastest known. A method by Coppersmith and Winograd runs in $O(n^{2.376})$ time. Practical issues against Strassen’s algorithm:

- Higher constant factor than the obvious $\Theta(n^3)$-time method.
- Not good for sparse matrices.
- Not numerically stable: larger errors accumulate than in the obvious method.
- Submatrices consume space, especially if copying.

Numerical stability problem is not as bad as previously thought. And can use index calculations to reduce space requirement.
Substitution method

1. Guess the solution.
2. Use induction to find the constants and show that the solution works.

Example

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1, \\
2T(n/2) + n & \text{if } n > 1.
\end{cases} \]

1. Guess: \( T(n) = n \log n + n \). [Here, we have a recurrence with an exact function, rather than asymptotic notation, and the solution is also exact rather than asymptotic. We’ll have to check boundary conditions and the base case.]

2. Induction:

Basis: \( n = 1 \Rightarrow n \log n + n = 1 = T(n) \)

Inductive step: Inductive hypothesis is that \( T(k) = k \log k + k \) for all \( k < n \). We’ll use this inductive hypothesis for \( T(n/2) \).

\[
T(n) = 2T\left(\frac{n}{2}\right) + n \\
= 2\left(\frac{n}{2} \log \frac{n}{2} + \frac{n}{2}\right) + n \quad \text{(by inductive hypothesis)} \\
= n \log \frac{n}{2} + n + n \\
= n(\log n - \log 2) + n + n \\
= n \log n - n + n + n \\
= n \log n + n.
\]
Substitution method

Generally, we use asymptotic notation:

- We would write $T(n) = 2T(n/2) + \Theta(n)$.
- We assume $T(n) = O(1)$ for sufficiently small $n$.
- We express the solution by asymptotic notation: $T(n) = \Theta(n \log n)$.
- We don’t worry about boundary cases, nor do we show base cases in the substitution proof.

  - $T(n)$ is always constant for any constant $n$.
  - Since we are ultimately interested in an asymptotic solution to a recurrence, it will always be possible to choose base cases that work.
  - When we want an asymptotic solution to a recurrence, we don’t worry about the base cases in our proofs.
  - When we want an exact solution, then we have to deal with base cases.

For the substitution method:

- Name the constant in the additive term.
- Show the upper ($O$) and lower ($\Omega$) bounds separately. Might need to use different constants for each.
Substitution method

Example

\[ T(n) = 2T(n/2) + \Theta(n). \]

If we want to show an upper bound of \( T(n) = 2T(n/2) + O(n) \), we write \( T(n) \leq 2T(n/2) + cn \) for some positive constant \( c \).

1. **Upper bound:**

   **Guess:** \( T(n) \leq dn \lg n \) for some positive constant \( d \). We are given \( c \) in the recurrence, and we get to choose \( d \) as any positive constant. It’s OK for \( d \) to depend on \( c \).

   **Substitution:**

   \[
   \begin{align*}
   T(n) & \leq 2T(n/2) + cn \\
   &= 2 \left( d \frac{n}{2} \lg \frac{n}{2} \right) + cn \\
   &= d n \lg \frac{n}{2} + cn \\
   &= d n \lg n - dn + cn \\
   \leq d n \lg n & \quad \text{if } -dn + cn \leq 0,
   \end{align*}
   \]

   Therefore, \( T(n) = O(n \lg n) \).
2. **Lower bound:** Write $T(n) \geq 2T(n/2) + cn$ for some positive constant $c$.

*Guess:* $T(n) \geq dn \lg n$ for some positive constant $d$.

*Substitution:*

\[
T(n) \geq 2T(n/2) + cn \\
= 2 \left( d \frac{n}{2} \lg \frac{n}{2} \right) + cn \\
= dn \lg \frac{n}{2} + cn \\
= dn \lg n - dn + cn \\
\geq dn \lg n \quad \text{if } -dn + cn \geq 0, \\
\quad \quad \quad \quad \text{ if } d \leq c
\]

Therefore, $T(n) = \Omega(n \lg n)$.

Therefore, $T(n) = \Theta(n \lg n)$. [For this particular recurrence, we can use $d = c$ for both the upper-bound and lower-bound proofs. That won’t always be the case.]  ■
Substitution method

Make sure you show the same exact form when doing a substitution proof.

Consider the recurrence

\[ T(n) = 8T(n/2) + \Theta(n^2). \]

For an upper bound:

\[ T(n) \leq 8T(n/2) + cn^2. \]

Guess: \[ T(n) \leq dn^3. \]

\[
\begin{align*}
T(n) & \leq 8d(n/2)^3 + cn^2 \\
& = 8d(n^3/8) + cn^2 \\
& = dn^3 + cn^2 \\
& \neq dn^3 \quad \text{doesn’t work!}
\end{align*}
\]

Remedy: Subtract off a lower-order term.
Substitution method

**Remedy:** Subtract off a lower-order term.

\[\text{Guess: } T(n) \leq dn^3 - d'n^2.\]

\[
\begin{align*}
T(n) & \leq 8(d(n/2)^3 - d'(n/2)^2) + cn^2 \\
& = 8d(n^3/8) - 8d'(n^2/4) + cn^2 \\
& = dn^3 - 2d'n^2 + cn^2 \\
& = dn^3 - d'n^2 - d'n^2 + cn^2 \\
& \leq dn^3 - d'n^2 \quad \text{if } -d'n^2 + cn^2 \leq 0, \quad d' \geq c
\end{align*}
\]

Be careful when using asymptotic notation.

The false proof for the recurrence \( T(n) = 4T(n/4) + n \), that \( T(n) = O(n) \):

\[
\begin{align*}
T(n) & \leq 4(c(n/4)) + n \\
& \leq cn + n \\
& = O(n) \quad \text{wrong!}
\end{align*}
\]

Because we haven’t proven the exact form of our inductive hypothesis (which is that \( T(n) \leq cn \)), this proof is false.
Recursion Tree

Use to generate a guess. Then verify by substitution method.

Consider the recurrence $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$
Recursion Tree

**Example**

\[ T(n) = T(n/3) + T(2n/3) + \Theta(n). \]

For upper bound, rewrite as \( T(n) \leq T(n/3) + T(2n/3) + cn \); for lower bound, as \( T(n) \geq T(n/3) + T(2n/3) + cn \).

By summing across each level, the recursion tree shows the cost at each level of recursion (minus the costs of recursive calls, which appear in subtrees):

- **Depth for leftmost branch:**
  - Subproblem size for a node at depth \( i \) is \( \frac{1}{3^i} n \).
  - Therefore, \( \frac{1}{3^i} n = 1 \) \( \rightarrow \) \( i = \log_3 n \).

- **Depth for rightmost:**
  - Subproblem size for a node at depth \( i \) is \( \left(\frac{2}{3}\right)^i n \).
  - Therefore, \( \left(\frac{2}{3}\right)^i n = 1 \) \( \rightarrow \) \( i = \log_{3/2} n \).
Recursion Tree

- There are $\log_3 n$ full levels, and after $\log_{3/2} n$ levels, the problem size is down to 1.
- Each level contributes $\leq cn$.
- Lower bound guess: $\geq dn \log_3 n = \Omega(n \log n)$ for some positive constant $d$.
- Upper bound guess: $\leq dn \log_{3/2} n = O(n \log n)$ for some positive constant $d$.
- Then prove by substitution.
1. **Upper bound:**

   **Guess:** $T(n) \leq dn \lg n$.

   **Substitution:**
   
   $T(n) \leq T(n/3) + T(2n/3) + cn$
   
   $\leq d(n/3) \lg(n/3) + d(2n/3) \lg(2n/3) + cn$
   
   $= (d(n/3) \lg n - d(n/3) \lg 3)$
   
   $\quad + (d(2n/3) \lg n - d(2n/3) \lg(3/2)) + cn$
   
   $= dn \lg n - d((n/3) \lg 3 + (2n/3) \lg(3/2)) + cn$
   
   $= dn \lg n - d((n/3) \lg 3 + (2n/3) \lg 3 - (2n/3) \lg 2) + cn$
   
   $= dn \lg n - dn(\lg 3 - 2/3) + cn$
   
   $\leq dn \lg n$ if $-dn(\lg 3 - 2/3) + cn \leq 0$,

   $d \geq \frac{c}{\lg 3 - 2/3}$.

   Therefore, $T(n) = O(n \lg n)$.

   **Note:** Make sure that the symbolic constants used in the recurrence (e.g., $c$) and the guess (e.g., $d$) are different.
2. **Lower bound:**

*Guess:* \( T(n) \geq d n \lg n \).

*Substitution:* Same as for the upper bound, but replacing \( \leq \) by \( \geq \). End up needing

\[
0 < d \leq \frac{c}{\lg 3 - 2/3}.
\]

Therefore, \( T(n) = \Omega(n \lg n) \).

Since \( T(n) = O(n \lg n) \) and \( T(n) = \Omega(n \lg n) \), we conclude that \( T(n) = \Theta(n \lg n) \).
Master Method

Upper Bound: If

\[ T(n) \leq aT(n/b) + f(n) \]

and

\[ f(n) = O(n^k), \]

then

\[ T(n) = \begin{cases} 
O(n^k) & \text{if } a < b^k \\
O(n^k \log n) & \text{if } a = b^k \\
O(n^{\log_b a}) & \text{if } a > b^k. 
\end{cases} \]

Lower Bound: If

\[ T(n) \geq aT(n/b) + f(n) \]

and

\[ f(n) = \Omega(n^k), \]

then

\[ T(n) = \begin{cases} 
\Omega(n^k) & \text{if } a < b^k \\
\Omega(n^k \log n) & \text{if } a = b^k \\
\Omega(n^{\log_b a}) & \text{if } a > b^k. 
\end{cases} \]

Exact: If

\[ T(n) = aT(n/b) + f(n) \]

and

\[ f(n) = \Theta(n^k), \]

then

\[ T(n) = \begin{cases} 
\Theta(n^k) & \text{if } a < b^k \\
\Theta(n^k \log n) & \text{if } a = b^k \\
\Theta(n^{\log_b a}) & \text{if } a > b^k. 
\end{cases} \]

Let \( a, b, \) and \( k \) be integers satisfying \( a \geq 1, b \geq 2, \) and \( k \geq 0 \)
\n\( n/b \) can be either floor or ceiling function.
\nFloor: \( T(0) = u \) is given
\nCeiling: \( T(1) = u \) is given

Examples:

\[ T(n) = 4T(n/2) + n \Rightarrow T(n) \in ? \]

\[ T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ? \]

\[ T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ? \]
Master Method

Used for many divide-and-conquer recurrences of the form
\[ T(n) = aT(n/b) + f(n), \]
where \( a \geq 1, b > 1, \) and \( f(n) > 0. \)
Based on the \textit{master theorem} (Theorem 4.1).

Compare \( n^{\log_b a} \) vs. \( f(n) \):

1. If \( f(n) = O(n^{\log_b a - \epsilon}) \) for some constant \( \epsilon > 0, \) then \( T(n) = \Theta(n^{\log_b a}). \)
2. If \( f(n) = \Theta(n^{\log_b a}), \) then \( T(n) = \Theta(n^{\log_b a} \lg n). \)
3. If \( f(n) = \Omega(n^{\log_b a + \epsilon}) \) for some constant \( \epsilon > 0, \) and if \( af(n/b) \leq cf(n) \) for some constant \( c < 1 \) and all sufficiently large \( n, \) then \( T(n) = \Theta(f(n)) \).

\( F(n) \) is \textit{polynomially} smaller than \( n^{\log_b a} \)

\( F(n) \) is the same size as \( n^{\log_b a} \)

\( F(n) \) is \textit{polynomially} larger than \( n^{\log_b a} \). It should also satisfy the regularity condition
Master Method

Used for many divide-and-conquer recurrences of the form

\[ T(n) = aT(n/b) + f(n), \]

where \( a \geq 1, b > 1, \) and \( f(n) > 0. \)

Based on the master theorem (Theorem 4.1).

Compare \( n^{\log_b a} \) vs. \( f(n) \):

**Case 1:** \( f(n) = O(n^{\log_b a - \epsilon}) \) for some constant \( \epsilon > 0. \)
\( (f(n) \) is polynomially smaller than \( n^{\log_b a}. \)

**Solution:** \( T(n) = \Theta(n^{\log_b a}). \)
(Intuitively: cost is dominated by leaves.)

**Case 2:** \( f(n) = \Theta(n^{\log_b a} \lg^k n) \), where \( k \geq 0. \)

[This formulation of Case 2 is more general than in Theorem 4.1, and it is given in Exercise 4.6-2.]

\( (f(n) \) is within a polylog factor of \( n^{\log_b a}, \) but not smaller.)

**Solution:** \( T(n) = \Theta(n^{\log_b a} \lg^{k+1} n). \)
(Intuitively: cost is \( n^{\log_b a} \) \( \lg^k n \) at each level, and there are \( \Theta(\lg n) \) levels.)

**Simple case:** \( k = 0 \Rightarrow f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg n). \)

**Case 3:** \( f(n) = \Omega(n^{\log_b a + \epsilon}) \) for some constant \( \epsilon > 0 \) and \( f(n) \) satisfies the regularity condition \( af(n/b) \leq cf(n) \) for some constant \( c < 1 \) and all sufficiently large \( n. \)

\( (f(n) \) is polynomially greater than \( n^{\log_b a}. \)

**Solution:** \( T(n) = \Theta(f(n)). \)
(Intuitively: cost is dominated by root.)
What’s with the Case 3 regularity condition?

- Generally not a problem.
- It always holds whenever \( f(n) = n^k \) and \( f(n) = \Omega(n^\log_b a + \epsilon) \) for constant \( \epsilon > 0 \). [Proving this makes a nice homework exercise. See below.] So you don’t need to check it when \( f(n) \) is a polynomial.

[Here’s a proof that the regularity condition holds when \( f(n) = n^k \) and \( f(n) = \Omega(n^{\log_b a + \epsilon}) \) for constant \( \epsilon > 0 \).

Since \( f(n) = \Omega(n^{\log_b a + \epsilon}) \) and \( f(n) = n^k \), we have that \( k > \log_b a \). Using a base of \( b \) and treating both sides as exponents, we have \( b^k > b^{\log_b a} = a \), and so \( a/b^k < 1 \). Since \( a, b, \) and \( k \) are constants, if we let \( c = a/b^k \), then \( c \) is a constant strictly less than 1. We have that \( af(n/b) = a(n/b)^k = (a/b^k)n^k = cf(n) \), and so the regularity condition is satisfied.]
Examples

- $T(n) = 5T(n/2) + \Theta(n^2)$
  $n^{\log_2 5}$ vs. $n^2$
  Since $\log_2 5 - \epsilon = 2$ for some constant $\epsilon > 0$, use Case 1 $\Rightarrow T(n) = \Theta(n^{\log_5^2})$

- $T(n) = 27T(n/3) + \Theta(n^3 \lg n)$
  $n^{\log_3 27} = n^3$ vs. $n^3 \lg n$
  Use Case 2 with $k = 1$ $\Rightarrow T(n) = \Theta(n^3 \lg^2 n)$

- $T(n) = 5T(n/2) + \Theta(n^3)$
  $n^{\log_2 5}$ vs. $n^3$
  Now $\lg 5 + \epsilon = 3$ for some constant $\epsilon > 0$
  Check regularity condition (don’t really need to since $f(n)$ is a polynomial):
  \[ af(n/b) = 5(n/2)^3 = 5n^3/8 \leq cn^3 \text{ for } c = 5/8 < 1 \]
  Use Case 3 $\Rightarrow T(n) = \Theta(n^3)$

- $T(n) = 27T(n/3) + \Theta(n^3 / \lg n)$
  $n^{\log_3 27} = n^3$ vs. $n^3 / \lg n = n^3 \lg^{-1} n \neq \Theta(n^3 \lg^k n)$ for any $k \geq 0$.
  Cannot use the master method.
Chapter 5: Probabilistic Analysis and Randomized Algorithms

Introduction to probabilistic analysis and randomized algorithms.
Note: It is assumed that the students are familiar with the basic probability material in Appendix C.
The primary goals of this chapter is:

• explain the difference between probabilistic analysis and randomized algorithms,
• present the technique of indicator random variables, and
• give another example of the analysis of a randomized algorithm (permuting an array in place).
The hiring problem

Scenario

- You are using an employment agency to hire a new office assistant.
- The agency sends you one candidate each day.
- You interview the candidate and must immediately decide whether or not to hire that person. But if you hire, you must also fire your current office assistant—even if it’s someone you have recently hired.
- Cost to interview is $c_i$ per candidate (interview fee paid to agency).
- Cost to hire is $c_h$ per candidate (includes cost to fire current office assistant + hiring fee paid to agency).
- Assume that $c_h > c_i$.
- You are committed to having hired, at all times, the best candidate seen so far. Meaning that whenever you interview a candidate who is better than your current office assistant, you must fire the current office assistant and hire the candidate. Since you must have someone hired at all times, you will always hire the first candidate that you interview.

Goal

Determine what the price of this strategy will be.
**Pseudocode to model this scenario**

Assumes that the candidates are numbered 1 to \( n \) and that after interviewing each candidate, we can determine if it’s better than the current office assistant. Uses a dummy candidate 0 that is worse than all others, so that the first candidate is always hired.

**Hire-Assistant** \((n)\)

\[
\begin{align*}
best &= 0 & \quad \text{// candidate 0 is a least-qualified dummy candidate} \\
\text{for } i = 1 \text{ to } n \\
\quad &\text{interview candidate } i \\
\quad &\text{if candidate } i \text{ is better than candidate } best \\
\quad &\quad best = i \\
\quad &\text{hire candidate } i
\end{align*}
\]
Cost

If $n$ candidates, and we hire $m$ of them, the cost is $O(nc_i + mc_h)$.

- Have to pay $nc_i$ to interview, no matter how many we hire.
- So we focus on analyzing the hiring cost $mc_h$.
- $mc_h$ varies with each run—it depends on the order in which we interview the candidates.
- This is a model of a common paradigm: we need to find the maximum or minimum in a sequence by examining each element and maintaining a current “winner.” The variable $m$ denotes how many times we change our notion of which element is currently winning.

Worst-case analysis

In the worst case, we hire all $n$ candidates.

This happens if each one is better than all who came before. In other words, if the candidates appear in increasing order of quality.

If we hire all $n$, then the cost is $O(nc_i + nc_h) = O(nc_h)$ (since $c_h > c_i$).
Probabilistic Analysis and Randomized Algorithms

Probabilistic analysis

In general, we have no control over the order in which candidates appear. We could assume that they come in a random order:

- Assign a rank to each candidate: \( rank(i) \) is a unique integer in the range 1 to \( n \).
- The ordered list \( \langle rank(1), rank(2), \ldots, rank(n) \rangle \) is a permutation of the candidate numbers \( \langle 1, 2, \ldots, n \rangle \).
- The list of ranks is equally likely to be any one of the \( n! \) permutations.
- Equivalently, the ranks form a uniform random permutation: each of the possible \( n! \) permutations appears with equal probability.

Essential idea of probabilistic analysis

We must use knowledge of, or make assumptions about, the distribution of inputs.

- The expectation is over this distribution.
- This technique requires that we can make a reasonable characterization of the input distribution.
Randomized algorithms

We might not know the distribution of inputs, or we might not be able to model it computationally.
Instead, we use randomization within the algorithm in order to impose a distribution on the inputs.

For the hiring problem

Change the scenario:

• The employment agency sends us a list of all $n$ candidates in advance.
• On each day, we randomly choose a candidate from the list to interview (but considering only those we have not yet interviewed).
• Instead of relying on the candidates being presented to us in a random order, we take control of the process and enforce a random order.
What makes an algorithm randomized

An algorithm is **randomized** if its behavior is determined in part by values produced by a **random-number generator**.

- **RANDOM**($a, b$) returns an integer $r$, where $a \leq r \leq b$ and each of the $b - a + 1$ possible values of $r$ is equally likely.
- In practice, **RANDOM** is implemented by a **pseudorandom-number generator**, which is a deterministic method returning numbers that “look” random and pass statistical tests.
Indicator random variables

A simple yet powerful technique for computing the expected value of a random variable. Helpful in situations in which there may be dependence.

Given a sample space and an event $A$, we define the indicator random variable $I\{A\}$ as:

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs,} \\ 0 & \text{if } A \text{ does not occur.} \end{cases}$$

Lemma

For an event $A$, let $X_A = I\{A\}$. Then $E[X_A] = \Pr\{A\}$.

Proof: Letting $\overline{A}$ be the complement of $A$, we have

$$E[X_A] = E[I\{A\}]$$
$$= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\overline{A}\} \quad \text{(definition of expected value)}$$
$$= \Pr\{A\}. \quad \blacksquare$$

Simple example

Determine the expected number of heads when we flip a fair coin one time.

- Sample space is $\{H, T\}$.
- $\Pr\{H\} = \Pr\{T\} = 1/2$.
- Define indicator random variable $X_H = I\{H\}$. $X_H$ counts the number of heads in one flip.
- Since $\Pr\{H\} = 1/2$, lemma says that $E[X_H] = 1/2$. 

Probabilistic Analysis and Randomized Algorithms

*Slightly more complicated example*

Determine the expected number of heads in \( n \) coin flips.

- Let \( X \) be a random variable for the number of heads in \( n \) flips.
- Could compute \( E[X] = \sum_{k=0}^{n} k \cdot \Pr\{X = k\} \). In fact, this is what the book does in equation (C.37).
- Instead, we’ll use indicator random variables.
- For \( i = 1, 2, \ldots, n \), define \( X_i = I\{\text{the } i\text{ th flip results in event } H\} \).
- Then \( X = \sum_{i=1}^{n} X_i \).
- Lemma says that \( E[X_i] = \Pr\{H\} = 1/2 \) for \( i = 1, 2, \ldots, n \).
- Expected number of heads is \( E[X] = E[\sum_{i=1}^{n} X_i] \).

**Problem:** We want \( E[\sum_{i=1}^{n} X_i] \). We have only the individual expectations \( E[X_1], E[X_2], \ldots, E[X_n] \).

**Solution:** Linearity of expectation says that the expectation of the sum equals the sum of the expectations. Thus,

\[
E[X] = E\left[\sum_{i=1}^{n} X_i\right] \\
= \sum_{i=1}^{n} E[X_i] \\
= \sum_{i=1}^{n} 1/2 \\
= n/2 .
\]
Probabilistic Analysis and Randomized Algorithms

Analysis of the hiring problem

Assume that the candidates arrive in a random order.
Let $X$ be a random variable that equals the number of times we hire a new office assistant.
Define indicator random variables $X_1, X_2, \ldots, X_n$, where

$$X_i = I\{\text{candidate } i \text{ is hired}\}.$$

**Useful properties:**

- $X = X_1 + X_2 + \cdots + X_n$.
- Lemma $\Rightarrow \mathbb{E}[X_i] = \Pr\{\text{candidate } i \text{ is hired}\}$.

We need to compute $\Pr\{\text{candidate } i \text{ is hired}\}$.

- Candidate $i$ is hired if and only if candidate $i$ is better than each of candidates $1, 2, \ldots, i - 1$.
- Assumption that the candidates arrive in random order $\Rightarrow$ candidates $1, 2, \ldots, i$ arrive in random order $\Rightarrow$ any one of these first $i$ candidates is equally likely to be the best one so far.
- Thus, $\Pr\{\text{candidate } i \text{ is the best so far}\} = 1/i$.
- Which implies $\mathbb{E}[X_i] = 1/i$.

Now compute $\mathbb{E}[X]$: 
Now compute $E[X]$:

$$E[X] = E \left[ \sum_{i=1}^{n} X_i \right]$$

$$= \sum_{i=1}^{n} E[X_i]$$

$$= \sum_{i=1}^{n} \frac{1}{i}$$

$$= \ln n + O(1) \quad \text{(equation (A.7): the sum is a harmonic series)}.$$

Thus, the expected hiring cost is $O(c_h \ln n)$, which is much better than the worst-case cost of $O(nc_h)$. 
Randomized algorithms

Instead of assuming a distribution of the inputs, we impose a distribution.

The hiring problem

For the hiring problem, the algorithm is deterministic:

- For any given input, the number of times we hire a new office assistant will always be the same.
- The number of times we hire a new office assistant depends only on the input.
- In fact, it depends only on the ordering of the candidates’ ranks that it is given.
- Some rank orderings will always produce a high hiring cost. Example: \(1, 2, 3, 4, 5, 6\), where each candidate is hired.
- Some will always produce a low hiring cost. Example: any ordering in which the best candidate is the first one interviewed. Then only the best candidate is hired.
- Some may be in between.
Instead of always interviewing the candidates in the order presented, what if we first randomly permuted this order?

- The randomization is now in the algorithm, not in the input distribution.
- Given a particular input, we can no longer say what its hiring cost will be. Each time we run the algorithm, we can get a different hiring cost.
- In other words, each time we run the algorithm, the execution depends on the random choices made.
- No particular input always elicits worst-case behavior.
- Bad behavior occurs only if we get “unlucky” numbers from the random-number generator.
Pseudocode for randomized hiring problem

\textsc{Randomized-Hire-Assistant}(n)

randomly permute the list of candidates

\textsc{Hire-Assistant}(n)

Lemma

The expected hiring cost of \textsc{Randomized-Hire-Assistant} is $O(c_h \ln n)$. 

Proof  After permuting the input array, we have a situation identical to the probabilistic analysis of deterministic \textsc{Hire-Assistant}. 

$\blacksquare$
Randomly permuting an array

Your book considers two methods of randomly permuting an n-element array. The first method assigns a random priority in the range 1 to \( n^3 \) to each position and then reorders the array elements into increasing priority order. The second method is better: it works in place, it runs in linear time without requiring sorting, and it needs fewer random bits (n random numbers in the range 1 to n rather than the range 1 to \( n^3 \)). Let us consider the second method.

**Goal**

Produce a uniform random permutation (each of the n! permutations is equally likely to be produced).
The following procedure permutes the array $A[1 \ldots n]$ in place (i.e., no auxiliary array is required).

**Randomize-In-Place** $(A, n)$

```
for $i = 1$ to $n$
    swap $A[i]$ with $A[\text{Random}(i, n)]$
```

**Idea**
- In iteration $i$, choose $A[i]$ randomly from $A[i \ldots n]$.
- Will never alter $A[i]$ after iteration $i$.

**Time**

$O(1)$ per iteration $\Rightarrow O(n)$ total.
**Correctness**

Given a set of \( n \) elements, a \textit{k-permutation} is a sequence containing \( k \) of the \( n \) elements. There are \( n!/(n-k)! \) possible \( k \)-permutations.

**Lemma**

\texttt{RANDOMIZE-IN-PLACE} computes a uniform random permutation.

**Proof** Use a loop invariant:

**Loop invariant:** Just prior to the \( i \)th iteration of the for loop, for each possible \((i-1)\)-permutation, subarray \( A[1..i-1] \) contains this \((i-1)\)-permutation with probability \((n-i+1)!/n! \).

**Initialization:** Just before first iteration, \( i = 1 \). Loop invariant says that for each possible 0-permutation, subarray \( A[1..0] \) contains this 0-permutation with probability \( n!/n! = 1 \). \( A[1..0] \) is an empty subarray, and a 0-permutation has no elements. So, \( A[1..0] \) contains any 0-permutation with probability 1.
Probabilistic Analysis and Randomized Algorithms

**Maintenance:** Assume that just prior to the \( i \)th iteration, each possible \((i - 1)\)-permutation appears in \( A[1 \ldots i - 1] \) with probability \((n - i + 1)!/n! \). Will show that after the \( i \)th iteration, each possible \( i \)-permutation appears in \( A[1 \ldots i] \) with probability \((n - i)!/n! \). Incrementing \( i \) for the next iteration then maintains the invariant.

Consider a particular \( i \)-permutation \( \pi = (x_1, x_2, \ldots, x_i) \). It consists of an \((i - 1)\)-permutation \( \pi' = (x_1, x_2, \ldots, x_{i-1}) \), followed by \( x_i \).

Let \( E_1 \) be the event that the algorithm actually puts \( \pi' \) into \( A[1 \ldots i - 1] \). By the loop invariant, \( \Pr \{ E_1 \} = (n - i + 1)!/n! \).

Let \( E_2 \) be the event that the \( i \)th iteration puts \( x_i \) into \( A[i] \).

We get the \( i \)-permutation \( \pi \) in \( A[1 \ldots i] \) if and only if both \( E_1 \) and \( E_2 \) occur \( \Rightarrow \) the probability that the algorithm produces \( \pi \) in \( A[1 \ldots i] \) is \( \Pr \{ E_2 \cap E_1 \} \).

Equation (C.14) \( \Rightarrow \) \( \Pr \{ E_2 \cap E_1 \} = \Pr \{ E_2 \mid E_1 \} \Pr \{ E_1 \} \).

The algorithm chooses \( x_i \) randomly from the \( n - i + 1 \) possibilities in \( A[i \ldots n] \) \( \Rightarrow \) \( \Pr \{ E_2 \mid E_1 \} = 1/(n - i + 1) \). Thus,

\[
\Pr \{ E_2 \cap E_1 \} = \Pr \{ E_2 \mid E_1 \} \Pr \{ E_1 \} = \frac{1}{(n - i + 1)} \frac{(n - i + 1)!}{n!} = \frac{(n - i)!}{n!}.
\]

**Termination:** At termination, \( i = n + 1 \), so we conclude that \( A[1 \ldots n] \) is a given \( n \)-permutation with probability \((n - n)!/n! = 1/n! \). \( \blacksquare \) (lemma)
1 Heapsort
   • Introduction
   • Heap data structure
   • Heap algorithms
   • Heapsort algorithm
   • Priority queue
   • Conclusion
Contents

1 Heapsort
   - Introduction
     - Heap data structure
     - Heap algorithms
     - Heapsort algorithm
     - Priority queue
     - Conclusion
- $O(n \lg n)$ in the worst case – like *merge sort*. 
- $O(n \log n)$ in the worst case – like *merge sort*.
- Sorts *in place* – like *insertion sort*.
- $O(n \log n)$ in the worst case – like merge sort.
- Sorts in place – like insertion sort.
- Combines the best of both algorithms.
- \( O(n \lg n) \) in the worst case – like *merge sort*.
- Sorts *in place* – like *insertion sort*.
- Combines the best of both algorithms.
- Uses a data structure called the *heap*, which is also extensively used in other applications.
1 Heapsort
   • Introduction
   • Heap data structure
   • Heap algorithms
   • Heapsort algorithm
   • Priority queue
   • Conclusion
A data structure that provides worst-case $O(1)$ time access to the largest (max heap) or smallest (min heap) element.
A data structure that provides worst-case $O(1)$ time access to the largest (max heap) or smallest (min heap) element.

A data structure that provides worst-case $\Theta(\lg n)$ time extract the largest (max heap) or smallest (min heap) element.
A data structure that provides worst-case $O(1)$ time access to the largest (max heap) or smallest (min heap) element.

A data structure that provides worst-case $\Theta(\lg n)$ time extract the largest (max heap) or smallest (min heap) element.

Priority queue is a prototypical application, where the keys are retrieved by priority.
A data structure that provides worst-case $O(1)$ time access to the largest (max heap) or smallest (min heap) element.

A data structure that provides worst-case $\Theta(\lg n)$ time extract the largest (max heap) or smallest (min heap) element.

Priority queue is a prototypical application, where the keys are retrieved by priority.

Heapsort is another application, where the keys can be sorted by repeatedly extracting the largest from the heap.
A data structure that provides worst-case $O(1)$ time access to the largest (max heap) or smallest (min heap) element.

A data structure that provides worst-case $\Theta(\lg n)$ time extract the largest (max heap) or smallest (min heap) element.

Priority queue is a prototypical application, where the keys are retrieved by priority.

Heapsort is an another application, where the keys can be sorted by repeatedly extracting the largest from the heap.

Max vs. Min Heap

Unless explicitly stated as max heap or min heap, heap means max heap in this course.
Heap-ordered tree

Definition
A binary tree is heap-ordered if:

1. the value at a node is $\geq$ the value at each of its children.

Example of (max) heap
Heap-ordered tree

Definition

A binary tree is heap-ordered if:

1. the value at a node is $\geq$ the value at each of its children.
2. the tree is **almost-complete**.

Example of complete tree (or *not*)
Heap-ordered tree

Definition
A binary tree is heap-ordered if:
1. the value at a node is $\geq$ the value at each of its children.
2. the tree is almost-complete.

Example of complete tree (or not)
Heap-ordered tree

Definition

A binary tree is heap-ordered if:
1. the value at a node is $\geq$ the value at each of its children.
2. the tree is almost-complete.

Example of complete tree (or not)

![Complete Binary Tree](image1.png)

![Non-Complete Binary Tree](image2.png)
**Heap-ordered tree**

**Definition**

A binary tree is heap-ordered if:

1. the value at a node is $\geq$ the value at each of its children.
2. the tree is *almost-complete*.

**Example of complete tree (or not)**

![Example of complete tree](image)
Heap-ordered tree

Definition
A binary tree is heap-ordered if:

1. the value at a node is $\geq$ the value at each of its children.
2. the tree is almost-complete.

Example of complete tree (or not)
Heap-ordered tree

Definition

A binary tree is heap-ordered if:

1. the value at a node is $\geq$ the value at each of its children.
2. the tree is almost-complete.

Example of complete tree (or not)

![Examples of complete and not complete trees]
Heap-ordered tree

**Definition**

A binary tree is heap-ordered if:

1. the value at a node is $\geq$ the value at each of its children.
2. the tree is almost-complete. Height of tree is $\Theta(\lg n)$.

**Example of complete tree (or not)**

[Diagrams of complete and incomplete binary trees]
Heap-ordered tree

Definition
A binary tree is heap-ordered if:
1. the value at a node is $\geq$ the value at each of its children.
2. the tree is almost-complete. Height of tree is $\Theta(\lg n)$. Why?

Example of complete tree (or not)
Height of a heap-ordered tree

- Height $h$ of a tree is the maximum distance of any leaf node to the root.
Height of a heap-ordered tree

- Height $h$ of a tree is the maximum distance of any leaf node to the root.
- A heap of height $h$ has the most number of elements if the tree is complete, so $n$ equals the sum of nodes at each level.

\[
n \leq 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^h
\]

\[
= \frac{2^{h+1} - 1}{2 - 1}
= 2^{h+1} - 1.
\]
Height of a heap-ordered tree

- Height $h$ of a tree is the maximum distance of any leaf node to the root.
- A heap of height $h$ has the most number of elements if the tree is complete, so $n$ equals the sum of nodes at each level.

\[
  n \leq 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^h \\
  = \frac{2^{h+1} - 1}{2 - 1} \\
  = 2^{h+1} - 1.
\]

- It has the least number of elements if the lowest level has a single element and all higher levels are complete, so $n \geq 2^h - 1 + 1 = 2^h$. 
Height of a heap-ordered tree

- Height $h$ of a tree is the maximum distance of any leaf node to the root.
- A heap of height $h$ has the most number of elements if the tree is complete, so $n$ equals the sum of nodes at each level.

\[
 n \leq 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^h = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1.
\]

- It has the least number of elements if the lowest level has a single element and all higher levels are complete, so $n \geq 2^h - 1 + 1 = 2^h$.
- $2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1} \Rightarrow h \leq \lg n < h + 1$. Since $h$ is an integer, $h = \lfloor \lg n \rfloor = \Theta(\lg n)$. 
Heap – array representation of heap-ordered tree
Heap – array representation of heap-ordered tree
Heap – array representation of heap-ordered tree
Heap – array representation of heap-ordered tree
Heap – array representation of heap-ordered tree
Heap – array representation of heap-ordered tree
Heap – accessing parent and children

\[
\text{MAXIMUM}(A) \\
\text{return } A[1]
\]
Heap – accessing parent and children

MAXIMUM\( (A) \)

\[
\text{return } A[1]
\]

PARENT\( (i) \)

\[
\text{return } \lfloor i/2 \rfloor
\]
Heap – accessing parent and children

MAXIMUM(A)
    return A[1]

PARENT(i)
    return \lfloor i/2 \rfloor

Question
What if PARENT(i) < 1?
MAXIMUM(A)
  return A[1]

PARENT(i)
  return \lfloor i/2 \rfloor

LEFT(i)
  return 2i
Heap – accessing parent and children

MAXIMUM(A)
  return A[1]

PARENT(i)
  return \lfloor i/2 \rfloor

LEFT(i)
  return 2i

Question
What if LEFT(i) > n?
MAXIMUM(A)
return A[1]

PARENT(i)
return \[i/2\]

LEFT(i)
return 2i

RIGHT(i)
return 2i + 1
Heap – accessing parent and children

MAXIMUM($A$)

\[ \text{return } A[1] \]

PARENT($i$)

\[ \text{return } \lfloor i/2 \rfloor \]

LEFT($i$)

\[ \text{return } 2i \]

RIGHT($i$)

\[ \text{return } 2i + 1 \]

**Question**

What if $\text{RIGHT}(i) > n$?
Heap – accessing parent and children

- **MAXIMUM(A)**
  ```
  return A[1]
  ```

- **PARENT(i)**
  ```
  return \lfloor i/2 \rfloor
  ```

- **LEFT(i)**
  ```
  return 2i
  ```

- **RIGHT(i)**
  ```
  return 2i + 1
  ```

**Lemma**

All nodes $i > \lfloor \text{length}[A]/2 \rfloor$ (or equivalently, $i > \lfloor \text{heap-size}[A]/2 \rfloor$) are leaf nodes.
Heapsort

Heap – accessing parent and children

**MAXIMUM**\( (A) \)

\[
\text{return } A[1]
\]

**PARENT**(\( i \))

\[
\text{return } [i/2]
\]

**LEFT**(\( i \))

\[
\text{return } 2i
\]

**RIGHT**(\( i \))

\[
\text{return } 2i + 1
\]

**Definition (Heap property)**

**Heap property**: For every node \( i \) other than the root,

\[
A[\text{PARENT}(i)] \geq A[i].
\]
Heap – accessing parent and children

MAXIMUM($A$)

\[ \text{return } A[1] \]

PARENT($i$)

\[ \text{return } \lfloor i/2 \rfloor \]

LEFT($i$)

\[ \text{return } 2i \]

RIGHT($i$)

\[ \text{return } 2i + 1 \]

Question

Why do we insist that a heap-ordered tree be a complete binary tree? (Hint: draw the array representation of a tree that is not complete and see the gaps).
Contents

1 Heapsort
   • Introduction
   • Heap data structure
   • Heap algorithms
   • Heapsort algorithm
   • Priority queue
   • Conclusion
**MAX-HEAPIFY**($A, i$) – Ensure the heap property of $A$ starting at node $i$. Also known as “sink” operation since it sinks the lighter elements down the tree.
Operations on heap

1. **MAX-HEAPIFY**(A, i) – Ensure the heap property of A starting at node i. Also known as “sink” operation since it sinks the lighter elements down the tree.

2. **MAX-HEAP-INSERT**(A, key) – Insert key in the heap A, maintaining A’s heap property.
Operations on heap

1. MAX-HEAPIFY\((A, i)\) – Ensure the heap property of \(A\) starting at node \(i\). Also known as “sink” operation since it sinks the lighter elements down the tree.

2. MAX-HEAP-INSERT\((A, key)\) – Insert \(key\) in the heap \(A\), maintaining \(A\)'s heap property.

3. BUILD-MAX-HEAP\((A)\) – Build a \textit{max heap} given an array \(A\).
Operations on heap

1. MAX-HEAPIFY\((A, i)\) – Ensure the heap property of \(A\) starting at node \(i\). Also known as “sink” operation since it sinks the lighter elements down the tree.

2. MAX-HEAP-INSERT\((A, key)\) – Insert \(key\) in the heap \(A\), maintaining \(A\)'s heap property.

3. BUILD-MAX-HEAP\((A)\) – Build a max heap given an array \(A\).

4. HEAPSORT\((A)\) – Sort the elements in array \(A\) using the heap operations.
Operations on heap

1. **MAX-HEAPIFY**\((A, i)\) – Ensure the heap property of \(A\) starting at node \(i\). Also known as “sink” operation since it sinks the lighter elements down the tree.

2. **MAX-HEAP-INSERT**\((A, key)\) – Insert \(key\) in the heap \(A\), maintaining \(A\)'s heap property.

3. **BUILD-MAX-HEAP**\((A)\) – Build a max heap given an array \(A\).

4. **HEAPSORT**\((A)\) – Sort the elements in array \(A\) using the heap operations.

5. **HEAP-INCREASE-KEY**\((A, i, key)\) – Increase the value of element at node \(i\) to \(key\), and ensure the heap property of \(A\) by moving larger elements upwards. Also known as “swim” operation as it moves larger elements upwards.
Operations on heap

1. **MAX-HEAPIFY**(A, i) – Ensure the heap property of A starting at node i. Also known as “sink” operation since it sinks the lighter elements down the tree.

2. **MAX-HEAP-INSERT**(A, key) – Insert key in the heap A, maintaining A’s heap property.

3. **BUILD-MAX-HEAP**(A) – Build a max heap given an array A.

4. **HEAPSORT**(A) – Sort the elements in array A using the heap operations.

5. **HEAP-INCREASE-KEY**(A, i, key) – Increase the value of element at node i to key, and ensure the heap property of A by moving larger elements upwards. Also known as “swim” operation as it moves larger elements upwards.

6. **HEAP-EXTRACT-MAX**(A) – Extract the largest element from heap A.
Assumes that the trees Left(i) and Right(i) are max-heaps, but that A[i] might be smaller than its children.
Example of MAX-HEAPIFY ("sink") operation
Example of MAX-HEAPIFY ("sink") operation
MAX-HEAPIFY algorithm

MAX-HEAPIFY(A, i)
1. \( l \leftarrow \text{left}(i) \)
2. \( r \leftarrow \text{right}(i) \)
3. if \( l \leq \text{heap-size}[A] \) and \( A[l] > A[i] \)
   then \( \text{largest} \leftarrow l \)
4. else \( \text{largest} \leftarrow i \)
5. if \( r \leq \text{heap-size}[A] \) and \( A[r] > A[\text{largest}] \)
   then \( \text{largest} \leftarrow r \)
6. if \( \text{largest} \neq i \)
   then exchange \( A[i] \leftarrow A[\text{largest}] \)
10. \( \text{MAX-HEAPIFY}(A, \text{largest}) \)
MAX-HEAPIFY algorithm

MAX-HEAPIFY(A, i)
1  l ← left(i)
2  r ← right(i)
4    then largest ← l
5    else largest ← i
7    then largest ← r
8  if largest ≠ i
9    then exchange A[i] ← A[largest]
10   MAX-HEAPIFY(A, largest)

Analysis – first way

Since the children’s subtrees each have at most size of 2n/3 (when the last row is exactly half full), we have

\[ T(n) \leq T(2n/3) + \Theta(1). \]

According to case 2 of the Master theorem, \( T(n) = O(\log n) \).
MAX-HEAPIFY algorithm

MAX-HEAPIFY(A, i)
1  l ← left(i)
2  r ← right(i)
4      then largest ← l
5      else largest ← i
7      then largest ← r
8  if largest ≠ i
9      then exchange A[i] ↔ A[largest]
10     MAX-HEAPIFY(A, largest)

Analysis – second way

The running time of MAX-HEAPIFY on a node of height h is
\[ T(n) = O(h) = O(\log n). \]
Example of HEAP-INCREASE-KEY ("swim") operation
Example of **HEAP-INCREASE-KEY** ("swim") operation.
Example of **HEAP-INCREASE-KEY** ("swim") operation
Example of **HEAP-INCREASE-KEY** ("swim") operation
HEAP-INCREASE-KEY \( (A, i, key) \)

1. if \( key < A[i] \)
2. then error "new key is smaller than current key"
3. \( A[i] \leftarrow key \)
4. while \( i > 1 \) and \( A[\text{PARENT}(i)] < A[i] \)
5. do exchange \( A[i] \leftrightarrow A[\text{parent}(i)] \)
6. \( i \leftarrow \text{PARENT}(i) \)
HEAP-INCREASE-KEY algorithm

HEAP-INCREASE-KEY(A, i, key)
1. if key < A[i]
2. then error "new key is smaller than current key"
3. A[i] ← key
4. while i > 1 and A[PARENT(i)] < A[i]
5. do exchange A[i] ← A[parent(i)]
6. i ← PARENT(i)

Analysis

A node may move all the way from a leaf node to the root because of increased value, so \( T(n) = O(h) = O(\lg n) \).
Example of **MAX-HEAP-INSERT** operation

```
16
 /    
14    10
 /  
8   7   9   3
 /  
2   4   1
```

Insert 17
Example of **MAX-HEAP-INSERT** operation
Example of MAX-HEAP-INSERT operation
Example of **MAX-HEAP-INSERT** operation
Example of **MAX-HEAP-INSERT** operation
MAX-HEAP-INSERT algorithm

MAX-HEAP-INSERT($A, key$)
1  $heap-size[A] \leftarrow heap-size[A] + 1$
2  $A[heap-size[A]] \leftarrow key$
3  $i \leftarrow heap-size[A]$
4  while $i > 1$ and $A[parent(i)] < A[i]$
5     do exchange $A[i] \leftrightarrow A[parent(i)]$
6     $i \leftarrow A[parent(i)]$
MAX-HEAP-INSERT algorithm

MAX-HEAP-INSERT(A, key)

1. heap-size[A] ← heap-size[A] + 1
2. A[heap-size[A]] ← key
3. i ← heap-size[A]
4. while i > 1 and A[parent(i)] < A[i]
   5. do exchange A[i] ← A[parent(i)]
   6. i ← A[parent(i)]

Can also be done using HEAP-INCREASE-KEY.

MAX-HEAP-INSERT(A, key)

1. heap-size[A] ← heap-size[A] + 1
2. A[heap-size[A]] ← −∞
3. HEAP-INCREASE-KEY(A, heap-size[A], key)
**MAX-HEAP-INSERT** algorithm

```plaintext
MAX-HEAP-INSERT(A, key)
1  heap-size[A] ← heap-size[A] + 1
2  A[heap-size[A]] ← key
3  i ← heap-size[A]
4  while i > 1 and A[PARENT(i)] < A[i]
5      do exchange A[i] ← A[parent(i)]
6      i ← A[PARENT(i)]

Can also be done using **HEAP-INCREASE-KEY**.

MAX-HEAP-INSERT(A, key)
1  heap-size[A] ← heap-size[A] + 1
2  A[heap-size[A]] ← −∞
3  HEAP-INCREASE-KEY(A, heap-size[A], key)
```

**Analysis**

\[ T(n) = O(h) = O(lg n). \]
Simple BUILD-MAX-HEAP algorithm

BUILD-MAX-HEAP’(A)

1 \( \text{heap-size}[A] \leftarrow 1 \)
2 \( \text{for } i \leftarrow 2 \text{ to } \text{length}[A] \)
3 \( \text{do } \text{MAX-HEAP-INSERT}(A, A[i]) \)
**Simple BUILD-MAX-HEAP algorithm**

**BUILD-MAX-HEAP’**(A)

1. \(\text{heap-size}[A] \leftarrow 1\)
2. \(\text{for } i \leftarrow 2 \text{ to } \text{length}[A]\)
3. \(\text{do } \text{MAX-HEAP-INSERT}(A, A[i])\)

**Analysis**

There are \(n - 1\) calls to \text{MAX-HEAP-INSERT}, each taking \(O(\lg n)\) time, so \(T(n) = O(n \lg n)\).
Simple BUILD-MAX-HEAP algorithm

BUILD-MAX-HEAP'(A)
1  heap-size[A] ← 1
2  for i ← 2 to length[A]
3      do MAX-HEAP-INSERT(A, A[i])

Analysis

There are \( n - 1 \) calls to MAX-HEAP-INSERT, each taking \( O(\lg n) \) time, so \( T(n) = O(n \lg n) \).

Better way?

A better way is to build up the heap from the smaller trees. See next.
Example of **BUILD-MAX-HEAP** ("heapify") operation

1 2 3 4 5 6 7 8 9 10
4 1 3 2 16 9 10 14 8 7

```
1

4

2

1

4

2

16

i

10

5

6

3

7

9

14

8

7

8

9
```

Example of BUILD-MAX-HEAP ("heapify") operation
Example of BUILD-MAX-HEAP ("heapify") operation

1 2 3 4 5 6 7 8 9 10

4 1 3 2 16 9 10 14 8 7

Diagram of max heap:

1

4

2 1

4 14 16

5

3

i 6

3 9 10

7

8 9 10

2 8 7
Example of **BUILD-MAX-HEAP** ("heapify") operation
Example of BUILD-MAX-HEAP ("heapify") operation

1 2 3 4 5 6 7 8 9 10
4 1 3 2 16 9 10 14 8 7
Example of **BUILD-MAX-HEAP** ("heapify") operation
BUILD-MAX-HEAP algorithm

1. heap-size[A] ← length[A]
2. for i ← ⌊length[A]/2⌋ downto 1
3. do MAX-HEAPIFY(A, i)
BUILD-MAX-HEAP algorithm

BUILD-MAX-HEAP(A)

1  heap-size[A] ← length[A]
2  for i ← ⌈length[A]/2⌉ downto 1
3      do MAX-HEAPIFY(A, i)

Analysis

\[ T(n) = O(n) \] (see textbook for details)
Build-Max-Heap

Correctness

**Loop invariant:** At start of every iteration of for loop, each node \(i + 1, i + 2, \ldots, n\) is root of a max-heap.

**Initialization:** By Exercise 6.1-7, we know that each node \([n/2] + 1, [n/2] + 2, \ldots, n\) is a leaf, which is the root of a trivial max-heap. Since \(i = [n/2]\) before the first iteration of the for loop, the invariant is initially true.

**Maintenance:** Children of node \(i\) are indexed higher than \(i\), so by the loop invariant, they are both roots of max-heaps. Correctly assuming that \(i + 1, i + 2, \ldots, n\) are all roots of max-heaps, MAX-HEAPIFY makes node \(i\) a max-heap root. Decrementing \(i\) reestablishes the loop invariant at each iteration.

**Termination:** When \(i = 0\), the loop terminates. By the loop invariant, each node, notably node 1, is the root of a max-heap.

BUILD-MAX-HEAP\( (A) \)

1. \(heap-size[A] \leftarrow length[A]\)
2. for \(i \leftarrow [\text{length}[A]/2]\) downto 1
3. do MAX-HEAPIFY\( (A, i)\)
1. Heapsort
   - Introduction
   - Heap data structure
   - Heap algorithms
   - Heapsort algorithm
   - Priority queue
   - Conclusion
Heap use - heapsort

A heap is a complete binary tree where each node is greater than or equal to its children (max heap) or less than or equal to its children (min heap). The diagram shows a max heap with the root node 10. The heap property is maintained throughout the tree.
Heap use - heapsort

The diagram shows a heap structure.

- The root node has the value 9.
- Node 8 is below node 9.
- Node 3 is below node 9.
- Nodes 4, 7, 1, and 2 are below nodes 8 and 3, respectively.
- Node i is below node 9.
- Nodes 10, 14, and 16 are below node i.
Heap use - heapsort
Heap use - heapsort
Heap use - heapsort

```
  3
 / \
2   1
 |   |
\i/ |  |
 4   7
`-\  |
  10 14 16
```
Heap use - heapsort
HEAPSORT algorithm

HEAPSORT(A)

1  BUILD-MAX-HEAP(A)  \hspace{1cm} \text{cost} \hspace{1cm} \text{times}
2  for \ i \leftarrow \text{length}[A] \text{ downto } 2  \\
3  \hspace{1cm} \text{do } \text{exchange } A[1] \leftrightarrow A[i]  \\
4  \hspace{1cm} \text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1  \\
5  \hspace{1cm} \text{MAX-HEAPIFY}(A, 1)  \\
   \Theta(n) \hspace{1cm} 1  \\
   \Theta(1) \hspace{1cm} n  \\
   \Theta(1) \hspace{1cm} n - 1  \\
   \Theta(1) \hspace{1cm} n - 1  \\
   \Theta(\lg n) \hspace{1cm} n - 1
HEAPSORT algorithm

HEAPSORT(\(A\))

\begin{align*}
1 & \text{ BUILD-MAX-HEAP(} A \text{)} & \Theta(n) & 1 \\
2 & \text{ for } i \leftarrow \text{length}[A] \text{ downto 2} & \Theta(1) & n \\
3 & \quad \text{ do exchange } A[1] \leftrightarrow A[i] & \Theta(1) & n - 1 \\
4 & \quad \text{ heap-size}[A] \leftarrow \text{heap-size}[A] - 1 & \Theta(1) & n - 1 \\
5 & \text{ MAX-HEAPIFY(} A, 1 \text{)} & \Theta(\log n) & n - 1
\end{align*}
HEAPSORT algorithm

HEAPSORT(A)

1. BUILD-MAX-HEAP(A)  \( \Theta(n) \)  \( n \)
2. for \( i \leftarrow \text{length}[A] \) downto 2  \( \Theta(1) \)  \( n \)
3. \hspace{1em} do exchange \( A[1] \leftarrow A[i] \)  \( \Theta(1) \)  \( n - 1 \)
4. \hspace{2em} heap-size[A] \leftarrow \text{heap-size}[A] - 1  \( \Theta(1) \)  \( n - 1 \)
5. MAX-HEAPIFY(A, 1)  \( \Theta(\lg n) \)  \( n - 1 \)
HEAPSORT algorithm

HEAPSORT(A)

1. BUILD-MAX-HEAP(A)
2. for i ← length[A] downto 2
4. MAX-HEAPIFY(A, 1)

<table>
<thead>
<tr>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Θ(n)</td>
<td>1</td>
</tr>
<tr>
<td>Θ(1)</td>
<td>n</td>
</tr>
<tr>
<td>Θ(1)</td>
<td>n - 1</td>
</tr>
<tr>
<td>Θ(lg n)</td>
<td>n - 1</td>
</tr>
</tbody>
</table>
HEAPSORT algorithm

**HEAPSORT**(A)

1. BUILD-MAX-HEAP(A)  \(\Theta(n)\)  1
2. for \(i \leftarrow \text{length}[A]\) downto 2  \(\Theta(1)\)  \(n\)
3. \hspace{1em} do exchange A[1] \(\leftrightarrow\) A[i]  \(\Theta(1)\)  \(n - 1\)
4. \hspace{1em} heap-size[A] \(\leftarrow\) heap-size[A] \(- 1\)  \(\Theta(1)\)  \(n - 1\)
5. MAX-HEAPIFY(A, 1)  \(\Theta(\lg n)\)  \(n - 1\)
**HEAPSORT** algorithm

HEAPSORT\((A)\)

1. BUILD-MAX-HEAP\((A)\) \(\Theta(n)\) 1
2. for \(i \leftarrow \text{length}[A]\) downto 2
3. do exchange \(A[1] \leftrightarrow A[i]\) \(\Theta(1)\) \(n - 1\)
4. heap-size\([A]\) \(\leftarrow\) heap-size\([A]\) \(- 1\) \(\Theta(1)\) \(n - 1\)
5. MAX-HEAPIFY\((A, 1)\) \(\Theta(\lg n)\) \(n - 1\)
HEAPSORT algorithm

HEAPSORT(A)

1. BUILD-MAX-HEAP(A)  
2. for i ← length[A] downto 2  
4. heap-size[A] ← heap-size[A] − 1  
5. MAX-HEAPIFY(A, 1)

<table>
<thead>
<tr>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Θ(n)</td>
<td>1</td>
</tr>
<tr>
<td>Θ(1)</td>
<td>n</td>
</tr>
<tr>
<td>Θ(1)</td>
<td>n − 1</td>
</tr>
<tr>
<td>Θ(1)</td>
<td>n − 1</td>
</tr>
<tr>
<td>Θ(lg n)</td>
<td>n − 1</td>
</tr>
</tbody>
</table>

Worst-case analysis

\[ T(n) = \Theta(n \lg n) \]
Contents

1 Heapsort
   • Introduction
   • Heap data structure
   • Heap algorithms
   • Heapsort algorithm
   • Priority queue
   • Conclusion
Definition (Priority Queue)

A priority queue is a data structure for maintaining a set $S$ of elements, each with an associated value called a key. A max-priority queue supports the following operations.

1. $\text{INSERT}(S, x)$ – inserts the element $x$ into the set $S$. This operation could be written as $S \leftarrow S \cup \{x\}$.
2. $\text{MAXIMUM}(S)$ – returns the element of $S$ with the largest key.
3. $\text{EXTRACT-MAX}(S)$ – removes and returns the element of $S$ with the largest key.
4. $\text{INCREASE-KEY}(S, x, k)$ – increases the value of element $x$’s key to $k$. Assume $k \geq x$’s current value.
Heap use - priority queue

Definition (Priority Queue)

A priority queue is a data structure for maintaining a set $S$ of elements, each with an associated value called a key. A max-priority queue supports the following operations.

1. $\text{INSERT}(S, x) -$ inserts the element $x$ into the set $S$. This operation could be written as $S \leftarrow S \cup \{x\}$.
2. $\text{MAXIMUM}(S) -$ returns the element of $S$ with the largest key.
3. $\text{EXTRACT-MAX}(S) -$ removes and returns the element of $S$ with the largest key.
4. $\text{INCREASE-KEY}(S, x, k) -$ increases the value of element $x$’s key to $k$. Assume $k \geq x$’s current value.

Used in many scheduling applications where jobs or tasks are scheduled according to priority.
Heap use - priority queue

Definition (Priority Queue)

A priority queue is a data structure for maintaining a set $S$ of elements, each with an associated value called a key. A max-priority queue supports the following operations.

1. $\text{INSERT}(S, x)$ – inserts the element $x$ into the set $S$. This operation could be written as $S \leftarrow S \cup \{x\}$.
2. $\text{MAXIMUM}(S)$ – returns the element of $S$ with the largest key.
3. $\text{EXTRACT-MAX}(S)$ – removes and returns the element of $S$ with the largest key.
4. $\text{INCREASE-KEY}(S, x, k)$ – increases the value of element $x$’s key to $k$. Assume $k \geq x$’s current value.

- Used in many scheduling applications where jobs or tasks are scheduled according to priority.
- A FIFO queue is a priority queue where the priority is inversely proportional to time of arrival.
Definition (Priority Queue)

A priority queue is a data structure for maintaining a set $S$ of elements, each with an associated value called a *key*. A *max-priority queue* supports the following operations:

1. **INSERT($S$, $x$)** – inserts the element $x$ into the set $S$. This operation could be written as $S \leftarrow S \cup \{x\}$.
2. **MAXIMUM($S$)** – returns the element of $S$ with the largest key.
3. **EXTRACT-MAX($S$)** – removes and returns the element of $S$ with the largest key.
4. **INCREASE-KEY($S$, $x$, $k$)** – increases the value of element $x$’s key to $k$. Assume $k \geq x$’s current value.

- Used in many scheduling applications where jobs or tasks are scheduled according to priority.
- A FIFO queue is a priority queue where the priority is inversely proportional to time of arrival.
- A LIFO stack is a priority queue where the priority is proportional to time of arrival.
Example of **HEAP-EXTRACT-MAX** operation
Example of **HEAP-EXTRACT-MAX** operation
Example of HEAP-EXTRACT-MAX operation
Example of HEAP-EXTRACT-MAX operation
Example of **HEAP-EXTRACT-MAX** operation
Example of HEAP-EXTRACT-MAX operation
Example of HEAP-EXTRACT-MAX operation
HEAP-EXTRACT-MAX algorithm

EXTRACT-MAX(A)

1. if heap-size[A] < 1
2. then error "heap underflow"
3. max ← A[1]
5. heap-size[A] ← heap-size[A] − 1
6. MAX-HEAPIFY(A, 1)

<table>
<thead>
<tr>
<th></th>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Θ(1)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Θ(1)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Θ(1)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Θ(1)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Θ(lg n)</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**HEAP-EXTRACT-MAX** algorithm

**EXTRACT-MAX**(A)

1. **if** heap-size[A] < 1
2. **then** error “heap underflow”
3. max ← A[1]
5. heap-size[A] ← heap-size[A] − 1
6. MAX-HEAPIFY(A, 1)

<table>
<thead>
<tr>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Θ(1)</td>
<td>1</td>
</tr>
<tr>
<td>Θ(1)</td>
<td>1</td>
</tr>
<tr>
<td>Θ(1)</td>
<td>1</td>
</tr>
<tr>
<td>Θ(1)</td>
<td>1</td>
</tr>
<tr>
<td>Θ(1)</td>
<td>1</td>
</tr>
<tr>
<td>Θ(lg n)</td>
<td>1</td>
</tr>
</tbody>
</table>
**HEAP-EXTRACT-MAX algorithm**

**EXTRACT-MAX(A)**

1. if $heap-size[A] < 1$
2. then error “heap underflow”
3. $max \leftarrow A[1]$
5. $heap-size[A] \leftarrow heap-size[A] - 1$
6. MAX-HEAPIFY($A$, 1)

<table>
<thead>
<tr>
<th>$cost$</th>
<th>$times$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(1)$</td>
<td>1</td>
</tr>
<tr>
<td>$\Theta(1)$</td>
<td>1</td>
</tr>
<tr>
<td>$\Theta(1)$</td>
<td>1</td>
</tr>
<tr>
<td>$\Theta(1)$</td>
<td>1</td>
</tr>
<tr>
<td>$\Theta(lg n)$</td>
<td>1</td>
</tr>
</tbody>
</table>
HEAP-EXTRACT-MAX algorithm

EXTRACT-MAX(A)

1. if heap-size[A] < 1
2. then error "heap underflow"
3. max ← A[1]
5. heap-size[A] ← heap-size[A] − 1
6. MAX-HEAPIFY(A, 1)

<table>
<thead>
<tr>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Theta(1))</td>
<td>1</td>
</tr>
<tr>
<td>(\Theta(1))</td>
<td>1</td>
</tr>
<tr>
<td>(\Theta(1))</td>
<td>1</td>
</tr>
<tr>
<td>(\Theta(1))</td>
<td>1</td>
</tr>
<tr>
<td>(\Theta(1))</td>
<td>1</td>
</tr>
<tr>
<td>(\Theta(\lg n))</td>
<td>1</td>
</tr>
</tbody>
</table>
HEAP-EXTRACT-MAX algorithm

EXTRACT-MAX(A)

1 if heap-size[A] < 1
2 then error "heap underflow"
3 max ← A[1]
5 heap-size[A] ← heap-size[A] − 1
6 MAX-HEAPIFY(A, 1)

<table>
<thead>
<tr>
<th></th>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Θ(1)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Θ(1)</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Θ(1)</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Θ(1)</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Θ(1)</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Θ(lg n)</td>
<td>1</td>
</tr>
</tbody>
</table>
EXTRACT-MAX($A$)

1. if heap-size[$A$] < 1
2. then error "heap underflow"
3. $max \leftarrow A[1]$
5. heap-size[$A$] $\leftarrow$ heap-size[$A$] $- 1$
6. MAX-HEAPIFY($A$, 1)
**Heap-Extract-Max** algorithm

**EXTRACT-MAX**($A$)

1. if heap-size[$A$] < 1
2. then error “heap underflow”
3. $max \gets A[1]$
4. $A[1] \gets A[\text{heap-size}[A]]$
5. $\text{heap-size}[A] \gets \text{heap-size}[A] - 1$
6. MAX-HEAPIFY($A$, 1)

<table>
<thead>
<tr>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(1)$</td>
<td>1</td>
</tr>
<tr>
<td>$\Theta(1)$</td>
<td>1</td>
</tr>
<tr>
<td>$\Theta(1)$</td>
<td>1</td>
</tr>
<tr>
<td>$\Theta(1)$</td>
<td>1</td>
</tr>
<tr>
<td>$\Theta(1)$</td>
<td>1</td>
</tr>
<tr>
<td>$\Theta(\lg n)$</td>
<td>1</td>
</tr>
</tbody>
</table>

**Worst-case analysis**

$T(n) = \Theta(\lg n)$
Chapter 6: Heapsort

Heapsort

- $O(n \lg n)$ worst case—like merge sort.
- Sorts in place—like insertion sort.
- Combines the best of both algorithms.

To understand heapsort, we’ll cover heaps and heap operations, and then we’ll take a look at priority queues.
Heapsort

Heap data structure

- Heap $A$ (*not* garbage-collected storage) is a nearly complete binary tree.
  - **Height** of node = # of edges on a longest simple path from the node down to a leaf.
  - **Height** of heap = height of root = $\Theta(\lg n)$.
- A heap can be stored as an array $A$.
  - Computing is fast with binary representation implementation.
Heapsort

Example
Of a max-heap. [Arcs above and below the array on the right go between parents and children. There is no significance to whether an arc is drawn above or below the array.]

Heap property
- For max-heaps (largest element at root), **max-heap property**: for all nodes \( i \), excluding the root, \( A[\text{PARENT}(i)] \geq A[i] \).
- For min-heaps (smallest element at root), **min-heap property**: for all nodes \( i \), excluding the root, \( A[\text{PARENT}(i)] \leq A[i] \).

By induction and transitivity of \( \leq \), the max-heap property guarantees that the maximum element of a max-heap is at the root. Similar argument for min-heaps.

The heapsort algorithm we’ll show uses max-heaps.

Note: In general, heaps can be \( k \)-ary tree instead of binary.
Heapsort

Maintaining the heap property

MAX-HEAPIFY is important for manipulating max-heaps. It is used to maintain
the max-heap property.

- Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
- Assume left and right subtrees of $i$ are max-heaps.
- After MAX-HEAPIFY, subtree rooted at $i$ is a max-heap.

MAX-HEAPIFY $(A, i, n)$

$$l = \text{LEFT}(i)$$
$$r = \text{RIGHT}(i)$$

\text{if } l \leq n \text{ and } A[l] > A[i]$

$$\text{largest} = l$$

\text{else } \text{largest} = i$

\text{if } r \leq n \text{ and } A[r] > A[\text{largest}]$

$$\text{largest} = r$$

\text{if } \text{largest} \neq i$

exchange $A[i]$ with $A[\text{largest}]$

MAX-HEAPIFY $(A, \text{largest}, n)$

[Parameter $n$ replaces attribute $A\.heap-size\.\]
The way MAX-HEAPIFY works:

- Compare $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$.
- If necessary, swap $A[i]$ with the larger of the two children to preserve heap property.
- Continue this process of comparing and swapping down the heap, until subtree rooted at $i$ is max-heap. If we hit a leaf, then the subtree rooted at the leaf is trivially a max-heap.
Run MAX-HEAPIFY on the following heap example.

(a)  

(b)  

(c)
Heapsort

Time
$O(lg n)$

Intuitive Analysis
Heap is almost-complete binary tree, hence must process $O(lg n)$ levels, with constant work at each level (comparing 3 items and maybe swapping 2).

For more formal analysis see your book using recurrence and Master method.
Build-Max-Heap

Building a heap

The following procedure, given an unordered array, will produce a max-heap.

**BUILD-MAX-HEAP** \((A, n)\)

\[
\text{for } i = \lfloor n/2 \rfloor \text{ downto } 1 \\
\quad \text{MAX-HEAPIFY}(A, i, n)
\]

[Parameter \(n\) replaces both attributes \(A.length\) and \(A.heap-size\).]

**Example**

Building a max-heap from the following unsorted array results in the first heap example.

- \(i\) starts off as 5.
- \(\text{MAX-HEAPIFY}\) is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.

\[
A = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
4 & 1 & 3 & 2 & 16 & 9 & 10 & 14 & 8 & 7 \\
\end{bmatrix}
\]

See next page for detail.
The Operation of BUILD_MAX_HEAP
Correctness

Loop invariant: At start of every iteration of for loop, each node \( i + 1, \ i + 2, \ldots, \ n \) is root of a max-heap.

Initialization: By Exercise 6.1-7, we know that each node \( \lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, \ n \) is a leaf, which is the root of a trivial max-heap. Since \( i = \lfloor n/2 \rfloor \) before the first iteration of the for loop, the invariant is initially true.

Maintenance: Children of node \( i \) are indexed higher than \( i \), so by the loop invariant, they are both roots of max-heaps. Correctly assuming that \( i + 1, i + 2, \ldots, n \) are all roots of max-heaps, MAX-HEAPIFY makes node \( i \) a max-heap root. Decrementing \( i \) reestablishes the loop invariant at each iteration.

Termination: When \( i = 0 \), the loop terminates. By the loop invariant, each node, notably node 1, is the root of a max-heap.
Analysis

- **Simple bound:** $O(n)$ calls to MAX-HEAPIFY, each of which takes $O(\lg n)$ time $\Rightarrow O(n \lg n)$. (Note: A good approach to analysis in general is to start by proving easy bound, then try to tighten it.)

- **Tighter analysis:** Observation: Time to run MAX-HEAPIFY is linear in the height of the node it’s run on, and most nodes have small heights. Have $\leq \left\lceil \frac{n}{2^h+1} \right\rceil$ nodes of height $h$ (see Exercise 6.3-3), and height of heap is $\lceil \lg n \rceil$ (Exercise 6.1-2).

The time required by MAX-HEAPIFY when called on a node of height $h$ is $O(h)$, so the total cost of BUILD-MAX-HEAP is

$$
\sum_{h=0}^{\lceil \lg n \rceil} \left\lceil \frac{n}{2^h+1} \right\rceil O(h) = O \left( n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^h} \right).
$$

Evaluate the last summation by substituting $x = 1/2$ in the formula (A.8) ($\sum_{k=0}^{\infty} kx^k$), which yields

$$
\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1 - 1/2)^2} = 2.
$$

Thus, the running time of BUILD-MAX-HEAP is $O(n)$.

Building a min-heap from an unordered array can be done by calling MIN-HEAPIFY instead of MAX-HEAPIFY, also taking linear time.
Heapsort

The heapsort algorithm

Given an input array, the heapsort algorithm acts as follows:

• Builds a max-heap from the array.
• Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.
• “Discard” this last node (knowing that it is in its correct place) by decreasing the heap size, and calling MAX-HEAPIFY on the new (possibly incorrectly-placed) root.
• Repeat this “discarding” process until only one node (the smallest element) remains, and therefore is in the correct place in the array.

HEAPSORT\( (A, n) \)

BUILD-MAX-HEAP\( (A, n) \)

for \( i = n \) downto 2


MAX-HEAPIFY\( (A, 1, i - 1) \)

[Parameter \( n \) replaces \( A.length \), and parameter value \( i - 1 \) in MAX-HEAPIFY call replaces decrementing of \( A.heap-size \).]
Example
Sort an example heap on the board. [Nodes with heavy outline are no longer in the heap.]

(a)  

(b)  

(c)  

(d)  

(e)
Heapsort Example
Heapsort

Analysis

• BUILD-MAX-HEAP: $O(n)$
• for loop: $n - 1$ times
• exchange elements: $O(1)$
• MAX-HEAPIFY: $O(\lg n)$

Total time: $O(n \lg n)$.

Though heapsort is a great algorithm, a well-implemented quicksort usually beats it in practice.
Heap implementation of priority queue

Heaps efficiently implement priority queues. These notes will deal with max-priority queues implemented with max-heaps. Min-priority queues are implemented with min-heaps similarly.

A heap gives a good compromise between fast insertion but slow extraction and vice versa. Both operations take $O(\lg n)$ time.

Priority queue

- Maintains a dynamic set $S$ of elements.
- Each set element has a key—an associated value.
- Max-priority queue supports dynamic-set operations:
  - \textsc{Insert}(S, x): inserts element $x$ into set $S$.
  - \textsc{Maximum}(S): returns element of $S$ with largest key.
  - \textsc{Extract-Max}(S): removes and returns element of $S$ with largest key.
  - \textsc{Increase-Key}(S, x, k): increases value of element $x$'s key to $k$. Assume $k \geq x$'s current key value.
Priority Queues

- Example max-priority queue application: schedule jobs on shared computer.
- Min-priority queue supports similar operations:
  - $\text{INSERT}(S, x)$: inserts element $x$ into set $S$.
  - $\text{MINIMUM}(S)$: returns element of $S$ with smallest key.
  - $\text{EXTRACT-MIN}(S)$: removes and returns element of $S$ with smallest key.
  - $\text{DECREASE-KEY}(S, x, k)$: decreases value of element $x$'s key to $k$. Assume $k \leq x$'s current key value.
- Example min-priority queue application: event-driven simulator.

Note: Actual implementations often have a handle in each heap element that allows access to an object in the application, and objects in the application often have a handle (likely an array index) to access the heap element.
Finding the maximum element

Getting the maximum element is easy: it’s the root.

\textbf{Heap-Maximum}(A) \\
\hspace{1em} \textbf{return} \ A[1]

\textit{Time} \\
\Theta(1).

Extracting max element

Given the array \(A\):

- Make sure heap is not empty.
- Make a copy of the maximum element (the root).
- Make the last node in the tree the new root.
- Re-heapify the heap, with one fewer node.
- Return the copy of the maximum element.

Note: Because we need to decrement the heap size \(n\) in the following pseudocode, assume that it is passed by reference, not by value.

\textit{[This issue does not come up in the pseudocode in the book, because it uses the attribute \(A.heap\text{-}size\) instead of passing in the heap size as a parameter.]}

Priority Queues

**HEAP-EXTRACT-MAX**($A, n$)

```
if $n < 1$
    error "heap underflow"
max = $A[1]$
n = n - 1
MAX-HEAPIFY($A, 1, n$) // remakes heap
return max
```

*Analysis*

Constant-time assignments plus time for MAX-HEAPIFY.

*Time*

$O(\lg n)$. 

Priority Queues

Example
Run \textsc{Heap-Extract-Max} on first heap example.

- Take 16 out of node 1.
- Move 1 from node 10 to node 1.
- Erase node 10.
- \textsc{Max-Heapify} from the root to preserve max-heap property.
- Note that successive extractions will remove items in reverse sorted order.

Increasing key value

Given set $S$, element $x$, and new key value $k$:

- Make sure $k \geq x$'s current key.
- Update $x$'s key value to $k$.
- Traverse the tree upward comparing $x$ to its parent and swapping keys if necessary, until $x$'s key is smaller than its parent's key.
**Heap-Increase-Key** \((A, i, key)\)

- **if** \(key < A[i]\)
  - **error** “new key is smaller than current key”
  - \(A[i] = key\)
- **while** \(i > 1\) and \(A[\text{PARENT}(i)] < A[i]\)
  - exchange \(A[i]\) with \(A[\text{PARENT}(i)]\)
  - \(i = \text{PARENT}(i)\)

**Analysis**

Upward path from node \(i\) has length \(O(\log n)\) in an \(n\)-element heap.

**Time**

\(O(\log n)\).

**Example**

Increase key of node 9 in first heap example to have value 15. Exchange keys of nodes 4 and 9, then of nodes 2 and 4.
Heap-Increase-key Example

(a) Original heap:
- 16
- 14
- 8
- 4
- 2
- 7
- 1

(b) After increasing key 4 to 15:
- 16
- 14
- 8
- 15
- 2
- 7
- 1

(c) After adjusting the heap:
- 16
- 14
- 15
- 7
- 9
- 3
- 2
- 8
- 1

(d) Final heap:
- 16
- 14
- 8
- 7
- 9
- 3
- 1
- 2
- 15

Priority Queues

Inserting into the heap

Given a key $k$ to insert into the heap:

- Increment the heap size.
- Insert a new node in the last position in the heap, with key $-\infty$.
- Increase the $-\infty$ key to $k$ using the HEAP-INCREASE-KEY procedure defined above.

Note: Again, the parameter $n$ is passed by reference, not by value.

**MAX-HEAP-INSERT** $(A, key, n)$

\[
\begin{align*}
    n &= n + 1 \\
    A[n] &= -\infty \\
    \text{HEAP-INCREASE-KEY} (A, n, key)
\end{align*}
\]

**Analysis**

Constant time assignments + time for **HEAP-INCREASE-KEY**.

**Time**

$O(\lg n)$.

Min-priority queue operations are implemented similarly with min-heaps.
Chapter 7: Quicksort

Quicksort

• Worst-case running time: $\Theta(n^2)$.  
• Expected running time: $\Theta(n \lg n)$.  
• Constants hidden in $\Theta(n \lg n)$ are small.  
• Sorts in place.

Description of quicksort

Quicksort is based on the three-step process of divide-and-conquer.

• To sort the subarray $A[p..r]$:
  
  **Divide:** Partition $A[p..r]$, into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$, such that each element in the first subarray $A[p..q-1]$ is $\leq A[q]$ and $A[q]$ is $\leq$ each element in the second subarray $A[q+1..r]$.
  
  **Conquer:** Sort the two subarrays by recursive calls to QUICKSORT.
  
  **Combine:** No work is needed to combine the subarrays, because they are sorted in place.

• Perform the divide step by a procedure PARTITION, which returns the index $q$ that marks the position separating the subarrays.
Quicksort

\[
\text{QUICKSORT}(A, p, r) \\
\quad \text{if } p < r \\
\quad \quad q = \text{PARTITION}(A, p, r) \\
\quad \quad \text{QUICKSORT}(A, p, q - 1) \\
\quad \quad \text{QUICKSORT}(A, q + 1, r)
\]

Initial call is \text{QUICKSORT}(A, 1, n).
Partitioning

Partition subarray $A[p \ldots r]$ by the following procedure:

\[\text{PARTITION}(A, p, r)\]

1. $x = A[r]$
2. $i = p - 1$
3. for $j = p$ to $r - 1$
   - if $A[j] \leq x$
     - $i = i + 1$
     - exchange $A[i]$ with $A[j]$
   - exchange $A[i + 1]$ with $A[r]$
4. return $i + 1$

- PARTITION always selects the last element $A[r]$ in the subarray $A[p \ldots r]$ as the pivot—the element around which to partition.
- As the procedure executes, the array is partitioned into four regions, some of which may be empty:

  **Loop invariant:**
  1. All entries in $A[p \ldots i]$ are $\leq$ pivot.
  2. All entries in $A[i + 1 \ldots j - 1]$ are $> \text{pivot}$.

It’s not needed as part of the loop invariant, but the fourth region is $A[j \ldots r - 1]$, whose entries have not yet been examined, and so we don’t know how they compare to the pivot.
**Partitioning Example**

```
\begin{algorithm}
\textbf{PARTITION}(A, p, r)
\begin{align*}
x &= A[r] \\
i &= p - 1 \\
\text{for } j &= p \text{ to } r - 1 \\
\text{if } A[j] &\leq x \\
&\quad i = i + 1 \\
\text{exchange } A[i] \text{ with } A[j] \\
\text{exchange } A[i + 1] \text{ with } A[r] \\
\text{return } i + 1
\end{align*}
\end{algorithm}
```

[The index \( j \) disappears because it is no longer needed once the \textbf{for} loop is exited.]
Partitioning

Correctness
Use the loop invariant to prove correctness of PARTITION:

Initialization: Before the loop starts, all the conditions of the loop invariant are satisfied, because \( r \) is the pivot and the subarrays \( A[p..i] \) and \( A[i+1..j-1] \) are empty.

Maintenance: While the loop is running, if \( A[j] \leq \text{pivot} \), then \( A[j] \) and \( A[i+1] \) are swapped and then \( i \) and \( j \) are incremented. If \( A[j] > \text{pivot} \), then increment only \( j \).

Termination: When the loop terminates, \( j = r \), so all elements in \( A \) are partitioned into one of the three cases: \( A[p..i] \leq \text{pivot} \), \( A[i+1..r-1] > \text{pivot} \), and \( A[r] = \text{pivot} \).

The last two lines of PARTITION move the pivot element from the end of the array to between the two subarrays. This is done by swapping the pivot and the first element of the second subarray, i.e., by swapping \( A[i+1] \) and \( A[r] \).

Time for partitioning
\( \Theta(n) \) to partition an \( n \)-element subarray.
Quicksort

Performance of quicksort

The running time of quicksort depends on the partitioning of the subarrays:
- If the subarrays are balanced, then quicksort can run as fast as mergesort.
- If they are unbalanced, then quicksort can run as slowly as insertion sort.

Worst case

- Occurs when the subarrays are completely unbalanced.
- Have 0 elements in one subarray and \( n - 1 \) elements in the other subarray.
- Get the recurrence
  \[
  T(n) = T(n - 1) + T(0) + \Theta(n) \\
  = T(n - 1) + \Theta(n) \\
  = \Theta(n^2).
  \]
- Same running time as insertion sort.
- In fact, the worst-case running time occurs when quicksort takes a sorted array as input, but insertion sort runs in \( O(n) \) time in this case.

Best case

- Occurs when the subarrays are completely balanced every time.
- Each subarray has \( \leq n/2 \) elements.
- Get the recurrence
  \[
  T(n) = 2T(n/2) + \Theta(n) \\
  = \Theta(n \lg n).
  \]
Quicksort

Balanced partitioning

- Quicksort’s average running time is much closer to the best case than to the worst case.
- Imagine that PARTITION always produces a 9-to-1 split.
- Get the recurrence
  \[ T(n) \leq T(9n/10) + T(n/10) + \Theta(n) \]
  \[ = O(n \lg n) \, . \]
- Intuition: look at the recursion tree.
  - It’s like the one for \( T(n) = T(n/3) + T(2n/3) + O(n) \) in Section 4.4.
  - Except that here the constants are different; we get \( \log_{10} n \) full levels and \( \log_{10/9} n \) levels that are nonempty.
  - As long as it’s a constant, the base of the log doesn’t matter in asymptotic notation.
  - Any split of constant proportionality will yield a recursion tree of depth \( \Theta(\lg n) \).
Quicksort

\[ O(n \log n) \]
Quicksort

Intuition for the average case

- Splits in the recursion tree will not always be constant.
- There will usually be a mix of good and bad splits throughout the recursion tree.
- To see that this doesn’t affect the asymptotic running time of quicksort, assume that levels alternate between best-case and worst-case splits.

The extra level in the left-hand figure only adds to the constant hidden in the $\Theta$-notation.

There are still the same number of subarrays to sort, and only twice as much work was done to get to that point.

Both figures result in $O(n \lg n)$ time, though the constant for the figure on the left is higher than that of the figure on the right.
Randomized Quicksort

Randomized version of quicksort

- We have assumed that all input permutations are equally likely.
- This is not always true.
- To correct this, we add randomization to quicksort.
- We could randomly permute the input array.
- Instead, we use random sampling, or picking one element at random.
- Don’t always use $A[r]$ as the pivot. Instead, randomly pick an element from the subarray that is being sorted.

```plaintext
RANDOMIZED-PARTITION(A, p, r)
    i = RANDOM(p, r)
    exchange $A[r]$ with $A[i]$
    return PARTITION(A, p, r)
```

Randomly selecting the pivot element will, on average, cause the split of the input array to be reasonably well balanced.
Randomized Quicksort

\texttt{RANDOMIZED-QUICKSORT}(A, p, r)

\textbf{if} \( p < r \)

\hspace{10pt} \( q = \texttt{RANDOMIZED-PARTITION}(A, p, r) \)

\hspace{10pt} \texttt{RANDOMIZED-QUICKSORT}(A, p, q - 1) \)

\hspace{10pt} \texttt{RANDOMIZED-QUICKSORT}(A, q + 1, r) \)

Randomization of quicksort stops any specific type of array from causing worst-case behavior. For example, an already-sorted array causes worst-case behavior in non-randomized \texttt{QUICKSORT}, but not in \texttt{RANDOMIZED-QUICKSORT}. 
Chapter 8 overview

How fast can we sort?

We will prove a lower bound, then beat it by playing a different game.

Comparison sorting

• The only operation that may be used to gain order information about a sequence is comparison of pairs of elements.
• All sorts seen so far are comparison sorts: insertion sort, selection sort, merge sort, quicksort, heapsort, treesort.
Comparison Sorts

Lower bounds for sorting

Lower bounds

- $\Omega(n)$ to examine all the input.
- All sorts seen so far are $\Omega(n \lg n)$.
- We’ll show that $\Omega(n \lg n)$ is a lower bound for comparison sorts.

Decision tree

- Abstraction of any comparison sort.
- Represents comparisons made by
  - a specific sorting algorithm
  - on inputs of a given size.
- Abstracts away everything else: control and data movement.
- We’re counting *only* comparisons.
Comparison Sorts

For insertion sort on 3 elements:

Each internal node is labeled by indices of array elements from their original positions. Each leaf is labeled by the permutation of orders that the algorithm determines.

How many leaves on the decision tree? There are \( \geq n! \) leaves, because every permutation appears at least once.

For any comparison sort,

- 1 tree for each \( n \).
- View the tree as if the algorithm splits in two at each node, based on the information it has determined up to that point.
- The tree models all possible execution traces.
Comparison Sorts

What is the length of the longest path from root to leaf?

- Depends on the algorithm
- Insertion sort: $\Theta(n^2)$
- Merge sort: $\Theta(n \log n)$

Lemma
Any binary tree of height $h$ has $\leq 2^h$ leaves.

In other words:

- $l =$ # of leaves,
- $h =$ height,
- Then $l \leq 2^h$.

(We’ll prove this lemma later.)
Using equation 3.19: \( lg(n!) = \Theta(n \lg n) \) we have
\[
  h \geq lg(n!)
\]
\[
  = \Theta(n \lg n)
\]
Therefore,
\[
h = \Omega(n \lg n)
\]
Now to prove the lemma:

**Proof** By induction on $h$.

**Basis:** $h = 0$. Tree is just one node, which is a leaf. $2^h = 1$.

**Inductive step:** Assume true for height $= h - 1$. Extend tree of height $h - 1$ by making as many new leaves as possible. Each leaf becomes parent to two new leaves.

\[
\text{# of leaves for height } h = 2 \cdot (\text{# of leaves for height } h - 1) \\
= 2 \cdot 2^{h-1} \\
= 2^h. \\
\text{(ind. hypothesis)} \\
\text{(lemma)}
\]

**Corollary**

Heapsort and merge sort are asymptotically optimal comparison sorts.
Linear Time sorting: Counting Sort

Note: Counting sort is **stable** (keys with same value appear in same order in output as they did in input) because of how the last loop works.

**COUNTING-SORT**\((A, B, n, k)\)

Let \(C[0..k]\) be a new array

\[
\text{for } i = 0 \text{ to } k \\
C[i] = 0
\]

\[
\text{for } j = 1 \text{ to } n \\
C[A[j]] = C[A[j]] + 1
\]

\[
\text{for } i = 1 \text{ to } k \\
C[i] = C[i] + C[i - 1]
\]

\[
\text{for } j = n \text{ down to } 1 \\
C[A[j]] = C[A[j]] - 1
\]
Analysis

$\Theta(n + k)$, which is $\Theta(n)$ if $k = O(n)$. How big a $k$ is practical?

- Good for sorting 32-bit values? No.
- 16-bit? Probably not.
- 8-bit? Maybe, depending on $n$.
- 4-bit? Probably (unless $n$ is really small).

Counting sort will be used in radix sort.
Radix Sort

Key idea: Sort least significant digits first.
To sort $d$ digits:

\[
\text{Radix-Sort}(A, d)
\]

\[
\text{for } i = 1 \text{ to } d \\
\text{use a stable sort to sort array } A \text{ on digit } i
\]

Example

<table>
<thead>
<tr>
<th>326</th>
<th>690</th>
<th>704</th>
<th>326</th>
</tr>
</thead>
<tbody>
<tr>
<td>453</td>
<td>751</td>
<td>608</td>
<td>435</td>
</tr>
<tr>
<td>608</td>
<td>453</td>
<td>326</td>
<td>453</td>
</tr>
<tr>
<td>835</td>
<td>704</td>
<td>835</td>
<td>608</td>
</tr>
<tr>
<td>751</td>
<td>835</td>
<td>435</td>
<td>690</td>
</tr>
<tr>
<td>435</td>
<td>435</td>
<td>751</td>
<td>704</td>
</tr>
<tr>
<td>704</td>
<td>326</td>
<td>453</td>
<td>751</td>
</tr>
<tr>
<td>690</td>
<td>608</td>
<td>690</td>
<td>835</td>
</tr>
</tbody>
</table>
Correctness

- Induction on number of passes ($i$ in pseudocode).
- Assume digits $1, 2, \ldots, i - 1$ are sorted.
- Show that a stable sort on digit $i$ leaves digits $1, \ldots, i$ sorted:
  - If 2 digits in position $i$ are different, ordering by position $i$ is correct, and positions $1, \ldots, i - 1$ are irrelevant.
  - If 2 digits in position $i$ are equal, numbers are already in the right order (by inductive hypothesis). The stable sort on digit $i$ leaves them in the right order.

This argument shows why it’s so important to use a stable sort for intermediate sort.
An Analysis
Assume that we use counting sort as the intermediate sort.

- $\Theta(n + k)$ per pass (digits in range $0, \ldots, k$)
- $d$ passes
- $\Theta(d(n + k))$ total
- If $k = O(n)$, time $= \Theta(dn)$.

How to break each key into digits?

- $n$ words.
- $b$ bits/word.
- Break into $r$-bit digits. Have $d = \lceil b / r \rceil$.
- Use counting sort, $k = 2^r - 1$.
  Example: 32-bit words, 8-bit digits. $b = 32$, $r = 8$, $d = \lceil 32 / 8 \rceil = 4$, $k = 2^8 - 1 = 255$.
- Time $= \Theta\left(\frac{b}{r} (n + 2^r)\right)$.

How to choose $r$? Balance $b/r$ and $n + 2^r$. Choosing $r \approx \lg n$ gives us $\Theta\left(\frac{b}{\lg n} (n + n)\right) = \Theta(bn / \lg n)$.

- If we choose $r < \lg n$, then $b/r > b/\lg n$, and $n + 2^r$ term doesn’t improve.
- If we choose $r > \lg n$, then $n + 2^r$ term gets big. Example: $r = 2\lg n \Rightarrow 2^r = 2^{2\lg n} = (2^{\lg n})^2 = n^2$.

So, to sort $2^{16}$ 32-bit numbers, use $r = \lg 2^{16} = 16$ bits. $\lceil b/r \rceil = 2$ passes.
Radix Sort

Compare radix sort to merge sort and quicksort:

- 1 million \((2^{20})\) 32-bit integers.
- Radix sort: \(\lceil 32/20 \rceil = 2\) passes.
- Merge sort/quick sort: \(\lg n = 20\) passes.
- Remember, though, that each radix sort “pass” is really 2 passes—one to take census, and one to move data.
Bucket Sort

Assumes the input is generated by a random process that distributes elements uniformly over \([0, 1]\).

**Idea**
- Divide \([0, 1]\) into \(n\) equal-sized buckets.
- Distribute the \(n\) input values into the buckets.
- Sort each bucket.
- Then go through buckets in order, listing elements in each one.

**Input:** \(A[1..n]\), where \(0 \leq A[i] < 1\) for all \(i\).

**Auxiliary array:** \(B[0..n-1]\) of linked lists, each list initially empty.

**BUCKET-SORT**(\(A\))
1. let \(B[0..n-1]\) be a new array
2. \(n = A.length\)
3. for \(i = 0\) to \(n-1\)
   4. make \(B[i]\) an empty list
5. for \(i = 1\) to \(n\)
   6. insert \(A[i]\) into list \(B[[nA[i]]]\)
7. for \(i = 0\) to \(n-1\)
   8. sort list \(B[i]\) with insertion sort
9. concatenate the lists \(B[0], B[1], \ldots, B[n-1]\) together in order
**Correctness**

Consider \( A[i] \), \( A[j] \). Assume without loss of generality that \( A[i] \leq A[j] \). Then \( \lfloor n \cdot A[i] \rfloor \leq \lfloor n \cdot A[j] \rfloor \). So \( A[i] \) is placed into the same bucket as \( A[j] \) or into a bucket with a lower index.

- If same bucket, insertion sort fixes up.
- If earlier bucket, concatenation of lists fixes up.
**Analysis**

- Relies on no bucket getting too many values.
- All lines of algorithm except insertion sorting take $\Theta(n)$ altogether.
- Intuitively, if each bucket gets a constant number of elements, it takes $O(1)$ time to sort each bucket $\Rightarrow O(n)$ sort time for all buckets.
- We “expect” each bucket to have few elements, since the average is 1 element per bucket.
- But we need to do a careful analysis.
**Analysis**

- Relies on no bucket getting too many values.
- All lines of algorithm except insertion sorting take $\Theta(n)$ altogether.
- Intuitively, if each bucket gets a constant number of elements, it takes $O(1)$ time to sort each bucket $\Rightarrow O(n)$ sort time for all buckets.
- We “expect” each bucket to have few elements, since the average is 1 element per bucket.
- But we need to do a careful analysis.

Define a random variable:

- $n_i = \text{the number of elements placed in bucket } B[i].$

Because insertion sort runs in quadratic time, bucket sort time is
Bucket Sort

\[ T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2). \]

Take expectations of both sides:

\[
E[T(n)] = E\left[ \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \right] \\
= \Theta(n) + \sum_{i=0}^{n-1} E\left[ O(n_i^2) \right] \quad \text{(linearity of expectation)} \\
= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \quad \text{(E[ax] = aE[x])}
\]

Claim

\[ E[n_i^2] = 2 - (1/n) \text{ for } i = 0, \ldots, n-1. \]

Note: We are not covering the formal proof of this claim in cs312
Bucket Sort

Therefore:

\[
E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n)
\]

\[
= \Theta(n) + O(n)
\]

\[
= \Theta(n)
\]

- Again, not a comparison sort. Used a function of key values to index into an array.
- This is a \textit{probabilistic analysis}—we used probability to analyze an algorithm whose running time depends on the distribution of inputs.
- Different from a \textit{randomized algorithm}, where we use randomization to impose a distribution.
- With bucket sort, if the input isn’t drawn from a uniform distribution on \([0, 1)\), all bets are off (performance-wise, but the algorithm is still correct).
Many applications require a dynamic set that supports only the *dictionary operations* **INSERT**, **SEARCH**, and **DELETE**. Example: a symbol table in a compiler.

A hash table is effective for implementing a dictionary.

- The expected time to search for an element in a hash table is $O(1)$, under some reasonable assumptions.
- Worst-case search time is $\Theta(n)$, however.

A hash table is a generalization of an ordinary array.

- With an ordinary array, we store the element whose key is $k$ in position $k$ of the array.
- Given a key $k$, we find the element whose key is $k$ by just looking in the $k$th position of the array. This is called **direct addressing**.
- Direct addressing is applicable when we can afford to allocate an array with one position for every possible key.
Hash Tables

We use a hash table when we do not want to (or cannot) allocate an array with one position per possible key.

- Use a hash table when the number of keys actually stored is small relative to the number of possible keys.
- A hash table is an array, but it typically uses a size proportional to the number of keys to be stored (rather than the number of possible keys).
- Given a key $k$, don’t just use $k$ as the index into the array.
- Instead, compute a function of $k$, and use that value to index into the array. We call this function a **hash function**.

Issues that we’ll explore in hash tables:

- How to compute hash functions. We’ll look at the multiplication and division methods.
- What to do when the hash function maps multiple keys to the same table entry. We’ll look at chaining and open addressing.
Direct Address Tables

Direct-address tables

Scenario
- Maintain a dynamic set.
- Each element has a key drawn from a universe $U = \{0, 1, \ldots, m - 1\}$ where $m$ isn’t too large.
- No two elements have the same key.

Represent by a direct-address table, or array, $T[0 \ldots m - 1]$:
- Each slot, or position, corresponds to a key in $U$.
- If there’s an element $x$ with key $k$, then $T[k]$ contains a pointer to $x$.
- Otherwise, $T[k]$ is empty, represented by NIL.
Dictionary operations are trivial and take $O(1)$ time each:

**DIRECT-ADDRESS-SEARCH**($T, k$)

return $T[k]$

**DIRECT-ADDRESS-INSERT**($T, x$)

$T[\text{key}[x]] = x$

**DIRECT-ADDRESS-DELETE**($T, x$)

$T[\text{key}[x]] = \text{NIL}$
Hash Tables

Hash tables

The problem with direct addressing is if the universe $U$ is large, storing a table of size $|U|$ may be impractical or impossible.

Often, the set $K$ of keys actually stored is small, compared to $U$, so that most of the space allocated for $T$ is wasted.

• When $K$ is much smaller than $U$, a hash table requires much less space than a direct-address table.
• Can reduce storage requirements to $\Theta(|K|)$.
• Can still get $O(1)$ search time, but in the average case, not the worst case.

Idea

Instead of storing an element with key $k$ in slot $k$, use a function $h$ and store the element in slot $h(k)$.

• We call $h$ a hash function.
• $h : U \rightarrow \{0, 1, \ldots, m - 1\}$, so that $h(k)$ is a legal slot number in $T$.
• We say that $k$ hashes to slot $h(k)$.

Collisions

When two or more keys hash to the same slot.

• Can happen when there are more possible keys than slots ($|U| > m$).
• For a given set $K$ of keys with $|K| \leq m$, may or may not happen. Definitely happens if $|K| > m$.
• Therefore, must be prepared to handle collisions in all cases.
• Use two methods: chaining and open addressing.
• Chaining is usually better than open addressing. We’ll examine both.
Hashing with Chaining

Collision resolution by chaining

Put all elements that hash to the same slot into a linked list.

[This figure shows singly linked lists. If we want to delete elements, it's better to use doubly linked lists.]

- Slot $j$ contains a pointer to the head of the list of all stored elements that hash to $j$ [or to the sentinel if using a circular, doubly linked list with a sentinel],
- If there are no such elements, slot $j$ contains NIL.
Hashing with Chaining

How to implement dictionary operations with chaining:

- **Insertion:**
  
  \[
  \text{CHAINED-HASH-INSERT}(T, x)
  \]
  
  insert \( x \) at the head of list \( T[h(key[x])] \)

  - Worst-case running time is \( O(1) \).
  - Assumes that the element being inserted isn’t already in the list.
  - It would take an additional search to check if it was already inserted.

- **Search:**

  \[
  \text{CHAINED-HASH-SEARCH}(T, k)
  \]
  
  search for an element with key \( k \) in list \( T[h(k)] \)

  Running time is proportional to the length of the list of elements in slot \( h(k) \).

- **Deletion:**

  \[
  \text{CHAINED-HASH-DELETE}(T, x)
  \]
  
  delete \( x \) from the list \( T[h(key[x])] \)

  - Given pointer \( x \) to the element to delete, so no search is needed to find this element.
  - Worst-case running time is \( O(1) \) time if the lists are doubly linked.
  - If the lists are singly linked, then deletion takes as long as searching, because we must find \( x \)’s predecessor in its list in order to correctly update \textit{next} pointers.
Hashing with Chaining

Analysis of hashing with chaining

Given a key, how long does it take to find an element with that key, or to determine that there is no element with that key?

- Analysis is in terms of the \textit{load factor} $\alpha = n/m$:
  - $n$ = \# of elements in the table.
  - $m$ = \# of slots in the table = \# of (possibly empty) linked lists.
  - Load factor is average number of elements per linked list.
  - Can have $\alpha < 1$, $\alpha = 1$, or $\alpha > 1$.

- Worst case is when all $n$ keys hash to the same slot $\Rightarrow$ get a single list of length $n$
  $\Rightarrow$ worst-case time to search is $\Theta(n)$, plus time to compute hash function.

- Average case depends on how well the hash function distributes the keys among the slots.
Hashing with Chaining

We focus on average-case performance of hashing with chaining.

- Assume *simple uniform hashing*: any given element is equally likely to hash into any of the \( m \) slots.
- For \( j = 0, 1, \ldots, m - 1 \), denote the length of list \( T[j] \) by \( n_j \). Then \( n = n_0 + n_1 + \cdots + n_{m-1} \).
- Average value of \( n_j \) is \( E[n_j] = \alpha = n/m \).
- Assume that we can compute the hash function in \( O(1) \) time, so that the time required to search for the element with key \( k \) depends on the length \( n_{h(k)} \) of the list \( T[h(k)] \).

We consider two cases:

- If the hash table contains no element with key \( k \), then the search is unsuccessful.
- If the hash table does contain an element with key \( k \), then the search is successful.

*[In the theorem statements that follow, we omit the assumptions that we're resolving collisions by chaining and that simple uniform hashing applies.]*
Unsuccessful search

Theorem
An unsuccessful search takes expected time $\Theta(1 + \alpha)$.

Proof  Simple uniform hashing $\Rightarrow$ any key not already in the table is equally likely to hash to any of the $m$ slots.

To search unsuccessfully for any key $k$, need to search to the end of the list $T[h(k)]$. This list has expected length $E[n_{h(k)}] = \alpha$. Therefore, the expected number of elements examined in an unsuccessful search is $\alpha$.

Adding in the time to compute the hash function, the total time required is $\Theta(1 + \alpha)$.  ■
Hashing with Chaining

Successful search
- The expected time for a successful search is also $\Theta(1 + \alpha)$.
- The circumstances are slightly different from an unsuccessful search.
- The probability that each list is searched is proportional to the number of elements it contains.

Theorem
A successful search takes expected time $\Theta(1 + \alpha)$.

Proof is not covered in CS312.

Interpretation
If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1)$, which means that searching takes constant time on average.

Since insertion takes $O(1)$ worst-case time and deletion takes $O(1)$ worst-case time when the lists are doubly linked, all dictionary operations take $O(1)$ time on average.
Hash Functions

Hash functions

We discuss some issues regarding hash-function design and present schemes for hash function creation.

What makes a good hash function?

• Ideally, the hash function satisfies the assumption of simple uniform hashing.
• In practice, it’s not possible to satisfy this assumption, since we don’t know in advance the probability distribution that keys are drawn from, and the keys may not be drawn independently.
• Often use heuristics, based on the domain of the keys, to create a hash function that performs well.
Hash Functions

Keys as natural numbers

• Hash functions assume that the keys are natural numbers.
• When they’re not, have to interpret them as natural numbers.
• *Example:* Interpret a character string as an integer expressed in some radix notation. Suppose the string is CLRS:

  • ASCII values: C = 67, L = 76, R = 82, S = 83.
  • There are 128 basic ASCII values.
  • So interpret CLRS as \((67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0) = 141,764,947\).
Hash Functions: Division Method

Division method

\[ h(k) = k \mod m. \]

**Example:** \( m = 20 \) and \( k = 91 \) \( \Rightarrow h(k) = 11. \)

**Advantage:** Fast, since requires just one division operation.

**Disadvantage:** Have to avoid certain values of \( m \):

- Powers of 2 are bad. If \( m = 2^p \) for integer \( p \), then \( h(k) \) is just the least significant \( p \) bits of \( k \).
- If \( k \) is a character string interpreted in radix \( 2^p \) (as in CLRS example), then \( m = 2^p - 1 \) is bad: permuting characters in a string does not change its hash value (Exercise 11.3-3).

**Good choice for \( m \):** A prime not too close to an exact power of 2.
Hash Functions: Multiplication Method

**Multiplication method**

1. Choose constant $A$ in the range $0 < A < 1$.
2. Multiply key $k$ by $A$.
3. Extract the fractional part of $kA$.
4. Multiply the fractional part by $m$.
5. Take the floor of the result.

Put another way, $h(k) = \lfloor m (kA \mod 1) \rfloor$, where $kA \mod 1 = kA - \lfloor kA \rfloor =$ fractional part of $kA$.

**Disadvantage:** Slower than division method.

**Advantage:** Value of $m$ is not critical.
(Relatively) easy implementation:

- Choose \( m = 2^p \) for some integer \( p \).
- Let the word size of the machine be \( w \) bits.
- Assume that \( k \) fits into a single word. (\( k \) takes \( w \) bits.)
- Let \( s \) be an integer in the range \( 0 < s < 2^w \). (\( s \) takes \( w \) bits.)
- Restrict \( A \) to be of the form \( s/2^w \).
Hash Functions: Multiplication Method

\[ m = 2^p \]
\[ 0 < s < 2^w \]
\[ A = \frac{s}{2^w} \]

- Multiply \( k \) by \( s \).
- Since we’re multiplying two \( w \)-bit words, the result is \( 2w \) bits, \( r_1 2^w + r_0 \), where \( r_1 \) is the high-order word of the product and \( r_0 \) is the low-order word.
- \( r_1 \) holds the integer part of \( kA \) (\( \lfloor kA \rfloor \)) and \( r_0 \) holds the fractional part of \( kA \) (\( kA \mod 1 = kA - \lfloor kA \rfloor \)). Think of the “binary point” (analog of decimal point, but for binary representation) as being between \( r_1 \) and \( r_0 \). Since we don’t care about the integer part of \( kA \), we can forget about \( r_1 \) and just use \( r_0 \).
- Since we want \( \lfloor m (kA \mod 1) \rfloor \), we could get that value by shifting \( r_0 \) to the left by \( p = \lg m \) bits and then taking the \( p \) bits that were shifted to the left of the binary point.
- We don’t need to shift. The \( p \) bits that would have been shifted to the left of the binary point are the \( p \) most significant bits of \( r_0 \). So we can just take these bits after having formed \( r_0 \) by multiplying \( k \) by \( s \).
Hash Functions: Multiplication Method

- **Example**: \( m = 8 \) (implies \( p = 3 \)), \( w = 5 \), \( k = 21 \). Must have \( 0 < s < 2^5 \); choose \( s = 13 \) \( \Rightarrow A = 13/32 \).

  - Using just the formula to compute \( h(k) \): \( kA = 21 \cdot 13/32 = 273/32 = 8\frac{17}{32} \)
    \( \Rightarrow k A \mod 1 = 17/32 \Rightarrow m (k A \mod 1) = 8 \cdot 17/32 = 17/4 = 4\frac{1}{4} \Rightarrow 
    \lfloor m (k A \mod 1) \rfloor = 4 \), so that \( h(k) = 4 \).

  - Using the implementation: \( ks = 21 \cdot 13 = 273 = 8 \cdot 2^5 + 17 \Rightarrow r_1 = 8, r_0 = 17 \). Written in \( w = 5 \) bits, \( r_0 = 10001 \). Take the \( p = 3 \) most significant bits of \( r_0 \), get 100 in binary, or 4 in decimal, so that \( h(k) = 4 \).

**How to choose \( A \):**

- The multiplication method works with any legal value of \( A \).
- But it works better with some values than with others, depending on the keys being hashed.
- Knuth suggests using \( A \approx (\sqrt{5} - 1)/2 \).
Hash Tables: Open Addressing

Open addressing

An alternative to chaining for handling collisions.

Idea

- Store all keys in the hash table itself.
- Each slot contains either a key or NIL.
- To search for key $k$:
  - Compute $h(k)$ and examine slot $h(k)$. Examining a slot is known as a **probe**.
  - If slot $h(k)$ contains key $k$, the search is successful. If this slot contains NIL, the search is unsuccessful.
  - There’s a third possibility: slot $h(k)$ contains a key that is not $k$. We compute the index of some other slot, based on $k$ and on which probe (count from 0: 0th, 1st, 2nd, etc.) we’re on.
  - Keep probing until we either find key $k$ (successful search) or we find a slot holding NIL (unsuccessful search).
Hash Tables: Open Addressing

- We need the sequence of slots probed to be a permutation of the slot numbers \( \langle 0, 1, \ldots, m - 1 \rangle \) (so that we examine all slots if we have to, and so that we don’t examine any slot more than once).
- Thus, the hash function is \( h : U \times \{0, 1, \ldots, m - 1\} \rightarrow \{0, 1, \ldots, m - 1\} \).
  
  \[
  \begin{array}{cccc}
  \text{probe number} & \text{slot number} \\
  h(k, 0) & h(k, 1) & \ldots & h(k, m - 1)
  \end{array}
  \]
- The requirement that the sequence of slots be a permutation of \( \langle 0, 1, \ldots, m - 1 \rangle \) is equivalent to requiring that the **probe sequence** \( \langle h(k, 0), h(k, 1), \ldots, h(k, m - 1) \rangle \) be a permutation of \( \langle 0, 1, \ldots, m - 1 \rangle \).
- To insert, act as though we’re searching, and insert at the first **NIL** slot we find.
Hash Tables: Open Addressing

**Pseudocode for searching**

\textbf{Hash-Search}(T, k)

1. \(i = 0\)
2. \textbf{repeat}
   1. \(j = h(k, i)\)
   2. \textbf{if} \(T[j] == k\) \textbf{then return} \(j\) \textbf{end if}
   3. \(i = i + 1\)
3. \textbf{until} \(T[j] == \text{NIL} \text{ or } i = m\)
4. \textbf{return} \text{NIL}

\textbf{Hash-Search} returns the index of a slot containing key \(k\), or \text{NIL} if the search is unsuccessful.
Hash Tables: Open Addressing

**Pseudocode for insertion**

\[ \text{Hash-Insert}(T, k) \]

\[
i = 0 \\
\text{repeat} \\
\quad j = h(k, i) \\
\quad \text{if } T[j] == \text{NIL} \\
\quad \quad T[j] = k \\
\quad \quad \text{return } j \\
\quad \text{else } i = i + 1 \\
\text{until } i == m \\
\text{error} "\text{hash table overflow}"\]

\text{Hash-Insert} returns the number of the slot that gets key } k, \text{ or it flags a "hash table overflow" error if there is no empty slot in which to put key } k.\]

// It is a number between 0 and m-1
Hash Tables: Open Addressing

Deletion
Cannot just put NIL into the slot containing the key we want to delete.

- Suppose we want to delete key $k$ in slot $j$.
- And suppose that sometime after inserting key $k$, we were inserting key $k'$, and during this insertion we had probed slot $j$ (which contained key $k$).
- And suppose we then deleted key $k$ by storing NIL into slot $j$.
- And then we search for key $k'$.
- During the search, we would probe slot $j$ before probing the slot into which key $k'$ was eventually stored.
- Thus, the search would be unsuccessful, even though key $k'$ is in the table.

Solution: Use a special value DELETED instead of NIL when marking a slot as empty during deletion.

- Search should treat DELETED as though the slot holds a key that does not match the one being searched for.
- Insertion should treat DELETED as though the slot were empty, so that it can be reused.

The disadvantage of using DELETED is that now search time is no longer dependent on the load factor $\alpha$. 
Hash Tables: Open Addressing

How to compute probe sequences

The ideal situation is uniform hashing: each key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \ldots, m - 1 \rangle$ as its probe sequence. (This generalizes simple uniform hashing for a hash function that produces a whole probe sequence rather than just a single number.)

It’s hard to implement true uniform hashing, so we approximate it with techniques that at least guarantee that the probe sequence is a permutation of $\langle 0, 1, \ldots, m - 1 \rangle$. None of these techniques can produce all $m!$ probe sequences. They will make use of auxiliary hash functions, which map $U \rightarrow \{0, 1, \ldots, m - 1\}$. 
Hash Tables: Open Addressing

Linear probing

Given auxiliary hash function $h'$, the probe sequence starts at slot $h'(k)$ and continues sequentially through the table, wrapping after slot $m - 1$ to slot 0.

Given key $k$ and probe number $i$ ($0 \leq i < m$), $h(k, i) = (h'(k) + i) \mod m$.

The initial probe determines the entire sequence $\Rightarrow$ only $m$ possible sequences.

Linear probing suffers from primary clustering: long runs of occupied sequences build up. And long runs tend to get longer, since an empty slot preceded by $i$ full slots gets filled next with probability $(i + 1)/m$. Result is that the average search and insertion times increase.
Quadratic probing
As in linear probing, the probe sequence starts at $h'(k)$. Unlike linear probing, it jumps around in the table according to a quadratic function of the probe number:

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m,$$

where $c_1, c_2 \neq 0$ are constants.

Must constrain $c_1$, $c_2$, and $m$ in order to ensure that we get a full permutation of \{0, 1, ..., m−1\}. (Problem 11-3 explores one way to implement quadratic probing.)

Can get secondary clustering: if two distinct keys have the same $h'$ value, then they have the same probe sequence.
Hash Tables: Open Addressing

Double hashing

Use two auxiliary hash functions, \( h_1 \) and \( h_2 \). \( h_1 \) gives the initial probe, and \( h_2 \) gives the remaining probes: \( h(k, i) = (h_1(k) + ih_2(k)) \mod m \).

Must have \( h_2(k) \) be relatively prime to \( m \) (no factors in common other than 1) in order to guarantee that the probe sequence is a full permutation of \( (0, 1, \ldots, m - 1) \).

- Could choose \( m \) to be a power of 2 and \( h_2 \) to always produce an odd number > 1.
- Could let \( m \) be prime and have \( 1 < h_2(k) < m \).

\( \Theta(m^2) \) different probe sequences, since each possible combination of \( h_1(k) \) and \( h_2(k) \) gives a different probe sequence.

Example: Hash table size 13 with \( h_1(k) = k \mod 13 \) and \( h_2(k) = 1 + (k \mod 11) \)
Consider \( k = 14 \). \( h_1(14) = 1 \) and \( h_2(14) = 4 \).
Therefore, we probe 1, 5, 9, ...
Analysis of open-address hashing

Assumptions

- Analysis is in terms of load factor $\alpha$. We will assume that the table never completely fills, so we always have $0 \leq n < m \Rightarrow 0 \leq \alpha < 1$.
- Assume uniform hashing.
- No deletion.
- In a successful search, each key is equally likely to be searched for.
**Theorem**
The expected number of probes in an unsuccessful search is at most $1/(1 - \alpha)$.

**Proof** Since the search is unsuccessful, every probe is to an occupied slot, except for the last probe, which is to an empty slot.

Define random variable $X = \#$ of probes made in an unsuccessful search.

Define events $A_i$, for $i = 1, 2, \ldots$, to be the event that there is an $i$th probe and that it's to an occupied slot.

$X \geq i$ if and only if probes $1, 2, \ldots, i - 1$ are made and are to occupied slots $\Rightarrow$  
$\Pr\{X \geq i\} = \Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$.

By Exercise C.2-5,

$\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2}\}$.
**Claim**

\[ \Pr\{A_{j} \mid A_{1} \cap A_{2} \cap \cdots \cap A_{j-1}\} = \frac{n-j+1}{m-j+1}. \] Boundary case: \( j = 1 \) \( \Rightarrow \) \( \Pr\{A_{1}\} = \frac{n}{m}. \)

**Proof** For the boundary case \( j = 1 \), there are \( n \) stored keys and \( m \) slots, so the probability that the first probe is to an occupied slot is \( n/m \).

Given that \( j-1 \) probes were made, all to occupied slots, the assumption of uniform hashing says that the probe sequence is a permutation of \( \{0, 1, \ldots, m-1\} \), which in turn implies that the next probe is to a slot that we have not yet probed. There are \( m-j+1 \) slots remaining, \( n-j+1 \) of which are occupied. Thus, the probability that the \( j \)th probe is to an occupied slot is \( \frac{n-j+1}{m-j+1} \). \( \blacksquare \) (claim)
Hash Tables: Open Addressing

Using this claim,

\[ \Pr\{X \geq i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \]

\[ \text{i - 1 factors} \]

\[ n < m \Rightarrow \frac{n-j}{m-j} \leq \frac{n}{m} \text{ for } j \geq 0, \text{ which implies} \]

\[ \Pr\{X \geq i\} \leq \left(\frac{n}{m}\right)^{i-1} \]

\[ = \alpha^{i-1}. \]

By equation (C.25),

\[ \mathbb{E}[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\} \]

\[ \leq \sum_{i=1}^{\infty} \alpha^{i-1} \]

\[ = \sum_{i=0}^{\infty} \alpha^{i} \]

\[ = \frac{1}{1 - \alpha} \]  

(equation (A.6)) .  

\[ \blacksquare \text{ (theorem)} \]
Hash Tables: Open Addressing

Interpretation
If $\alpha$ is constant, an unsuccessful search takes $O(1)$ time.
- If $\alpha = 0.5$, then an unsuccessful search takes an average of $1/(1 - 0.5) = 2$ probes.
- If $\alpha = 0.9$, takes an average of $1/(1 - 0.9) = 10$ probes.

Corollary
The expected number of probes to insert is at most $1/(1 - \alpha)$.

Proof Since there is no deletion, insertion uses the same probe sequence as an unsuccessful search.

Theorem
The expected number of probes in a successful search is at most $\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$.

Proof: not covered in CS312
Chapter 12 overview

Search trees

- Data structures that support many dynamic-set operations.
- Can be used as both a dictionary and as a priority queue.
- Basic operations take time proportional to the height of the tree.
  - For complete binary tree with $n$ nodes: worst case $\Theta(\lg n)$.
  - For linear chain of $n$ nodes: worst case $\Theta(n)$.
- Different types of search trees include binary search trees, red-black trees (covered in Chapter 13), and B-trees (covered in Chapter 18).
Binary search trees

Binary search trees are an important data structure for dynamic sets.

- Accomplish many dynamic-set operations in $O(h)$ time, where $h =$ height of tree.
- As in Section 10.4, we represent a binary tree by a linked data structure in which each node is an object.
- $T.root$ points to the root of tree $T$.
- Each node contains the attributes
  - $key$ (and possibly other satellite data).
  - $left$: points to left child.
  - $right$: points to right child.
  - $p$: points to parent. $T.root.p = \text{NIL}$.
- Stored keys must satisfy the \textit{binary-search-tree property}.
  - If $y$ is in left subtree of $x$, then $y.key \leq x.key$.
  - If $y$ is in right subtree of $x$, then $y.key \geq x.key$. 
Binary Search Trees

• Tree representation:
  – A linked data structure in which each node is an object

• Node representation:
  – Key field
  – Satellite data
  – Left: pointer to left child
  – Right: pointer to right child
  – p: pointer to parent (p[root[T]] = NIL)

• Satisfies the binary-search-tree property
Binary Search Tree Example

• Binary search tree property:
  – If \( y \) is in left subtree of \( x \), then \( \text{key}[y] \leq \text{key}[x] \)
  – If \( y \) is in right subtree of \( x \), then \( \text{key}[y] \geq \text{key}[x] \)
Traversing a Binary Search Tree

- **Inorder** tree walk:
  - Prints the keys of a binary tree in sorted order
  - Root is printed between the values of its left and right subtrees: left, root, right

- **Preorder** tree walk:
  - root printed first: root, left, right

- **Postorder** tree walk: left, right, root
  - root printed last

![Binary Search Tree Diagram]

Inorder: 2 3 5 5 7 9
Preorder: 5 3 2 5 7 9
Postorder: 2 5 3 9 7 5
Traversing a Binary Search Tree

**Alg:** INORDER-TREE-WALK(x)

1. if x ≠ NIL
2. then INORDER-TREE-WALK (left [x])
3. print key [x]
4. INORDER-TREE-WALK (right [x])

- *E.g.:*

```
      5
     / \
    3   7
   / \ /\
  2  5 9
```

Output: 2 3 5 5 7 9

- Running time:
  - $\Theta(n)$, where $n$ is the size of the tree rooted at $x$
Searching for a Key

• Given a pointer to the root of a tree and a key $k$:
  – Return a pointer to a node with key $k$
    if one exists
  – Otherwise return NIL

• Idea
  – Starting at the root: trace down a path by comparing $k$ with the key of the current node:
    • If the keys are equal: we have found the key
    • If $k < \text{key}[x]$ search in the left subtree of $x$
    • If $k > \text{key}[x]$ search in the right subtree of $x$
Searching for a Key

Alg: TREE-SEARCH(\(x, k\))

1. if \(x = \text{NIL}\) or \(k = \text{key}[x]\)
2. then return \(x\)
3. if \(k < \text{key}[x]\)
4. then return TREE-SEARCH(\text{left}[x], k\)
5. else return TREE-SEARCH(\text{right}[x], k\)

Running Time: \(O(h)\),
\(h\) – the height of the tree
Example: TREE-SEARCH

- Search for key 13:
  - 15 → 6 → 7 → 13
Iterative Tree Search

**Alg:** ITERATIVE-TREE-SEARCH(x, k)

1. while \( x \neq \text{NIL} \) and \( k \neq \text{key}[x] \)
2. do if \( k < \text{key}[x] \)
3. then \( x \leftarrow \text{left}[x] \)
4. else \( x \leftarrow \text{right}[x] \)
5. return \( x \)
Finding the Minimum in a Binary Search Tree

- Goal: find the minimum value in a BST
  - Following left child pointers from the root, until a NIL is encountered

**Alg:** TREE-MINIMUM(x)

1. while left [x] ≠ NIL
2. do x ← left [x]
3. return x

Running time: \(O(h), h - \) height of tree
Finding the Maximum in a Binary Search Tree

• Goal: find the maximum value in a BST
  – Following right child pointers from the root, until a NIL is encountered

Alg: TREE-MAXIMUM(x)
1. while right[x] ≠ NIL
2. do x ← right[x]
3. return x

• Running time: $O(h)$, $h$ – height of tree

Maximum = 20
**Def:** successor($x$) = $y$, such that \( \text{key}[y] \) is the smallest key > \( \text{key}[x] \)

- **E.g.:** successor(15) = 17  
  successor(13) = 15  
  successor(9) = 13

- **Case 1:** right($x$) is non empty  
  - successor($x$) = the minimum in right($x$)

- **Case 2:** right($x$) is empty  
  - go up the tree until the current node is a left child: successor($x$) is the parent of the current node  
  - if you cannot go further (and you reached the root): $x$ is the largest element
Finding the Successor

**Alg:** TREE-SUCCESSOR\( (x) \)

1. if right\( [x] \) \( \neq \) NIL
2. then return TREE-MINIMUM(right\( [x] \))
3. \( y \leftarrow p[x] \)
4. while \( y \neq \) NIL and \( x = \) right\( [y] \)
5. do \( x \leftarrow y \)
6. \( y \leftarrow p[y] \)
7. return \( y \)

Running time: \( O (h) \), \( h \) – height of the tree
**Def**: \( \text{predecessor}(x) = y \), such that \( \text{key}[y] \) is the biggest \( \text{key} < \text{key}[x] \)

- **E.g.**: \( \text{predecessor}(15) = 13 \)
  - \( \text{predecessor}(9) = 7 \)
  - \( \text{predecessor}(13) = 9 \)

- **Case 1**: \( \text{left}(x) \) is non empty
  - \( \text{predecessor}(x) = \) the maximum in \( \text{left}(x) \)

- **Case 2**: \( \text{left}(x) \) is empty
  - go up the tree until the current node is a right child:
    - \( \text{predecessor}(x) \) is the parent of the current node
  - if you cannot go further (and you reached the root):
    - \( x \) is the smallest element
Binary Search Trees

*Time*

For both the `TREE-SUCCESSOR` and `TREE-PREDECESSOR` procedures, in both cases, we visit nodes on a path down the tree or up the tree. Thus, running time is \( O(h) \), where \( h \) is the height of the tree.
Insertion

- **Goal:**
  - Insert value $v$ into a binary search tree

- **Idea:**
  - If $\text{key}[x] < v$ move to the right child of $x$,
  - else move to the left child of $x$
  - When $x$ is NIL, we found the correct position
  - If $v < \text{key}[y]$ insert the new node as $y$’s left child
    - else insert it as $y$’s right child
  - Beginning at the root, go down the tree and maintain:
    - Pointer $x$: traces the downward path (current node)
    - Pointer $y$: parent of $x$ (“trailing pointer”)

Insert value 13

![Binary search tree diagram](image-url)
Example: TREE-INSERT

Insert 13:

```
x, y=NIL
```

```
x = NIL
y = 15
```
**Alg:** TREE-INSERT($T, z$)

1. $y \leftarrow \text{NIL}$
2. $x \leftarrow \text{root}[T]$
3. while $x \neq \text{NIL}$
4. \hspace{1em} do $y \leftarrow x$
5. \hspace{2em} if key[z] < key[x]
6. \hspace{3em} then $x \leftarrow \text{left}[x]$
7. \hspace{2em} else $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. if $y = \text{NIL}$
10. \hspace{1em} then $\text{root}[T] \leftarrow z$
11. \hspace{1em} else if key[z] < key[y]
12. \hspace{2em} then $\text{left}[y] \leftarrow z$
13. \hspace{2em} else $\text{right}[y] \leftarrow z$

Tree $T$ was empty

Running time: $O(h)$
Binary Search Trees

- To insert value $v$ into the binary search tree, the procedure is given node $z$, with $z.key = v$, $z.left = \text{NIL}$, and $z.right = \text{NIL}$.
- Beginning at root of the tree, trace a downward path, maintaining two pointers.
  - Pointer $x$: traces the downward path.
  - Pointer $y$: “trailing pointer” to keep track of parent of $x$.
- Traverse the tree downward by comparing the value of node at $x$ with $v$, and move to the left or right child accordingly.
- When $x$ is $\text{NIL}$, it is at the correct position for node $z$.
- Compare $z$’s value with $y$’s value, and insert $z$ at either $y$’s left or right, appropriately.

Example
Run TREE-INSERT($T$, $C$) on the first sample binary search tree. Result:

![Binary Search Tree Diagram]

Time
Same as TREE-SEARCH. On a tree of height $h$, procedure takes $O(h)$ time.
Deletion

Conceptually, deleting node $z$ from binary search tree $T$ has three cases:

1. If $z$ has no children, just remove it.
2. If $z$ has just one child, then make that child take $z$’s position in the tree, dragging the child’s subtree along.
3. If $z$ has two children, then find $z$’s successor $y$ and replace $z$ by $y$ in the tree. $y$ must be in $z$’s right subtree and have no left child. The rest of $z$’s original right subtree becomes $y$’s new right subtree, and $z$’s left subtree becomes $y$’s new left subtree.

This case is a little tricky because the exact sequence of steps taken depends on whether $y$ is $z$’s right child.

The code organizes the cases a bit differently. Since it will move subtrees around within the binary search tree, it uses a subroutine, TRANSPLANT, to replace one subtree as the child of its parent by another subtree.
**TRANSPLANT**(\(T, u, v\))

\[
\text{if } u.p == \text{NIL} \\
T.root = v \\
\text{elseif } u == u.p.left \\
u.p.left = v \\
\text{else } u.p.right = v \\
\text{if } v \neq \text{NIL} \\
v.p = u.p
\]

**TRANSPLANT**\((T, u, v)\) replaces the subtree rooted at \(u\) by the subtree rooted at \(v\):

- Makes \(u\)’s parent become \(v\)’s parent (unless \(u\) is the root, in which case it makes \(v\) the root).
- \(u\)’s parent gets \(v\) as either its left or right child, depending on whether \(u\) was a left or right child.
- Doesn’t update \(v\).\(left\) or \(v\).\(right\), leaving that up to **TRANSPLANT**’s caller.
Tree-Delete($T, z$) has four cases when deleting node $z$ from binary search tree $T$:

- If $z$ has no left child, replace $z$ by its right child. The right child may or may not be NIL. (If $z$'s right child is NIL, then this case handles the situation in which $z$ has no children.)

- If $z$ has just one child, and that child is its left child, then replace $z$ by its left child.
Otherwise $z$ has two children. Find $z$’s successor $y$. $y$ must lie in $z$’s right subtree and have no left child (Exercise 12.2-5).

Goal is to replace $z$ by $y$, splicing $y$ out of its current location.

- If $y$ is $z$’s right child, replace $z$ by $y$ and leave $y$’s right child alone.

- Otherwise, $y$ lies within $z$’s right subtree but is not the root of this subtree. Replace $y$ by its own right child. Then replace $z$ by $y$. 
**Binary Search Trees**

TREE-DELETE \((T, z)\)

- If \(z.left == NIL\)
  - TRANSPLANT \((T, z, z.right)\)  // \(z\) has no left child

- Elseif \(z.right == NIL\)
  - TRANSPLANT \((T, z, z.left)\)  // \(z\) has just a left child

- Else  // \(z\) has two children.
  - \(y = \text{TREE-MINIMUM}(z.right)\)  // \(y\) is \(z\)'s successor
  - If \(y.p \neq z\)
    - \(y\) lies within \(z\)'s right subtree but is not the root of this subtree.
      - TRANSPLANT \((T, y, y.right)\)
      - \(y.right = z.right\)
      - \(y.right.p = y\)
    - // Replace \(z\) by \(y\).
      - TRANSPLANT \((T, z, y)\)
      - \(y.left = z.left\)
      - \(y.left.p = y\)

Note that the last three lines execute when \(z\) has two children, regardless of whether \(y\) is \(z\)'s right child.
Example
On this binary search tree $T$,

run the following. [You can either start with the original tree each time or start with the result of the previous call. The tree is designed so that either way will elicit all four cases.]

- **Tree-Delete**$(T, I)$ shows the case in which the node deleted has no left child.
- **Tree-Delete**$(T, G)$ shows the case in which the node deleted has a left child but no right child.
- **Tree-Delete**$(T, K)$ shows the case in which the node deleted has both children and its successor is its right child.
- **Tree-Delete**$(T, B)$ shows the case in which the node deleted has both children and its successor is not its right child.
**Binary Search Trees**

---

**Time**

\(O(h)\), on a tree of height \(h\). Everything is \(O(1)\) except for the call to `TREE-MINIMUM`.

---

**Minimizing running time**

We’ve been analyzing running time in terms of \(h\) (the height of the binary search tree), instead of \(n\) (the number of nodes in the tree).

- Problem: Worst case for binary search tree is \(\Theta(n)\)—no better than linked list.
- Solution: Guarantee small height (balanced tree)—\(h = O(\lg n)\).

In later chapters, by varying the properties of binary search trees, we will be able to analyze running time in terms of \(n\).

- Method: Restructure the tree if necessary. Nothing special is required for querying, but there may be extra work when changing the structure of the tree (inserting or deleting).