Divide-and-Conquer Algorithms
Part Two
Recap from Last Time
Divide-and-Conquer Algorithms

• A **divide-and-conquer** algorithm is one that works as follows:
  • *(Divide)* Split the input apart into multiple smaller pieces, then recursively invoke the algorithm on those pieces.
  • *(Conquer)* Combine those solutions back together to form the overall answer.
• Can be analyzed using **recurrence relations**.
Two Important Recurrences

\[
T(0) = \Theta(1) \\
T(1) = \Theta(1) \\
T(n) = T(\lfloor n / 2 \rfloor) + T(\lceil n / 2 \rceil) + \Theta(n)
\]

Solves to \( O(n \log n) \)

\[
T(0) = \Theta(1) \\
T(1) = \Theta(1) \\
T(n) \leq T(\lfloor n / 2 \rfloor) + \Theta(1)
\]

Solves to \( O(\log n) \)
Outline for Today

- **More Recurrences**
  - Other divide-and-conquer relations.

- **Algorithmic Lower Bounds**
  - Showing that certain problems cannot be solved within certain limits.

- **Binary Heaps**
  - A fast data structure for retrieving elements in sorted order.
Another Algorithm: 
Maximizing Unimodal Arrays
An array is called **unimodal** iff it can be split into an increasing sequence followed by a decreasing sequence.
procedure unimodalMax(list A, int low, int high):
    if low = high - 1:
        return A[low]

    let mid = ⌊(high + low) / 2⌋
        return unimodalMax(A, mid + 1, high)
    else:
        return unimodalMax(A, low, mid + 1)

T(1) = Θ(1)
T(n) ≤ T(⌈n / 2⌉) + Θ(1)

O(log n)
A weakly unimodal array is one that can be split into a nondecreasing sequence followed by a nonincreasing sequence.
procedure weakUnimodalMax(list A, int low, int high):
    if low = high - 1:
        return A[low]

    let mid = ⌊(high + low) / 2⌋
        return weakUnimodalMax(A, mid + 1, high)
        return weakUnimodalMax(A, low, mid + 1)
    else
        return max(weakUnimodalMax(A, low, mid + 1)
                   weakUnimodalMax(A, mid + 1, high))

T(1) = Θ(1)
T(n) ≤ T(⌈n / 2⌉) + T(⌊n / 2⌋) + Θ(1)
procedure weakUnimodalMax(list A, int low, int high):
  if low = high - 1:
    return A[low]

  let mid = \lfloor (high + low) / 2 \rfloor
    return weakUnimodalMax(A, mid + 1, high)
    return weakUnimodalMax(A, low, mid + 1)
  else
    return max(weakUnimodalMax(A, low, mid + 1),
                weakUnimodalMax(A, mid + 1, high))

T(1) ≤ c
T(n) ≤ 2T(n / 2) + c
\[ T(1) \leq c \]

\[ T(n) \leq 2T(n/2) + c \]

\[
T(n) \leq 2T\left(\frac{n}{2}\right) + c \\
\leq 2 \left( 2T\left(\frac{n}{4}\right) + c \right) + c \\
\leq 4T\left(\frac{n}{4}\right) + 2c + c \\
= 4T\left(\frac{n}{4}\right) + 3c \\
\leq 4 \left( 2T\left(\frac{n}{8}\right) + c \right) + 3c \\
= 8T\left(\frac{n}{8}\right) + 4c + 3c \\
= 8T\left(\frac{n}{8}\right) + 7c \\
\ldots \\
\leq 2^k T\left(\frac{n}{2^k}\right) + (2^k - 1)c \]
\[ T(1) \leq c \]
\[ T(n) \leq 2T(n/2) + c \]

\[
T(n) \leq 2^k T\left(\frac{n}{2^k}\right) + (2^k - 1)c
\]

\[
\leq 2^{\log_2 n} T(1) + (2^{\log_2 n} - 1)c
\]

\[
= n T(1) + c(n-1)
\]

\[
\leq c n + c(n-1)
\]

\[
= 2cn - c
\]

\[ = O(n) \]
\[ T(1) \leq c \]

\[ T(n) \leq 2T(n/2) + c \]

\[ 2cn - c \]
Another Recurrence Relation

- The recurrence relation solves to $T(n) = O(n)$

\[
\begin{align*}
T(1) &= \Theta(1) \\
T(n) &\leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(1)
\end{align*}
\]

solves to $T(n) = O(n)$

- Intuitively, the recursion tree is “bottomheavy:” the bottom of the tree accounts for almost all of the work.
Unimodal Arrays

- Our recurrence shows that the work done is $O(n)$, but this might not be a tight bound.
- Does our algorithm ever do $\Omega(n)$ work?
- **Yes:** What happens if all array values are equal to one another?
- Can we do better?
A Lower Bound

- **Claim**: Every correct algorithm for finding the maximum value in a unimodal array must do $\Omega(n)$ work in the worst-case.

- Note that this claim is over *all possible algorithms*, so the argument had better be watertight!
A Lower Bound

• We will prove that any algorithm for finding the maximum value of a unimodal array must, on at least one input, inspect all \( n \) locations.

• *Proof idea*: Suppose that the algorithm didn't do this.
A Lower Bound

• We will prove that any algorithm for finding the maximum value of a unimodal array must, on at least one input, inspect all \( n \) locations.

• Proof idea: Suppose that the algorithm didn't do this.
A Lower Bound

• We will prove that any algorithm for finding the maximum value of a unimodal array must, on at least one input, inspect all $n$ locations.

• *Proof idea:* Suppose that the algorithm didn't do this.
Algorithmic Lower Bounds

- The argument we just saw is called an adversarial argument and is often used to establish algorithmic lower bounds.

- Idea: Show that if an algorithm doesn't do enough work, then it cannot distinguish two different inputs that require different outputs.

- Therefore, the algorithm cannot always be correct.
o Notation

- Let \( f, g : \mathbb{N} \rightarrow \mathbb{N} \).
- We say that \( f(n) = o(g(n)) \) (\( f \) is little-\( o \) of \( g \)) iff
  \[
  \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0
  \]
- In other words, \( f \) grows strictly slower than \( g \).
- Often used to describe impossibility results.
- For example: There is no \( o(n) \)-time algorithm for finding the maximum element of a weakly unimodal array.
What Does This Mean?

• In the worst-case, our algorithm must do $\Omega(n)$ work.
• That's the same as a linear scan over the input array!
• Is our algorithm even worth it?
• **Yes**: In most cases, the runtime is $\Theta(\log n)$ or close to it.
Binary Heaps
Data Structures Matter

- We have seen two instances where a better choice of data structure improved the runtime of an algorithm:
  - Using adjacency lists instead of adjacency matrices in graph algorithms.
  - Using a double-ended queue in 0/1 Dijkstra's algorithm.
- Today, we'll explore a data structure that is useful for improving algorithmic efficiency.
- We'll come back to this structure in a few weeks when talking about Prim's algorithm and Kruskal's algorithm.
Priority Queues

- A **priority queue** is a data structure for storing elements associated with *priorities* (often called *keys*).
- Optimized to find the element that currently has the smallest key.
- Supports the following operations:
  - **enqueue** \((k, v)\) which adds element \(v\) to the queue with key \(k\).
  - **is-empty**, which returns whether the queue is empty.
  - **dequeue-min**, which removes the element with the least priority from the queue.
- Many implementations are possible with varying tradeoffs.
A Naive Implementation

- One simple way to implement a priority queue is with an unsorted array key/value pairs.
- To enqueue $v$ with key $k$, append $(k, v)$ to the array in time $O(1)$.
- To check whether the priority queue is empty, check whether the underlying array is empty in time $O(1)$.
- To dequeue-min, scan across the array to find an element with minimum key, then remove it in time $O(n)$.
- Doing $n$ enqueues and $n$ dequeues takes time $O(n^2)$. 
A Better Implementation

This tree obeys the **heap property**: each node's key is less than or equal to all its descendants' keys.
A Better Implementation

This is a complete binary tree: every level except the last one is filled in completely.
A Better Implementation
A Better Implementation
A Better Implementation

```
    0
   / \
  1   2
 / \ / \   
2  5 9  8
/   /   /   
4   3   9   8
```
A Better Implementation

1

2

4

3

5

9

8

2
A Better Implementation
A Better Implementation
Binary Heap Efficiency

- The enqueue and dequeue operations on a binary heap all run $O(h)$, where $h$ is the height of the tree.
- In a perfect binary tree of height $h$, there are $1 + 2 + 4 + 8 + \ldots + 2^h = 2^{h+1} - 1$ nodes.
- If there are $n$ nodes, the maximum height would be found by setting $n = 2^{h+1} - 1$.
- Solving, we get that $h = \log_2 (n + 1) - 1$
- Thus $h = \Theta(\log n)$, so enqueue and dequeue take time $O(\log n)$. 
Implementing Binary Heaps

- It is extremely rare to actually implement a binary heap as a tree structure.
- Can encode the heap as an array:

\[
\begin{array}{cccccccc}
1 & 3 & 8 & 4 & 5 & 9 & & \\
\end{array}
\]

Assuming one-indexing:
- Node \( n \) has children at positions \( 2n \) and \( 2n + 1 \).
- Node \( n \) has its parent at position \( \lfloor n / 2 \rfloor \).
Application: **Heapsort**
Heapsort

- The **heapsort** algorithm is as follows:
  - Build a max-heap from the array elements, using the array itself to represent the heap.
  - Repeatedly dequeue from the heap until all elements are placed in sorted order.
- This algorithm runs in time $O(n \log n)$, since it does $n$ enqueues and $n$ dequeues.
- Only requires $O(1)$ auxiliary storage space, compared with $O(n)$ space required in mergesort.
An Optimization: **Heapify**
Making a Binary Heap

• Suppose that you have \( n \) elements and want to build a binary heap from them.

• One way to do this is to enqueue all of them, one after another, into the binary heap.

• We can upper-bound the runtime as \( n \) calls to an \( O(\log n) \) operation, giving a total runtime of \( O(n \log n) \).

• Is that a tight bound?
Making a Binary Heap

Total Runtime: $\Theta(n \log n)$
Quickly Making a Binary Heap

- Here is a slightly different algorithm for building a binary heap out of a set of data:
  - Put the nodes, in any order, into a complete binary tree of the right size. (Shape property holds, but heap property might not.)
  - For each node, starting at the bottom layer and going upward, run a bubble-down step on that node.
Analyzing the Runtime

- At most half of the elements start one layer above that and can move down at most once.
- At most a quarter of the elements start one layer above that and can move down at most twice.
- At most an eighth of the elements start two layers above that and can move down at most thrice.
- More generally: At most $n / 2^k$ of the elements can move down $k$ steps.
- Can upper-bound the runtime with the sum

$$T(n) \leq \sum_{i=0}^{\lfloor \log_2 n \rfloor} \frac{ni}{2^i} = n \sum_{i=0}^{\lfloor \log_2 n \rfloor} \frac{i}{2^i}$$
Simplifying the Summation

• We want to simplify the sum

\[ \sum_{i=0}^{\lceil \log_2 n \rceil} \frac{i}{2^i} \]

• Let's introduce a new variable \( x \), then evaluate the sum when \( x = \frac{1}{2} \):

\[ \sum_{i=0}^{\lfloor \log_2 n \rfloor} i x^i \]

• If \( x < 1 \), each term is less than the previous, so

\[ \sum_{i=0}^{\lfloor \log_2 n \rfloor} i x^i < \sum_{i=0}^{\infty} i x^i \]
Solving the Summation

\[
\sum_{i=0}^{\infty} i x^i = x \sum_{i=0}^{\infty} i x^{i-1}
\]

\[
= x \sum_{i=0}^{\infty} \frac{d}{dx} x^i
\]

\[
= x \frac{d}{dx} \left( \sum_{i=0}^{\infty} x^i \right)
\]

\[
= x \frac{d}{dx} \left( \frac{1}{1-x} \right)
\]

\[
= x \frac{1}{(1-x)^2}
\]

\[
= \frac{x}{(1-x)^2}
\]
The Finishing Touches

• We know that

\[ T(n) \leq n \sum_{i=0}^{\lfloor \log_2 n \rfloor} i x^i < n \sum_{i=0}^{\infty} i x^i = \frac{nx}{(1-x)^2} \]

• Evaluating at \( x = \frac{1}{2} \), we get

\[ T(n) \leq \frac{n(1/2)}{(1-(1/2))^2} = \frac{n(1/2)}{(1/2)^2} = 2n \]

• So at most \( 2n \) swaps are performed!

• We visit each node once and do at most \( O(n) \) swaps, so the runtime is \( \Theta(n) \).