Divide-and-Conquer Algorithms
Part One
Announcements

• Problem Set One completely due right now. Solutions distributed at the end of lecture.

• Programming section today in Gates B08 from 3:45PM – 5:00PM.
  • Resumes at normal Thursday schedule (4:15PM – 5:05PM) next week.
Where We've Been

- We have just finished discussing fundamental algorithms on graphs.
- These algorithms are indispensable and show up everywhere.
- You can now solve a large class of problems by recognizing that they reduce to a problem you already know how to solve.
Where We're Going

- We are about to explore the **divide-and-conquer** paradigm, which gives a useful framework for thinking about problems.

- We will explore several major techniques:
  - Solving problems recursively.
  - Intuitively understanding how the structure of recursive algorithms influences runtime.
  - Recognizing when a problem can be solved by reducing it to a simpler case.
Outline for Today

- **Recurrence Relations**
  - Representing an algorithm's runtime in terms of a simple recurrence.

- **Solving Recurrences**
  - Determining the runtime of a recursive function from a recurrence relation.

- **Sampler of Divide-and-Conquer**
  - A few illustrative problems.
Insertion Sort

- As we saw in Lecture 00, insertion sort can be used to sort an array in time $\Omega(n)$ and $O(n^2)$.
  - It's $\Theta(n^2)$ in the average case.
- Can we do better?
A Better Sorting Algorithm: Mergesort
Thinking About $O(n^2)$

$T(n)$

$T(\frac{1}{2}n) \approx \frac{1}{4}T(n)$
procedure merge(list A, list B):
    let result be an empty list.
    while both A and B are nonempty:
        if head(A) < head(B):
            append head(A) to result
            remove head(A) from A
        else:
            append head(B) to result
            remove head(B) from B
    append all elements remaining in A to result
    append all elements remaining in B to result
    return result

Complexity: $\Theta(m + n)$, where $m$ and $n$ are the lengths of the input lists.
Motivating Mergesort

- Splitting the input array in half, sorting each half, and merging them back together will take roughly half as long as sorting the original array.
- So why not split the array into fourths? Or eighths?
- Question: What happens if we *never stop splitting*?
High-Level Idea

- A recursive sorting algorithm!
- **Base Case:**
  - An empty or single-element list is already sorted.
- **Recursive step:**
  - Break the list in half and recursively sort each part.
  - Merge the sorted halves back together.
- This algorithm is called *mergesort*.
procedure mergesort(list A):
    if length(A) ≤ 1:
        return A

    let left be the first half of the elements of A
    let right be the second half of the elements of A

    return merge(mergesort(left), mergesort(right))

T(0) = \Theta(1)
T(1) = \Theta(1)
T(n) = T(\lfloor n / 2 \rfloor) + T(\lfloor n / 2 \rfloor) + \Theta(n)
Recurrence Relations

- A **recurrence relation** is a function or sequence whose values are defined in terms of earlier values.
- In our case, we get this recurrence for the runtime of mergesort:

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

- We can **solve** a recurrence by finding an explicit expression for its terms, or by finding an asymptotic bound on its growth rate.
- How do we solve this recurrence?
Simplifying our Recurrence

- It is often difficult to solve recurrences involving floors and ceilings, as ours does.

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

- Note that if we only consider \( n = 1, 2, 4, 8, 16, \ldots \), then the floors and ceilings are always equivalent to standard division.

- **Simplifying Assumption 1:** We will only consider the recurrence as applied to powers of two.

- We need to justify why this is safe, which we'll do later.
Simplifying our Recurrence

- Without knowing the actual functions hidden by the $\Theta$ notation, we cannot get an exact value for the terms in this recurrence.

\[
\begin{align*}
T(1) &= c_1 \\
T(n) &= 2T(n/2) + c_2n
\end{align*}
\]

- If the $\Theta(1)$ just hides a constant and $\Theta(n)$ just hides a multiple of $n$, this would be a lot easier to manipulate!

- **Simplifying Assumption 2:** We will pretend that $\Theta(1)$ hides some constant and $\Theta(n)$ hides a multiple of $n$.

- We need to justify why this is safe, which we'll do later.
Simplifying our Recurrence

- Working with two constants $c_1$ and $c_2$ is most accurate, but it makes the math a lot harder.

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

- If all we care about is getting an asymptotic bound, these constants are unlikely to make a noticeable difference.

- **Simplifying Assumption 3**: Set $c = \max\{c_1, c_2\}$ and replace the equality with an upper bound.

- This is less exact, but is easier to manipulate.
The Final Recurrence

• Here is the final version of the recurrence we'll be working with:

\[
\begin{align*}
T(1) & \leq c \\
T(n) & \leq 2T(n/2) + cn
\end{align*}
\]

• As before, we will justify why all of these simplifications are safe later on.

• The analysis we're about to do (without justifying the simplifications) is at the level we will expect for most of our discussion of divide-and-conquer algorithms.
Getting an Intuition

- Simple recurrence relations often give rise to surprising results.
- It is often useful to build up an intuition for what the recursion solves to before trying to formally prove it.
- We will explore two methods for doing this:
  - The *iteration method*.
  - The *recursion-tree method*. 

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + cn \]

\[ \leq 2\left(2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn \]

\[ = 4T\left(\frac{n}{4}\right) + cn + cn \]

\[ = 4T\left(\frac{n}{4}\right) + 2cn \]

\[ \leq 4\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + 2cn \]

\[ = 8T\left(\frac{n}{8}\right) + cn + 2cn \]

\[ = 8T\left(\frac{n}{8}\right) + 3cn \]

\[ \cdots \]

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

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

\[
= 2^{\log_2 n} T(1) + cn \log_2 n
\]

\[
= nT(1) + cn \log_2 n
\]

\[
\leq cn + cn \log_2 n
\]

\[
= O(n \log n)
\]
The Iteration Method

• What we just saw is an example of the iteration method.

• Keep plugging the recurrence into itself until you spot a pattern, then try to simplify.

• Doesn't always give an exact answer, but useful for building up an intuition.
\[ T(1) \leq c \]
\[ T(n) \leq 2T(n/2) + cn \]

\[ cn \log_2 n + cn \]
The Recursion Tree Method

- This diagram is called a recursion tree and accounts for how much total work each recursive call makes.
- Often useful to sum up the work across the layers of the tree.
A Formal Proof

• Both the iteration and recursion tree methods suggest that the runtime is at most

\[ cn \log_2 n + cn \]

• Neither of these lines of reasoning are perfectly rigorous; how could we formalize this?

• **Induction!**
**Theorem:** If $n$ is a power of 2, $T(n) \leq cn \log_2 n + cn$

**Proof:** By induction. As a base case, if $n = 2^0 = 1$, then

$$T(n) = T(1) \leq c = cn \log_2 n + cn.$$ 

For the inductive step, assume the claim holds for all $n' < n$ that are powers of two. Then

$$T(n) \leq 2T(n/2) + cn$$
$$= 2((cn/2) \log_2 (n/2) + cn/2) + cn$$
$$= cn \log_2 (n/2) + cn + cn$$
$$= cn (\log_2 n - 1) + cn + cn$$
$$= cn \log_2 n - cn + cn + cn$$
$$= cn \log_2 n + cn \blacksquare$$
What This Means

• We have shown that as long as we only look at powers of two, the runtime for mergesort is bounded from above by $cn \log_2 n + cn$.

  In most cases, it's perfectly safe to stop here and claim we have a working bound. Mergesort is indeed $O(n \log n)$.

• For completeness, let's take some time to see why it is safe to stop here.

• In the future, we won't go into this level of detail.
Replacing $\Theta$

- Our original recurrence was

\[
\begin{align*}
T(0) &= \Theta(1) \\
T(1) &= \Theta(1) \\
T(n) &\leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)
\end{align*}
\]

- We claimed it was safe to remove the $\Theta$ notation and rewrite it as

\[
\begin{align*}
T(0) &\leq c \\
T(1) &\leq c \\
T(n) &\leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn
\end{align*}
\]

- Why can we do this?
Fat Base Cases

- When $n \geq n_0$, we can replace $\Theta(n)$ by $cn$ for some constant $c$.

- Our simplification in the previous step assumed that $n_0 = 0$. What if this isn't the case?

- Can always rewrite the recurrence to use a "fat base case:"

$$T(n) \leq T(\lfloor n / 2 \rfloor) + T(\lfloor n / 2 \rfloor) + cn \quad \text{(if } n \geq n_0)$$
$$T(n) \leq c \quad \text{(otherwise)}$$

- Makes the induction a lot harder to do, but the result would come out the same.
Non Powers of Two

- Consider this recurrence:

\[
\begin{align*}
T(0) & \leq c \\
T(1) & \leq c \\
T(n) & \leq T(\lfloor n / 2 \rfloor) + T(\lceil n / 2 \rceil) + cn
\end{align*}
\]

- We know that for powers of two, this is upper bounded by \( cn \log_2 n + cn \).

- Does that upper bound still hold for values other than powers of two?

- If not, is our bound even useful?
Non Powers of Two

- Can we claim that since $T(n) \leq cn \log_2 n + cn$ when $n$ is a power of two, that $T(n) = O(n \log n)$?
- Without more work, no. Consider this function:

$$f(n) = \begin{cases} 
  n \log_2 n & \text{if } n = 2^k \\
  n! & \text{otherwise}
\end{cases}$$

- Only looking at inputs that are powers of two, we might claim that $f(n) = \Theta(n \log n)$, even though this isn't the case!
- We need to do extra work to show that $T(n)$ is "well-behaved" enough to extrapolate.
\( b(n) = \Theta(a(n)) \)
Our Proof Strategy

• We will proceed as follows:
  • Show that the values generated by the recurrence are nondecreasing.
  • For each non power-of-two $n$, provide an upper bound $T(n)$ using our upper bound on the next power of two greater than $n$.
  • Show that the upper bound we find this way is asymptotically equivalent (in terms of $\Theta$) to our original bound.
Making Things Easier

- We are given this recurrence:

\[
\begin{align*}
T(0) & \leq c \\
T(1) & \leq c \\
T(n) & \leq T([n / 2]) + T([n / 2]) + cn
\end{align*}
\]

- This only gives an upper bound on \(T(n)\); we don't know the exact values.

- Let's define a new function \(f(n)\) as follows:

\[
\begin{align*}
f(0) & = c \\
f(1) & = c \\
f(n) & = f([n / 2]) + f([n / 2]) + cn
\end{align*}
\]

- Note that \(T(n) \leq f(n)\) for all \(n \in \mathbb{N}\).
Lemma: \( f(n + 1) \geq f(n) \) for all \( n \in \mathbb{N} \).

Proof: By induction on \( n \). As a base case, note that

\[
f(1) = c \geq c = f(0)
\]

For the inductive step, assume that for some \( n \) that the lemma holds for all \( n' < n \). Then

\[
f(n + 1) = f(\lceil (n+1) / 2 \rceil) + f(\lfloor (n+1) / 2 \rfloor) + c(n+1)
\]
\[
\geq f(\lfloor n / 2 \rfloor) + f(\lceil n / 2 \rceil) + cn
\]
\[
= f(n) \quad \blacksquare
\]
Theorem: \( T(n) = O(n \log n) \)

Proof: Consider any \( n \in \mathbb{N} \) with \( n \geq 1 \). Let \( k \) be such that \( 2^k \leq n < 2^{k+1} \). Thus \( 2^{k+1} \leq 2n < 2^{k+2} \).

From our lemma, we know that

\[
T(n) \leq f(n) \leq f(2^{k+1})
\]

Using our upper bound for powers of two:

\[
f(2^{k+1}) \leq c(2^{k+1}) \log_2 (2^{k+1}) + c(2^{k+1})
\]

Therefore

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

\[
\leq c(2n) \log_2 (2n) + 2cn
\]

\[
= 2cn (\log_2 n + 1) + 2cn
\]

\[
= 2cn \log_2 n + 4cn
\]

So for any \( n \geq 1 \), \( T(n) \leq 2cn \log_2 n + 4cn \). Thus \( T(n) = O(n \log n) \). ■
Summary

- We can safely extrapolate from the runtime bounds at powers of two for the following reasons:
  - The runtime is nondecreasing, so we can use powers of two to provide upper bounds on other points.
  - The runtime grows only polynomially, so this upper bounding strategy does not produce values that are “too much” bigger than the actual values.
- In the future, we will assume that this line of proof works and will not repeat it.
Perfectly Safe Assumptions

- For the purposes of this class, you can safely simplify recurrences by
  - Only evaluating the recurrences at powers of some number to avoid ceilings and floors.
  - Replace $\Theta(f(n))$ or $O(f(n))$ terms in a recurrence with a constant multiple of $f(n)$.
  - Replace all constants with a single constant equal to the max of all of the constants.
A Different Problem:
Maximum Single-Sell Profit
Maximum Single-Sell Profit

procedure maxProfit(list prices):
    if length(prices) ≤ 1:
        return 0

    let left be the first half of prices
    let right be the second half of prices

    return max(maxProfit(left), maxProfit(right), max(right) - min(left))
Analyzing the Algorithm

procedure maxProfit(list prices):
    if length(prices) ≤ 1:
        return 0
    let left be the first half of prices
    let right be the second half of prices
    return max(maxProfit(left), maxProfit(right), max(right) - min(left))

T(0) = Θ(1)
T(1) = Θ(1)
T(n) ≤ T(⌈n / 2⌉) + T(⌊n / 2⌋) + Θ(n)

T(n) = O(n log n)
The Divide-and-Conquer Framework

• The two algorithms we have just seen are examples of divide-and-conquer algorithms.

• These algorithms usually have two steps:
  • (Divide) Split the input apart into multiple smaller pieces, recursively solving each piece.
  • (Conquer) Combine the solutions to each smaller piece together into the overall solution.

• Typically, correctness is proven inductively and runtime is proven by solving a recurrence relation.

• In many cases, the runtime is determined without actually solving the recurrence; more on that later.
Another Algorithm: *Binary Search*
procedure binarySearch(list A, int low, int high, value key):
  if low ≥ high:
    return false

  let mid = ⌊(high + low) / 2⌋
  if A[mid] = key:
    return true
  else if A[mid] > key:
    return binarySearch(a, low, mid)
  else:
    return binarySearch(a, mid + 1, high)

T(1) ≤ c
T(n) ≤ T(n / 2) + c
The Iteration Method

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

\[
T(n) \leq T\left(\frac{n}{2}\right) + c \\
\leq T\left(\frac{n}{4}\right) + c + c \\
= T\left(\frac{n}{4}\right) + 2c \\
\leq T\left(\frac{n}{8}\right) + c + 2c \\
= T\left(\frac{n}{8}\right) + 3c \\
... \\
\leq T\left(\frac{n}{2^k}\right) + kc
\]
The Iteration Method

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

\[
T(n) \leq T\left(\frac{n}{2^k}\right) + kc
\]
\[
= T(1) + c \log_2 n
\]
\[
\leq c + c \log_2 n
\]
\[
= O(\log n)
\]
The Recursion Tree Method

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

\[ \cdots \]

\[ c \log_2 n + c \]
Formalizing Our Argument

- To formalize correctness, it's useful to use this invariant:
  
  \[\text{If } key = A[i] \text{ for some } i, \text{ then } \ \text{low} \leq i < \text{high}\]

- You can prove this is true by induction on the number of calls made.

- We can also formalize the runtime bound by induction to prove the \(O(\log n)\) upper bound, but it's not super exciting to do so.