Divide and Conquer: Sorting and Recurrences
Divide & Conquer: The Pattern

- **Divide** the problem into several independent smaller instances of exactly the same problem

- **Delegate** each smaller instance to the **Recursive Leap of Faith** (technically known as induction hypothesis)

- **Combine** the solutions for the smaller instances
Review: Merge Sort

**MergeSort**(\(L\)):

if \(L\) has one element
  return \(L\)

Divide \(L\) into two halves \(A\) and \(B\)

\(A \leftarrow \text{MergeSort}(A)\)
\(B \leftarrow \text{MergeSort}(B)\)

\(L \leftarrow \text{Merge}(A, B)\)

return \(L\)
Merge Step: $\Theta(n)$

- Scan sorted lists from left to right
- Compare element by element; create new merged list

\[
\begin{array}{c}
\text{a} \\
2 & 4 & 9 & 11 & 12
\end{array}
\quad
\begin{array}{c}
b \\
1 & 3 & 5 & 7 & 13 & 14
\end{array}
\]
Merge Step: $\Theta(n)$

Is $a[i] \leq b[j]$?
- Yes, $a[i]$ appended to $c$, advance $i$
- No, $b[j]$ appended to $c$, advance $j$

```
merged list c
```
Merge Step: $\Theta(n)$

Is $a[i] \leq b[j]$?
- Yes, $a[i]$ appended to $c$, advance $i$
- No, $b[j]$ appended to $c$, advance $j$
Merge Step: $\Theta(n)$

Is $a[i] \leq b[j]$?
- Yes, $a[i]$ appended to $c$, advance $i$
- No, $b[j]$ appended to $c$, advance $j$
Merge Step: $\Theta(n)$

Is $a[i] \leq b[j]$?
- Yes, $a[i]$ appended to $c$, advance $i$
- No, $b[j]$ appended to $c$, advance $j$

```
Is a[i] <= b[j] ?
- Yes, a[i] appended to c, advance i
- No, b[j] appended to c, advance j
```

```
merged list c
```
Is $a[i] \leq b[j]$?
- Yes, $a[i]$ appended to $c$, advance $i$
- No, $b[j]$ appended to $c$, advance $j$
Merge Step: $\Theta(n)$

Is $a[i] \leq b[j]$?
- Yes, $a[i]$ appended to $c$, advance $i$
- No, $b[j]$ appended to $c$, advance $j$
Yada yada yada...
Merge Step: $\Theta(n)$

Is $a[i] \leq b[j]$?

- Yes, $a[i]$ appended to $c$, advance $i$
- No, $b[j]$ appended to $c$, advance $j$

**Diagram**:  

- List $a$: 2 4 9 11 12  
- List $b$: 1 3 5 7 13 14  
- Merged list $c$: 1 2 3 4 5 7 9 11 12 13 14
Correctness: D&C Algorithms

• **Proving Correctness** (often follow *proof by induction* pattern)
  - Show *base case* holds
  - Assume your recursive calls return the correct solution (induction hypothesis)
  - **Inductive step**: crux of the proof
    - Must show that the solutions returned by the recursive calls are “combined” correctly
Correctness Sketch: Merge Sort

• Claim. (Combine step.) Merge subroutine correctly merges two sorted subarrays $A[1,\ldots,i]$ and $B[1,\ldots,j]$ where $i + j = n$.

  • Will prove that for the first $k$ iterations of the loop, correctly merges $A$ and $B$ (from $n = 0$ to $n = k$).

• Invariant: Merged array is sorted after every iteration.

• Base case: $k = 0$

  • Algorithm correctly merges two empty subarrays

• For inductive step, there are multiple cases, including $a_i \leq b_j$, $a_i > b_j$

  • for each case, must show that newly added element maintains sorted-ness
Analyzing Running Time

- For this topic, our main focus will be on analysis of running time.
- We analyze the running time of recursive functions by:
  - **Considering the recursive calls**: both the number of calls made and the size of the inputs to each call.
    - e.g., merge sort on an input of size $n$ makes two recursive calls each on an input of size $n/2$.
  - **The time spent “combining” solutions (“non-recursive work”) returned by recursive calls**.
    - e.g. merge step combines the sorted arrays in $\Theta(n)$ time.
- Using the two, we typically write a **running time recurrence**.
Running Time Recurrence

• Let $T(n)$ represent the worst-case running time of merge sort on an input of size $n$

• $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n)$

• Base case: $T(1) = 1$; often ignored

• We will ignore the floors and ceilings (turns out it doesn't matter for asymptotic bounds; we’ll show this later)

• So the recurrence simplifies to:
  • $T(n) = 2T(n/2) + O(n)$
  • The answer to this ends up being $T(n) = O(n \log n)$
  • The next slides will discuss different ways to derive this
Method 1. Unfolding the recurrence

- Assume \( n = 2^\ell \) (that is, \( \ell = \log n \))
- Because we don’t care about constant factors and are only upper-bounding, we can always choose smallest power of 2 that is greater than \( n \). That is, \( n < n' = 2^\ell < 2n \)
- We can explicitly add in our constants

\[
T(n) = 2T(n/2) + cn = 2T(2^{\ell-1}) + c2^\ell \text{ (change of variable, replace } n) \\
= 2(2T(2^{\ell-2}) + c2^{\ell-1}) + c2^\ell = 2^2T(2^{\ell-2}) + 2 \cdot c2^\ell \\
= 2^3T(2^{\ell-3}) + 3 \cdot c2^\ell \\
= \ldots \\
= 2^\ell T(2^0) + c\ell 2^\ell = O(n \log n)
\]
Recurrences: Recursion Tree

**Method 2.** Recursion Trees

- Work done at each level $2^i \cdot (n/2^i) = n$
- Total $\log_2 n$ levels

\[
\begin{align*}
\text{Recommended Method!}
\end{align*}
\]
Recurrences: Recursion Tree

- This is really a method of visualization
- Very similar to unrolling, but much easier to keep track of what’s going on
- It’s not (quite) a proof, but generally it is sufficient for reasoning about running times in this class
  - “Solve the recurrence” can be done by drawing the recursion tree and explaining the solution
Recurrences: Guess & Verify

**Method 3.** Guess and Verify

- Eyeball recurrence and make a guess
- Verify guess using induction

- More on this later…
• What aspect(s) of the course do you like most?
• What aspect(s) of the course do you like least?
• In what CS courses that you’ve taken so far have you needed to spend more time than you have in this course?
• In what CS courses that you’ve taken so far have you needed to spend less time than you have in this course?
• Is there anything that you’d like me to know at this point in the semester?
Acknowledgments

• Some of the material in these slides are taken from
  • Jeff Erickson’s Algorithms Book (http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf)