Reminders

> HW2 is posted: due in one week
  – start early!
  – added hint on problem 1
    > (use domain knowledge to limit ternary search range)
Outline for Today

> Divide & Conquer definition
> Some familiar problems
> Master Theorem
Divide & Conquer

Algorithmic approach:

1. **Divide** the input data into 2+ parts

2. **Recursively** solve the problem on each part
   - i.e., solve the *same problem* on each part

3. **Combine** those solutions to solve the original problem
Divide & conquer uses the solutions to sub-problems to solve the given problem

- used term “sub-problem” before as any other problem I solve as a step toward solving the larger problem
  - e.g., to find the N-th special composite, we solve the sub-problem of finding the next largest such number given all the previous ones
- here, sub-problem is another instance of the same type of problem
  - with N-th special composite, we changed the type of problem

We will see the same idea with dynamic programming

- but has a more general use of sub-problems
Outline for Today

> Divide & Conquer definition
> Some familiar problems
> Master Theorem
Merge Sort

1. Divide the array A into two halves: A[0..n/2-1] and A[n/2..n-1]
   - where n is the length of A

2. Recursively sort the two halves

3. Combine two sorted arrays into a single sorted array
   - merging sorted arrays is easier than sorting
   - use a “two finger” algorithm
     > note that this needs attention to detail
Divide takes no work at all
– we don’t even copy the arrays

To do that, we need to *generalize* the inputs:

```c
void mergeSort(int[] A, int start, int end)
```

This issue will come up again with dynamic programming
– but there it will be less obvious
– keep this example in mind for later...
Merge Sort: Combine

- merge with a “two finger” algorithm
  - one finger for the next largest element of each sub-array
  - copy smaller to the output and move that finger forward

- note that this needs attention to detail...
/** Make B[i..k-1] the sorted merge of A[i..j-1] and A[j..k-1]. */
void merge(int[] A, int i, int j, int k, int[] B) {
    int a = i, b = j, c = i;

    // Inv: B[i..c-1] is merge of A[i..a-1] and A[j..b-1]
    while (c < k) {
        else B[c++] = A[b++];
    }
}

What's wrong?
** Make B[i..k-1] the sorted merge of A[i..j-1] and A[j..k-1]. */
void merge(int[] A, int i, int j, int k, int[] B) {
    int a = i, b = j, c = i;

    // Inv: B[i..c-1] is merge of A[i..a-1] and A[j..b-1]
    while (a < j && b < k) {
        else B[c++] = A[b++];
    }
    B[c++] = A[a++];

    What’s still wrong?
**Merge Sort: Combine**

/** Make B[i..k-1] the sorted merge of A[i..j-1] and A[j..k-1]. */

void merge(int[] A, int i, int j, int k, int[] B) {
    int a = i, b = j, c = i;

    // Inv: B[i..c-1] is merge of A[i..a-1] and A[j..b-1]
    while (a < j && b < k) {
        else B[c++] = A[b++];
    }

    while (a < j) B[c++] = A[a++];
    while (b < k) B[c++] = A[b++];
}

Merge Sort: Combine
Merge Sort: Combine

> merge with a “two finger” algorithm
  - one finger for the next largest element of each sub-array
  - copy smaller to the output and move that finger forward

> note that this needs attention to detail
  - easily to forget array out of bounds cases & left over elements
  - you’ll spot the error when it crashes, but that’s no help for interviews...

> Runs in O(n) time (where \( n = k - i \))
  - every iteration copies one value to B
  - only \( k - i \) values to copy
Merge Sort: running time

> Total time is $T(n)$ where

\[
T(1) = O(1) \\
T(n) = 2 \ T(n/2) + O(n) \quad \text{for } n > 1
\]

> Takes $O(1)$ time to sort 1 element
  – just return

> Time to sort $n$ elements is the time for
  2 recursive calls on half the data + $O(n)$ merge
  – for now, ignore issues about rounding $n/2$ to an integer
Merge Sort: running time

\[
T(1) = D \\
T(n) = 2 \ T(n / 2) + C \ n \quad \text{for } n > 1
\]

> Can solve by repeated substitution...

\[
T(n) = 2 \ (2 \ T(n / 4) + C \ (n/2)) + C \ n \\
= 4 \ T(n / 4) + 2 \ C \ (n/2) + C \ n \\
= 4 \ T(n / 4) + C \ n + C \ n \\
= 4 \ T(n / 4) + 2C \ n
\]
Merge Sort: running time

\[ T(1) = D \]
\[ T(n) = 2 \, T(n / 2) + C \, n \quad \text{for } n > 1 \]

> Can solve by repeated substitution...

\[ T(n) = 4 \, T(n / 4) + 2C \, n \]
\[ = 4 \, (2 \, T(n / 8) + C \, (n / 4)) + 2C \, n \]
\[ = 8 \, T(n / 8) + 4 \, C \, (n / 4) + 2C \, n \]
\[ = 8 \, T(n / 8) + 3C \, n \]
Merge Sort: running time

\[
T(1) = D  \\
T(n) = 2 \cdot T(n/2) + C \cdot n \quad \text{for } n > 1
\]

> Can solve by repeated substitution...

\[
T(n) = 8 \cdot T(n/8) + 3C \cdot n  \\
= \ldots \\
= 2^k \cdot T(n/2^k) + kC \cdot n
\]
Merge Sort: running time

\[
\begin{align*}
T(1) &= D \\
T(n) &= 2 \cdot T(n / 2) + C \cdot n \quad \text{for } n > 1
\end{align*}
\]

Can solve by repeated substitution...

\[
T(n) = 2^k \cdot T(n / 2^k) + kC \cdot n
\]

When \( k = \log_2 n \), we have \( 2^k = 2^{\log_2 n} = n \), so...

\[
\begin{align*}
T(n) &= n \cdot T(n / n) + (\log n) \cdot C \cdot n \\
&= n \cdot T(1) + C \cdot n \cdot \log n
\end{align*}
\]
Merge Sort: running time

\[
\begin{align*}
T(1) &= D \\
T(n) &= 2 \cdot T(n/2) + Cn \quad \text{for } n > 1 \\
\end{align*}
\]

> When \( k = \log_2 n \), we have \( 2^k = 2^{\log n} = n \), so ...

\[
\begin{align*}
T(n) &= n \cdot T(n/n) + (\log n) \cdot Cn \\
&= n \cdot T(1) + Cn \cdot \log n \\
&= Dn + Cn \cdot \log n \\
&= O(n \log n)
\end{align*}
\]
Almost certainly the best algorithm for linked lists
– unlike array version, does not require any extra memory
  > still needs extra space, but...
  > gets by with the space in the pointers of the linked list nodes
– with arrays, quick sort was at one time considered fastest
  > merge sort is probably more commonly used there also now
  > changes in processor architecture had an impact

> Use the same divide & conquer approach...
  – split, recurse, merge
Merge Sort 2: Divide

> How do you split a linked list in two halves?

1. Find an element in the middle and disconnect the lists there
2. Put the even elements in one list and the odds in another

> Let’s look at the second one...
/** Put half of A’s elements into B and half into C */
<T> void split(LinkedList<T> A, List<T> B, List<T> C) {
    while (A.size() > 0) {
        B.add(A.removeFirst());
        C.add(A.removeFirst());
    }
}
/* Put half of A’s elements into B and half into C */
<T> void split(LinkedList<T> A, List<T> B, List<T> C) {
    while (A.size() > 0) {
        if (A.size() % 2 == 0)
            B.add(A.removeFirst());
        else
            C.add(A.removeFirst());
    }
}
Merge Sort 2: Combine

> Merge two sorted linked lists into one with the same idea
  – “two finger” algorithm

> one finger points to next node in each list
  – smallest value not yet merged into the result

> merged list is initially empty
  – add smallest node to the end
Now, the divide is not $O(1)$, it is $O(n)$

Combine is still an $O(n)$ merge

Total time is $T(n)$ where

\[
T(1) = O(1) \\
T(n) = 2 \cdot T(n / 2) + O(n) \quad \text{for } n > 1
\]

Same formula, so same running time as before $O(n \log n)$
Finding the minimum

> **Problem**: Find the minimum value in $A[0..n-1]$

1. Divide the array into two halves: $A[0..n/2-1]$ and $A[n/2..n-1]$
2. Recursively find the minimums: $m_1$ and $m_2$
3. Combine to find the overall minimum: $\min(m_1, m_2)$

> Is this the best way to solve the problem?
  - not really...
Finding the minimum: running time

Total time is $T(n)$ where

$T(1) = O(1)$

$T(n) = 2 \cdot T(n/2) + O(1)$ for $n > 1$
Finding the minimum: running time

\[ T(1) = D \]
\[ T(n) = 2 \, T(n/2) + C \quad \text{for } n > 1 \]

> Solve by repeated substitution...

\[ T(n) = 2 \left( 2 \, T(n/4) + C \right) + C \]
\[ = 4 \, T(n/4) + C \left( 2 + 1 \right) \]
\[ = 4 \left( 2 \, T(n/8) + C \right) + C \left( 2 + 1 \right) \]
\[ = 8 \, T(n/8) + C \left( 4 + 2 + 1 \right) \]
Finding the minimum: running time

T(1) = D
T(n) = 2 \cdot T(n / 2) + C \quad \text{for } n > 1

> Solve by repeated substitution...

T(n) = 2^k \cdot T(n / 2^k) + C \cdot (2^k + \ldots + 2 + 1)

> When \( k = \log_2 n \), \( 2^k = n \), so ...

T(n) = n \cdot T(n / n) + C \cdot (2^k + \ldots + 2 + 1)
Finding the minimum: running time

\[ T(1) = D \]
\[ T(n) = 2 \, T(n / 2) + C \quad \text{for } n > 1 \]

> When \( k = \log_2 n \), \( 2^k = n \), so ...

\[ T(n) = n \, T(n / n) + C \, 2^k (1 + \ldots + 1/2^{k-1} + 1/2^k) \]
\[ = n \, T(1) + C \, n (1 + 1/2 + \ldots + 1/2^k) \]
\[ = D \, n + C \, n \, O(1) \]
\[ = O(n) \]
Outline for Today

> Divide & Conquer definition
> Some familiar problems
> Master Theorem
Master Theorem

> Solves recurrence relations of the form that arise in Divide & Conquer algorithms:

\[
\begin{align*}
T(1) &= O(1) \\
T(n) &\leq a \cdot T(n/b) + f(n) \\
&\quad - \text{divide into } a \text{ problems of size } n/b \\
&\quad - \text{divide + combine takes } f(n) \text{ time}
\end{align*}
\]
Theorem: Let \( T(n) \) be bounded as on the previous slide. Define \( C := \log_b a \). Then...

- If \( f(n) = O(n^{C-\varepsilon}) \) for any \( \varepsilon > 0 \), then \( T(n) = \Theta(n^C) \)
- If \( f(n) = \Theta(n^C) \), then \( T(n) = \Theta(n^C \log n) \)
- If \( f(n) = \Omega(n^{C+\varepsilon}) \) for any \( \varepsilon > 0 \) and ..., then \( T(n) = \Theta(f(n)) \)
  > ... and \( f \) must satisfy a \( f(n / b) < C f(n) \) for some \( C < 1 \)
  > ... almost always true for polynomial time algorithms
  > ... don’t worry about it

Master Theorem

*The size 1 problems are time dominated by all those \( O(1) \)'s*

*The time dominated by divide+combine of initial problem*
Example: Binary Search

> divide into 1 sub-problem of size n / 2
> so a = 1 and b = 2 and C = \log_2 1 = 0

> divide take O(1) time: compare middle element to x
> combine takes no time
  > actually, takes O(1) just for the return statement
> so f(n) = \Theta(1)
Example: Binary Search

> \[ C = \log_2 1 = 0 \]
> \[ f(n) = \Theta(1) \]

> Compare \( f(n) \) to \( n^C = n^0 = 1 \ldots \)

> \( f(n) = \Theta(n^C) \) since both are \( \Theta(1) \)
> Master theorem says time is \( \Theta(n^C \log n) = \Theta(\log n) \)
Example: Binary Search

> f(n) = \Theta(n^c) since both are \Theta(1)
> Master theorem says time is \Theta(n^c \log n) = \Theta(\log n)

> If we had f(n) = \Theta(n^{0.5})
  then master theorem says time is \Theta(f(n)) = \Theta(n^{0.5})
  – time is dominated by the divide + combine of the initial problem
Example: Merge Sort

> divide into 2 sub-problem of size n / 2
> so a = 2 and b = 2 and C = \log_2 2 = 1

> divide take no time: sub-arrays are already in place
> combine takes O(n) time
  > two-finger pass over the two sorted sub-arrays
> so f(n) = \Theta(n)
Example: Merge Sort

> \( C = \log_2 2 = 1 \)
> \( f(n) = \Theta(n) \)

> Compare \( f(n) \) to \( n^C = n^1 = n \)...

> \( f(n) = \Theta(n^C) \) since both are \( \Theta(n) \)
> Master theorem says time is \( \Theta(n^C \log n) = \Theta(n \log n) \)
Example: Merge Sort

> f(n) = \( \Theta(n^c) \) since both are \( \Theta(n) \)
> Master theorem says time is \( \Theta(n^c \log n) = \Theta(n \log n) \)

> If we had \( f(n) = \Theta(n^{0.5}) \),
  then master theorem says time is \( \Theta(n^c) = \Theta(n) \)
  – time is dominated by all the \( O(1) \) works in size 1 problems
f(n) = \Theta(n^C) since both are \Theta(n)

Master theorem says time is \Theta(n^C \log n) = \Theta(n \log n)

If we had f(n) = \Theta(n^2),

then master theorem says time is \Theta(n^2)

- time is dominated by the divide + combine of initial problem

Example: Merge Sort
Example: Merge Sort

> $f(n) = \Theta(n^c)$ since both are $\Theta(n)$
> Master theorem says time is $\Theta(n^c \log n) = \Theta(n \log n)$

> If we had $f(n) = \Theta(n \log n)$, then master theorem says...
Example: Merge Sort

> f(n) = \( \Theta(n^c) \) since both are \( \Theta(n) \)

> Master theorem says time is \( \Theta(n^c \log n) = \Theta(n \log n) \)

> If we had \( f(n) = \Theta(n \log n) \),
then master theorem says... nothing!
  – need \( f(n) = \Omega(n^{c+\epsilon}) \) for some \( \epsilon > 0 \)
  – even \( f(n) = \Omega(n^{1.00000001}) \) would work but \( \Theta(n \log n) \) does not
Understanding the Master Theorem

> Recursion tree
  - node for each recursive call

> Node records time for the divide + combine of that call

> Total time for any call is the sum of all nodes in the subtree
  - cost of the recursive calls is recorded in those child nodes
Understanding the Master Theorem

\[ f(n) \]

\[ f(n/b) \quad f(n/b) \quad \ldots \quad f(n/b) \]

\[ f(n/b^2) \quad f(n/b^2) \quad \ldots \quad f(n/b^2) \]

\[ f(n/b^3) \quad f(n/b^3) \quad \ldots \quad f(n/b^3) \]

\[ f(n/b^k) \quad f(n/b^k) \]

(a recursive calls)

(a^2 recursive calls)

W
Understanding the Master Theorem

> Bottom level has \( n / b^k = 1 \), so \( k = \log_b n \)
  - height of the tree is \( k + 1 \)

> Number of leaves is \( a^k \)
  - this is \( a^{\log_b n} = a^{\log_a n \log_b a} = n^{\log_b a} = n^c \)
Understanding the Master Theorem

> Time for the top call (divide + combine) is $f(n)$
> Time for all the calls in the leaves is $O(1) \times n^c = \Theta(n^c)$

> Master theorem asks us to compare these times
  - when $f(n) \gg \Theta(n^c)$, then the root divide + combine dominates everything
  - when $f(n) \ll \Theta(n^c)$, then the leaf node work dominates everything
  - when $f(n) = \Theta(n^c)$, then all the $\Theta(\log n)$ levels do the same work
Proof of Master Theorem (out of scope)

> Total time is

\[ \Theta(n^c) + \sum_{i=0}^{\log_{bn} - 1} a^i f(n/b^i) \]

- left term is leaves
- right term is all the divide + combines
- (need two terms since T(n) is defined by two formulas)
Case 2: $f(n) = \Theta(n^C)$

Let $f(n) \leq A n^C$

- so $A$ is the hidden constant in the big-O

Then $a^i f(n / b^i) \leq a^i A (n / b^i)^C = A n^C a^i / b^i C = A n^C (a / C b)^i$

- note that $b^C = b^{\log_b a} = a$
- so we have $a^i f(n / b^i) \leq A n^C (a / a)^i = A n^C$
- work on each level is $\leq A n^C$
- total work is $\leq \Theta(\log n) A n^C = \Theta(f(n) \log n)$

Proof of Master Theorem (out of scope)
Case 1: $f(n) \ll \Theta(n^C)$
- need to show summation is $O(n^C)$
- then the first term is just as big

Assume $f(n) \leq A \cdot n^{C-\varepsilon}$ for some $\varepsilon > 0$

Then $a^i f(n / b^i) \leq a^i A \cdot (n / b^i)^{C-\varepsilon} = (n / b^i)^{-\varepsilon} A \cdot n^C \cdot (a / b^C)^i = A \cdot n^{C-\varepsilon} \cdot b^{i\varepsilon}$
- only difference is factor of $(n / b^i)^{-\varepsilon}$
- as before, $a / b^C = a / a = 1$
- need to sum this over $i...$
Proof of Master Theorem (out of scope)

> **Case 1**: $f(n) \ll \Theta(n^c)$
>  
>  - need to show summation is $O(n^c)$
>  - then the first term is just as big

> **Time for all levels:**

$$
\sum_{i=0}^{\log_bn-1} a^i f(n/b^i) \leq An^{c-\varepsilon} \sum_{i=0}^{\log_bn-1} (b^\varepsilon)^i = An^{c-\varepsilon} \cdot n^\varepsilon = AnC
$$

- geometric series with largest term $(b^\varepsilon)^{\log_bn} = (b^{\log_bn})^\varepsilon = n^\varepsilon$
Proof of Master Theorem (out of scope)

> **Case 3:** $f(n) \gg \Theta(n^C)$
>   
>   - need to show summation is $\Theta(f(n))$, which dominates first term

> Will use: $a \cdot f(n / b) = C \cdot f(n)$ for some $C < 1$
> Then $a^i f(n / b^i) < C^i f(n)$, so...

\[
\sum_{i=0}^{\log_b n - 1} a^i f(n/b^i) \leq f(n) \sum_{i=0}^{\log_b n - 1} C^i = f(n) \cdot O(1)
\]

- geometric series with largest term $C^0 = 1$