Divide and Conquer

CPE 349

Theresa Migler-VonDollen

Divide and Conquer

Divide and Conquer is a strategy that solves a problem by:

1. Breaking the problem into subproblems that are themselves smaller instances of the same type of problem.
2. Recursively solving these subproblems.
3. Appropriately combining their answers.

Searching

Search

Input: A list of sorted numbers, $A = [a_0, a_1, a_2, a_3, \ldots a_{n-1}]$ and a number x.
Goal: Return the position of x in A or return a statement that x is not in the list.

- How would a four year old do this?
- How would a 2nd grader do this?
- How would you do it?

BinarySearch

```
BinarySearch(A, x, min, max)
```

Input: An array of sorted numbers A, values x, min, and max.
Output: The position of x in A, or a statement that x is not in A.

(Initially $min = 0$ and $max = n - 1$)

if max $<$ min **then**
  return "$x \notin A$"
else
  if $x < A[\lfloor (max + min)/2 \rfloor]$ **then**
    return BinarySearch(A, x, min, $\lfloor (max + min)/2 \rfloor - 1$)
  else if $x > A[\lfloor (max + min)/2 \rfloor]$ **then**
    return BinarySearch(A, x , $\lfloor (max + min)/2 \rfloor + 1, max$)
  else
    return $\lfloor (max + min)/2 \rfloor$

BinarySearch

We must ask ourselves two questions about this algorithm:

- Is it correct?
- What is the running time?
  - At each stage we divide the problem in half and it takes constant time to "combine" solutions:
  - $T(n) = T(n/2) + O(1)$

BinarySearch - Correctness

Is it correct?

Lemma

If x is an element in the list, BinarySearch *will return the correct position of x.*

Do we need a better statement of the lemma?

Proof.

By induction, in class. □

Lemma

If x is not an element of the list, BinarySearch *will return that there is no such element in the list.*

Running Time

Again, for any algorithm there are two (maybe three) important things to prove:

- Is it correct?
- How fast is it?
- (Maybe) How much space does it take?

Example:
Compare $f(x) = 5x + 100$ with $g(x) = x^2$

- If these functions represent running times, which is faster?
- We need to formalize what we mean by "faster".
  - "Big O" notation.

Big O Notation

- Big O notation describes the limiting behavior of a function when the input gets very large.
- Big O notation characterizes functions according to their growth rates:
  - Functions with the same growth rate may be represented using the same O notation.
- When we make statements such as, $f(x) = 2x^3 - 7x + 14 = O(x^3)$:
  - The first equals sign really means equality.
  - The second equals sign represents set inclusion.

Big O Notation

Rough Guide

class	in English	meaning	key phrases
$f(n) = o(g(n))$	little-oh	$f(n) << g(n)$	$f(n)$ is asymptotically better than $g(n)$ $f(n)$ grows slower than $g(n)$
$f(n) = O(g(n))$	big-oh	$f(n) \leq g(n)$	$f(n)$ is asymptotically no worse than $g(n)$ $f(n)$ grows no faster than $g(n)$
$f(n) = \Theta(g(n))$	big-theta	$f(n) \approx g(n)$	$f(n)$ is asymptotically equivalent to $g(n)$ $f(n)$ grows the same as $g(n)$
$f(n) = \Omega(g(n))$	big-omega	$f(n) \geq g(n)$	$f(n)$ is asymptotically no better than $g(n)$ $f(n)$ grows at least as fast as $g(n)$
$f(n) = \omega(g(n))$	little-omega	$f(n) >> g(n)$	$f(n)$ is asymptotically worse than $g(n)$ $f(n)$ grows faster than $g(n)$

Big O Notation

Formal Definitions

class	formally	working
$f(n) = o(g(n))$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$	$f(n) = O(g(n))$ but $g(n) \neq O(f(n))$
$f(n) = O(g(n))$	$\exists c > 0,\ \exists n_0,\ \forall n > n_0, f(n) \leq c \cdot g(n)$	
$f(n) = \Theta(g(n))$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} =$ some finite, non-zero constant	$f(n) = O(g(n))$ and $g(n) = O(f(n))$
$f(n) = \Omega(g(n))$	$\exists c > 0,\ \exists n_0,\ \forall n > n_0, c \cdot g(n) \leq f(n)$	$g(n) = O(f(n))$
$f(n) = \omega(g(n))$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$	$g(n) = O(f(n))$ but $f(n) \neq O(g(n))$

Big O Notation

Practical tricks:

- If $f(x)$ is a sum of several terms, then the one with the largest growth rate is kept, and all others omitted.
- If $f(x)$ is a product of several factors, any constants are omitted.

Example

For the following examples determine if $f = O(g)$, $f = \Omega(g)$, or both ($f = \Theta(g)$):

1. $f(x) = 4x^2$ and $g(x) = 1,000x^2 + 12x + 7$
2. $f(x) = x^2 3^x$ and $g(x) = 4^x$
3. $f(x) = 2^x$ and $g(x) = x!$
4. $f(x) = 10 \log x$ and $g(x) = \log x^2$

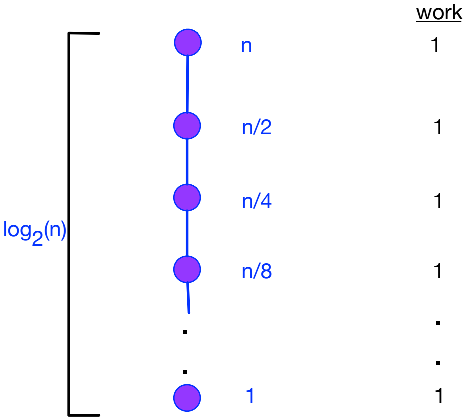
Recurrence Relations

You have two main choices when it comes to solving recurrence relations:

- The tree method (my favorite)
- The Master Theorem
  - If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for $a > 0, b > 1, d \geq 0$ then:
    - $T(n) = O(n^d)$ if $d > \log_b a$
    - $T(n) = O(n^d \log n)$ if $d = \log_b a$
    - $T(n) = O(n^{\log_b a})$ if $d < \log_b a$

Recurrence Relations - Binary Search

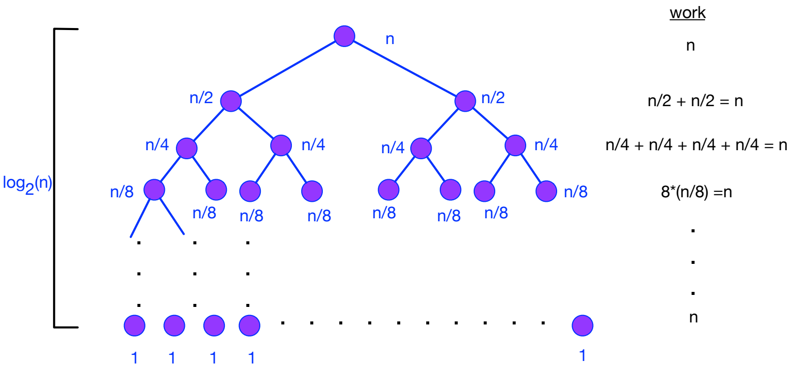
$T(n) = T(n/2) + O(1)$



$= O(1 + 1 + \cdots + 1) = O(\log_2(n))$

Recurrence Relations - Merge Sort

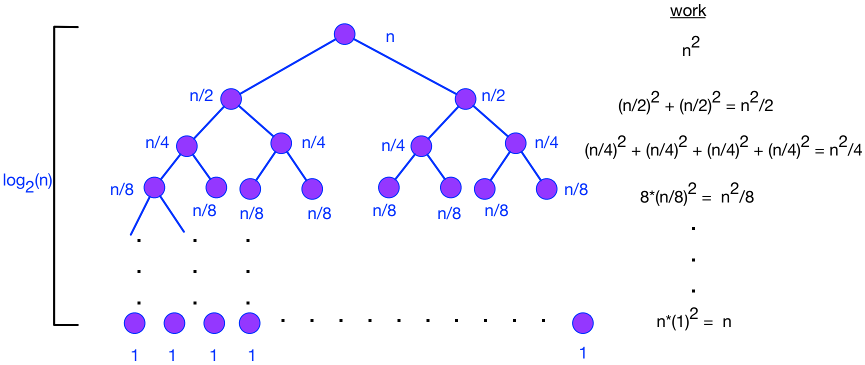
$T(n) = 2T(n/2) + O(n)$



$= O(n + n + n + \cdots + n) = O(n \log_2(n))$

Recurrence Relations - More Practice

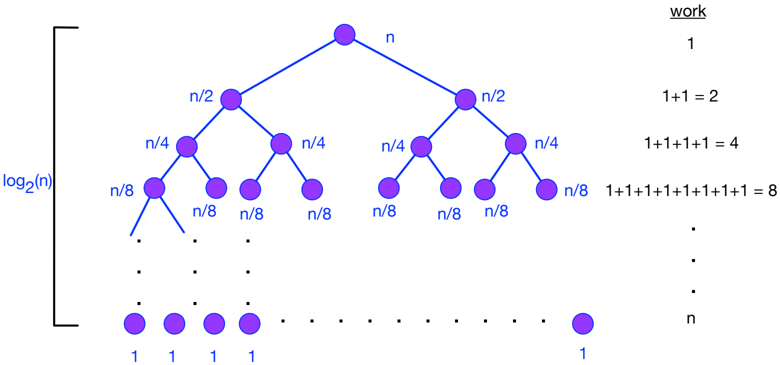
$$T(n) = 2T(n/2) + O(n^2)$$



$$= O(n^2 + n^2/2 + n^2/4 + n^2/8 \cdots + n^2/n) \leq O(2n^2) = O(n^2)$$

Recurrence Relations - More Practice

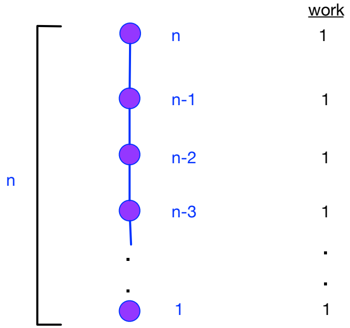
$T(n) = 2T(n/2) + O(1)$



$= O(1 + 2 + 4 + \cdots + n) = O(n + n/2 + n/4 + n/8 + \cdots + n/n) \leq O(2n) = O(n)$

Recurrence Relations - Reduce by One

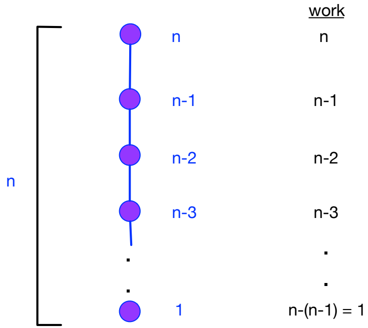
$$T(n) = T(n-1) + O(1)$$



$$= O(1 + 1 + 1 + \cdots + 1) = O(n)$$

Recurrence Relations - Reduce by One

$$T(n) = T(n-1) + O(n)$$



$$= O(n+(n-1)+(n-2)+(n-3)+\cdots+(n-(n-2))+(n-(n-1))) \approx O(n^2/2) = O(n^2)$$

Matrix Multiplication

Who remembers how to multiply two $n \times n$ matrices together?
Example:

$$\begin{pmatrix} 1 & 0 & 7 \\ 0 & 1 & 3 \\ 2 & 1 & 3 \end{pmatrix} \begin{pmatrix} 1 & 2 & 1 \\ 5 & 1 & 0 \\ 1 & 3 & 3 \end{pmatrix}$$

- What algorithm did you use?
- What is the running time?

Matrix Multiplication

Matrix Multiplication

Input: Two $n \times n$ matrices, X and Y.
Goal: Return the product XY.

To simplify analysis, suppose that n is a power of 2.

Matrix Multiplication

MMult(X,Y)

Input: Two $n \times n$ matrices, X and Y, (where n is a power of 2)
Output: The product XY

1: **if** $n = 1$ (i.e. $X = (x), Y = (y)$) **then**
  return $(\; x \times y \;)$
2: **else** Decompose X and Y into four $n/2 \times n/2$ blocks each:

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}, Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

$$\textbf{return} \begin{pmatrix} MMult(A,E) + MMult(B,G) & MMult(A,F) + MMult(B,H) \\ MMult(C,E) + MMult(D,G) & MMult(C,F) + MMult(D,H) \end{pmatrix}$$

Matrix Multiplication

- Is the algorithm correct?
- What is the running time?
  - $T(n) = 8T(n/2) + O(n^2) = O(n^3)$
  - There was no improvement from the linear algebra method.
- There is a way to only perform 7 multiplications (Strassen's method).

$$XY = \begin{pmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{pmatrix}$$

Where: $P_1 = A(F - H)$, $P_2 = (A + B)H$, $P_3 = (C + D)E$, $P_4 = D(G - E)$, $P_5 = (A + D)(E + H)$, $P_6 = (B - D)(G + H)$, and $P_7 = (A - C)(E + F)$.

  - Then $T(n) = 7T(n/2) + O(n^2) = O(n^{2.81})$.

There exists an $O(n^{2.3727})$ algorithm (Virginia Vassilevska Williams).

Sorting

Sorting

Input: A list of numbers, $a_1, a_2, a_3, \ldots a_n$.
Goal: Return a list of the same numbers sorted in increasing order.

Example:
Given: $4, 907, 34, 18, 42, 36, 71, 34, 16$
Return: $4, 16, 18, 34, 34, 36, 42, 71, 907$

SelectionSort

Sorting

Input: A list of numbers, $a_1, a_2, a_3, \ldots a_n$.
Goal: Return a list of the same numbers sorted in increasing order.

We will start with a straightforward algorithm:

SelectionSort($A[0, \ldots, n-1]$)

Input: A list of unsorted numbers $A[0, \ldots, n-1]$
Output: The same list sorted in increasing order

for $i = 0, \ldots, n-1$ **do**
    Find min of $A[i, \ldots, n-1]$.
    Suppose that the min occurs at position j.
    Swap $A[i]$ with $A[j]$.

SelectionSort - Correctness

Lemma

Upon completion of SelectionSort, *for any* $i \in \{1, \ldots, n-1\}$, $A[i-1] \leq A[i]$. *Furthermore, all elements of the input list are in* A.

Proof.

Suppose, for a contradiction, that there is a j such that upon completion of SelectionSort, $A[j-1] > A[j]$.
Let $A[j-1] = x$ and $A[j] = y$.
At iteration $j-1$ of the algorithm, $A[j-1]$ was set to be x (step 4).
Thus, at iteration $j-1$, x must have been the smallest remaining element (step 2).
Contradiction. Because y was a remaining element at iteration $j-1$ and $y < x$. □

How can we show that all elements of the input list are in A?

SelectionSort - Running Time

- Running Time:
  - $T(n) = T(n-1) + O(n) = O(n^2)$

MergeSort

Sorting

Input: A list of numbers, $a_1, a_2, a_3, \ldots a_n$.
Goal: Return a list of the same numbers sorted in increasing order.

MergeSort($A[0, \ldots, n-1]$)

Input: A list of unsorted numbers $A[0, \ldots, n-1]$
Output: The same list, sorted in increasing order

1: **if** $n \leq 1$ **then**
    return A
2: **else**
    return Merge(MergeSort($A[0, \ldots, \lfloor n/2 \rfloor]$),MergeSort($A[\lfloor n/2 \rfloor + 1, \ldots, n-1]$))

MergeSort

Sorting

Input: A sequence of numbers, $a_1, a_2, a_3, \ldots a_n$.
Goal: Return a list of the same numbers sorted in increasing order.

Merge($x[0,\ldots,k-1]$,$y[0,\ldots,\ell-1]$)

Input: Two sorted lists, $x[0,\ldots,k-1]$ and $y[0,\ldots,\ell-1]$
Output: One sorted list that contains all elements of both lists.

```
if x = ∅ then return y
if y = ∅ then return x
if x[0] ≤ y[0] then
        return x[0]∘Merge(x[1,...,k−1], y[0,...,ℓ−1])
else
        return y[0]∘Merge(x[0,...,k−1], y[1,...,ℓ−1])
```

MergeSort - Correctness

Theorem

Merge *correctly merges two sorted lists.*

Theorem

Given two sorted lists, x and y, of total size n Merge *returns a sorted list containing all elements from x and y.*

Proof.

We will procede by induction on the total size of the lists being merged.

- Base Case: $(n = 1)$
  This will only occur if either x or y is empty, and the other list has exactly 1 element.
  - Merge correctly merges the empty list with any other sorted list (steps 1 and 2).

MergeSort - Correctness

Proof (Cont.)

- Inductive Hypothesis: Suppose that, given two sorted lists, x and y, of total size h Merge returns a sorted list containing all elements from x and y.
- Inductive Step: Consider two sorted lists with total size $h + 1$.
  - In steps 3 and 4 of the algorithm, Merge correctly places the smallest element at the beginning of the list.
  - Merge then concatenates that element with the Merge of the remaining elements of the two lists.
    - The total size of the remaining two lists is h.
    - By the Inductive Hypothesis, Merge correctly merges the remainder.
- Conclusion: Therefore, by PMI, Merge correctly merges two sorted lists.

□

MergeSort - Running Time

- What is the running time?
  - $T(n) = 2T(n/2) + O(n) = O(n \log n)$

Median Finding

Definition

The *median* of a list of numbers is its 50th percentile. Half the numbers are bigger than the median and half the numbers are smaller.

For example, suppose the list of numbers is $14, 2, 3, 2, 7$.
The median is 3.
What if the list is even?
We choose the smaller of the two middle elements.

Median Finding

Input: A list of numbers, $a_1, a_2, a_3, \ldots a_n$.
Goal: Return the median element.

Any ideas?

Selection

It is surprisingly easier to consider a more general problem, *selection*.

Selection

Input: A list of numbers, $a_1, a_2, a_3, \ldots a_n$ and an integer k.
Goal: Return the kth smallest element of $a_1, a_2, a_3, \ldots a_n$.

If $k = 1$, then the minimum element is returned.
If $k = n$, then the maximum element is returned.
If $k = \lfloor \frac{n}{2} \rfloor$, then the median is returned.

Randomized Selection

RandomSelection($A[1, \ldots, n], k$)

Input: A list of unsorted numbers $A[1, \ldots, n]$ and an integer k.
Output: The kth smallest element of A.

if $n \leq 1$ **then**
  return A
else
  Randomly choose an element from A, call it x.
  Let A_L be the numbers in the list less than x, A_R be the numbers in the list greater than x, and A_x be the numbers in the list equal to x.
  if $k \leq |A_L|$ **then return** RandomSelection(A_L, k)
  if $|A_L| \leq k \leq |A_L| + |A_x|$ **then return** x
  else **return** RandomSelection($A_R, k - |A_L| - |A_x|$)

RandomSelection

What is the running time?
We can build, A_L, A_R, and A_x in linear time. If we could choose x so that roughly half of the elements in the list are in A_L and the other half are in A_R, then our running time would be:
$T(n) = T(n/2) + O(n) = O(n)$
But that would only work if x is the median!

- Worst Case?
  - In the worst case, we may select x to be the largest or smallest element over and over - that would only shrink our list by one at each iteration:
    $T(n) = T(n-1) + O(n) = O(n^2)$
    Luckily it turns out this is highly unlikely.
- Average Case?

RandomSelection Average Case Running Time

Let's call a choice of x "good" if it lies within the 25th to 75th percentile. Then:
$|A_L| \leq 3/4|A|$ and $A_R \leq 3/4|A|$
How many x values do we have to pick (on average) before a good one is found?

Lemma

On average a fair coin needs to be tossed twice before a "heads" is seen.

The proof hinges on the fact that if E is the expected number of tosses before a heads is seen, $E = 1 + \frac{1}{2}E$.
Therefore, on average, after two choices of x, the array will be reduced to at most 3/4 its original size.
The *expected* running time is:
$T(n) \leq T(3n/4) + O(n) = O(n)$

RandomSelection Correctness

With randomized algorithms, it's important to check that the algorithm terminates.
Does RandomSelection terminate?

Lemma

Given a list, A, of size n and a value, k, RandomSelection(A, k) *correctly finds the kth smallest element of A.*

Proof.

Use induction over n. □

In Class Exercise: Peaks

Peaks

Suppose you are given a list, A, with n entries, each entry holding a distinct number. You are told that the sequence of values $A[1], A[2], \ldots, A[n]$ is *unimodal*: For some index, p, between 1 and n, the values of the list increase up to position p and decrease until position n.

- Find the "peak" entry.

- Is your algorithm correct?
- What is the running time?

Peaks - Pseudocode

FindPeaks($A[1, \dots, n]$)

Input: A unimodal list of distinct numbers $A[1, \dots, n]$.
Output: The peak entry.

1: **if** $|A| = 1$ **then**
  return $A[1]$
2: **if** $|A| = 2$ **then**
  return $\max\{A[1], A[2]\}$
3: $mid = \lceil n/2 \rceil$
4: **if** $A[mid] > A[mid-1]$ AND $A[mid] > A[mid+1]$ **then**
  return $A[mid]$
5: **else if** $A[mid] > A[mid-1]$ AND $A[mid] < A[mid+1]$ **then**
  return FindPeaks($A[mid, \dots n]$)
6: **else if** $A[mid] < A[mid-1]$ AND $A[mid] > A[mid+1]$ **then**
  return FindPeaks($A[1, \dots, mid]$)

Peaks - Running Time

- We reduce a problem of size n to a single problem of size $n/2$.
- There are a constant number of comparisons at each level of recursion.
- $T(n) = T(n/2) + O(1) = O(\log_2 n)$

Peaks - Correctness

Theorem

FindPeaks *correctly finds the peak in a unimodal list of n distinct numbers.*

Proof.

We proceed by strong induction.

- Base Cases:
  - If there is one element (as in part 1 of FindPeaks), then FindPeaks correctly returns that element.
  - If there are two elements (as in part 2 of FindPeaks), then the larger of the two is the peak and FindPeaks correctly returns it.
- Induction Hypothesis:
  - Suppose that FindPeaks correctly finds the peak in a unimodal list of less than or equal to k distinct numbers.

Peaks - Correctness

Theorem

FindPeaks *correctly finds the peak in a unimodal list of n distinct numbers.*

Proof (Cont.)

- Consider a unimodal list of $k+1$ distinct numbers.
  - If the midpoint ($mid = \lceil (k+1)/2 \rceil$) is the peak, then $A[mid] > A[mid-1]$ and $A[mid] > A[mid+1]$. FindPeaks will correctly return $A[mid]$.
  - If mid is among the increasing portion of the list ($A[mid] > A[mid-1]$ and $A[mid] < A[mid+1]$), then the peak is in the second half of the list ($A[mid, \ldots, k]$). FindPeaks returns FindPeaks($A[mid, \ldots, n]$), which, by the induction hypothesis correctly finds the peak because the list has size less than or equal to k.

Peaks - Correctness

Theorem

FindPeaks *correctly finds the peak in a unimodal list of n distinct numbers.*

Proof (Cont.)

- - If mid is among the decreasing portion of the list ($A[mid] < A[mid-1]$ and $A[mid] > A[mid+1]$), then the peak is in the first half of the list ($A[1, \ldots, mid]$). FindPeaks returns FindPeaks($A[1, \ldots, mid]$), which, by the induction hypothesis correctly finds the peak because the list has size less than or equal to n.
- Therefore, by the principle of mathematical induction, we have the result.

□