CSCE 411
Design and Analysis of Algorithms

Set 3: Divide and Conquer
Slides by Prof. Jennifer Welch
Spring 2014
General Idea of Divide & Conquer

1. Take your problem and divide it up into smaller pieces
2. Solve one or more of the smaller problems
3. Combine solutions to subproblems
Varieties of Divide & Conquer
[Levitin]

- **Option 1:** Only use one of the subproblems
  - Ex: binary search
  - Levitin calls this “decrease & conquer”

- **Option 2:** Use all of the subproblems
  - Ex: mergesort

- Running time for these algorithms can often be stated as a recurrence and solved with the master theorem
Varieties of Decrease & Conquer

- Decrease by a constant
  - Ex: insertion sort (subtract 1) – $\Theta(n^2)$
    - incrementally build up longer and longer prefix of the array of keys that is in sorted order (unsorted suffix shrinks by 1)
    - take the current key, find correct place in sorted prefix, and shift to make room to insert it
  - Ex: another algorithm for topological sorting (subtract 1)
    - identify a source (node with no incoming edges) in the DAG
    - add this node to the list of nodes and remove all its outgoing edges
    - repeat until all nodes are removed
Varieties of Decrease & Conquer

- **Decrease by a constant factor – \( \Theta(\log n) \)**
  - **Ex: binary search (divide by 2)**
    - divide sequence into two halves by comparing search key to midpoint
    - recursively search in one of the two halves
    - combine step is empty
  - **Ex: fake coin problem – \( \Theta(\log n) \)**
    - Given a set of \( n \) coins (\( n-1 \) are real and have same weight, 1 is fake and is lighter), find the fake coin
    - divide set of coins into two piles of floor(\( n/2 \)) each; if \( n \) is odd, there is 1 left over
    - if the piles weigh the same, the leftover coin is the fake coin
    - otherwise continue recursively with the lighter pile
Varieties of Decrease & Conquer

- Decrease by a variable amount
  - Ex: searching (and inserting) in a binary search tree
    - compare search key to key in current node and decide to continue search in either left subtree or right subtree, not necessarily same size
  - Ex: Euclid’s algorithm for computing GCD (greatest common divisor)
    - From about 300 B.C.
    - Cf. Chapter 31, Section 2
Greatest Common Divisor

- gcd(a, b) is the largest integer that divides both a and b
  - Ex: gcd(24, 36) = 12

- First try: factor a and b into primes and then choose the common ones:
  - 24 = $2^3 \times 3$ and 36 = $2^2 \times 3^2$
  - so gcd(24, 36) = $2^2 \times 3 = 12$

- But factoring is not so easy
Euclid’s Algorithm

Key insight: $\gcd(a, b) = \gcd(b, a \mod b)$

- “$a \mod b$” means the remainder when dividing $a$ by $b$
- Ex: $\gcd(36, 24) = \gcd(24, 36 \mod 24)$
  
  $= \gcd(24, 12)$
  
  $= \gcd(12, 24 \mod 12)$
  
  $= \gcd(12, 0)$
  
  $= 12$

Why? Next slide...
GCD Recursion Theorem Proof

Strategy is to show that
- gcd(a, b) divides gcd(b, a mod b), and that
- gcd(b, a mod b) divides gcd(a, b).

So they must be equal.

To show gcd(a, b) divides gcd(b, a mod b):
- a mod b = a – floor(a/b)*b (remainder after dividing a by b)
- gcd(a, b) divides a and b, and so it divides a – floor(a,b)*b, which is (a mod b)
- since gcd(a, b) divides b and (a mod b), it divides gcd(b, a mod b)

To show gcd(b, a mod b) divides gcd(a, b), use similar argument
Euclid’s Algorithm

Euclid(a, b) // a and b nonnegative integers
  if b == 0 return a
  else return Euclid(b, a mod b)

Correct because of previous observation. Also, no infinite loop (why?)
Running Time of Euclid’s Algorithm

- Running time is proportional to the number of recursive calls made.
- WLOG, assume $a > b$ initially. Then first argument is larger than second in each recursive call.
- Show if $k \geq 1$ recursive calls are done, then $a \geq F_{k+2}$ and $b \geq F_{k+1}$.
  - Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$.
- Basis: $k = 1$. Then $b \geq 1 = F_2$ (since there is at least one recursive call), and $a \geq 2 = F_3$ (since $a > b$).
Running Time of Euclid’s Algorithm

- **Induction:** Euclid(a,b) recursively calls Euclid(b, a mod b), which in turn makes \( k - 1 \) recursive calls.
- By inductive hypothesis, since Euclid(b, a mod b) makes \( k - 1 \) recursive calls, \( b \geq F_{k+1} \) and \( (a \mod b) \geq F_k \).
- Must show \( a \geq F_{k+2} \), or equivalently \( F_{k+2} \leq a \):
  \[ F_{k+2} = F_{k+1} + F_k \]
  \[ \leq b + (a \mod b) \]
  \[ = b + (a - \text{floor}(a/b)*b) \]
  \[ \leq a \quad \text{since floor}(a/b) \text{ is at least 1} \]
Running Time of Euclid’s Algorithm

- Just showed if it takes $k$ recursive calls, then $b \geq F_{k+1}$.
- **Fact:** $F_{k+1}$ is approx. $\phi^k/\sqrt{5}$, where $\phi = (1+\sqrt{5})/2$ (the golden ratio)
  - see Ch 3, Sec 2
- So $b \geq \phi^k/\sqrt{5}$
- Solving for $k$ gives: $k \leq \log_{\phi} \sqrt{5} + \log_{\phi} b$
- Thus $k = O(\log b)$
  - base of logarithm doesn’t matter asymptotically

and running time is proportional to number of digits in $b$. 

Classic Divide & Conquer

- Sorting:
  - mergesort – \( \Theta(n \log n) \)
    - divide sequence in half
    - recursively sort the two halves
    - merge the sorted halves
  - quicksort – \( \Theta(n^2) \)
    - divide sequence into two (possibly unequal-sized) parts by comparing pivot to each key
    - recursively sort the two parts
    - combine step is empty
Graph algorithms: binary tree traversals

- Inorder traversal:
  - traverse left subtree of current vertex
  - visit current vertex
  - traverse right subtree of current vertex

- Preorder traversal similar, but visit current vertex first
- Postorder traversal similar, but visit current vertex last

- All three take O(n) time, where n is number of nodes in tree
- Note difference from searching in a binary tree
D&C Algorithm for Closest Pair

- Recall the problem: Given n points in the plane, find two that are the minimum distance apart.
- Brute force algorithm took \( \Theta(n^2) \) time.
- Try to do better with divide and conquer:
  - divide points into two disjoint subsets
  - recursively find closest pairs in the two subsets
  - somehow combine results to get final answer
D&C Algorithm for Closest Pair: Ideas

- Separate points into two equal-sized groups on either side of a vertical line
- Recursively compute closest pair for left group and for right group
  - what should base of the recursion be?
- Check if there is a smaller distance between two points on opposite sides of the vertical line
  - This is the tricky part
D&C Algorithm for Closest Pair: Ideas

- Separate points into two equal-sized groups on either side of a vertical line
- Recursively compute closest pair for left group and for right group
  - what should base of the recursion be?
- Check if there is a smaller distance between two points on opposite sides of the vertical line
  - This is the tricky part
D&C Algorithm for Closest Pair: Ideas

- \(d\) is min. of min. distance on right and min. distance on left
- any pair with distance < \(d\) must be in this strip of width 2\(d\) centered around dividing line
- consider points in strip from bottom to top
- for each such point, compare it against other points in the strip that could possibly be closer
- there are only a constant number of these other points!
D&C Algorithm for Closest Pair: Ideas

Each box is $d/2$ by $d/2$

No point in comparing $p$ against points in red area – more than $d$ away

Just need to worry about the six blue boxes

Each box contains at most one point, since maximum distance in a box is $d/\sqrt{2}$, which is $< d$
D&C Algorithm for Closest Pair: Pseudocode

- ClosestPairDist(P):
  - if n is small then return result of brute force algorithm
  - $P_l :=$ left half of P w.r.t. x-coordinate
  - $P_r :=$ right half of P w.r.t. x-coordinate
  - $d_l :=$ ClosestPairDist($P_l$)
  - $d_r :=$ ClosestPairDist($P_r$)
  - $d := \min(d_l,d_r)$
  - for each point p in S (2d-wide center strip) do
    - for each point q in one of the six boxes do
      - $d := \min(d,\text{dist}(p,q))$
  - return d
D&C Algorithm for Closest Pair: Implementation Notes

- Before calling recursive code, preprocess:
  - sort P into array PX by increasing x-coordinate
  - sort P into array PY by increasing y-coordinate

- Use PX to efficiently divide P into half w.r.t. x-coordinates

- Use PY to efficiently scan up the 2d-wide center strip
D&C Algorithm for Closest Pair: Running Time

- Preprocessing takes $O(n \log n)$ time
- Recursive code, if implemented carefully, has running time described by this recurrence:

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

- i.e., two recursive calls (left half and right half)
- rest of the work takes time linear in the number of points being handled
- Solution is $T(n) = O(n \log n)$

- Total time is $O(n \log n)$; beats brute force
D&C Algorithm for Convex Hull

- Divide points into two halves by x-coordinates
- Recursively compute the convex hulls of the two subsets
- Combine the two convex hulls into the convex hull for the entire set of points

- How to do the combining step?
Merging Hulls

- Find the upper tangent line and the lower tangent line to the two hulls
- Remove the interior points on the two hulls
Running Time

- **Claim**: Merging the two hulls can be done in $O(n)$ time.
  - see Preparata and Hong, CACM 1977 (original paper) and various textbooks and on-line resources for details
- Thus running time is $T(n) = 2T(n/2) + O(n)$
  - Why?
- By master theorem, $T(n) = O(n \log n)$
Another Convex Hull Algorithm: Graham’s Scan

- (Not a divide & conquer algorithm)
- Start with lowest point and work your way around the set of points counter-clockwise, deciding whether or not each point is in the convex hull

See Fig. 33.7 in [CLRS] for a more involved example
Graham’s Scan Pseudocode

- \( p_0 := \) point with minimum y-coordinate
- \( p_1, p_2, \ldots, p_m := \) remaining points in counter-clockwise order of polar angle around \( p_0 \) // drop collinear points
- \( S := \) empty stack
- \( S.\text{push}(p_0); S.\text{push}(p_1); S.\text{push}(p_2) \)
- for \( i = 3 \) to \( m \) do
  - while angle formed by \( S.\text{second}(), S.\text{top}(), \) and \( p_i \) does not form a left turn do
    - \( S.\text{pop()} \)
    - \( S.\text{push}(p_i) \)
- return \( S \) // contains CH vertices in CCW order
Ordering Points by Polar Angle

- Simple approach is to calculate angle that line segment $p_0p_i$ makes w.r.t. horizontal line passing through $p_0$ (using basic geometry) for each $p_i$, and sort by angle.

- There is also a way using cross products of vectors to avoid operations that are expensive and prone to round-off error (division and trig functions).
  - See Ex. 33.1-3 in [CLRS]
Determining if an Angle Makes a Left Turn

Given 3 points $u$, $v$, and $w$, does angle $\angle uvw$ turn left or right?

In other words, is line segment $uw$ counterclockwise or clockwise from line segment $uv$?
Determining if an Angle Makes a Left Turn

- Can check this using cross product:
  - \((w-u) \times (v-u)\) is defined to be:
    \[(w.x-u.x)(v.y-u.y) - (v.x-u.x)(w.y-u.y)\]
  - Using \(x\) and \(y\) to indicate \(x\) and \(y\) coordinates of \(u\), \(v\) and \(w\)
- **Claim:** If \((w-u) \times (v-u) < 0\), then counter-clockwise (left), if it is > 0, then clockwise (right), and if it is 0, then collinear

![Diagram](image.png)
Running Time of Graham’s Scan

- Determine point $p_0$ with smallest y-coordinate: $O(1)$
- Calculate polar angles of remaining points w.r.t. $p_0$ and sort them: $O(n \log n)$
- Each stack operation: $O(1)$
- Total time of for loop, excluding time taken by enclosed while loop: $O(n)$
  - $m < n$ iterations and remaining body consists of a single stack push
- Total time of while loop, over all iterations of enclosing for loop: $O(n)$
  - total number of pops $\leq$ total number of pushes
  - each point is pushed at most once, so at most $n$ pops
  - each while loop iteration does one pop
  - so at most $n$ iterations of while loop
  - also, while loop test (for left turn) takes $O(1)$ time
- Grand total is $O(n \log n)$
Why is Graham’s Scan Correct?

- Intuition is that as we move counterclockwise, we have in the stack exactly the points that form the convex hull of the points we have processed so far, and the points are in the stack (from bottom to top) in counterclockwise order.

- We can formalize this argument using induction on the number of iterations of the for loop.
Proof that Graham’s Scan is Correct

**Claim:** For all $i = 3$ to $n+1$, at start of iteration $i$ of for loop, stack $S$ equals the points of $CH(Q_{i-1})$ in CCW order ($Q_{i-1}$ is $p_0, p_1, ..., p_{i-1}$).

- When $i = n+1$ (i.e., last check of the for-loop condition), this will imply that $S$ equals the CH of all the points.

**Show this is true by induction in $i$.**

**Basis:** When starting the for loop, $S$ equals $p_0, p_1, p_2$, which is the CH of these 3 points.
Proof that Graham’s Scan is Correct

- **Claim:** For all $i = 3$ to $n+1$, at start of iteration $i$ of for loop, stack $S$ equals the points of $\text{CH}(Q_{i-1})$ in CCW order ($Q_{i-1}$ is $p_0$, $p_1$, $\ldots$, $p_{i-1}$).

- **Induction:** Assume claim is true for all iterations 3, 4, ..., $i$. Show claim is true for iteration $i+1$.

- During iteration $i$, $p_i$ is under consideration, and some points might be popped off $S$ in the while loop due to nonleft-turn check.

- Let $p_j$ be top of $S$ after all the popping: $j \leq i-1$.

- $S$ contains exactly what it contained at end of iteration $j$, and thus start of iteration $j+1$.

- Since $j+1 \leq i$, *inductive hypothesis* states that $S$ contains $\text{CH}(Q_j)$.

- At end of iteration $i$ (and start of iteration $i+1$), $S$ contains $\text{CH}(Q_j) \cup \{p_i\}$. Must show this is same as $\text{CH}(Q_i)$.
Proof that Graham’s Scan is Correct

check for non-left turns and perhaps pop some points off S
Proof that Graham’s Scan is Correct

No point popped off S during iteration i can belong to CH(Q_i).
Suppose p_t is popped and p_r is its predecessor in S.
Then p_t is inside triangle p_0p_r p_i and is not part of CH(Q_i).
Additional Convex Hull Algorithms

- **Quickhull**: also divide & conquer, similar to quicksort
  - $O(n^2)$ worst case time, but if points are distributed uniformly at random in a convex region, then average case time is $O(n)$

- **Jarvis’ march**:
  - $O(nh)$ time, where $h$ is number of points on the hull
  - ranges from $O(n^2)$ to $O(n)$

- Asymptotically optimal algorithm has time $O(n \log h)$
  - ranges from $O(n \log n)$ to $O(n)$
D&C Algorithm to Multiply Large Integers

- Cryptographic applications require manipulating very large integers
  - 100 decimal digits or more
- Too long to fit into a computer word
- How can we efficiently manipulate them?
  - in particular, multiply them
- What is the time of the brute force algorithm for multiplying two n-digit integers?
D&C Algorithm to Multiply Large Integers

- The answer is $\Theta(n^2)$: each digit of one number must be multiplied times each digit of the other number, and then some additions done
- Can this be done faster?
- Although it may be counter-intuitive, it turns out it can be!
- Key idea is to reuse multiplications of some digits
- Homework.
D&C Algorithm to Multiply Matrices

Now let’s consider the problem of multiplying two matrices.

- Matrices are used throughout mathematics, science, engineering, business, economics,...
- Many applications for multiplying matrices (e.g., determining existence of paths from one vertex to another in a graph/network)

What is the running time of the brute force algorithm for matrix multiplication?
D&C Algorithm to Multiply Matrices

- Following the definition of matrix multiplication gives us an algorithm with $\Theta(n^3)$ running time.

- Can we do better?

- It might seem counter-intuitive, but the answer is “yes”.

- Key is to reuse some multiplications of the matrix elements
  - sound familiar?
Strassen’s Matrix Multiplication Algorithm

<board work>
Representing Polynomials

 Polynomial $A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$ has degree $n-1$ (largest power of $x$ with nonzero coefficient).

 Two ways to represent polynomial $A(x)$:

 - with the $n$ coefficients: $a_0, a_1, \ldots, a_{n-1}$
 - with $n$ point-value pairs (one more than the degree): $(x_0, A(x_0)), (x_1, A(x_1)), \ldots, (x_{n-1}, A(x_{n-1}))$

  where $x_0, x_1, \ldots, x_{n-1}$ are distinct points

 See Theorem 30.1 in [CLRS] for why this works.
Operations on Polynomials

- evaluate $A$ at some point $x_0$
- add two polynomials $A(x)$ and $B(x)$:
  - sum is defined to be $C(x)$, where $c_j = a_j + b_j$, $0 \leq j \leq \max(\deg(a), \deg(b))$
- multiply two polynomials $A(x)$ and $B(x)$:
  - product is defined to be $C(x)$, where $c_j = \sum_k a_k b_{j-k}$, $0 \leq j \leq \deg(A) + \deg(B)$
- How can we do these operations with the two different representations?
Operations with Coefficient Representation

- Evaluating $A(x_0)$: Use Horner’s rule.
  - rewrite $A(x_0)$ as
    $$a_0 + x_0(a_1 + x_0(a_2 + \ldots + x_0(a_{n-2} + x_0(a_{n-1})))\ldots))$$
  - Pseudocode:
    ```
    val := a_{n-1}
    for i := n-2 downto 0 do
      val := x*val + a_i
    return val
    ```
  - Running time is $O(n)$
Operations with Coefficient Representation

- Adding two polynomials:
  - add the corresponding coefficients, as in the definition of the sum
  - $O(n)$ running time

- Multiplying two polynomials:
  - Follow the definition of the product
  - $O(n^2)$ running time
Operations with Point-Value Pairs Representation

- **Evaluation**:  interpolate (convert to coefficient form) and evaluate
  - [CLRS] explains how to interpolate in $O(n^2)$ time
  - Thus total time is $O(n^2)$

- **Addition**:  add the corresponding $n$ values
  - requires the pairs for the two polynomials to use the same set of points
  - $O(n)$ time
Operations with Point-Value Pairs Representation

- **Multiplication:** multiply the corresponding $n$ values
  - requires the pairs for the two polynomials to use the same set of points
  - also requires enough values: since degree of product is $\text{deg}(A) + \text{deg}(B) = 2(n-1)$, we need $2n-1$ points to start with
  - $O(n)$ running time
Comparing Representations

<table>
<thead>
<tr>
<th></th>
<th>Coefficients</th>
<th>Point-Value Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluation</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Addition</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Multiplication</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Can we get the best of both worlds?

Yes (almost), using a divide-and-conquer algorithm called the Fast Fourier Transform (FFT)!
Efficient Multiplication Using Coefficients: Overview

\[ a_0, a_1, \ldots, a_{n-1} \]
\[ b_0, b_1, \ldots, b_{n-1} \]
\[ c_0, c_1, \ldots, c_{2n-2} \]

\[ \Theta(n^2) \]

ordinary multiplication

\[ \Theta(n) \]

point-wise multiplication

\[ \Theta(n \log n) \]

evaluate at carefully chosen points

\[ \Theta(n \log n) \]

interpolate

2n pairs for A, 2n pairs for B

2n pairs for C

\[ \text{FFT} \]

\[ \text{FFT}^{-1} \]
FFT Details

- <board work>
Divide & Conquer Summary

- decrease & conquer
  - insertion sort
  - topological sort algorithm that successively removes sources
  - binary search
  - fake coin algorithm
  - Euclid’s GCD algorithm
Divide & Conquer Summary

- classic divide & conquer
  - mergesort, quicksort
  - binary tree traversals
  - closest pair algorithm with center strip
  - convex hull algorithm that merges left and right hulls
    - also Graham’s scan (not D&C)
  - multiplying large integers
  - Strassen’s matrix multiplication
  - FFT (applied to polynomial multiplication)