CSE 417
Divide & Conquer (pt 4)
More Examples
Reminders

> HW3 due Wednesday
  – get started right away on understanding the algorithm
  – will be quick to do the drawing once you understand it fully

> HW2 postscript
  – good resume material
  – saw the impact of regularization
    > penalty improves prediction accuracy (ex. of Occam’s razor)
  – esp. useful for model selection on “small data” problems
  – always keep an eye out for applications of binary search
Use master theorem to calculate the running time
Review from last Time

> Counting Inversions
  - combine: add inversions \((i, j)\) with \(i\) in first half and \(j\) in second half
  - optimized by using binary search ~> faster than \(O(n^{1+\varepsilon})\) for any \(\varepsilon > 0\)
    > eventually realized sequence of linear searches is \(O(n)\) ~> so \(O(n \log n)\) in total

> Voronoi diagrams

> Closest Pair of Points
  - optimize by comparing \((p,q)\) if \(|p.x - q.x| < d\) and \(|p.y - q.y| < d\)
  - careful analysis reveals this is \(O(n)\) worst case ~> \(O(n \log n)\) overall
    > would have seen it was fast in practice anyway
Outline for Today

- Maximum sub-array sum
- Maximum single-sell profit
- Intersecting horz & vert segments
Famous interview question from the 1980–90s
  – Jon Bentley wrote about the problem in CACM 1984
  – no longer used, I think, since it’s too well known

Good algorithms question: has multiple solutions
  – the best solution uses dynamic programming
  – can also be solved with divide & conquer (today)

HW4 is similar to this problem
**Maximum Sub-array Sum**

> **Problem:** Given an array $A$ of integers, find max of $A[i] + ... + A[j-1]$ over all $0 \leq i \leq j \leq n$
>  - we allow $i = j$ so that the sub-array $A[i:j-1]$ can be empty
>  - note that $A[i]$’s can be negative

> **Example:**

$$A = [31, -41, 59, 26, -53, 58, 97, -93, -23, 84]$$

>  - includes a negative number, -53
Maximum Sub-array Sum

> **Problem:** Given an array $A$ of integers, find max of $A[i] + \ldots + A[j-1]$ over all $0 \leq i \leq j \leq n$
  > we allow $i = j$ so that the sub-array $A[i:j-1]$ can be empty

> **Brute force solution 1:**
  > for every $i = 0 \ldots n$, for every $j = i \ldots n$, compute $A[i] + \ldots + A[j-1]
  >$ take the maximum of all these
  > takes $\Theta(n^3)$ time
Maximum Sub-array Sum

> **Problem:** Given an array $A$ of integers, find max of $A[i] + ... + A[j-1]$ over all $0 \leq i \leq j \leq n$
> - we allow $i = j$ so that the sub-array $A[i:j-1]$ can be empty

> **Brute force solution 1:**
> - takes $\Theta(n^3)$ time

> **Brute force solution 2:**
> - for every $i = 0 .. n$, for every $j = i .. n$, compute $B[i] - B[j]$
>   > take the maximum of all these
Problem: Given an array $A$ of integers, find max of $A[i] + \ldots + A[j-1]$ over all $0 \leq i \leq j \leq n$ we allow $i = j$ so that the sub-array $A[i:j-1]$ can be empty.

Brute force solution 1:
- takes $\Theta(n^3)$ time

Brute force solution 2:
- for every $i = 0 \ldots n$, for every $j = i \ldots n$, compute $B[i] - B[j]$
  - take the maximum of all these
- takes $\Theta(n^2)$ time
Maximum Sub-array Sum

> Apply divide & conquer...

1. Divide A into halves, A[0..n/2-1] and A[n/2..n-1]

2. Recursively solve the sub-problem on each half
   > sums that start & end on one side

3. Combine by considering sums starting on left, ending on right
   > max (A[i] + ... + A[n/2-1]) + (A[n/2] + ... + A[j-1])
Maximum Sub-array Sum

> Apply divide & conquer...

3. Combine by considering sums starting on left, ending on right

> maximize $A[i] + \ldots + A[j-1]$
Maximum Sub-array Sum

\[ \max A[i] + \ldots + A[j-1] \]


- can always do this since \( i \leq n/2 \leq j \)
- splits sum into two parts
  - sum of suffix of first half
  - sum of prefix of second half
Maximum Sub-array Sum

> Choice of i only affects first part and Choice of j only affects second part
  – can maximize the two independently

> \[ \max A[i] + \ldots + A[j-1] \]
  \[ = (\max A[i] + \ldots + A[n/2-1]) + (\max A[n/2] + \ldots + A[j-1]) \]
Maximum Sub-array Sum

> Apply divide & conquer...

3. Combine by considering sums starting on left, ending on right

> max (A[i] + ... + A[n/2-1]) + (A[n/2] + ... + A[j-1])
= (max A[i] + ... + A[n/2-1]) + (max A[n/2] + ... + A[j-1])

> that equation is the key insight of the algorithm!
  - max is achieved by \textit{separately} maximizing each half
  - similar ideas in dynamic programming (also greedy)
Apply divide & conquer...

3. Combine by considering sums starting on left, ending on right
   > max (A[i] + ... + A[n/2-1]) + (A[n/2] + ... + A[j-1])
     = (max A[i] + ... + A[n/2-1]) + (max A[n/2] + ... + A[j-1])
   > compute sums A[i] + ... + A[n/2-1] for each i = 0 .. n/2-1
     – takes $\Theta(n)$ time
     – take the maximum of these
   > compute sums A[n/2] + ... + A[j-1] for each j = n/2 .. n-1
     – takes $\Theta(n)$ time
   > take sum of two maximums
Maximum Sub-array Sum

> Apply divide & conquer...

1. Divide A into halves, A[0..n/2-1] and A[n/2..n-1]

2. Recursively solve the sub-problem on each half
   > sums that start & end on one side

3. Combine by considering sums starting on left, ending on right
   > find max sum crossing divide with scan left & scan right

> Divide + combine in Θ(n), so Θ(n log n) by master thm
Outline for Today

> Maximum sub-array sum
> Maximum single-sell profit
> Intersecting horz & vert segments
Maximum Single-Sell Profit

> **Problem**: Given a list of prices on each day, find the days on which to buy & sell that would have achieved max profit
  - maximize (sell price – buy price)
Maximum Single-Sell Profit

> **Issue:** actually want to minimize percentage increase
  > (i.e., increase per dollar bought)
  > can spend whatever amount you want on the stock (simplification)

> \[
> \max \left( \frac{\text{price}_{\text{sell}} - \text{price}_{\text{buy}}}{\text{price}_{\text{buy}}} \right) = \max \frac{\text{price}_{\text{sell}}}{\text{price}_{\text{buy}}} - 1
> \]

log is monotonically increasing, so it does not change order, so it does not change maximum

> = \max \frac{\text{price}_{\text{sell}}}{\text{price}_{\text{buy}}}
> = \max \log\left( \frac{\text{price}_{\text{sell}}}{\text{price}_{\text{buy}}} \right)
> = \max \log(\text{price}_{\text{sell}}) - \log(\text{price}_{\text{buy}})
> \]

– original problem with log prices instead
**Maximum Single-Sell Profit**

> Apply divide & conquer

1. Divide prices into first half and second half

2. Recursively solve each sub-problem
   - gives best with both buy & sell on same half

3. Combine by considering best buy on one half and sell on other half
   - only need to consider buying in first half and selling in second half because we cannot buy after we sell
     - (actually, you can... it’s called short selling, but that’s disallowed here)
Apply divide & conquer

1. Combine by considering best (buy in first half, sell in second half)
   
   Q: what is the best price to buy at in first half?
   A: minimum price

   Q: what is the best price to sell at in second half?
   A: maximum price
Maximum Single-Sell Profit

> Apply divide & conquer

1. Divide prices into first half and second half

2. Recursively solve each sub-problem
   > gives best with both buy & sell on same half

3. Combine by considering best buy on one half and sell on other half
   > find (min(first half), max(second half)) in O(n) time

> Running time is O(n log n) by master theorem
Maximum Single-Sell Profit

> Q: Is this *really* easier to solve by divide and conquer?
> A: No!

> Linear-time single-pass (right to left) algorithm:
  > keep track of max profit seen so far
  > keep track of the maximum price seen so far
    > best available price to sell in the future
  > if current price - max price > max profit:
    > make this the max price

> This is $O(n)$ — faster than divide & conquer solution
Maximum Single-Sell Profit

Q: Is this really easier to solve by divide and conquer?
A: No!

Divide & Conquer may still help us find the solution
BUT ask yourself when you’re done whether it’s really needed.
Outline for Today

- Maximum sub-array sum
- Maximum single-sell profit
- Intersecting horz & vert segments
Count Intersections of Horz & Vert Segments

> **Problem**: given a set of horizontal & vertical segments, count the number of points where they intersect

> **Brute force solution**:
- check every pair to see if they intersect
- $\Theta(n^2)$ pairs to check if $n/2$ vertical and $n/2$ horizontal segments

Count Intersections of Horizontal & Vertical Segments

- Apply divide & conquer...
  1. Divide horizontally into two halves
     - vertical segments end up on one side
     - but horizontal segments could cross the dividing line
     - leave those out
  2. Recursively solve each sub-problem
  3. Combine by adding intersections with missing ones
     - every other intersection is accounted for
     - those completely on left and right cannot intersect
Count Intersections of Horizontal & Vertical Segments

> **Sub-problem:** find intersections with segments left out
  - these are horizontal segments, so they intersect with vertical ones
  - have a list of all the left out horizontal segments

> **Idea:** sort vertical & left out horizontal segments together
  - horizontal segments appear *twice* (once for each endpoint)
  - can do the sort before-hand...
    > only adds $O(n \log n)$ to total running time
Count Intersections of Horz & Vert Segments

> **Idea:** sort vert & left out vert segments
  > horizontal segments appear **twice**
  > can do the sort before-hand...
  > only adds $O(n \log n)$ to total running time

> **Example:** $[1_A, 1_B, 2, 3, 4_C, 5_B, 6, 6_A, 7, 8_C]$
  > subscript indicates horizontal segment
  > (in Java, this would be a list of objects)

> **Approach:** scan from right & left to dividing line
  > track horizontal segments & look for intersections with vertical
Count Intersections of Horz & Vert Segments

> **Idea:** sort vert & left out vert segments
  - horizontal segments appear **twice**

> Example: [1_A, 1_B, 2, 3, 4_C, 5_B, 6, 6_A, 7, 8_C]
  - subscript indicates horizontal segment

> Scan right to dividing line
  - keep an **AVL tree** of horizontal segments seen so far
  - for each vertical segment:
    > use range search on AVL tree to find intersecting horz segments
    > takes $O(\log n + \#\text{intersections})$ time
Count Intersections of Horizontal & Vertical Segments

> Apply divide & conquer...

1. Divide horizontally into two halves
   > leaving out horizontal segments that span two halves

2. Recursively solve each sub-problem

3. Combine by adding intersections with missing ones
   > $O(n \log n + \#\text{intersections})$ scan to left & right to count these

> Running time is $O(n^{1+\varepsilon} + \#\text{intersections})$ by master
Count Intersections of Horizontal & Vertical Segments

> Q: Is this really easier to solve by divide and conquer?

> A: No!
  – could just do this linear scan the entire way across
  – running time is $O(n \log n + \#\text{intersections})$

> Many faster algorithms for this problem...
  – from Bentley, Tarjan, etc.