CSE 417
Divide & Conquer (pt 5)
Review

UNIVERSITY *of* WASHINGTON

Reminders

- **HW3 due today**
- **HW4 will be posted tomorrow**
  - coding assignment
  - need to invent a new divide & conquer algorithm
    - similar to examples seen previously

Divide & Conquer Review

- Apply the steps:
  1. Divide the input data into 2+ parts
  2. Recursively solve the problem on each part
  3. Combine the sub-problem solutions into a problem solution

- Key questions:
  1. Can you solve the problem by combining solutions from sub-problems?
  2. Is that easier than solving it directly?

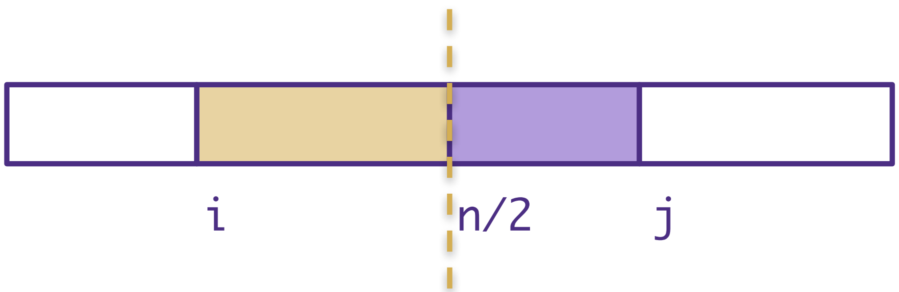
- Use master theorem to calculate the running time

Review from last Time

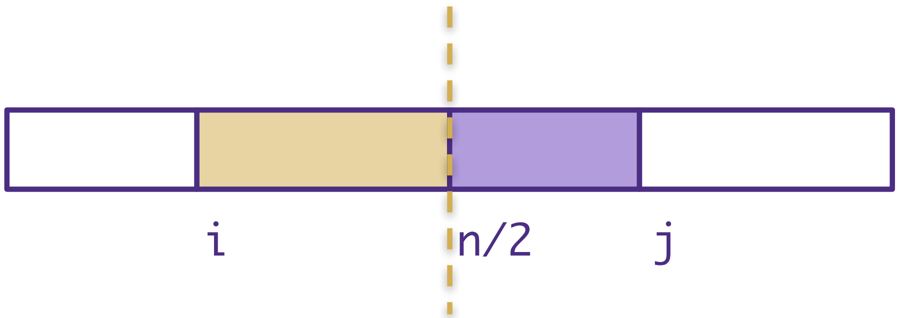
- **Max Sub-Array Sum**
  - combine separates into *independent* problems on left and right half…

Maximum Sub-array Sum

- max A[i] + … + A[j-1]
  = max A[i] + … + A[n/2-1] + A[n/2] + … + A[j-1]
  - can always do this since i ≤ n/2 ≤ 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] + … + A[n/2-1]) + (A[n/2] + … + A[j-1])
  = (max A[i] + … + A[n/2-1]) + (max A[n/2] + … + A[j-1])
  - nothing can be larger than this (≤) and it is achieved (=)

Review from last Time

- **Max Sub-Array Sum**
  - combine separates into *independent* problems on left and right half
  - dead give-away that divide & conquer will be useful

Review from last Time

- **Max Sub-Array Sum**
  - combine separates into *independent* problems on left and right half
  - dead give-away that divide & conquer will be useful
- **Max Single-Sell Profit**
  - equivalent problem with percentage change instead of absolute change…

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 $(\text{price}_{sell} - \text{price}_{buy}) / \text{price}_{buy}$
  $= \max \text{price}_{sell} / \text{price}_{buy} - 1$
  $\rightarrow \max \text{price}_{sell} / \text{price}_{buy}$
  $\rightarrow \max \log(\text{price}_{sell} / \text{price}_{buy})$
  $= \max \log(\text{price}_{sell}) - \log(\text{price}_{buy})$
  - *original problem with log prices instead*

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

Review from last Time

- **Max Sub-Array Sum**
  - combine separates into *independent* problems on left and right half
  - dead give-away that divide & conquer will be useful
- **Max Single-Sell Profit**
  - equivalent problem with percentage change instead of absolute change
  - (use in HW4)

Review from last Time

> **Max Sub-Array Sum**
  - combine separates into *independent* problems on left and right half
  - dead give-away that divide & conquer will be useful

> **Max Single-Sell Profit**
  - equivalent problem with percentage change instead of absolute change
  - (use in HW4)

> **Intersecting Horz & Vert Segments**
  - can be implemented in O(n log n) with divide & conquer
  - requires a subtle change to the problem formulation

Outline for Today

> Quicksort
> Implementing Partition
> Find by Rank
> Maximum Sub-array Average

Quicksort

- Previously looked at mergesort:
  - divide into A[0..n/2-1] and A[n/2..n-1] — easy!
  - combine by merging two sorted arrays into one — tricky but O(n)
- Quicksort is another divide & conquer sorting algorithm
- Uses a strategy that makes **combining** easy instead:
  - divide: rearrange so that (each of A[0], ..., A[k-1]) < (each of A[k], ..., A[n-1])
    - everything in left part is smaller than everything in right part
  - recursively sort A[0..k-1] and A[k..n-1]
  - combine by... nothing

Partition

- Key subroutine of mergesort is (sorted) “merge”
- Key subroutine of quicksort is “partition”
  - solves the following sub-problem

- **Sub-problem**: Given (unsorted) array A and *a number x*, rearrange so that A[0], ..., A[k-1] ≤ x and x < A[k], ..., A[n-1] and then return the index k

- We will later see this can be done in O(n) time...

Quicksort

> Apply divide & conquer...

A[0] is already in the right place

1. Divide using partition:
   > partition A[1..n-1] with x = A[0]
   > tells us that (each of A[0], ..., A[k-1]) < (each of A[k], ..., A[n-1])

2. Sort A[0..k-1] and A[k..n-1] recursively in place

3. Combine by doing nothing
   > already have $\underbrace{A[0] \le \dots \le A[k-1]}_{\text{sorted}} \underbrace{<}_{\text{partitioned}} \underbrace{A[k] \le \dots \le A[n-1]}_{\text{sorted}}$

Preview

- Apply divide & conquer…

  1. Divide using partition:
     - partition A[1..n-1] with x = A[0]
     - tells us that (each of A[0], …, A[k-1]) < (each of A[k], …, A[n-1])

- We actually know a little more than this…
  - A[1], …, A[k-1] ≤ A[0], so A[0] is as large as any of these
  - it will end up at A[k-1] when A is fully sorted
  - could just put it there by swapping A[0] and A[k-1]
    - that would let us recurse on A[0..k-2] and A[k..n-1]

Quicksort Run Time (out of scope)

- If $k = n/2$, then we have two recursive calls of half the size
  - $T(n) = 2\,T(n/2) + O(n)$
  - running time is $O(n \log n)$ by master theorem

- Unfortunately, there is no way to know that we'll get $k = n/2$
  - worst case would be if A[0] is the smallest or largest element
  - that would be true if A was already sorted!
  - get an $O(n^2)$ algorithm in that case
    - that's bad

Quicksort Run Time (out of scope)

> If k = n/2, then we have two recursive calls of half the size
  - T(n) = 2 T(n/2) + O(n)
  - running time is O(n log n) by master theorem

> Unfortunately, there is no way to know that we'll get k = n/2
  - nonetheless, this was often used in practice
  - alternative #1: pick a **random** element and swap it with A[0]
  - alternative #2: take the middle of A[0], A[n/2], and A[n-1]
    - "median of three"
    - works very well in practice
    - works perfectly on sorted data (no longer the worst case)

Quicksort Run Time (out of scope)

- It is sufficient if we have a $< 1/n$ probability of bad split
  - (call the split bad if it's in the first 5% or last 5% of numbers)
  - average running time is then $(n-1)/n\ O(n \log n) + (1/n)\ O(n^2) = O(n \log n) + O(n)$
  - still $O(n \log n)$ on average
- This is studied in more detail in *Randomized Algorithms* class
  - assumes familiarity with machinery of probability
- Quicksort works extremely well in practice whether or not we can get the theory right...

Outline for Today

> Quicksort
> Implementing Partition
> Find by Rank
> Maximum Sub-array Average

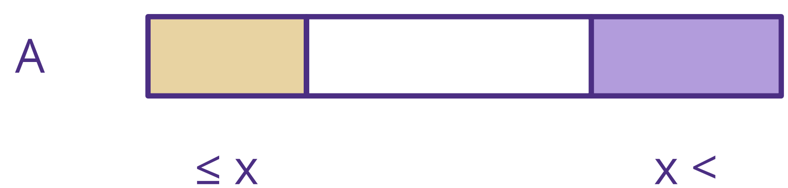
Implementing Partition

- **Problem**: Given (unsorted) array A and *a number x*, rearrange so that A[0], ..., A[k-1] ≤ x and x < A[k], ..., A[n-1] and then return the index k

- Another case where you need to careful attention to detail
  - i.e., you need to spell out your **loop invariant** in detail

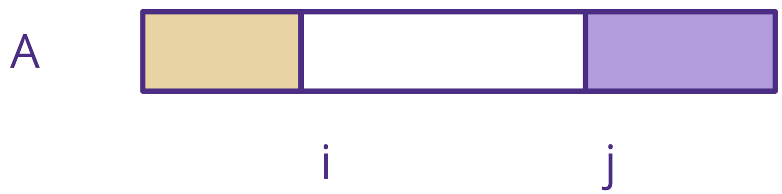
Implementing Partition

> **Problem**: Given (unsorted) array A and *a number x*, rearrange so that A[0], ..., A[k-1] ≤ x and x < A[k], ..., A[n-1] and then return the index k



Implementing Partition

> **Problem**: Given (unsorted) array A and *a number x*, rearrange so that $A[0], \ldots, A[k-1] \le x$ and $x < A[k], \ldots, A[n-1]$ and then return the index k



Loop Invariant: $A[0], .., A[i-1] \le x$ and $x < A[j], \ldots, A[n-1]$

- set $i = 0$ and $j = n$ to make it true initially
- done when $i = j$ $(= k)$

Implementing Partition

```
/** Return k with A[0], ..., A[k-1] ≤ x < A[k], ..., A[n-1]. */
void partition(int[] A, int x) {
  int i = 0, j = A.length;

  // Inv: A[0], ..., A[i-1] ≤ x and x < A[j], ..., A[n-1]
  while (i < j) {
    if (A[i] <= x) {
      i++;
    } else {
      swap(A, i, j-1);
      j--;
    }
  }

  return j;
}
```

invariant still holds since A[i-1] <= x

x < A[j-1]
x < A[j]   so invariant holds again

postcondition is true with k = j

Outline for Today

- > Quicksort
- > Implementing Partition
- > Find by Rank
- > Maximum Sub-array Average

Find By Rank

> **Problem**: Given an unsorted array A and a number m in 1 .. n, return the m-th smallest number in A.

> **Simple Solution**: (always start here)
  - sort A
  - return A[m-1]
  - takes Θ(n log n) time using mergesort

Find By Rank

- **Problem**: Given an unsorted array A and a number m in 1 .. n, return the m-th smallest number in A.

- **Q**: Is this optimal?
  - any solution must take Ω(n) time
  - **Q**: do we need to fully sort it?
    - algorithm returns A[m-1]
      so only needed the fact that A[m-1] is in the right place

Find By Rank

- **Problem**: Given an unsorted array A and a number m in 1 .. n, return the m-th smallest number in A.
- **Sub-problem**: partition so that A[m-1] is in the right spot
  - i.e., we need A[0], …, A[m-2] ≤ A[m-1] < A[m], …, A[n-1]
  - let's try using quicksort's partition…

Find By Rank

- **Problem**: Given an unsorted array A and a number m in 1 .. n, return the m-th smallest number in A.

- **Sub-problem**: partition so that A[m-1] is in the right spot

- **Idea**: partition A using A[0] then swapping...
  puts A[0] at its proper sorted position of A[k-1]

- **Q**: What's wrong with this?
- **A**: This tells us where A[0] belongs when sorted,
  but it may not belong at index m – 1!

Find By Rank

> **Problem**: Given an unsorted array A and a number m in 1 .. n, return the m-th smallest number in A.

> **Sub-problem**: partition so that A[m-1] is in the right spot

> **Idea**: partition A using A[0]
  – A[0] moves to A[k-1], but could have k < m or m < k

> **Q**: What should we do?
> **A**: binary search

Find By Rank

- **Problem**: Given an unsorted array A and a number m in 1 .. n, return the m-th smallest number in A.

- **Algorithm** (partition + binary search = "**quick select**")
  1. Partition A using A[0], moving it to index k – 1.
  2. If k > m, then recurse on A[0 .. k-2] looking for m-th smallest.
  3. If k < m, then recurse on A[k .. n-1] looking for (m – k)-th smallest.
  4. Otherwise, k = m, so return A[k-1].

- Running time <u>should</u> satisfy T(n) = T(n / b) + O(n).
  - solution is O(n) by master theorem *even if b = 1.001*

Outline for Today

> Quicksort
> Implementing Partition
> Find by Rank
> Maximum Sub-array Average

Maximum Sub-array Average

Previously looked at maximum sub-array sum...

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

Now consider...

only sub-arrays with 2+ elements

> **Problem**: Given an array A of integers,
> find max avg(A[i] + ... + A[j]) over $0 \le i < j \le n$

Maximum Sub-array Average

- **Problem**: Given an array A of integers,
  find max $(A[i] + \ldots + A[j]) / (j - i + 1)$ over all $0 \leq i < j \leq n$

- This seems hard... let's try to simplify it

- **Easier Problem**: given a number T,
  can we determine if max $\text{avg}(A[i] + \ldots + A[j]) \geq T$?

- **Q**: Why might that help?
- **A**: Binary search!

Maximum Sub-array Average

> **Problem**: Given an array A of integers,
> find max $(A[i] + \ldots + A[j]) / (j - i + 1)$ over all $0 \le i < j \le n$

> **Observation**:

$$
\begin{array}{ll}
(A[i] + \ldots + A[i+k-1]) / k \ge T & \text{iff} \\
A[i] + \ldots + A[i+k-1] \ge Tk & \text{iff} \\
A[i] + \ldots + A[i+k-1] - Tk \ge 0 & \text{iff} \\
(A[i] - Tk/k) + \ldots + (A[i+k-1] - Tk/k) \ge 0 & \text{iff} \\
(A[i] - T) + \ldots + (A[i+k-1] - T) \ge 0 &
\end{array}
$$

Maximum Sub-array Average

- **Problem**: Given an array A of integers, find max $(A[i] + \ldots + A[j]) / (j - i + 1)$ over all $0 \le i < j \le n$

- **Observation**:

  $(A[i] + \ldots + A[j]) / (j{-}i{+}1) \ge T$ iff
  $(A[i] - T) + \ldots + (A[j] - T) \ge 0$

  - max sub-array average $\ge T$ iff max sub-array sum of B ≥ 0, where $B[i] = A[i] - T$

Maximum Sub-array Average

- **Problem**: Given an array A of integers,
  find max $(A[i] + \ldots + A[j]) / (j - i + 1)$ over all $0 \le i < j \le n$

- **Observation**: max sub-array average $\ge T$ iff
  max sub-array sum of $B \ge 0$, where $B[i] = A[i] - T$

- **Solution**: use binary search to find the maximum average (T)
  - we can stop when $|b - a| < 1/n$
  - can use repeated doubling to find an upper bound
  - running time of $\Theta(n (\log n)^2)$ using previous algorithm
    - will later see now to solve max sub-array sum in $\Theta(n)$ time