Divide-and-Conquer

(Merge Sort, Quick Sort, Max sub-array)

Divide-and-Conquer

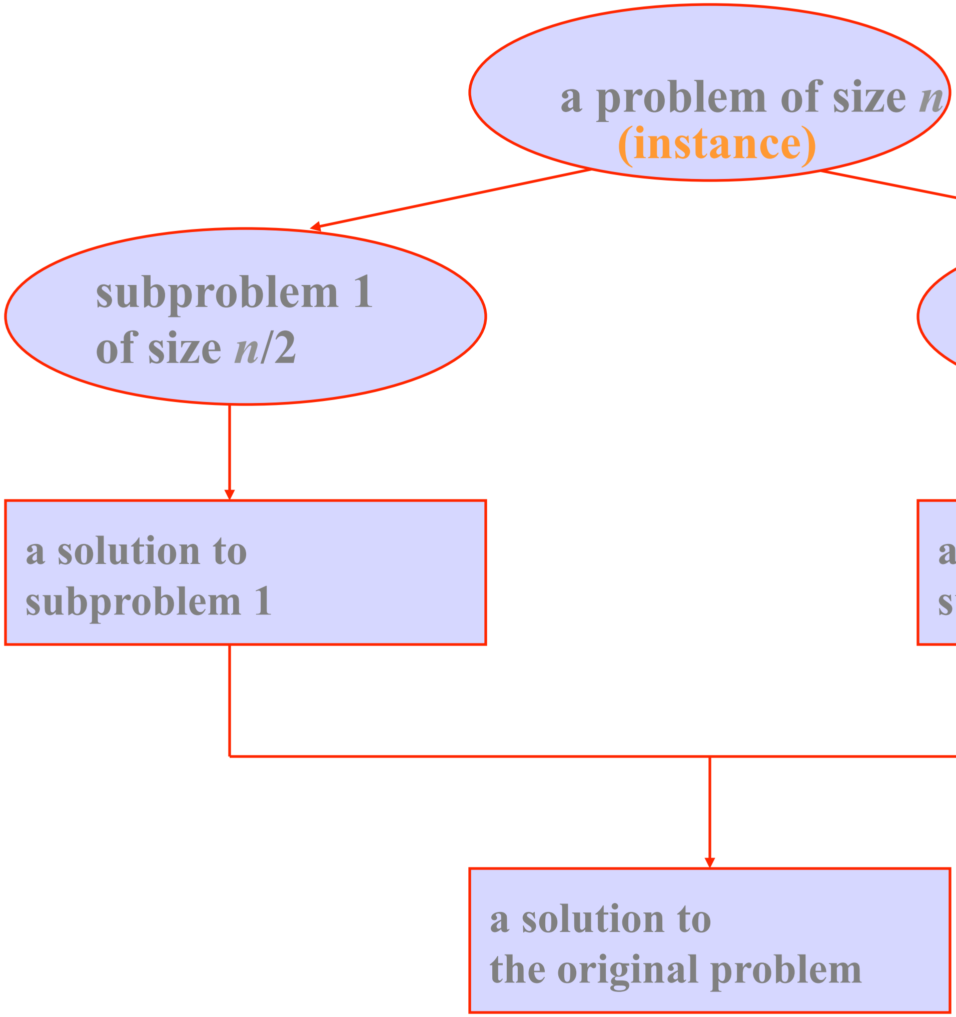
The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances

2. Solve smaller instances recursively

3. Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer Technique



It general leads to a recursive algorithm!

Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Binary search
- Max Sub-array problem
- Matrix multiplication: Strassen’s algorithm

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

<u>Master Theorem</u>: If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ.

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$ $\Theta(n\hat{}2)$

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$ $\Theta(n\hat{}2log\ n)$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$ $\Theta(n\hat{}3)$

Mergesort

- Split array A[0..n-1] into about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Pseudocode of Mergesort

ALGORITHM *Mergesort*($A[0..n-1]$)

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
  copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
  copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
  Mergesort($B[0..\lfloor n/2 \rfloor - 1]$)
  Mergesort($C[0..\lceil n/2 \rceil - 1]$)
  Merge(B, C, A)

Pseudocode of Merge

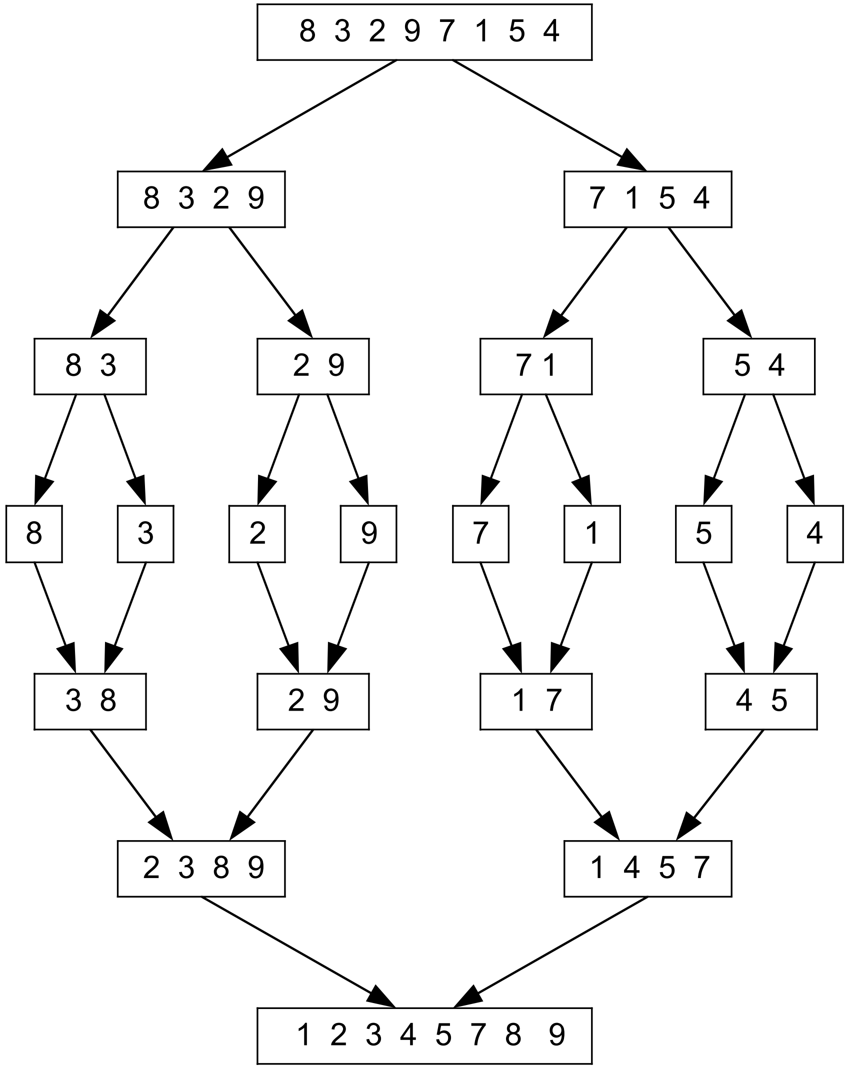
ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C

$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
  if $B[i] \leq C[j]$
    $A[k] \leftarrow B[i];\ i \leftarrow i+1$
  else $A[k] \leftarrow C[j];\ j \leftarrow j+1$
  $k \leftarrow k+1$
if $i = p$
  copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Time complexity: $\boldsymbol{\Theta(p+q) = \Theta(n)}$ comparisons

Mergesort Example



The non-recursive version of Mergesort starts from merging single elements into sorted pairs.

Analysis of Mergesort

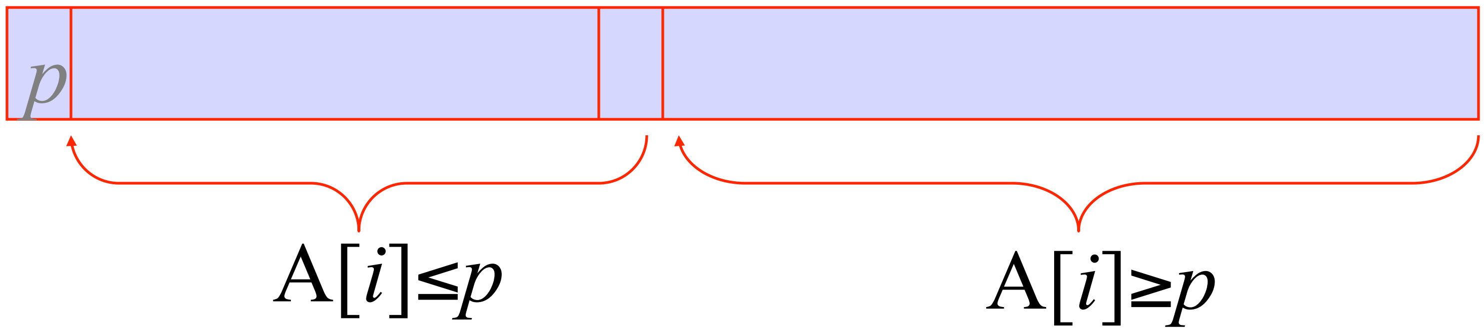
$$T(n) = 2T(n/2) + \Theta(n),\ T(1) = 0$$

- All cases have same efficiency: $\Theta(n \log n)$
- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:

$$\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n$$

- Space requirement: $\Theta(n)$ (not in-place)
- Can be implemented without recursion (bottom-up)

Quicksort



- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the elements in the remaining *n-s* positions are larger than or equal to the pivot (see next slide for an algorithm)

Partitioning Algorithm

```
Algorithm Partition(A[l..r])
//Partitions a subarray by using its first element as a pivot
//Input: A subarray A[l..r] of A[0..n − 1], defined by its left and right
//         indices l and r (l < r)
//Output: A partition of A[l..r], with the split position returned as
//          this function's value
p ← A[l]
i ← l;   j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])   //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

or $i > r$

or $j = l$

Time complexity: **Θ(*r*-*l*)** comparisons

Quicksort Example

5 3 1 9 8 2 4 7

2 3 1 4 5 8 9 7

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

Analysis of Quicksort

- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array! — $\Theta(n^2)$
- Average case: random arrays — $\Theta(n \log n)$
- Improvements:
  - better pivot selection: median of three partitioning
  - switch to insertion sort on small subfiles
  - elimination of recursion

  These combine to 20-25% improvement
- Considered the method of choice for internal sorting of large files ($n \geq 10000$)

Binary Search

Very efficient algorithm for searching in sorted array:

$$K$$

vs

$$A[0] \ . \ . \ . \ A[m] \ . \ . \ . \ A[n\text{-}1]$$

If $K = A[m]$, stop (successful search); otherwise, continue searching by the same method in $A[0..m\text{-}1]$ if $K < A[m]$ and in $A[m+1..n\text{-}1]$ if $K > A[m]$

$l \leftarrow 0$; $r \leftarrow n\text{-}1$
while $l \leq r$ do
    $m \leftarrow \lfloor (l+r)/2 \rfloor$
    if $K = A[m]$ return m
    else if $K < A[m]$ $r \leftarrow m\text{-}1$
    else $l \leftarrow m+1$
return -1

Analysis of Binary Search

- Time efficiency
  - worst-case recurrence: $C_w(n) = 1 + C_w(\lfloor n/2 \rfloor)$, $C_w(1) = 1$
    solution: $C_w(n) = \lceil \log_2(n+1) \rceil$

    This is VERY fast: e.g., $C_w(10^6) = 20$

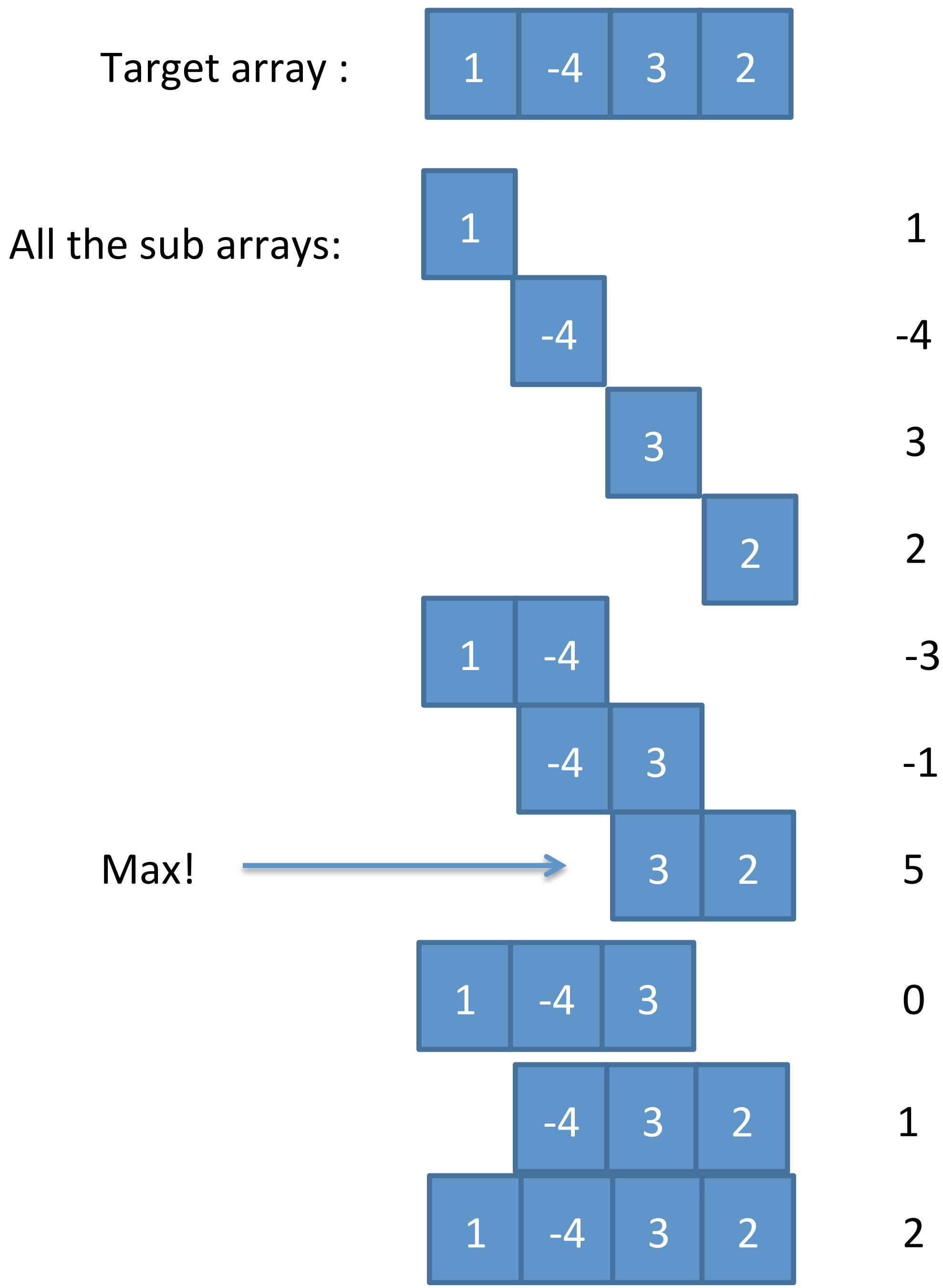
- Optimal for searching a sorted array

- Limitations: must be a sorted array (not linked list)

- Bad (degenerate) example of divide-and-conquer
  because only one of the sub-instances is solved

What is a maximum subarray?

The subarray with the largest sum



What is the brute-force T(n)?

$O(n^2)$

Maximum-subarray

- Can we do it in a divide-and-conquer manner?
- Yes, we can

Target array :

All the sub arrays:

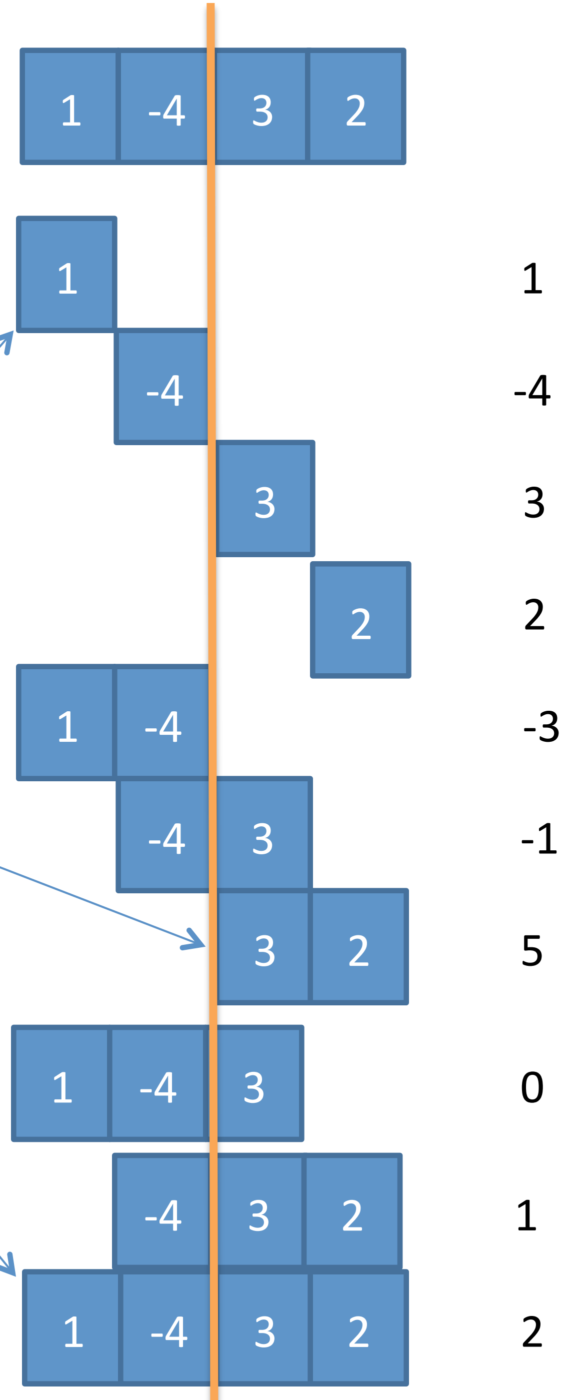
The problem can be then solved by:

1. Find the max in left sub arrays

2. Find the max in right sub arrays

3. Find the max in crossing sub arrays

4. Choose the largest one from those 3 as the final result



Divide target array into 2 arrays

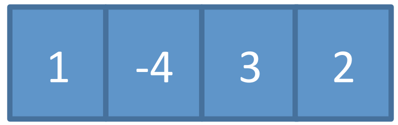
We then have 3 types of subarrays:

The ones belong to the left array

The ones belong to the right array

The ones crossing the mid point

How to find the max subarray crossing the midpoint?



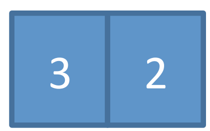
Find the left part

Sum=-4

Sum=-3 largest

Find the right part

Sum=3



Sum=5 largest

The largest crossing subarray is :

The whole algorithm

FindMaxSub ([1 | -4 | 3 | 2])

1. Find the max in left sub arrays **FindMaxSub** ([1 | -4])

2. Find the max in right sub arrays **FindMaxSub** ([3 | 2])

3. Find the max in crossing sub arrays

   Scan [1 | -4] once, and scan [3 | 2] once

4. Choose the largest one from those 3 as the final result

Maximum-subarray problem – divide-and-conquer algorithm

- Input: array A[i, …, j]
- Ouput: sum of maximum-subarray

- **FindMaxSubarray**:

1. if(j<=i) return (A[i]);
2. mid = floor(i+j);
3. sumLeft = **FindMaxSubarray**(A, i, mid);
4. sumRight = **FindMaxSubarray**(A, mid+1, j);
5. sumCross = **FindMaxCrossingSubarray**(A, i, j, mid);
6. Return the largest one from those 3

Time complexity? $T(n)=2T(n/2)+\Theta(n)$ $O(n lg n)$

Maximum-subarray problem – divide-and-conquer algorithm

FindMaxCrossingSubarray(A, i, j, mid)

1. Scan A[i, mid] once, find the largest A[left, mid]

2. Scan A[mid+1, j] once, find the largest A[mid +1, right]

3. Return (sum of A[left, mid] and A[mid+1, right])

Maximum-subarray problem – divide-and-conquer algorithm

- Input: array A[i, …, j]
- Ouput: sum of maximum-subarray, start point of maximum-subarray, end point of maximum-subarray
- **FindMaxSubarray**:

1. if(j<=i) return (A[i], i, j);
2. mid = floor(i+j);
3. (sumCross, startCross, endCross) = **FindMaxCrossingSubarray**(A, i, j, mid);
4. (sumLeft, startLeft, endLeft) = **FindMaxSubarray**(A, i, mid);
5. (sumRight, startRight, endRight) = **FindMaxSubarray**(A, mid+1, j);
6. Return the largest one from those 3

Time complexity? $T(n)=2T(n/2)+\Theta(n)$ $O(nlgn)$

Maximum-subarray problem – divide-and-conquer algorithm

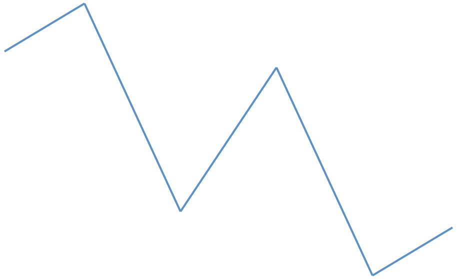
FindMaxCrossingSubarray(A, i, j, mid)

1. Scan A[i, mid] once, find the largest A[left, mid]
2. Scan A[mid+1, j] once, find the largest A[mid+1, right]
3. Return (sum of A[left, mid] and A[mid+1, right], left, right)

Maximum Subarray (Example)

- You can buy a unit of stock, only *one* time, then sell it at a later date
  - Buy/sell at end of day
- Strategy: buy low, sell high
  - The lowest price may appear after the highest price
- Assume you know future prices
- Can you maximize profit by buying at lowest price and selling at highest price?

Buy lowest sell highest



Transformation

- Find sequence of days so that:
  - the net change from last to first is maximized
- Look at the daily change in price
  - Change on day *i*: price day *i* minus price day *i*-1
  - We now have an array of changes (numbers), e.g.

    12,-3,-24,20,-3,-16,-23,18,20,-7,12,-5,-22,14,-4,6
  - Find contiguous subarray with largest sum
    - **maximum subarray**
  - E.g.: buy after day 7, sell after day 11

Time analysis (Divide & Conquer)

- Find-Max-Cross-Subarray: $O(n)$ time

- Two recursive calls on input size $n/2$

- Thus:

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

  $T(n) = O(n \log n)$