CS 457, Fall 2016

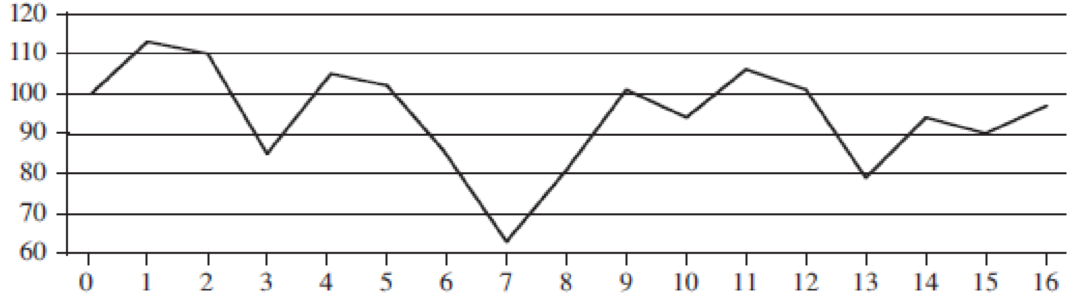
Drexel University, Department of Computer Science

Lecture 6

Maximum Subarray Problem

- Input: $\boldsymbol{n}$ price points
- Output: $(\boldsymbol{t_b}, \boldsymbol{t_s})$ s.t. $\mathbf{0} \leq \boldsymbol{t_b} < \boldsymbol{ts} \leq \boldsymbol{n}$, and $\boldsymbol{p(t_s)} - \boldsymbol{p(t_b)}$ is maximized

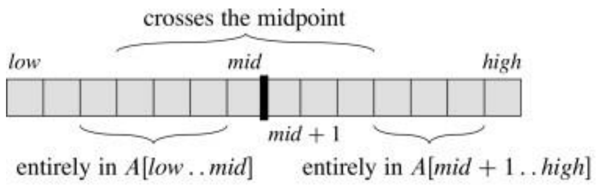
1. Brute force running time?
2. Improvements?
3. Divide-and-Conquer?



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Maximum Subarray Problem

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
// Find a maximum subarray of the form A[i..mid].
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
// Find a maximum subarray of the form A[mid + 1..j].
right-sum = −∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
// Return the indices and the sum of the two subarrays.
return (max-left, max-right, left-sum + right-sum)
```



(a)

Maximum Subarray Problem

Divide-and-conquer procedure for the maximum-subarray problem

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])                // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Initial call: FIND-MAXIMUM-SUBARRAY(A, 1, n)

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

Today's Lecture

- Quicksort
  - Average Case Running Time
- Heapsort

Quicksort

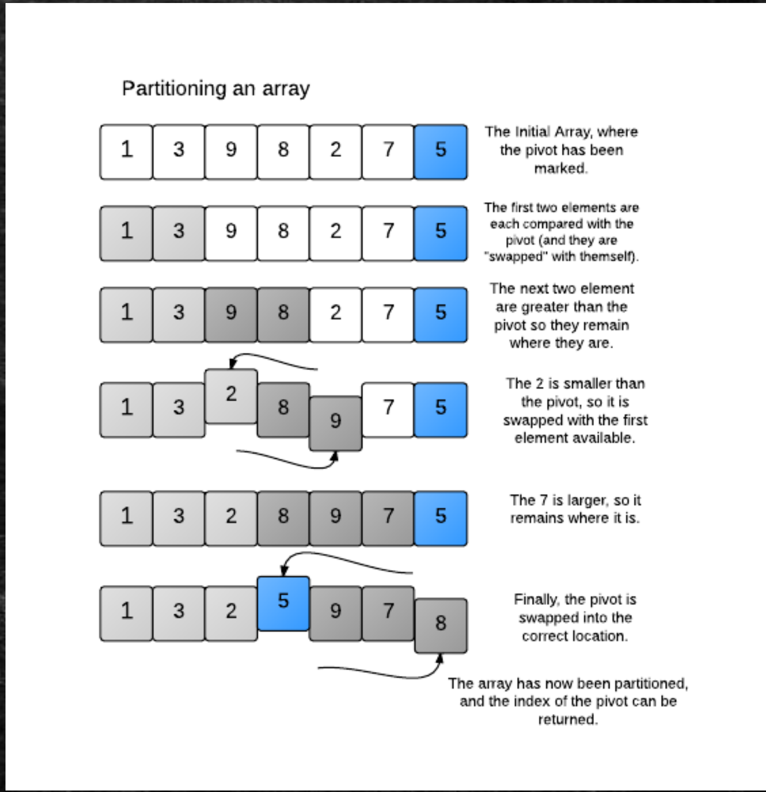
QUICKSORT (*A*, *p*, *r*)

```
if p < r                              // Check for base case
    q = PARTITION(A, p, r)            // Divide step
    QUICKSORT (A, p, q - 1)           // Conquer step.
    QUICKSORT (A, q + 1, r)           // Conquer step.
```

Quicksort

PARTITION (*A*, *p*, *r*)

```
x = A[r]
i = p – 1
for  j = p  to  r – 1
        if  A[ j] <= x
                i = i + 1
                exchange A[i] with A[j]
exchange A[i+1] with A[r]
return i+1
```



Quicksort (Running Time)

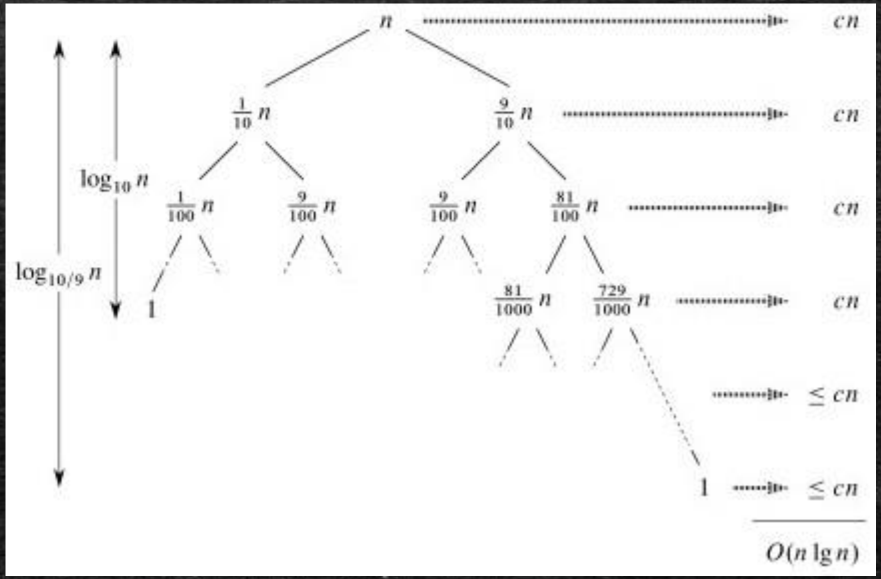
QUICKSORT (A, p, r)

1. **if** $p < r$ // Check for base case
2. q = PARTITION(A, p, r) // Divide step
3. QUICKSORT (A, p, q - 1) // Conquer step.
4. QUICKSORT (A, q + 1, r) // Conquer step.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(q) + T(n - q - 1) + \Theta(n) & \text{otherwise} \end{cases}$$

Quicksort (Running Time)

- Even if $q \approx \frac{1}{10}$



Quicksort (Running Time)

QUICKSORT (A, p, r)

1. **if** $p < r$ // Check for base case
2. q = PARTITION(A, p, r) // Divide step
3. QUICKSORT (A, p, q - 1) // Conquer step.
4. QUICKSORT (A, q + 1, r) // Conquer step.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(q) + T(n-q-1) + \Theta(n) & \text{otherwise} \end{cases}$$

Quicksort (Running Time)

QUICKSORT (A, p, r)

```
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT (A, p, q - 1)
    QUICKSORT (A, q + 1, r)
```

PARTITION (A, p, r)

```
1. x = A[r]
2. i = p – 1
3. for j = p to r – 1
4.     if A[ j] <= x
5.         i = i + 1
6.         exchange A[i] with A[j]
7. exchange A[i+1] with A[r]
8. return i+1
```

<u>Lemma</u>:
Let $\boldsymbol{X}$ be the number of comparisons performed in line 4 of PARTITION over the entire execution of QUICKSORT on an n-element array.
Then the running time of QUICKSORT is $\boldsymbol{O(n+X)}$.

Quicksort (Running Time)

- Denote the sorted elements of the array by $z_1, z_2, \dots, z_n$
- Let $X_{ij} = I\{z_i \text{ is compared to } z_j\}$
- Then, $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$ and $E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared to } z_j\}$

- Let $Z_{ij} = \{z_i, z_{i+1}, \dots, z_j\}$
- $Pr\{z_i \text{ is compared to } z_j\} = Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\} = \frac{2}{j-i+1}$
- So, $E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-1} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log n) = O(n \log n)$

Insertion Sort (Running Time)

INSERTION_SORT (*A*)		COST	TIMES
for j = 2 **to** A.length	--------->	c_1	n
key = *A*[*j*]	--------->	c_2	n-1
// Insert *A*[*j*] into the sorted sequence *A*[1 .. *j* – 1].	--------->	$c_3=0$	n-1
i = *j* – 1	--------->	c_4	n-1
while *i* > 0 and *A*[*i*] > key	--------->	c_5	$\sum_{j=2}^{n} t_j$
A[*i* +1] = *A*[*i*]	--------->	c_6	$\sum_{j=2}^{n}(t_j-1)$
i = *i* – 1	--------->	c_7	$\sum_{j=2}^{n}(t_j-1)$
A[*i* + 1] = key	--------->	c_8	n-1

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1)$$

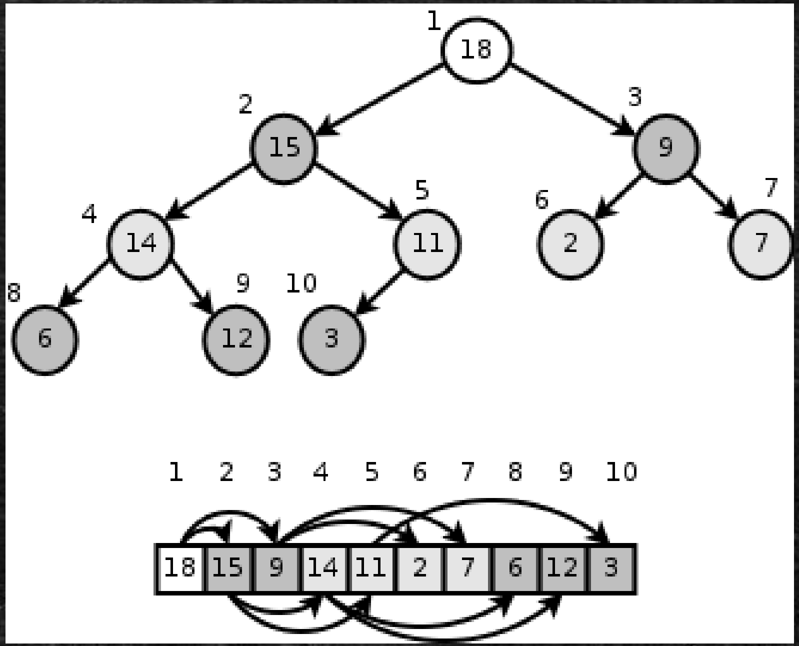
– $t_j \geq 1$ so $\sum_{j=2}^{n} t_j \geq n-1$ and $T(n) \geq (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$

– $t_j \leq j$ so $\sum_{j=2}^{n} t_j \leq \frac{n(n+1)}{2} - 1$ and $T(n) \leq C_1 n^2 + C_2 n + C_3$

Heapsort

Heap data structure:

PARENT (i)
    return $\lfloor i/2 \rfloor$
LEFT (i)
    return $2i$
RIGHT (i)
    return $2i + 1$



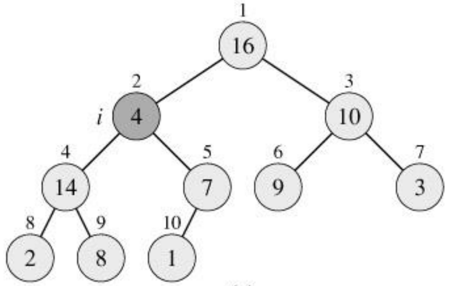
Max-heap property: $A[\text{PARENT}(i)] \geq A[i]$

Heapsort

Max-Heapify (A, *i*)

```
l = left [i]
r = right [i]
if  l ≤ A.heap-size and A[l] > A[i]
        largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
if largest  ≠ i
        exchange A[i] with A[largest]
        Max-Heapify (A, largest)
```

Call to **Max-Heapify (A,2)**

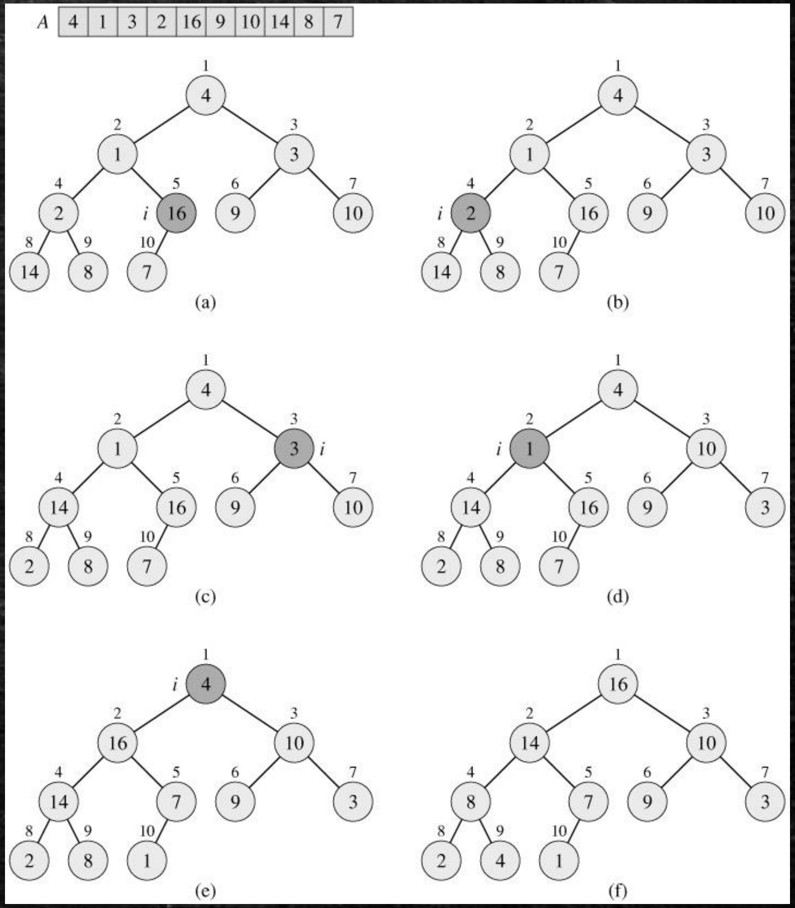


(a)

Heapsort

Build-Max-Heap (A)

A.heap-size = A.length
for $i = \lfloor \text{A.length}/2 \rfloor$ **down to** 1
    Max-Heapify(A, i)



Heapsort

Build-Max-Heap (A)

1. A.heap-size = A.length
2. **for** $i = \lfloor \text{A.length}/2 \rfloor$ **down to** 1
3. Max-Heapify(A, i)

<u>Loop Invariant:</u>

At the beginning of each iteration of the loop of lines 2-3, each node

$$i+1,\ \ i+2,\ \ \ldots\ ,\ \ n,$$

is the root of a max-heap.

Heapsort

Heapsort(A)

```
Build-Max-Heap(A)
for i = A.length down to 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size - 1
    Max-Heapify(A,1)
```