COT 6405 Introduction to Theory of Algorithms

Topic 11. Order Statistics

Order statistic

- The *i*-th order statistic in a set of *n* elements is the *i*-th smallest element
  - The *minimum* is thus the 1st order statistic
  - The *maximum* is the *n*-th order statistic
  - The *median* is the *n*/2 order statistic
    - If *n* is even, we have 2 medians: <u>lower median</u> n/2 and <u>upper median</u> n/2+1
    - By our convention, “median” normally refers to the lower median

How to calculate

- How can we calculate order statistics?
- What is the running time?
  - Simple method: Sort first, e.g., Heapsort O(n lg n)
  - then return the i-th element

Find the minimum

- How many comparisons are needed to find the minimum element in a set? Or the maximum?

```
MINIMUM(A)
  min=A[1]
  for i=2 to A.length
        if min > A[i]
              min = A[i]
  return min
```

Find both the minimum & the maximum

- We can find the minimum with n-1 comparisons
- We can find the maximum with n-1 comparisons
- So we can find both the minimum and the maximum with 2(n-1) comparisons

Can we reduce the cost?

- Can we find the minimum and maximum with less than twice the cost, 2(n-1) ?
- Yes: walk through elements by pairs
  - Compare each element in pair to the other
  - Compare the larger one to maximum, the smaller one to minimum
- Total cost: 3 comparisons per 2 elements = O(3n/2)

Finding order statistics: The Selection Problem

- A more interesting problem is the selection problem
  - finding the *i*-th smallest element of a set
- A naïve way is to sort the set
  - Running time takes O(nlgn)
- We will study a practical randomized algorithm with O(n) expected running time
- We will then study an algorithm with O(n) worst-case running time

5	2	7	4	3	9	8	6

Find the 3rd smallest element

We first partition this array. Assume pivot = 6, after partition

5	2	4	3	6	7	9	8

4 elements are smaller than the pivot and 3 are larger than the Pivot. So the pivot is the 5th smallest element of the original array

The index of the pivot is 5, and 3 is less than 5. Hence, the 3rd smallest element of the original array is indeed the 3rd smallest element of the first subarray after partition

5	2	4	3

5	2	7	4	3	9	8	6

Find the 7th smallest element

We first partition this array. Assume pivot = 6, after partition

5	2	4	3	6	7	9	8

4 elements are smaller than the pivot and 3 are larger than the Pivot. So the pivot is the 5th smallest element of the original array

The index of the pivot is 5, and 7 is larger 5. Hence, the 7th smallest element of the original array is indeed the 2ed smallest element of the second subarray after partition

7	9	8

Randomized Selection

- Key idea: use partition() from Quicksort
  - But, only need to examine one subarray
  - This savings shows up in running time: O(n)
- We will again use a randomized partition

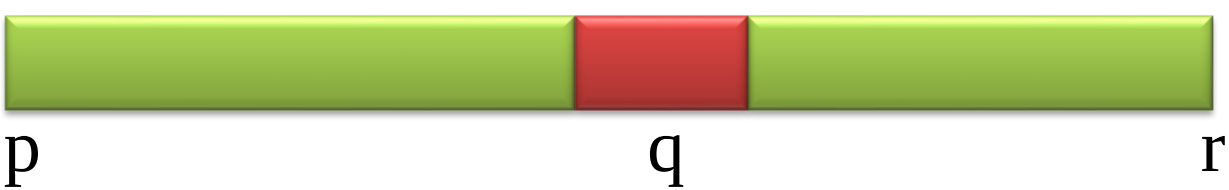
q = RANDOMIZED-PARTITION*(A, p, r)*

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

$i \leftarrow$ RANDOM*(p, r)*

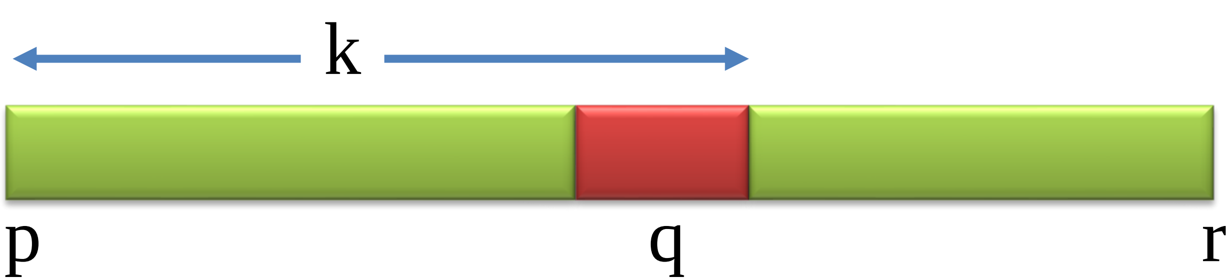
exchange $A[r] \leftrightarrow A[i]$

return PARTITION*(A, p, r)*



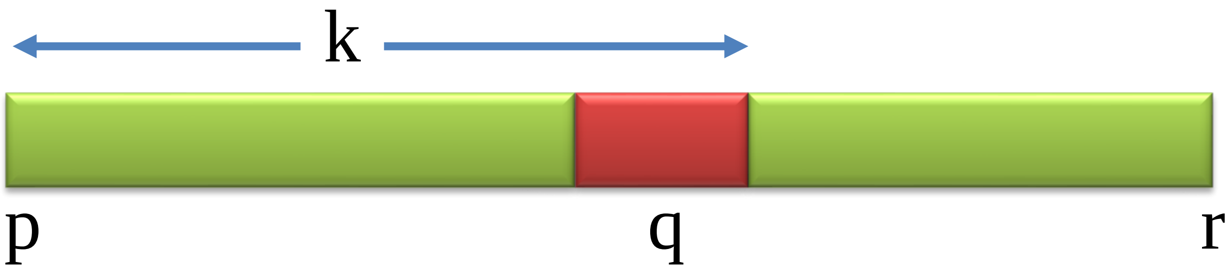
Randomized Selection

```
RandomizedSelect(A, p, r, i)
    if (p == r) then return A[p];
    q = RandomizedPartition(A, p, r)
    k = q - p + 1;
    if (i == k) then return A[q];
    if (i < k) then
        return RandomizedSelect(A, p, q-1, ?);
    else
        return RandomizedSelect(A,q+1,r, ?? );
```



Randomized Selection

```
RandomizedSelect(A, p, r, i)
    if (p == r) then return A[p];
    q = RandomizedPartition(A, p, r)
    k = q - p + 1;
    if (i == k) then return A[q];
    if (i < k) then
        return RandomizedSelect(A, p, q-1, i);
    else
        return RandomizedSelect(A,q+1,r, i-k);
```



Analyzing Randomized-Select()

- Worst case: partition always 0:n-1
  - $T(n) = T(n-1) + O(n) = O(n^2)$
  - No better than sorting!
- "Best" case: suppose a 9:1 partition
  - $T(n) \leq T(9n/10) + O(n) = O(n)$ (why?)
  - Master Theorem, case 3
  - Better than sorting!

Average case analysis

- We can upper-bound the time needed for the recursive call by the time needed for the recursive call on the largest possible input
- In other words, to obtain an upper bound, we assume that the i-th element is always on the side of the partition with the greater number of elements

Average case analysis (cont'd)

- We have a total of *n* partition outcome, and k can range between 1 and *n.* Therefore, the expected average case time T(*n*) is

$$T(n) \leq \frac{1}{n}\sum_{k=1}^{n} T(\max(k-1, n-k)) + O(n)$$

Average case analysis (cont'd)

- If n is even, T($\lceil n/2 \rceil$) up to T(n-1) appears exactly twice.
  - E.g., n = 4, T(n) $=$ 1/4(T(max(0, 3)) + T(max(1, 2)) + T(max(2, 1)) + T(max(3, 0)) = 2/4 (T(3) + T(2))
- If n is odd, all these terms appear twice and T($\lfloor n/2 \rfloor$) appears once
  - E.g., n = 5, T(n) $=$ $1/5$(T(max(0, 4)) + T(max(1, 3)) + $\boxed{\text{T(max(2, 2))}}$ + T(max(3, 1)) + T(max(4, 0))
    = 2/4 (T(4) + T(3)) + T(2)

Average case analysis (cont'd)

$$T(n) \le \frac{1}{n}\sum_{k=1}^{n} T(\max(k-1, n-k)) + O(n)$$

n is even

$$= \frac{2}{n}\sum_{k=\lceil n/2\rceil}^{n-1} T(k) + O(n)$$

$$\le \frac{2}{n}\sum_{k=\lfloor n/2\rfloor}^{n-1} T(k) + O(n)$$

Average case analysis (cont'd)

$$T(n) \le \frac{1}{n}\sum_{k=1}^{n} T(\max(k-1, n-k)) + O(n)$$

n is odd

$$= \frac{2}{n}\sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + \frac{1}{n}T(\lfloor n/2 \rfloor) + O(n)$$

$$\le \frac{2}{n}\sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + \frac{2}{n}T(\lfloor n/2 \rfloor) + O(n)$$

$$= \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} T(k) + O(n)$$

Average case analysis (cont'd)

- Use substitution method: Assume T(k) $\leq ck$, for sufficiently large c
- $T(n) \leq \frac{2}{n} \sum_{k=\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} ck + an$

$$= \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an$$

$$= \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{\left(\left\lfloor \frac{n}{2} \right\rfloor - 1\right)\left\lfloor \frac{n}{2} \right\rfloor}{2} \right) + an$$

Average case analysis (cont'd)

- T(n) $\leq \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{\left(\left\lfloor\frac{n}{2}\right\rfloor-1\right)\left\lfloor\frac{n}{2}\right\rfloor}{2}\right) + an$

$$\leq \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{\left(\frac{n}{2}-2\right)\left(\frac{n}{2}-1\right)}{2}\right) + an$$

$$= \frac{2c}{n}\left(\frac{n^2-n}{2} - \frac{\frac{n^2}{4}-\frac{3n}{2}+2}{2}\right) + an$$

$$= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + an$$

Average case analysis (cont'd)

- T(n) $\leq \frac{c}{n}(\frac{3n^2}{4} - \frac{n}{2} - 2)$ + *an*

$=c(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n})$ + *an*

$\leq \frac{3cn}{4} + \frac{c}{2} + an$

$\leq \frac{3cn}{4} + \frac{n}{2} + an$ when $n \geq c$

$= cn - (\frac{cn}{4} - \frac{n}{2} - an)$

$\leq cn$ when

$\frac{cn}{4} - \frac{n}{2} - an \geq 0 -> c \geq 4a + 2$

Worst-Case Linear-Time Selection

- Randomized selection algorithm works well in practice
- We now examine a selection algorithm whose running time is O(n) in the worst case.

Worst-Case Linear-Time Selection

- The worst-case happens when a 0:n-1 split is generated. Thus, to achieve O(n) running time, we *guarantee* a good split upon partitioning the array.
- Basic idea:
  – Generate a good partitioning element

Selection algorithm

1. Divide *n* elements into groups of 5
2. Find median of each group (How? How long?)
3. Use Select() recursively to find median *x* of the $\lceil n/5 \rceil$ medians
4. Partition the *n* elements around *x*. Let *k* = rank(*x*)
5. **if** (i == k) **then** return x

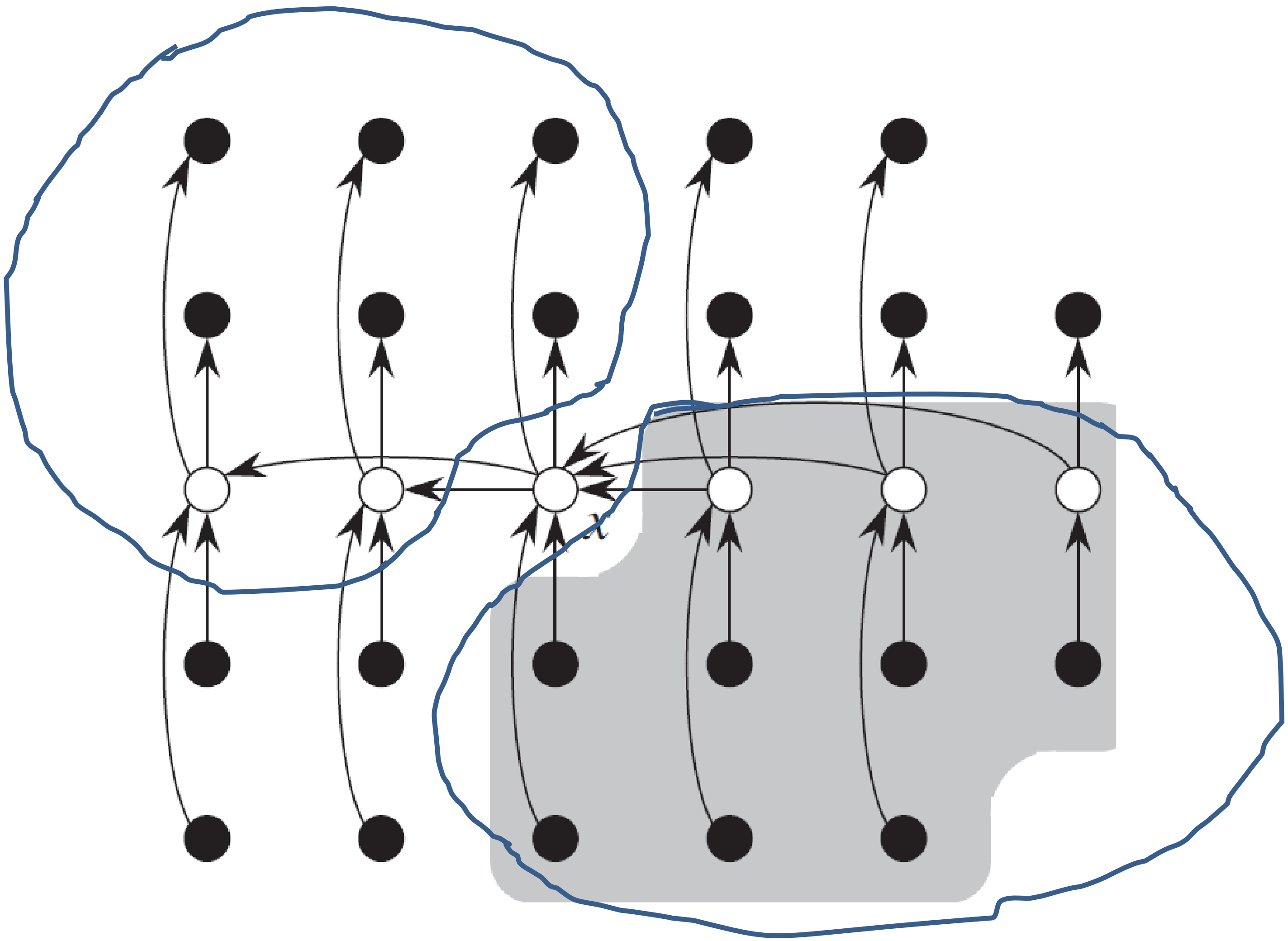
   if (i < k) **then**

   use Select() recursively to find *i*-th smallest element in the low side of the partition

   else

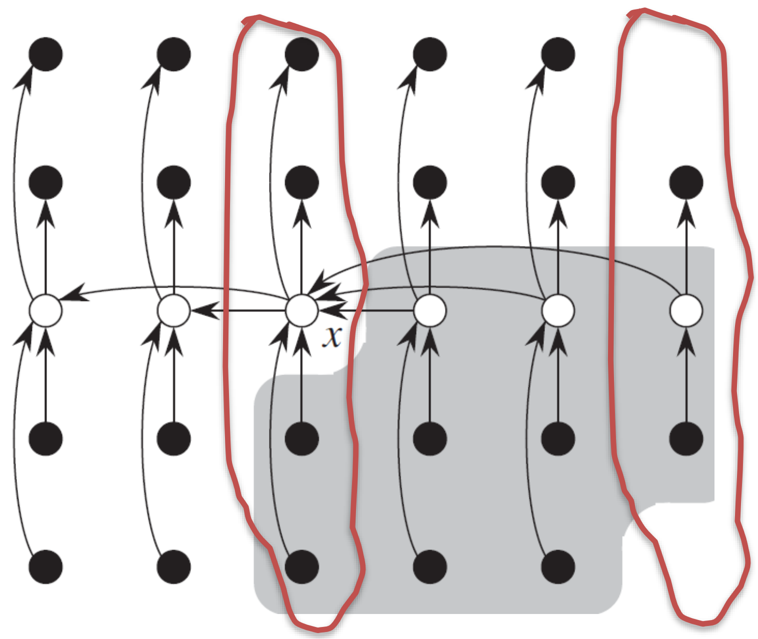
   (i > k) use Select() recursively to find (*i*-*k*)-th smallest element in the high side of the partition

Example



Running time analysis

- At least half of the $\lceil n/5 \rceil$ groups contribute at least 3 elements that are greater than x,
  - except for the one group that has fewer than 5 elements, and the one group containing x itself



Running time analysis (Cont'd)

- The number of elements greater than x is at least

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

- At most $\frac{7n}{10} + 6$ elements are less than x. This means that, in the worst case, step 5 calls SELECT recursively on at most $\frac{7n}{10} + 6$ elements.

Running time analysis (cont'd)

- Step 1 takes O(n) time
- Step 2 consists of O(n) calls of insertion sort on sets of size O(1)
- Step 3 takes time T($\lceil n/5 \rceil$)
- Step 4 takes O(n) time
- Step 5 takes time at most T(7n/10 + 6)

1. Divide *n* elements into groups of 5
2. Find median of each group (How? How long?)
3. Use Select() recursively to find median *x* of the $\lceil n/5 \rceil$ medians
4. Partition the *n* elements around *x*. Let *k* = rank(*x*)
5. **if** (i == k) **then** return x

   if (i < k) **then**

   use Select() recursively to find *i*-th smallest element in the low side of the partition

   else

   (i > k) use Select() recursively to find (*i*-*k*)-th smallest element in the high side of the partition

Running time analysis (cont'd)

- We can therefore obtain the recurrence
- $T(n) \leq T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$
- Assume $T(k) \leq ck$ for $k < n$, use the substitution method
- $T(n) \leq c\lceil n/5 \rceil + c(7n/10 + 6) + an$

$$\leq cn/5 + c + 7cn/10 + 6c + an$$

$$= 9cn/10 + 7c + an$$

$$= cn + (-cn/10 + 7c + an)$$

Running time analysis (cont'd)

- $T(n) \leq cn + (-cn/10 + 7c + an)$

  $\leq cn + (-cn/10 + 7n + an)$ when $n \geq c$
- Which is at most cn if
  - $-cn/10 + 7n + an \leq 0$ when $c \geq 70 + 10a$

Worst-case Quicksort

- Worst-case O(n lg n) quicksort
  - Find median *x* and partition around it
  - Recursively quicksort two halves
  - $T(n) = 2T(n/2) + O(n) = O(n \lg n)$

Worst-case Quicksort (Cont'd)

```
Quicksort(A, p, r)
{  if (p < r)
    {
        q = Median-Partition(A, p, r);
        Quicksort(A, p, q-1);
        Quicksort(A, q+1, r);
    }
}
Median-Partition(A, p, r)
{       median = ⌈(p-r+1)/2⌉;
        x = Select(A, p, r, median);
        i = rank(x);
        exchange A[r] and A[i]
        return Partition(A, p, r)
}
```

Summary

- Selection() does not require assumptions on the input
  - Do not need to sort the whole array, then pick i-th element
  - Counting/Radix/Bucket sort assume certain inputs