Computational Intelligence on Automation Lab @ NCTU

ECM5605(5075) S'20 Algorithms

Lecture 03:
Linear Time Sorting, Order Statistics and Quicksort (Reprise)

Review of Heapsort and Quicksort

- Heapsort
  - What is Max-Heap property ?
  - Max-Heapify() and Build-Max-Heapify()
  - Heapsort algorithm and its complexity
- Quicksort
  - What're important properties of Quicksort ?
  - What're the key ideas of Quicksort ?
  - What're two key factors to decide its performance ?
  - Best-case? Worst-case? Average-case?

Gnome Sort (Stupid Sort)

- Gnome (stupid) sort is the simplest sort algorithm
  - proposed by Dr. Hamid Sarbazi-Azad in 2000
  - not recursive and only one loop

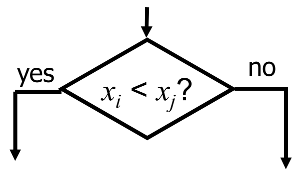
```
StupidSort(array A, integer n)
    i ← 0;
    while ( i < n ) do
       if ( i=0 || A[i]>=A[i-1])
         i ← i+1;
       else
         swap( A[i], A[i-1] );
         i ← i-1;
```

Current array	pos	Condition in effect	Action to take
[**5**, 3, 2, 4]	0	pos == 0	increment pos
[5, **3**, 2, 4]	1	a[pos] < a[pos-1]	swap, decrement pos
[**3**, 5, 2, 4]	0	pos == 0	increment pos
[3, **5**, 2, 4]	1	a[pos] ≥ a[pos-1]	increment pos
[3, 5, **2**, 4]	2	a[pos] < a[pos-1]	swap, decrement pos
[3, **2**, 5, 4]	1	a[pos] < a[pos-1]	swap, decrement pos
[**2**, 3, 5, 4]	0	pos == 0	increment pos
[2, **3**, 5, 4]	1	a[pos] ≥ a[pos-1]	increment pos
[2, 3, **5**, 4]	2	a[pos] ≥ a[pos-1]	increment pos:
[2, 3, 5, **4**]	3	a[pos] < a[pos-1]	swap, decrement pos
[2, 3, **4**, 5]	2	a[pos] ≥ a[pos-1]	increment pos
[2, 3, 4, **5**]	3	a[pos] ≥ a[pos-1]	increment pos
[2, 3, 4, 5]	4	pos == length(a)	finished

  - implement your own stupidsort
  - runtime complexity? worst-case? average-case?

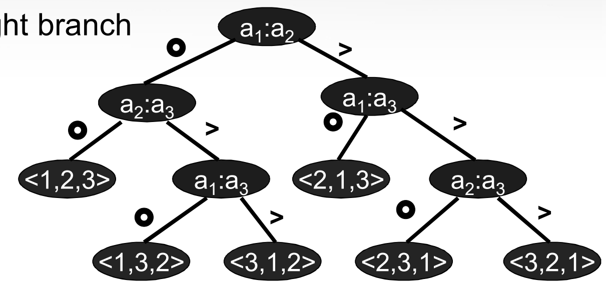
Comparison-based Sorting

- Many sorting algorithms are comparison-based.
  - sort by making comparisons between pairs of objects
  - Examples: *selection sort*, *bubble sort*, *shell sort*, insertion sort, heap sort, merge sort, quick sort, ...
- We can derive a lower bound on the running time of any algorithm that uses comparisons to sort *n* elements, $x_1, x_2, \ldots, x_n$.



Counting Comparisons

- Let us just count comparisons then.
- Each possible run of the algorithm corresponds to a root-to-leaf path in a ***decision tree***
  - ○ ⇨ left branch
  - **>** ⇨ right branch



- Question: number of leaf nodes ? Answer: **n!**

Lower Bound of Comparison Tree

- Worst-case # of comparisons = # of edges longest path (***height***) in the tree $\mathbf{T}$ of size $\boldsymbol{n}$
  - at most 2^h leaves if $\mathbf{T}$ is binary
  - Because $2^h \geq n! \Rightarrow h \geq \lg(n!)$
- Any comparison-based algorithm takes time at least Why ? Stirling's approximation:

  $h \geq \lg(n!) = \Theta(n\lg n)$ $\quad n! > (\frac{n}{e})^n$

- Therefore, any comparison-based sorting algorithm must run in $\Omega(n\lg n)$ time.
  - Mergesort and heapsort are asymptotically optimal comparison-based sorts

Linear Time Sorts

- Can we sort faster than $\Omega(n\lg n)$?
  - Yes, if we do not ***compare*** items
  - But that also means we need more information about the ***structure*** of items
- Examples of sorting algorithms that do not use comparisons
  - Counting-Sort
  - Radix-Sort
  - Bucket-Sort

Counting-Sort

- Assume that input integers are known in $[\boldsymbol{1..k}]$
- ***Basic idea***:
  - Count (accumulate) # of elements $\leq$ element i
  - Use that # to place i in position k of the new sorted array
- Features: no comparisons!
  - ***Stable***: *2 elements having the same value appear in the same order in the sorted sequence as in the input sequence*
  - Not ***in-place***: $\mathbf{O}(n)$ array to hold sorted output + $\mathbf{O}(k)$ array for scratch storage

Example of Counting-Sort

```
Counting-Sort(A,B,k)
for i ←1 to k do C[i] ← 0
for j ←1 to n do C[A[j]] ← C[A[j]] + 1
for i ←2 to k do C[i] ←C[i] + C[i-1]
for j ←n downto 1 do
      B[C[A[j]]] ← A[j]
      C[A[j]] ← C[A[j]] – 1
```

n=8 input list, A (j ↓ at 7, j ↓ at 8):

1	2	3	4	5	6	7	8
3	6	4	1	3	4	1	4

k=6 counts, B:

1	2	3	4	5	6
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

sorted list, C:

1	1	3	3	4	4	4	6

Runtime complexity? $\mathbf{O}(n+k)$

Radix-Sort

- Assumption: a list of n items; each item has d digits where $d_i \in [1..k]$, $1 \le i \le d$
- ***Basic idea***:
  - Using a stable sort on each digit
  - Start from least-significant bit to most-significant bit (right ⇒ left)
- Pseudocode of Radix-Sort

```
Radix-Sort(A,d,k)
1 for i ←1 to d do
2       stable-sort(A,d,i);
3       /* apply a stable sort to A on d_i */
```

  - Can use Counting-Sort as the stable sort in line 2
  - Worst-case complexity: $\mathbf{O}(d(n+k))$

Proof of Radix Sort

- Sketch of an inductive argument (induction on the number of passes):
  - Assume lower-order digits $\{j: j<i\}$ are sorted
  - Show that sorting next digit i leaves array correctly sorted:

  (case1) If two digits at position i are different, ordering numbers by that digit is correct (lower-order digits irrelevant)

  (case2) If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order

Example of Radix-Sort

```
Radix-Sort(A,d,k)
1 for i ←1 to d do
2       Counting-Sort(A,d,i);
3       /* apply Couting-Sort to A on d_i */
```

3 2 9	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

- Can sort from the most significant bit? No ! But why?

  Not stable! Can you imagine what will happen?

Example of Radix-Sort (cont'd)

- A variation of application for Radix-Sort
  - Sort 2 digits for each card: d_1d_2
  - d_1 = ♠♥♣♦: base 4 and its order: ♦ ≤ ♣ ≤ ♥ ≤ ♠
  - d_2 = A, 2, 3, ...J, Q, K: base 13 and its order: A ≤ 2 ≤ 3 ≤ ... ≤ J ≤ Q ≤ K
  - Therefore, ♦2 ≤ ♣2 ≤ ♥5 ≤ ♠K
- Any other applications??

Summary on Sort (1/2)

Algorithm	runtime: best-case	runtime: avg-case	runtime: worst-case	in-place?
comparison-based sort				
insertion	$O(n)$	$O(n^2)$	$O(n^2)$	**Yes**
merge	$O(n\lg n)$	$O(n\lg n)$	$O(n\lg n)$	**No**
heap	$O(n\lg n)$	$O(n\lg n)$	$O(n\lg n)$	**Yes**
quick	$O(n\lg n)$	$O(n\lg n)$	$O(n^2)$	**Yes**
non-comparison-based sort				
counting	$O(n+k)$	$O(n+k)$	$O(n+k)$	**No**
radix	$O(d(n+k))$	$O(d(n+k))$	$O(d(n+k))$	**No**
bucket	—	$O(n^2)$	—	**No**

- Q: Which are stable sorts?
  ⇨ insertion sort, merge sort, counting sort and radix sort

Summary on Sort (2/2)

- Cool! *Why not always using counting sort?*
  - Because it depends on range k of elements
- Ex: we cannot afford counting sort to sort 32 bit integers because k grows too fast
  - 2^{32} = 4,294,967,296.
- Another problem: how do we sort 1 million 64-bit numbers efficiently? Hint: radix sort
  - Treat as 4-digit radix 2^{16} numbers
  - Can sort in just four passes with radix sort!
  - Quicksort requires approximately lg*n* = 20 operations per number

Order Statistics (Selection Problem)

- Definition: The ***i-th order statistic*** in a set of *n* elements is the *i*-th ***smallest*** element
  - ***Minimum***: 1-st order statistic
  - ***Maximum***: *n*-th order statistic
  - ***Median***: ($\lfloor (n+1)/2 \rfloor$ and $\lceil (n+1)/2 \rceil$) order statistic
- Finding ***i-th*** order statistics ⇨ a.k.a. ***The Selection Problem***
  - **input**: a set ***A*** of ***n*** (distinct) elements and a number ***i***, $1 \le i \le n$
  - output: element $x \in A$ that is larger than exactly (***i-1***) elements of ***A***
- Naïve idea: sort ***A*** into ***A'*** and return ***A'[i]***
  - Time complexity: O(*n*lg*n*) but can we do better?

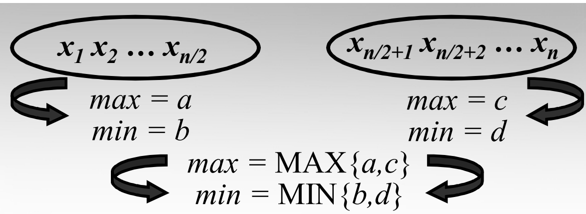
Finding Minimum/Maximum

```
Minimum(A)
1  min ← A[1]
2  for i ←2 to n do
3       if min > A[i]
4       then min ← A[i]
5  return min
```

```
Maximum(A)
1  max ← A[n]
2  for j ←(n-1) downto 1 do
3       if max < A[j]
4       then max ← A[j]
5  return max
```

- This algorithm executes exactly (***n-1***) comparisons
  - Can we do better?
  - Expected # of times executed for line 4: **O**(lg***n***)
    - ⇨ Why and how?
- Naïve simultaneous minimum and maximum requires (***2n-2***) comparisons
  - Can we do better? Yes! The optimal is (***3n/2-2***).

Simultaneous Minimum/Maximum



- T(n): # of comparisons used for n elements

$$T(n) = \begin{cases} 1, & \text{if } n = 2, \\ 2T(n/2) + 2, & \text{if } n > 2. \end{cases}$$

Assume $n = 2^k$

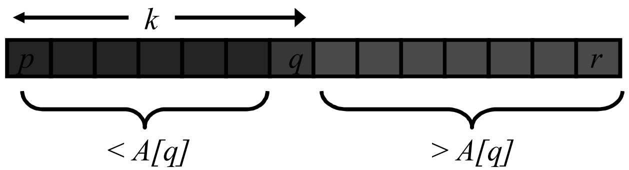
$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 2^{k-1} + 2^k - 2 \\ &= 3n/2 - 2 \quad \square \end{aligned}$$

Finding *i*-th order statistic

- If we apply Minimum algorithm to find the i-th order statistics, then we need
  - T(n) = $\mathbf{O}(in)$
- Therefore, finding the median will require
  - T(n) = $\mathbf{O}(n^2)$
  - How come?? even worse than the best sorting algorithm $\mathbf{O}(n\lg n)$
- Do we have an alternative to find i-th order statistic, like in $\mathbf{O}(n)$
  - Hint: Quicksort idea

Selection in Linear Expected Time

```
RAND-SELECT(A, p, r, i)    /* find i-th order */
  if p = r then return A[p]
  q ← RAND-PARTITION(A, p, r)
  k ← q-p+1                     /* k = rank(A[q]) */
  if i = k then return A[q]
  if i < k then return RAND-SELECT(A, p, q, i)
  else  return RAND-SELECT(A, q+1, r, i-k)
```



- Unlucky: T(n)=T(n-1)+Θ(n)=Θ(n^2) ⇐ bad partition
- Lucky: T(n)=T(9n/10)+Θ(n)=Θ(n) ⇐ good partition

Example of RAND-SELECT

Select 7-th order statistics $\Rightarrow$ *i=7*

6	10	13	5	11	3	2	8

pivot

Partition $\Rightarrow$ *find k=4 and k<i* $\therefore$ *select 7-4=3-th order stat.*

2	5	3	6	11	13	10	8

pivot

Partition $\Rightarrow$ *find k=3 and k=i return A[q]*

2	5	3	6	10	8	11	13

Stop!! $\Rightarrow$ *7-th order statistics* is 11. Done!!

Average-case Runtime

- Average-case runtime ≡ expected runtime E[**T(*n*)**] where
  - Let **T(*n*)** = the random variable for the running time of RAND-SELECT on an input of size ***n*** assuming random numbers are independent.
  - For k=0,1,…,(n-1), define ***indicator random variable***

$$x_k = \begin{cases} 1, \text{ if RAND-PARTITION can generate a } k:(n-k-1) \text{ split} \\ 0, \text{ otherwise} \end{cases}$$

- $E[x_k] = \Pr\{x_k = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.

Analysis of Expected Time

- To obtain the upper bound, assume the ***i–th*** order statistic always fall into the larger side of the partition

$$T(n) = \begin{cases} T(\max\{0, n-1\} + \Theta(n)) \text{ if } 0:(n\text{-}1) \text{ split,} \\ T(\max\{1, n-2\} + \Theta(n)) \text{ if } 1:(n\text{-}2) \text{ split,} \\ \dots \\ T(\max\{n-1, 0\} + \Theta(n)) \text{ if } (n-1):0 \text{ split,} \end{cases}$$

$$= \sum_{k=0}^{n-1} x_k (T(\max\{k, n-k-1\}) + \Theta(n))$$

indicator random variable

- Take expectations on both sides

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} x_k T(\max\{k, n-k-1\}) + \Theta(n)\right]$$

Calculating Expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} x_k (T(\max\{k, n-k-1\}) + \Theta(n))\right]$$

$$= \sum_{k=0}^{n-1} E[x_k (T(\max\{k, n-k-1\}) + \Theta(n))]$$

$$= \sum_{k=0}^{n-1} E[x_k] \cdot E[T(\max\{k, n-k-1\}) + \Theta(n)]$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(\max\{k, n-k-1\})] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$

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

Analysis of Expected Time

- Solve $T(n) \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} T(k) + \Theta(n)$
  – by substitution method, then assume T(*n*)=*cn*

Summary (Part 1)

- Implement your own i-th order statistics
- Comparison-based vs. Non-comparison-based sorting algorithms
  – Which one is ***optimal*** comparison-based sort?
  – What's ***lower bound*** for comparison-based sort?
  – Why cannot we always use linear-time sorting?
- Order Statistics
  – How to achieve (***3n/2-2***) comparisons for simultaneous minimum/maximum operations
  – What is ***the selection problem***?
  – How to derive a O(n) algorithm to find the i-th order statistic

More Intuition on Quicksort (Reprise)

- Suppose we alternate lucky, unlucky, lucky, unlucky, lucky, ….
  – Lucky cases: **L**(*n*) = 2**U**(*n*/2) + **O**(*n*)
  – Unlucky cases: **U**(*n*) = **L**(*n*-1) + **O**(*n*)
- Let's solve this recurrence

  L(*n*) = 2**U**(*n*/2) + **O**(*n*)
  = 2(**L**(*n/2*-1) + **O**(*n/2*)) + **O**(*n*)
  = 2**L**(*n*/2-1) + **O**(*n*) ⇨ still lucky !!!
  = **O**(*n*lg*n*)
- How do we ensure the good luck?

Randomized Quicksort (Revisited)

- Expect to get average-case behavior of Quicksort on all inputs
  – Randomization!!
- Two approaches
  1. Randomly permute input
  2. Choose the pivot randomly at each iteration

```
RANDOMIZED-PARTITION(A, p, r)
i ← RANDOM(p, r)
exchange A[r] ↔ A[i]
return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r then
q ← RANDOMIZED-PARTITION(A, p, r)
RANDOMIZED-QUICKSORT(A, p, q)
RANDOMIZED-QUICKSORT(A, q+1, r)
```

Analyze Randomized Quicksort

- Randomization will result in average-case behavior ≡ expected runtime E[***T(n)***] where
  - Let ***T(n)*** = the random variable for the running time of **RANDOMIZED-QUICKSORT** on an input of size ***n*** assuming random numbers are independent.
  - For k=0,1,…,(n-1), define ***indicator random variable***

$$x_k = \begin{cases} 1, \text{if RAND-PARTITION can generate a } k:(n-k-1) \text{ split} \\ 0, \text{otherwise} \end{cases}$$

- $E[x_k] = \Pr\{x_k = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.

Analysis of Expected Time

- To obtain the upper bound, assume the ***i–th*** order statistic always fall into the larger side of the partition

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) \text{ if } 0:(n-1) \text{ split,} \\ T(1) + T(n-2) + \Theta(n) \text{ if } 1:(n-2) \text{ split,} \\ \cdots \\ T(n-1) + T(0) + \Theta(n) \text{ if } (n-1):0 \text{ split,} \end{cases}$$

$$= \sum_{k=0}^{n-1} x_k (T(k) + T(n-k-1) + \Theta(n))$$

- Take expectations on both sides

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} x_k (T(k) + T(n-k-1) + \Theta(n))\right]$$

Calculating Expectation

$$E[T(n)] = E[\sum_{k=0}^{n-1} x_k (T(k) + T(n-k-1) + \Theta(n))]$$

$$= \sum_{k=0}^{n-1} E[x_k (T(k) + T(n-k-1) + \Theta(n))]$$

$$= \sum_{k=0}^{n-1} E[x_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

$$= \frac{1}{n} \sum_{k=0}^{n-1} E[T(k) + T(n-k-1)] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n)$$

$$= \frac{2}{n} \sum_{k=1}^{n-1} E[T(k)] + \Theta(n)$$

Why??

Analysis of Expected Time (cont'd)

- Solve $T(n) \le \frac{2}{n} \sum_{k=1}^{n-1} T(k) + \Theta(n)$

  by substitution method, then assume T(n)=***an***lg***n***+**b**

$$T(n) \le \frac{2}{n} \sum_{k=1}^{n-1} (ak\lg k + b) + \Theta(n)$$

$$= \frac{2a}{n} \sum_{k=1}^{n-1} k\lg k + \frac{2b}{n}(n-1) + \Theta(n)$$

$$\sum_{k=1}^{n-1} k\lg k = \left(\sum_{k=1}^{\lceil n/2 \rceil - 1} k\lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k\lg k\right)$$

$$\le (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k \qquad \textbf{\textit{Why??}}$$

$$\le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$$

Analysis of Expected Time (cont'd)

$$T(n) \le \frac{2a}{n}\sum_{k=1}^{n-1} k\lg k + \frac{2b}{n}(n-1)+\Theta(n)$$

$$= \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + \frac{2b}{n}(n-1)+\Theta(n)$$

$$= an\lg n - \left(\frac{an}{4} - \Theta(n)\right)$$

$$= \Theta(n\lg n) \quad \square$$

- Practically, Quicksort runs 2X-3X faster than Mergesort or Heapsort even if it's not a optimal sorting in worst-case.

O(n) Algorithm for i-th Order Statistic

- Works fast: linear expected time.
  - –Excellent algorithm in practice.
  - –But, the worst case is *very* bad: $\Theta(n^2)$.
- *Q.* Is there an algorithm that runs in linear time in the worst case?
  - –Yes, due to Blum, Floyd, Pratt, Rivest and Tarjan in 1973 (BFPRT-Select)
  - –Key idea: Guarantee a good split recursively to have a balance partition sizes
  - –Pure theoretical interests: Non-obvious & unintuitive

BFPRT-SELECT Pseudocode

BFPRT-SELECT(i, n)

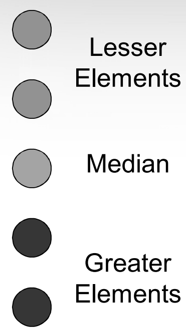
1 Divide the n elements into groups of 5. Find the median of each 5-element group by rote.
2 Recursively SELECT the median x of the $\lceil n/5 \rceil$ group medians to be the pivot.

} Find a good split

3 Partition around the pivot x. Let k= rank(x)
4 if i = k then return x
5 elseif i< k
6 then recursively SELECT the (i)-th smallest element in the ***lesser*** part
7 else recursively SELECT the (i–k)-th smallest element in the ***greater*** part

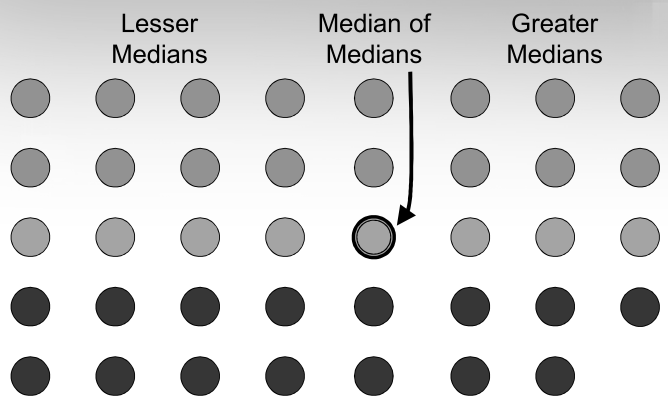
} Same as RAND-SELECT

Choosing the Pivot (1/3)



One group of 5 elements.

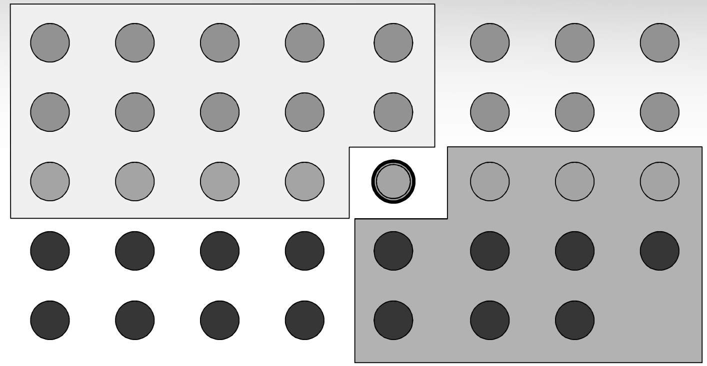
Choosing the Pivot (2/3)



All groups of 5 elements.
(And at most one smaller group.)

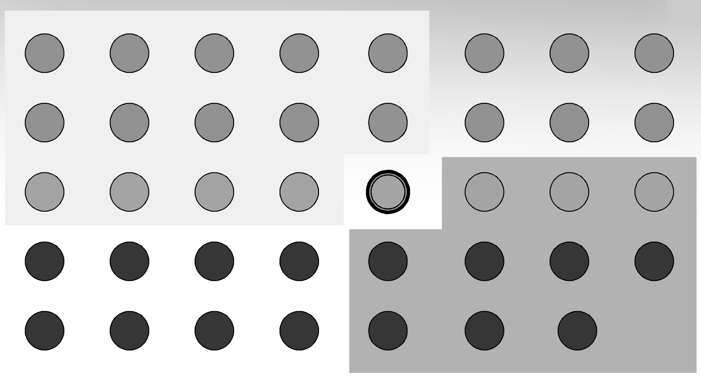
Choosing the Pivot (3/3)

Definitely Lesser Elements



Definitely Greater Elements

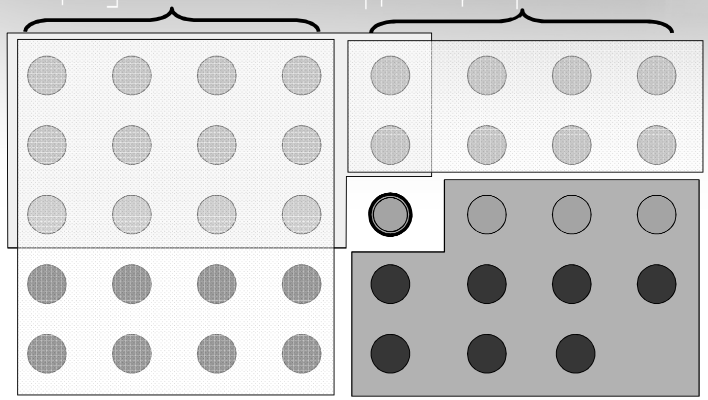
Order Statistics: Analysis 1



Must recur on all elements outside one of these boxes. How many?

Order Statistics: Analysis 1

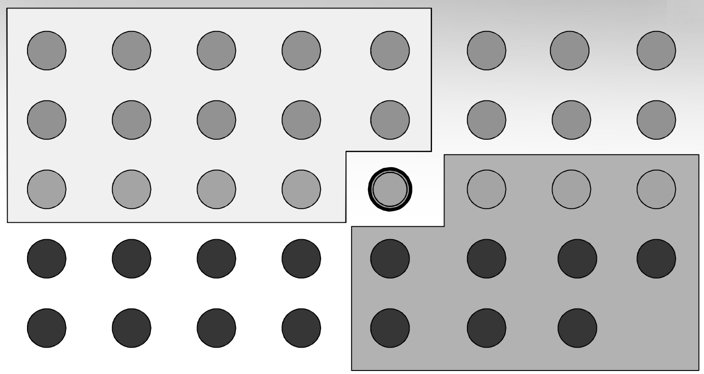
$\lfloor \lceil n/5 \rceil / 2 \rfloor$ full groups of 5 $\quad \lceil \lceil n/5 \rceil / 2 \rceil$ partial groups of 2



Count elements outside smaller box

At most

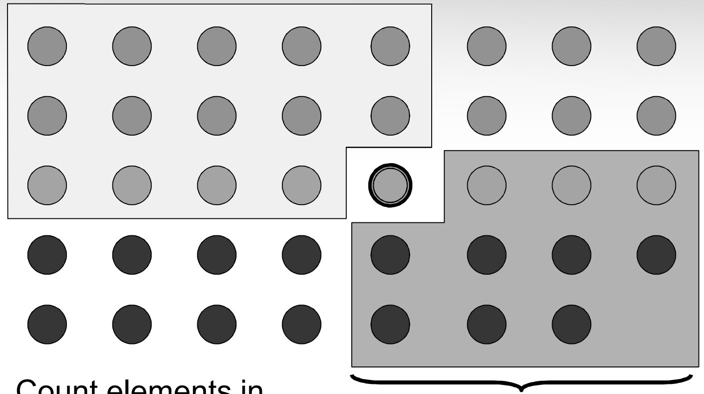
Order Statistics: Analysis 2



Equivalently, must recur on all elements not inside one of these smaller boxes. How many?

Order Statistics: Analysis 2

At most $n - \left(3\left(\left\lceil\left\lceil\frac{n}{5}\right\rceil\Big/2\right\rceil - 1\right)+1\right) \le \frac{7n}{10} + 2$



Count elements in smaller box & pivot

groups of 3

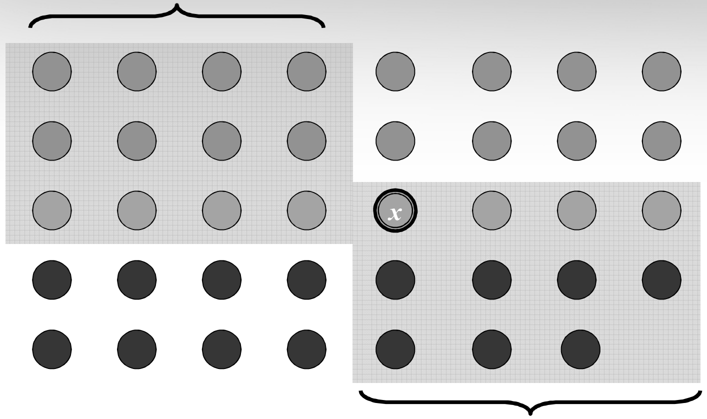
Runtime Analysis of SELECT

$$T(n) = T\left(\left\lceil\frac{n}{5}\right\rceil\right) + T\left(\frac{7n}{10}+2\right) + O(n)$$

- By substitution method, then assume T(k)=ck

Simplified Point of View

more than $\frac{1}{4}$ elements ≤ x



more than elements ≥ x

Simplified Point of View (cont'd)

- Partition original group into $\lceil n/5 \rceil$ subgroups
- Select median-of-median from only $\lceil n/5 \rceil$ median elements
  - –More than $n/4$ elements $\leq \boldsymbol{x}$
  - –More than $n/4$ elements $\geq \boldsymbol{x}$
- If $i \neq k$ ⇨ recursively select median-of-median from at most 3*n*/4 elements
- If $i = k$ ⇨ done!

$$T(n) = T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{3n}{4}\right) + O(n)$$

Prove it.

Summary (Part 2)

- Order Statistics
  - –How to define? How to analyze the average-case of RAND-SELECT?
  - –What's the key idea to have a $\mathbf{O}(n)$ Algorithm?
  - –Quiz: Why groups of 5? Can we use 3?
- Quicksort Review
  - –How to analyze the average-case?
  - –How to avoid the worst-case?
  - –Can we guarantee a $\mathbf{O}(n\lg n)$ quicksort??
- Next Lecture ⇨ Dynamic Programming