Week 4. Lecture Notes

Topics: Linear Time Sorting
Counting Sort
Radix Sort and Bucket Sort
Order Statistics
Randomized Order Statistics
Worst Case Linear time order Statistics.

Sorting in Linear Time

Counting Sort

No comparison between elements.

Input: $A[1, \ldots, n]$, where $A[j] \in \{1, 2, \ldots, K\}$

Output: $B[1, \ldots, n]$, sorted

Auxiliary Storage

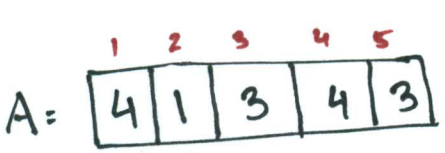
$C[1, \ldots, k]$

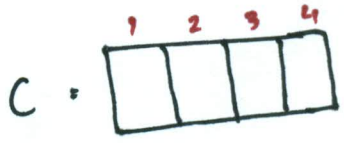
Counting Sort : Pseudocode

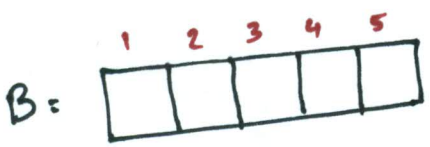
```
1.    for i ← 1 to K
2.        do C[i] ← 0
3.    for j ← 1 to n
4.        do C[A[j]] ← C[A[j]] + 1
5.    for i ← 2 to K
6.        do C[i] ← C[i] + C[i-1]
7.    for j ← n down to 1
8.        do B[C[A[j]]] ← A[j]
9.           C[A[j]] ← C[A[j]] - 1
```

Example:

A = | 4 | 1 | 3 | 4 | 3 |







Loop 1:

```
for i ← 1 to K
    do C[i] ← 0
```

A: | 4 | 1 | 3 | 4 | 3 |

C: | 0 | 0 | 0 | 0 |

B: | | | | | |

Loop 2:

```
for j ← 1 to n
  do C[A[j]] ← C[A[j]] + 1
```

A: | 4 | 1 | 3 | 4 | 3 |

C: | 1 | 0 | 2 | 2 |

B: | | | | | |

Loop 3:

```
for i ← 2 to K
  do C[i] ← C[i] + C[i-1]
```

A: | 4 | 1 | 3 | 4 | 3 |

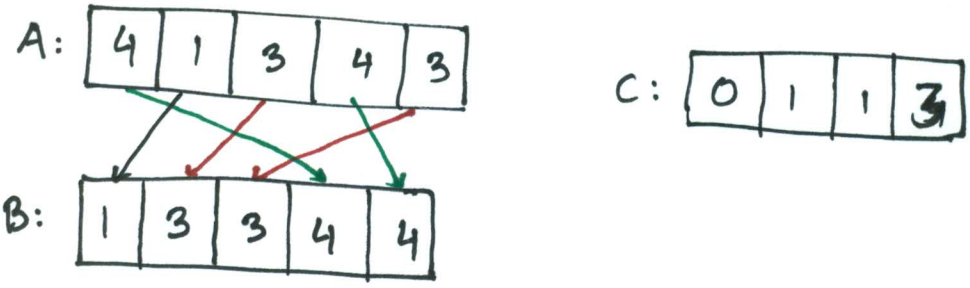
C: | 1 | 0 | 2 | 2 |

B: | | | | | |

C': | 1 | 1 | 3 | 5 |

Loop 4

```
for j ← n down to 1
  do B[C[A[j]]] ← A[j]
     C[A[j]] ← C[A[j]] - 1
```



Analysis of the Pseudocode

$\theta(k)$
```
for i ← 1 to k
  do C[i] ← 0
```

$\theta(n)$
```
for j ← 1 to n
  do C[A[j]] ← C[A[j]] + 1
```

$\theta(k)$
```
for i ← 2 to k
  do C[i] ← C[i] + C[i-1]
```

$\theta(n)$
```
for j ← n down to 1
  do B[C[A[j]]] ← A[j]
     C[A[j]] ← C[A[j]] - 1
```

$\theta(n+k)$

Running Time

If $K = O(n)$, then counting sort takes $\Theta(n)$ time.

- But, sorting takes $\Omega(n \log n)$ time.

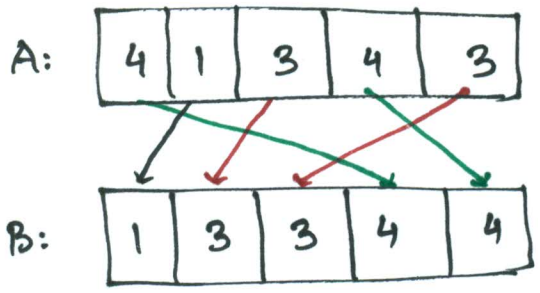
  Where is the fallacy?

Answer:

- Comparison sorting takes $\Omega(n \log n)$ time
- Counting sort is not a comparison sort
- In fact, not a single comparison between element occurs.

Stable Sorting

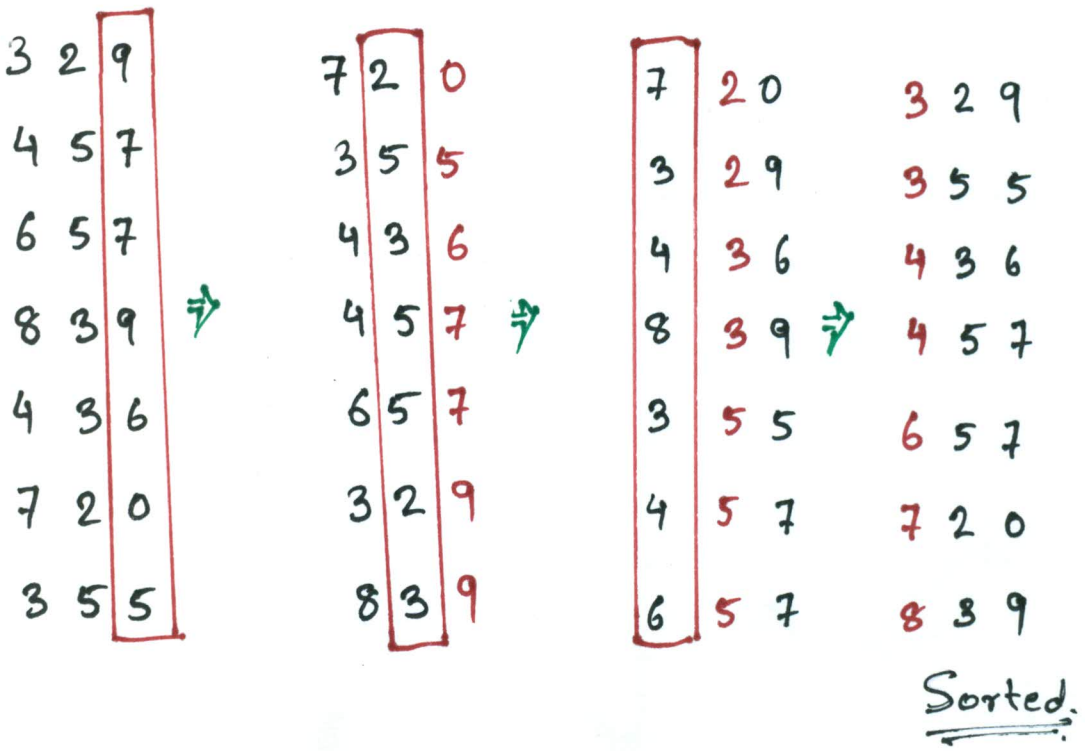
Counting sort is a stable sort, i.e. it preserves the input order among equal elements.



Radix Sort

- Origin: Herman Hollerith's card-sorting machine for the 1890 U.S. Census.
- Digit-by-digit sort
- Hollerith's original (bad) idea: sort on most-significant digit first
- Good idea - Sort on least-significant digit first with auxiliary stable sort.

Operation of Radix Sort



Correctness of Radix Sort

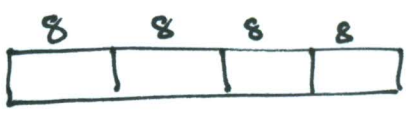
Induction on digit position.

- Assume that the numbers are sorted by their low-order $t-1$ digits
- Sort on digit t.
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input
    $\Rightarrow$ Correct order.

Analysis of Radix Sort

- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base-2^r digits.

Example: 32 bit word



$r = 8 \Rightarrow b/r = 4$ passes of Counting sort on base-2^8 digits; or $r = 16 \Rightarrow b/r = 2$ passes of Counting sort on base-2^{16} digits.

How many passes should we make?

Recall:

Counting Sort takes $\Theta(n+k)$ time to sort n numbers in the range 0 to $k-1$

If each b-bit word is broken in b/r equal pieces, each pass of counting sort takes $\Theta(n+2^r)$ time. Since there are b/r passes, we have

$$T(n, b) = \Theta\left(\frac{b}{r}(n+2^r)\right)$$

Choose r to minimize $T(n,b)$

- Increasing r means fewer passes but as $r \geq \log n$, the time grows exponentially.

Choosing r

$$T(n, b) = \Theta\left(\frac{b}{r}(n+2^r)\right)$$

Minimize $T(n,b)$ by differentiating and setting to 0.

Or just observe that we don't want $2^r >> n$, and there's is no harm asymptotically in choosing r as large as possible, subject to this constraint.

Choosing $r = \log n \Rightarrow T(n,b) = \Theta(bn/\log n)$

- For numbers in the range from 0 to $n^d - 1$, we have $b = d\log n \Rightarrow$ radix sort runs in $\Theta(dn)$ time

Conclusions

In practice Radix Sort is fast for large inputs as well as simple to code and maintain.

Example (32-bit numbers)

- At most 3 ~~phases~~ passes when sorting $\geq$ 2000 numbers.
- Merge sort and quicksort do atleast $\lceil \log 2000 \rceil = 11$ passes.

Downside:

Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.

Bucket Sort

Idea:

- Divide the interval $[0, n)$ into n equal-sized subinterval, or buckets
- Distribute the n input numbers into the buckets.

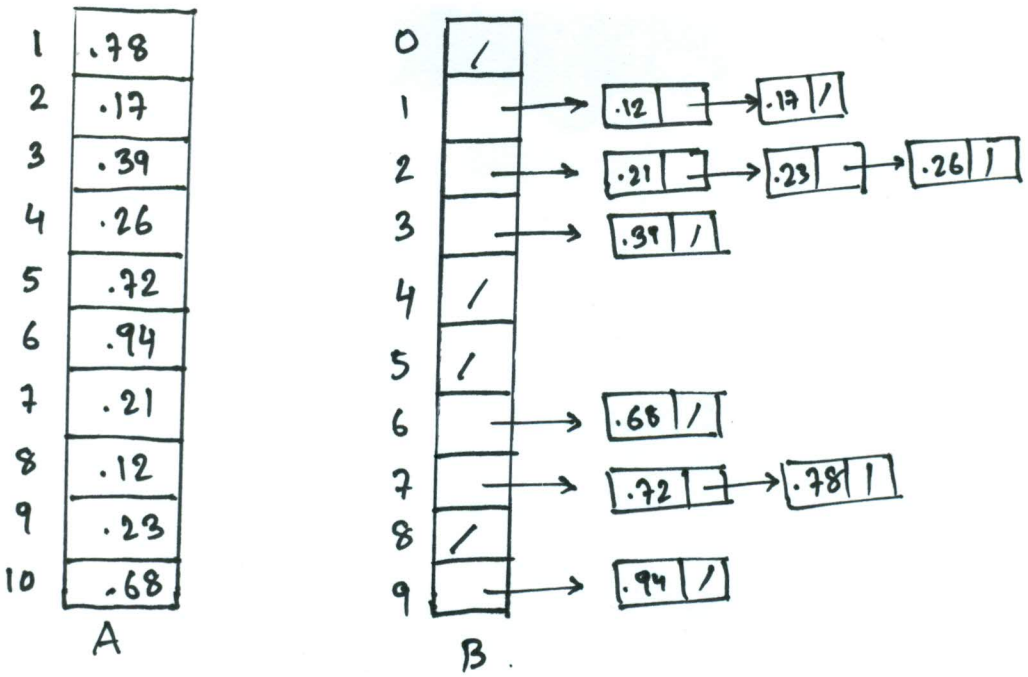
Since the inputs are assumed to be uniformly distributed over $[0,1)$, many numbers do not fall into each bucket.

To produce the output, simply sort the numbers in each bucket and then go through the buckets, in order, listing the elements in each.

Pseudocode of Bucket sort

```
BUCKET-SORT(A)
1.  n ← length[A]
2.  for i ← 1 to n
3.      do insert A[i] into list B[⌊n A[i]⌋]
4.  for i ← 0 to n-1
5.      do sort list B[i] with insertion sort
6.  Concatenate the lists B[0], B[1] ...
        B[n-1] together in order.
```

Example



A: input array

B: array of sorted lists (buckets) after line 5 of the above algorithm.

Correctness of Pseudocode

Consider two elements $A[i] \leq A[j]$

Since $\lfloor n A[i] \rfloor \leq \lfloor n A[j] \rfloor$, element $A[i]$ is placed either into the same bucket as $A[j]$ or a bucket with lower index.

If they are placed in same bucket then the for loop of lines 4-5 puts them in proper order.

If they are placed in different buckets line 6 puts them in proper order.

Therefore, bucket sort works correctly.

Analysis of Running Time

- Observe that all lines except line 5 takes $O(n)$ time in worst case.

- We need to balance the total time taken by the n calls to insertion sort in line 5.

Let n_i be the random variable denoting the number of elements placed in bucket $B[i]$.

So, running of bucket sort is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \quad \left[\because \text{insertion sort runs in } O(n^2)\right]$$

$$\Rightarrow E[T(n)] = E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right]$$

$$= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)]$$

$$= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \quad \text{——— (1)}$$

We claim that

$$E[n_i^2] = 2 - \frac{1}{n} \quad \text{for } i = 0, 1, \ldots, n-1. \quad \text{——— (2)}$$

Define

$$X_{ij} = I\{A[j] \text{ falls in bucket } i\}$$

for $i = 0, 1, \ldots n-1$, $j = 1, 2, \ldots, n$.

$$\Rightarrow n_i = \sum_{j=1}^{n} X_{ij}$$

$$\Rightarrow \quad E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right]$$

$$= E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij} X_{ik}\right]$$

$$= E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\neq j}} X_{ij} X_{ik}\right]$$

$$= \sum_{j=1}^{n} E\left[X_{ij}^2\right] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\neq j}} E\left[X_{ij} X_{ik}\right]$$

As, Indicator variable X_{ij} is 1 with probability $1/n$ and 0 other wise, so

$$E\left[X_{ij}^2\right] = 1\cdot 1/n + 0\,(1-1/n) = \frac{1}{n}$$

When $k \neq j$, X_{ij} and X_{ik} are independent, so

$$E[X_{ij} X_{ik}] = E[X_{ij}]\, E[X_{ik}]$$

$$= \frac{1}{n}\cdot\frac{1}{n} = \frac{1}{n^2}$$

So,

$$E[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\neq j}} \frac{1}{n^2}$$

$$= n\cdot\frac{1}{n} + n(n-1)\,1/n$$

$$= 1 + \frac{n-1}{n}$$

$$= 2 - \frac{1}{n}$$

which proves (2)

Using this expected value in (1),
We can say that the running time of bucket sort is expected to be

$$T(n) = \theta(n) + n \cdot O(2 - 1/n)$$

$$= \theta(n).$$

- Thus, the entire bucket sort algorithm runs in linear expected time.

Conclusion

- Bucket Sort runs in linear time when the input is drawn from a uniform distribution.
- Like Counting sort, it is fast.
- Even if the input is not drawn from a uniform distribution, bucket sort may run in linear time, as long as the input has the property that the sum of the squares of the bucket ~~sor~~ sizes is linear in the total number of elements.

Order Statistics

Select the i^{th} smallest of n elements (the element with rank i).

- $i=1$, minimum
- $i=n$, maximum
- $i = \lfloor (n+1)/2 \rfloor$ or $\lceil (n+1)/2 \rceil$, median.

Naive Algorithm:

Sort and index i^{th} element.

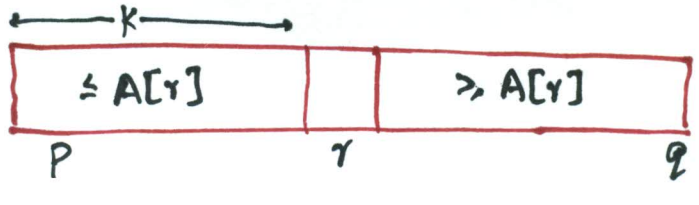
Worst case running time = $\theta(n \log n) + \theta(1)$

$= \theta(n \log n)$

using Merge sort or Heapsort (not Quicksort)

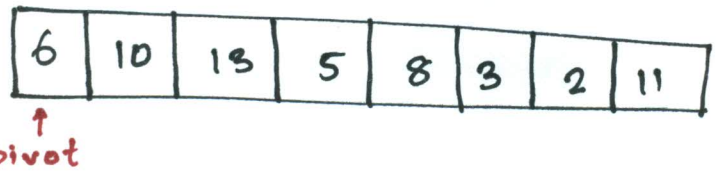
Randomized divide-and-conquer algorithm.

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

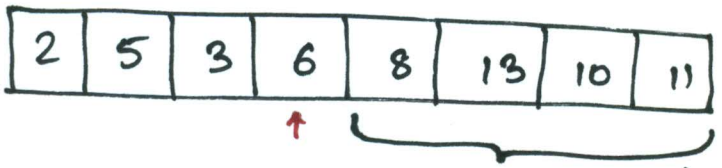


Example

Select the $i = 7^{th}$ smallest



Partition:



Select the $i = 7-4 = 3^{rd}$ element recursively

Intuition for Analysis

All our analysis here assume that all the elements are distinct.

Lucky:

$$T(n) = T(9n/10) + \Theta(n) \quad \left[n^{\log_{10/9} 1} = n^0 = 1, \text{ case 3} \right]$$

$$= \Theta(n)$$

Unlucky:

$$T(n) = T(n-1) + \Theta(n) \quad [\text{arithmetic series}]$$

$$= \Theta(n^2)$$

Worse than sorting

Analysis of Expected Time

The analysis follows that of randomized quicksort, but it's a little different.

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 $k = 0, 1, \ldots, n-1$, define the indicator random variable X_k as

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n-k-1 \text{ split} \\ 0 & \text{otherwise} \end{cases}$$

To obtain an upper bound, assume that the i^{th} element always fall in the larger side of the partition.

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

$$= \sum_{k=0}^{n-1} X_k \left(T(\max\{k, n-k-1\}) + \Theta(n) \right).$$

Calculating Expectation

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

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

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

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

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

Prove: $E[T(n)] \leq cn$ for constant $c > 0$

- The constant c can be chosen large enough so that $E[T(n)] \leq cn$ for the base cases.
- Use fact: $\sum_{k=\lfloor n/2 \rfloor}^{n-1} k \leq \frac{3}{8} n^2$

We have,

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + \Theta(n) \quad \left[\begin{array}{l}\text{Substituting} \\ E(T(n)) \leq cn\end{array}\right.$$

So, using the fact stated above

$$E[T(n)] \leq \frac{2c}{n}\left(\frac{3}{8}n^2\right) + \Theta(n)$$

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

[Expressed as desired - residual]

$$\leq cn$$

if c is chosen large enough so that $cn/4$ dominates the $\Theta(n)$.

Summary of randomized order-statistic selection

- 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?

A. Yes, due to Blum, Floyd, Pratt, Rivest and Tarjan [1973]

Idea: Generate a good point recursively.

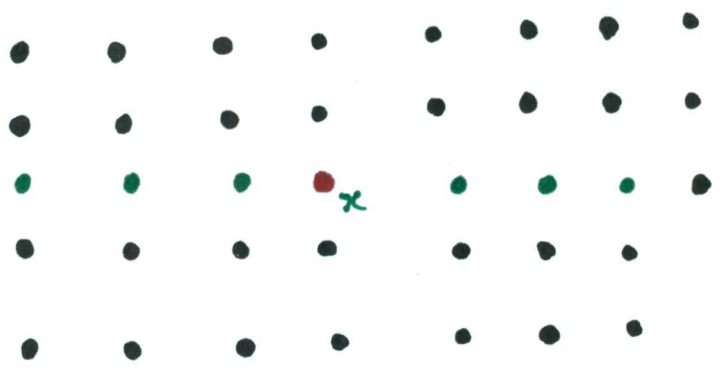
Worst Case linear Time order statistics

SELECT (i, n)

1. Divide the n elements into groups of 5. Find the median of each 5-element group
2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.
3. Partition around the pivot x. Let $k = \text{rank}(x)$.
4. if $i = k$ then return x
   else if $i < k$
   then recursively SELECT the i^{th} smallest element in lower part.
   else recursively SELECT the $(i-k)^{th}$ smallest element in upper part.

} Same as RAND-SELECT

Choosing the Pivot



- Divide n into groups of 5
- Recursively SELECT the median 'x' of $\lfloor n/5 \rfloor$ group medians as pivot.

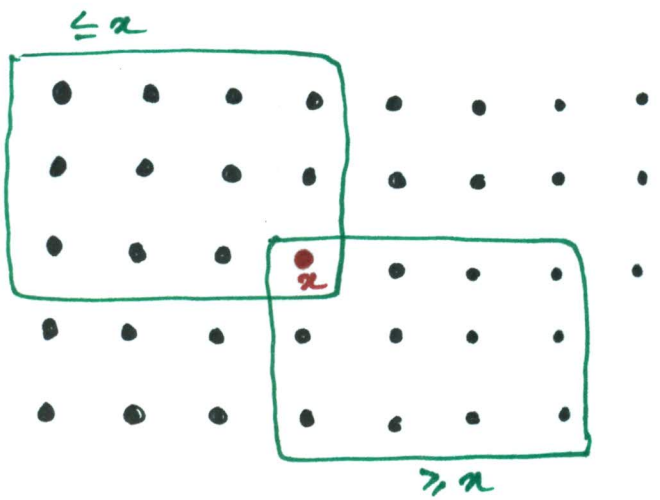
Analysis

Assume all elements are distinct

Atleast half the group medians $\leq x$, which is atleast $\lfloor \lceil n/5 \rceil / 2 \rfloor = \lfloor n/10 \rfloor$ group medians.

Therefore, at least $3\lfloor n/10 \rfloor$ elements are $\leq x$

Similarly, at least $3\lfloor n/10 \rfloor$ elements are $\geq x$



Minor Simplification

- For $n \geq 50$, we have $3\lfloor n/10 \rfloor \geq n/4$
- Therefore, for $n \geq 50$, the recursive call to SELECT in step 4 is executed on $\leq 3n/4$ elements.
- Thus, the recurrence for running time can assume that step 4 takes time $T(3n/4)$ in the worst case.
- for $n < 50$, we know that the wort case time is $T(n) = \theta(1)$

Developing the recurrence

SELECT (i, n)

$\theta(n)$ 1. Divide the n elements into groups of 5. Find the median of each 5-element group.

$T(n/5)$ 2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.

$\theta(n)$ 3. ~~Pivot~~ Partition around the pivot x.
Let $k = \text{rank}(x)$.

$T(3n/4)$ 4. if $i = k$ then return x
else if $i < k$
then recursively SELECT the i^{th} smallest element in the lower part.
else recursively SELECT the $(i-k)^{th}$ smallest element in the upper part.

Thus the recurrence is,

$$T(n) = T\left(\frac{1}{5}n\right) + T\left(\frac{3}{4}n\right) + \theta(n).$$

Solving the recurrence.

We have

$$T(n) = T(n/5) + T(3n/4) + \theta(n)$$

Substituting $T(n) \leq cn$

$$T(n) \leq \frac{1}{5}cn + \frac{3}{4}cn + \theta(n)$$

$$= \frac{19}{20}cn + \theta(n)$$

$$= cn - \left(\frac{1}{20}cn - \theta(n)\right)$$

$$\leq cn$$

if c is chosen large enough to handle both the $\theta(n)$ and the initial conditions.

Conclusion

- Since work at each level of recursion is a constant fraction $(19/20)$ smaller, the work per level is a geometric series dominated by the linear work at the root.
- In practice, this algorithm runs slowly, because the constant in front of n is large.
- The randomized algorithm is far more practical.