EL9343

Data Structure and Algorithm

Lecture 5: Randomized Quick Sort, Sorting Lower Bound, Order Statistics & Selection

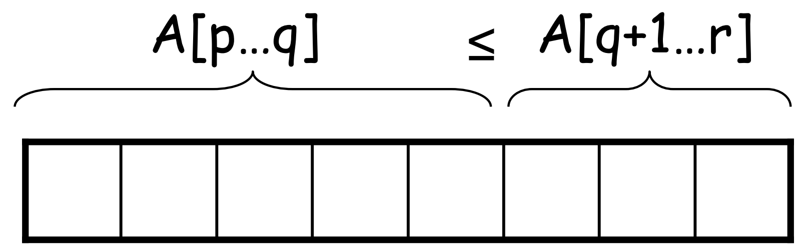
Instructor: Yong Liu

Last Lecture

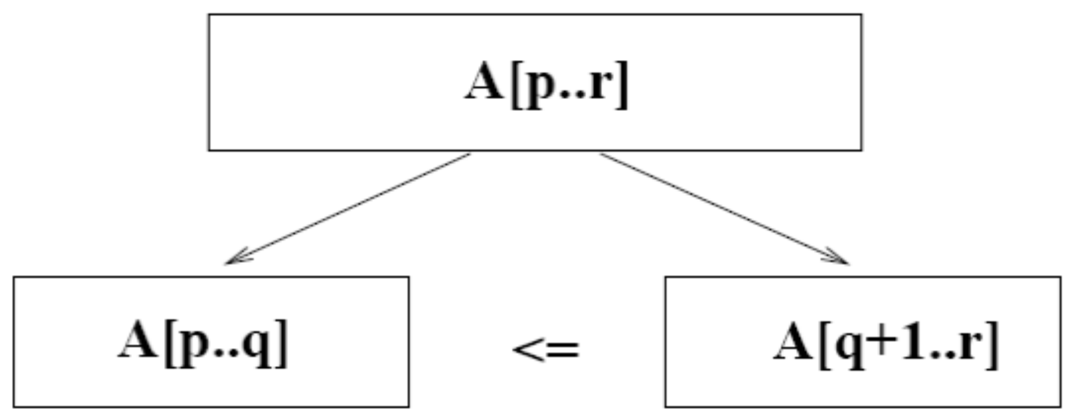
- Heapsort & Priority Queue
  - MAX-HEAPIFY $O(\lg n)$
  - BUILD-MAX-HEAP $O(n)$
  - HEAP-SORT $O(n\lg n)$
  - MAX-HEAP-INSERT $O(\lg n)$
  - HEAP-EXTRACT-MAX $O(\lg n)$
  - HEAP-INCREASE-KEY $O(\lg n)$
  - HEAP-MAXIMUM $O(1)$

Last Lecture

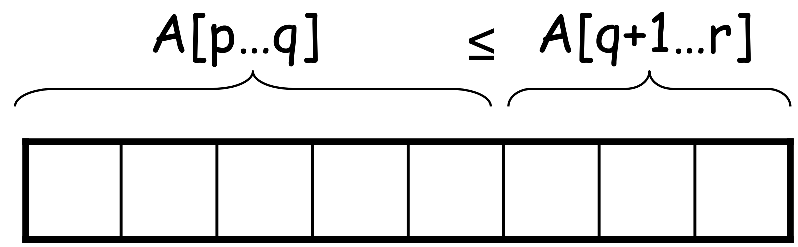
Quicksort



- **Divide**
  - Partition the array A into 2 subarrays A[p..q] and A[q+1..r], such that each element of A[p..q] is smaller than or equal to each element in A[q+1..r]
  - Need to find index q to partition the array



Quicksort



- **Conquer**
  - Recursively sort A[p..q] and A[q+1..r] by calls to Quicksort
- **Combine (unlike merge sort)**
  - Trivial: the arrays are sorted in place
  - No additional work is required to combine them
  - The entire array is now sorted

Quicksort: Recurrence

Alg.: QUICKSORT(A, p, r)

if $p < r$

then $q \leftarrow$ PARTITION(A, p, r)

QUICKSORT (A, p, q)

QUICKSORT (A, q+1, r)

Initially: p=1, r=n

Recurrence: $T(n) = T(q) + T(n - q) + n$

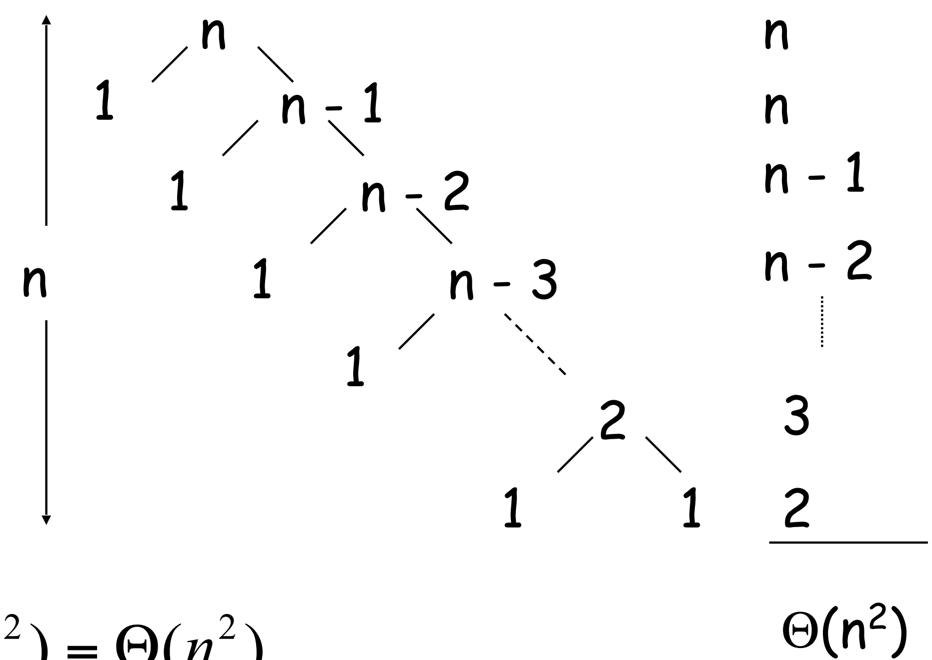
Analyzing Quicksort: Worst Case Partitioning

- Worst-case partitioning
  - One region has one element and the other has n – 1 elements
  - Maximally unbalanced
- Recurrence: q=1

$T(n) = T(1) + T(n - 1) + n,$

- $T(1) = \Theta(1)$
- $T(n) = T(n - 1) + n$

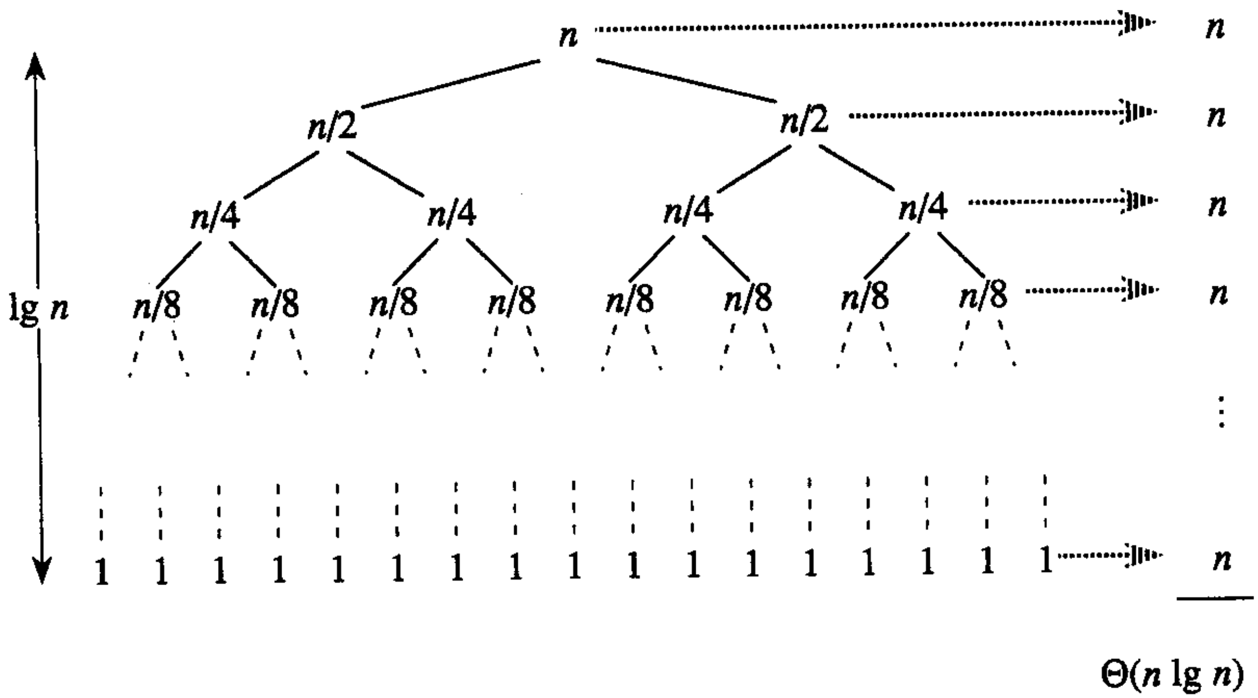
$$= n + \left(\sum_{k=1}^{n} k\right) - 1 = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$



When does the worst case happen?

Analyzing Quicksort: Best Case Partitioning

- Best-case partitioning
  - Partitioning produces two regions of size n/2
- Recurrence: q=n/2
  - $T(n) = 2T(n/2) + \Theta(n)$
  - $T(n) = \Theta(n\lg n)$ (Master theorem)



Randomized Algorithm

- No input can elicit worst case behavior
  - Worst case occurs only if we get "unlucky" numbers from the random number generator
- Worst case becomes less likely
  - Randomization can NOT eliminate the worst-case but it can make it less likely!

Randomizing Quicksort

- Randomly permute the elements of the input array before sorting
- OR ... modify the PARTITION procedure
  - At each step of the algorithm we exchange element A[p] with an element chosen at random from A[p…r]
  - The pivot element x = A[p] is equally likely to be any one of the r – p + 1 elements of the subarray

Randomizing PARTITION

Alg.: RANDOMIZED-PARTITION(A, p, r)

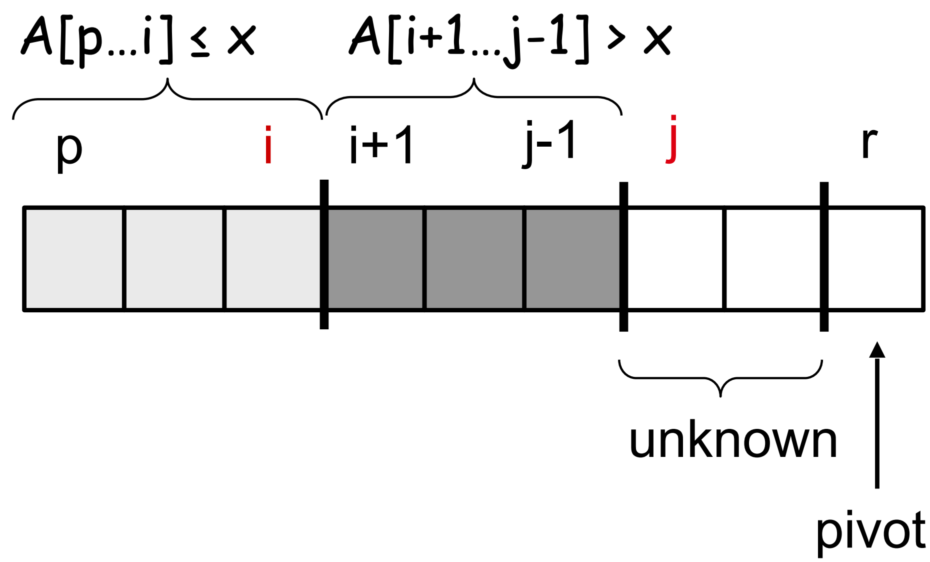
i ← RANDOM(p, r)

exchange A[p] ↔ A[i]

return PARTITION(A, p, r)

Another Partitioning: Lomuto's Partition

- Given an array A, partition the array into the following subarrays:
  - A pivot element x = A[q]
  - Subarray A[p..q-1] such that each element of A[p..q-1] is smaller than or equal to x (the pivot)
  - Subarray A[q+1..r], such that each element of A[p..q+1] is strictly greater than x (the pivot)
- The pivot element is not included in any of the two subarrays



Randomizing Quicksort using Lomuto's partition

Alg.: RANDOMIZED-QUICKSORT(A, p, r)

if $p < r$

then $q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, q - 1)

RANDOMIZED-QUICKSORT(A, q + 1, r)

The pivot is no longer included in any of the subarrays!!

Analysis of Randomized Quicksort

Alg.: RANDOMIZED-QUICKSORT(A, p, r)

if p < r

then q ← RANDOMIZED-PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, q - 1)

RANDOMIZED-QUICKSORT(A, q + 1, r)

PARTITION is called at most n times

- (at each call a pivot is selected and never again included in future calls)

Partition

Alg.: PARTITION*(A, p, r)*

$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j \leftarrow p$ **to** $r - 1$
  do if $A[j] \le x$
    then $i \leftarrow i + 1$
      exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$

O(1) - constant

# of comparisons: X_k between the pivot and the other elements

O(1) - constant

Amount of work at call k: $c + X_k$

Average-Case Analysis of Quicksort

- Let X = **total number of comparisons performed in <u>all calls</u> to PARTITION:**

- The total work done over the **entire** execution of Quicksort is

$$O(nc+X)=O(n+X)$$

- Need to estimate E(X): the average number of comparisons in all calls with random partition at each call.

Review of Probabilities

- **Definitions**

  - random experiment: an experiment whose result is not certain in advance (e.g., throwing a die)

  - outcome: the result of a random experiment

  - sample space: the set of all possible outcomes (e.g., {1,2,3,4,5,6})

  - event: a subset of the sample space (e.g., obtain an odd number in the experiment of throwing a die = {1,3,5})

Review of Probabilities

- **Probability of an event**

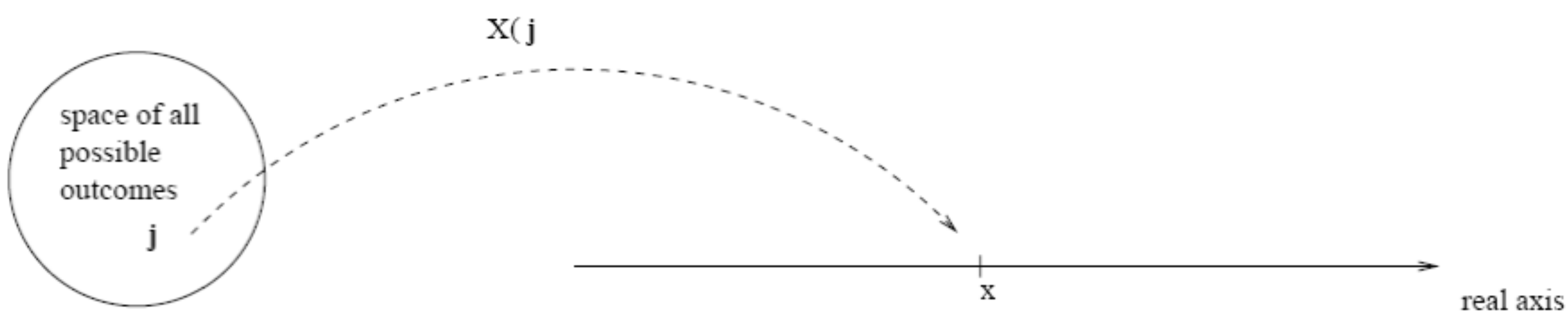
  - The likelihood that an event will occur if the underlying random experiment is performed

$$P(event) = \frac{number\ of\ favorable\ outcomes}{total\ number\ of\ possible\ outcomes}$$

Example: $P(obtain\ an\ odd\ number) = 3/6 = 1/2$

Random Variables

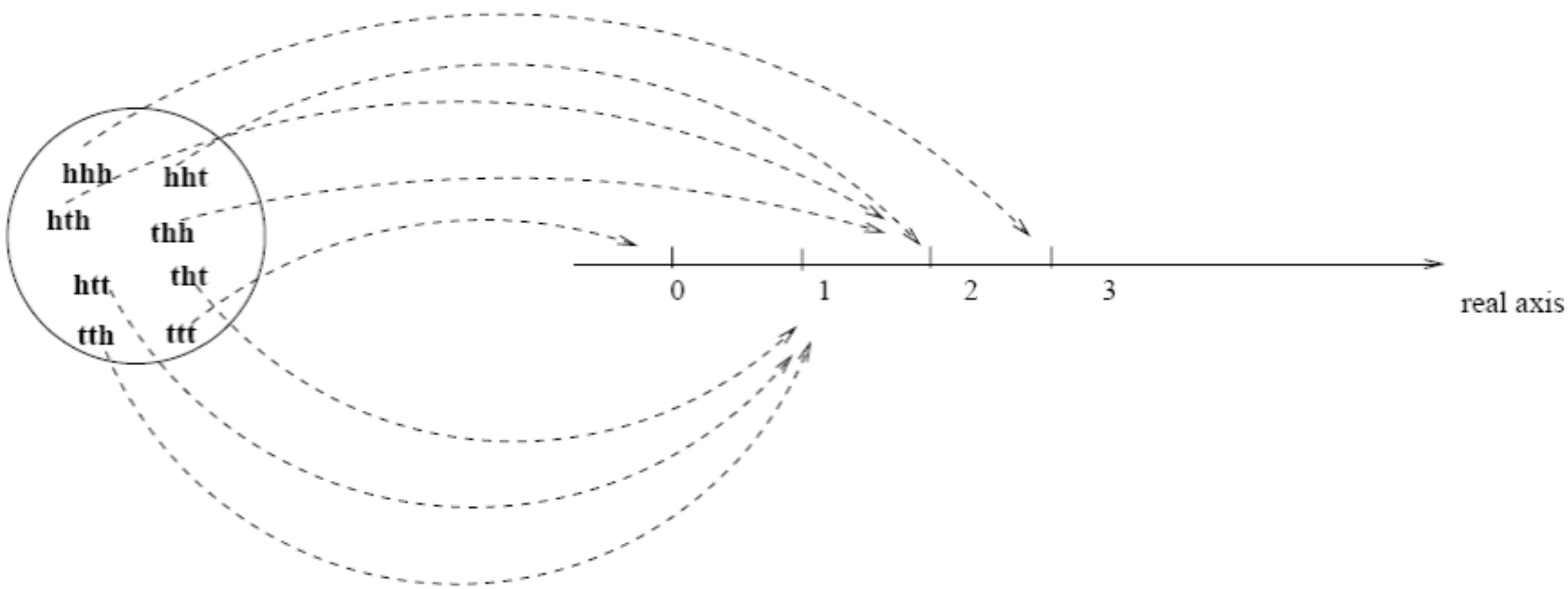
- *Def.:* ***(Discrete) random variable X:*** *a function from a sample space S to the real numbers.*
  - It associates a real number with each possible outcome of an experiment.



Random Variables

E.g.: Toss a coin three times

define X = “numbers of heads”



Computing Probabilities Using Random Variables

- Example: consider the experiment of throwing a pair of dice

Define the r.v. X="sum of dice"

$X = x$ corresponds to the event A_x=$\{s \in S / X(s) = x\}$

(e.g., $X = 5$ corresponds to A_5=$\{(1,4),(4,1),(2,3),(3,2)\}$

$$P(X = x) = P(A_x) = \sum_{s:X(s)=x} P(s)$$

$$(P(X = 5) = P((1,4)) + P((4,1)) + P((2,3)) + P((2,3)) = 4/36 = 1/9)$$

Expectation

- Expected value (expectation, mean) of a discrete random variable X is:

$$E[X] = \Sigma_x \, x \Pr\{X = x\}$$

  - “Average” over all possible values of random variable X

Examples

Example: X = face of one fair dice

$$E[X] = 1\cdot 1/6 + 2\cdot 1/6 + 3\cdot 1/6 + 4\cdot 1/6 + 5\cdot 1/6 + 6\cdot 1/6$$
$$= 3.5$$

Example: : X="sum of dice"

Events												
Sum	1	2	3	4	5	6	7	8	9	10	11	12
Probability	0/36	1/36	2/36	3/36	4/36	5/35	6/36	5/36	4/360	3/36	2/36	1/36

$$E(X) = 1P(X=1) + 2P(X=2) + \ldots + 12P(X=12) = (0+2+\ldots+12)/36 = 7$$

Indicator Random Variables

- Given a sample space S and an event *A*, we define the ***indicator random variable*** I{A} associated with A:
  - $$I\{A\} = \begin{cases} 1 & \text{if A occurs} \\ 0 & \text{if A does not occur} \end{cases}$$
- The expected value of an indicator random variable $X_A = I\{A\}$:
  - $E[X_A] = \Pr\{A\}$
- Proof: $E[X_A] = E[I\{A\}] = 1 * \Pr\{A\} + 0 * \Pr\{\bar{A}\} = \Pr\{A\}$

Average-Case Analysis of Quicksort

- Let X = **total number of comparisons performed in all calls to PARTITION:**

- The total work done over the **entire** execution of Quicksort is $O(n+X)$

- Need to estimate $E(X)$: the average number of comparisons in all calls with random partition at each call.

Comparisons in PARTITION : Observation 1

- Each pair of elements is compared **at most once** during the entire execution of the algorithm
  - Elements are compared only to the pivot point
  - Pivot point is excluded from future calls to PARTITION

Notation

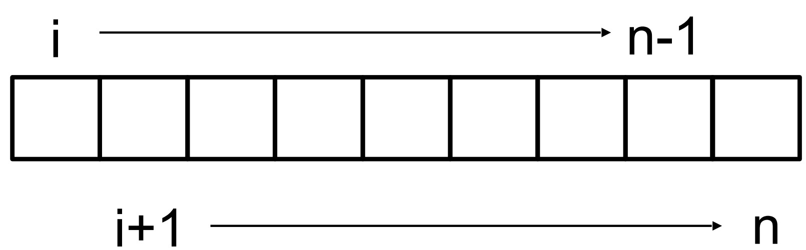
z_2	z_9	z_8	z_3	z_5	z_4	z_1	z_6	z_{10}	z_7
2	9	8	3	5	4	1	6	10	7

- Rename the elements of A as $z_1, z_2, \ldots, z_n$, with z_i being the *i*-th smallest element
- Define the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ the set of elements between z_i and z_j, inclusive

Total Number of Comparisons in PARTITION

- Define X_{ij} = I {z_i is compared to z_j }
- Total number of comparisons X performed by the algorithm:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$



Expected Number of Total Comparisons in PARTITION

- Compute the expected value of X:

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}] =$$

by linearity
of expectation

indicator
random variable

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \textit{ is compared to } z_j\}$$

the expectation of X_{ij} is equal
to the probability of the event
"z_i is compared to z_j"

Comparisons in PARTITION : Observation 2

- Only the pivot is compared with elements in both partitions!

z_2	z_9	z_8	z_3	z_5	z_4	z_1	z_6	z_{10}	z_7
2	9	8	3	5	4	1	6	10	7

$Z_{1,6} = \{1, 2, 3, 4, 5, 6\}$ $\{7\}$ $Z_{8,9} = \{8, 9, 10\}$

pivot

Elements between different partitions
are <u>never</u> compared!

Comparisons in PARTITION

z_2	z_9	z_8	z_3	z_5	z_4	z_1	z_6	z_{10}	z_7
2	9	8	3	5	4	1	6	10	7

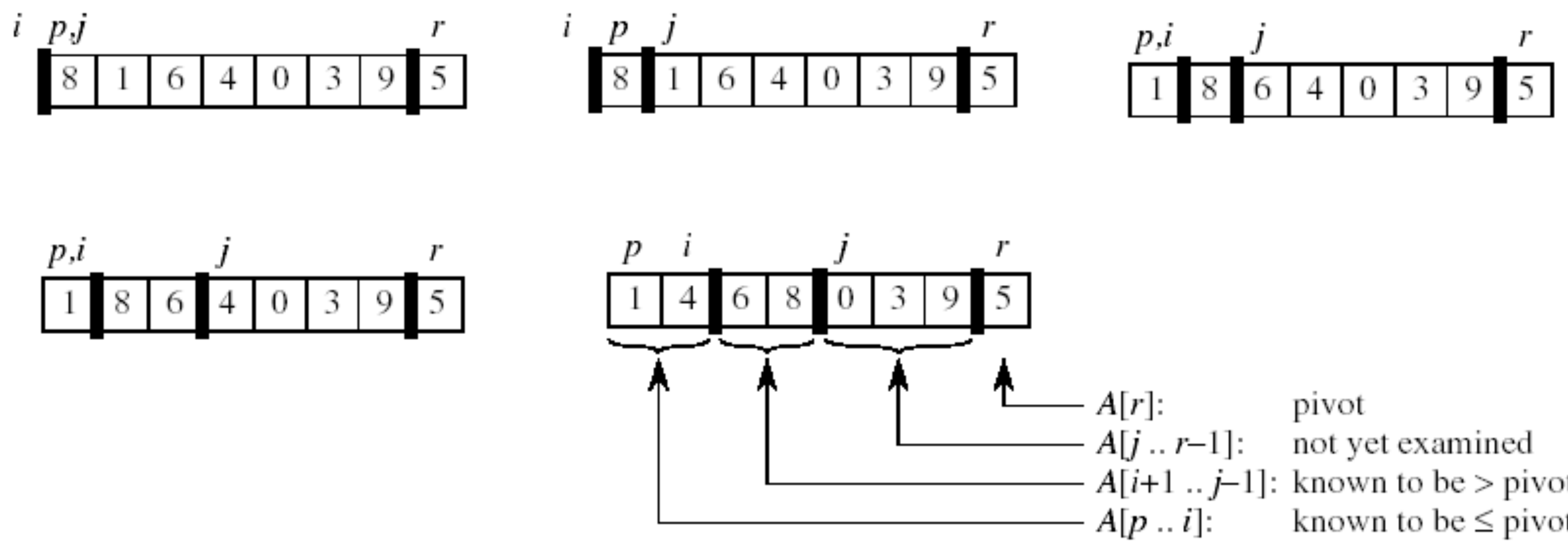
$Z_{1,6}$= {1, 2, 3, 4, 5, 6}   {7} $Z_{8,9}$ = {8, 9, 10}

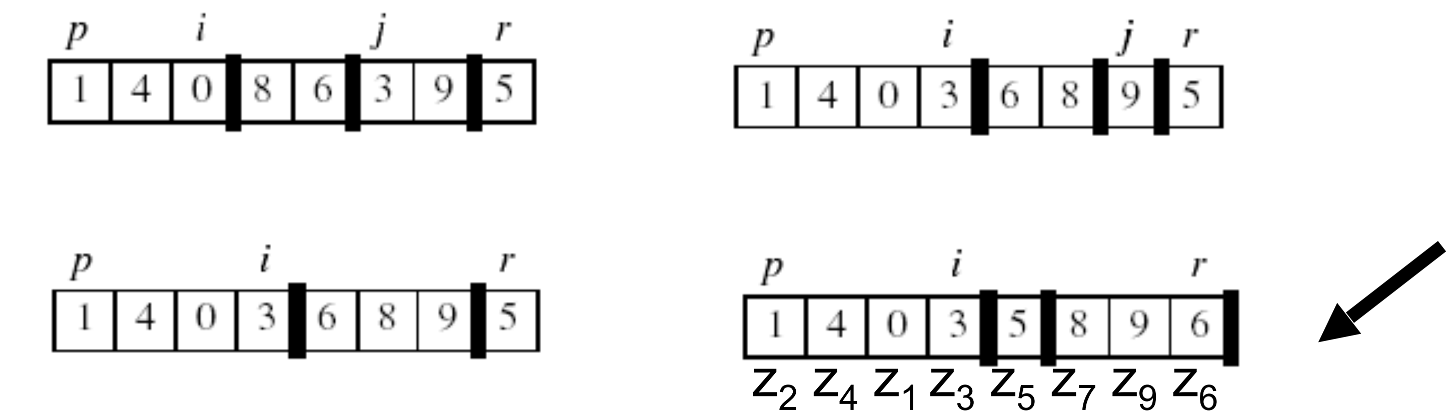
$$\Pr\{z_i \text{ is compared to } z_j\}?$$

- Case 1: pivot x chosen such as: $z_i < x < z_j$
  - z_i and z_j will never be compared
- Case 2: z_i or z_j is the pivot
  - z_i and z_j will be compared
  - only if one of them is chosen as pivot before any other element in range z_i to z_j

30

Let's See Why





z2 will never be compared with z6 since z5 (which belongs to [z_2, z_6]) was chosen as a pivot first !

Probability of comparing z_i with z_j

Pr{ z_i is compared to z_j }

= Pr{ z_i is the first pivot chosen from Z_{ij} } +

Pr{ z_j is the first pivot chosen from Z_{ij} }

= 1/(j - i + 1) + 1/(j - i + 1) = 2/(j - i + 1)

- There are j – i + 1 elements between z_i and z_j
  - Pivot is chosen randomly and independently
  - The probability that any particular element is the first one chosen is 1/(j - i + 1)

Number of Comparisons in PARTITION

Expected number of comparisons in PARTITION:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \textit{ is compared to } z_j\}$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\lg n)$$

(set k=j-i) (harmonic series)

$$= O(n \lg n)$$

⇒ Expected running time of Quicksort using RANDOMIZED-PARTITION is O(nlgn)

Sorting So Far ...

- Insertion sort:
  - Easy to code
  - Fast on small inputs (less than ~50 elements)
  - Fast on nearly-sorted inputs
  - $O(n^2)$ worst case
  - $O(n^2)$ average (equally-likely inputs) case
  - $O(n^2)$ reverse-sorted case

Sorting So Far ...

- Merge sort:
  - Divide-and-conquer appoarch
    - Split array in half
    - Recursively sort subarrays
    - Linear-time merge step
  - O(nlgn) worst case
  - Doesn’t sort in place

Sorting So Far ...

- Heap sort:
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children's keys
  - O(nlgn) worst case
  - Sorts in place
  - Not stable

Sorting So Far ...

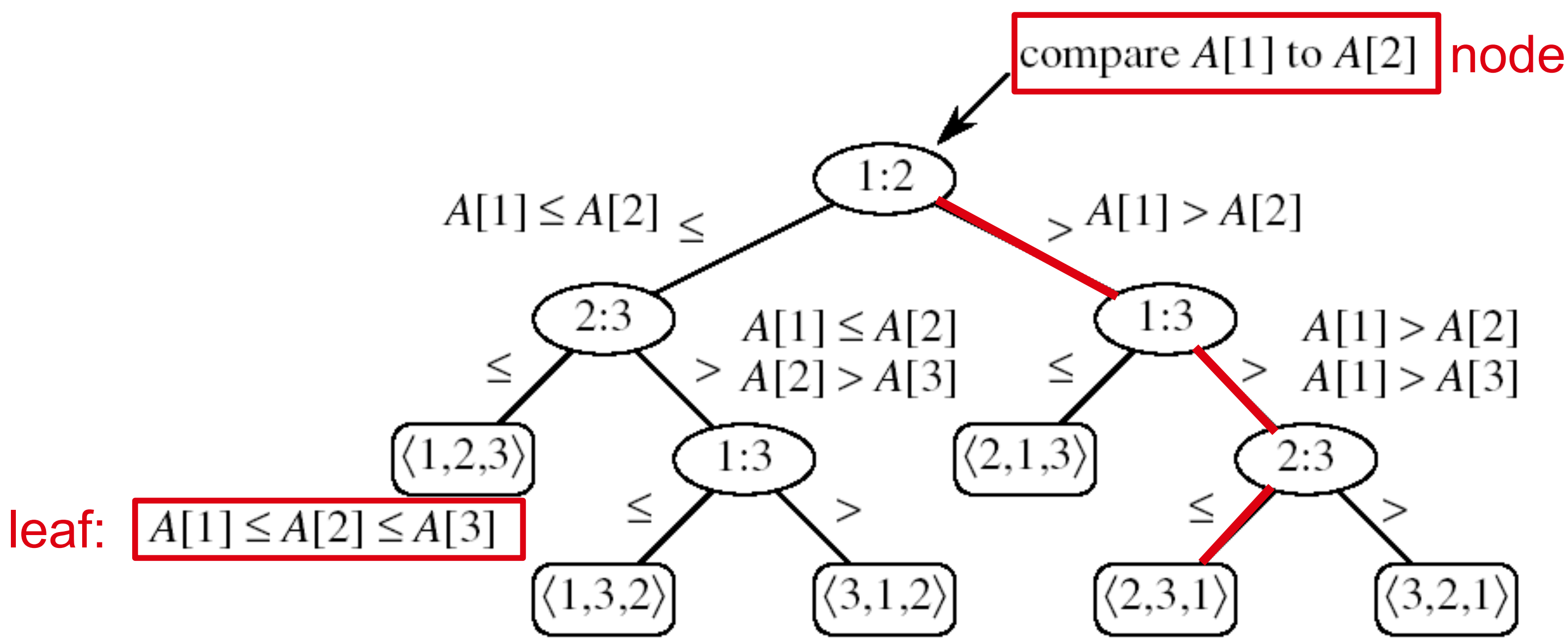
- Quick sort:
  - Divide-and-conquer:
    - Partition array into two subarrays, recursively sort
    - No merge step needed!
  - $O(n\lg n)$ average case
  - Fast in practice
  - $O(n^2)$ worst case
    - Naïve implementation: worst case on sorted input
    - Address this with randomized quicksort

How Fast Can We Sort?

- We will provide a lower bound, then beat it
  - *How do you suppose we'll beat it?*
- First, an observation: all of the sorting algorithms so far are *comparison sorts*
  - The only operation used to gain ordering information about a sequence is the pairwise comparison of two elements
  - Theorem: all comparison sorts are $\Omega(n \lg n)$

Decision Trees

- *Decision trees* provide an abstraction of comparison sorts
  - A decision tree represents the comparisons made by a comparison sort. Every thing else ignored
- *What do the leaves represent?*
- *How many leaves must there be?*

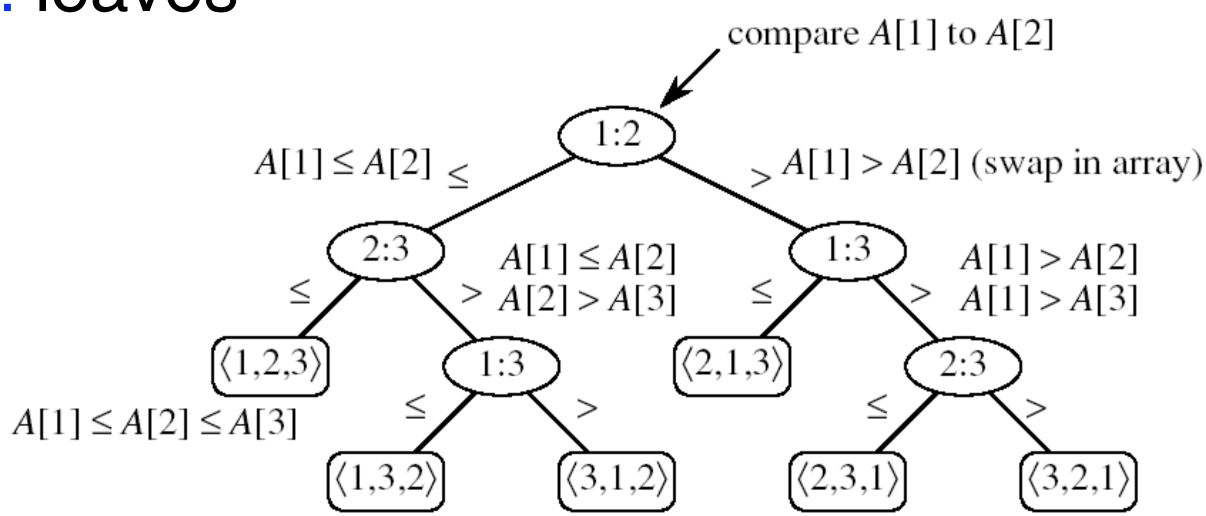


Decision Trees

- Decision trees can model comparison sorts. For a given algorithm:
  - One tree for each *n*
  - Tree paths are all possible execution traces
  - Worst-case number of comparisons depends on the length of the longest path from the root to a leaf (i.e., the height of the decision tree)
- *What is the asymptotic height of any decision tree for sorting n elements?*
  - Answer: $\Omega(n \lg n)$ (now let's prove it…)

The minimum # of leaves of a decision tree

- All permutations on n elements must appear as one of the leaves in the decision tree: n! permutations
- At least n! leaves



Any binary tree of height h has at most 2^h leaves

Lower Bound For Comparison Sorting

- *Theorem:* Any decision tree that sorts *n* elements has height $\Omega(n \lg n)$
  - So we have: $n! \le 2^h$
  - Taking logarithms: $\lg(n!) \le h$
  - Stirling's approximation tells us: $n! > \left(\frac{n}{e}\right)^n$
  - Thus: $h \ge \lg\left(\frac{n}{e}\right)^n$

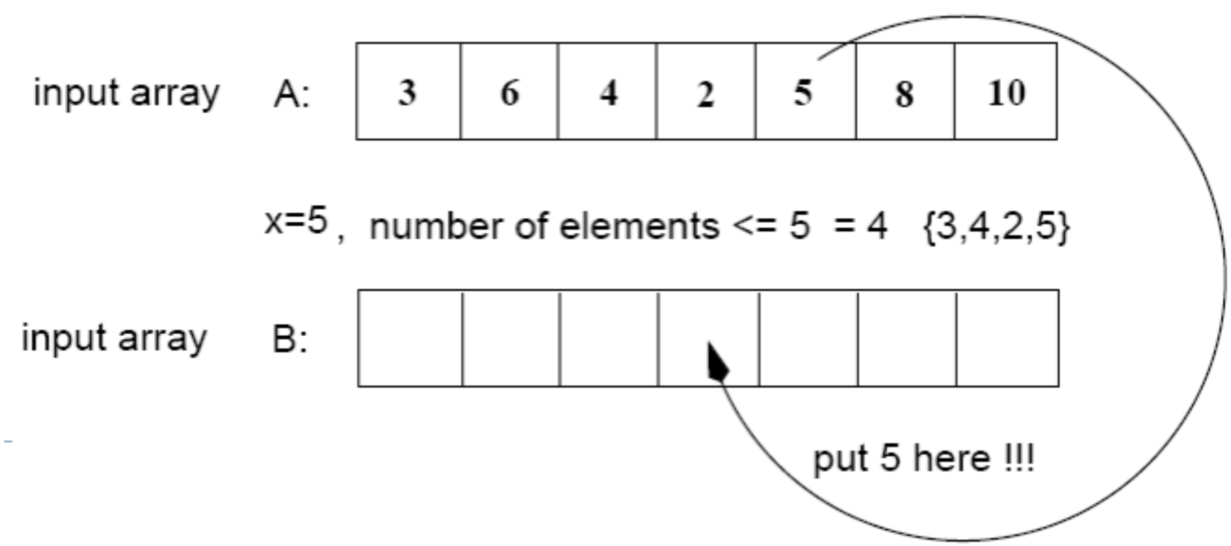
$$= n \lg n - n \lg e$$

$$= \Omega(n \lg n)$$

Sorting In Linear Time

Counting sort - no comparisons between elements!

- Assumptions
  - Sort n integers which are in the range [0 ... r]
  - r is in the order of n, that is, r=O(n)
- Idea
  - For each element x, find the number of elements <=x
  - Place x into its correct position in the output array



Step1: find the no. of times integer A[i] appears in A

input array A:

3	6	4	1	3	4	1	4

allocate C

1	2	3	4	5	6
0	0	0	0	0	0

Allocate $C[1..r]$ (histogram)

i=1, A[1]=3

1	2	3	4	5	6
0	0	1	0	0	0

C[A[1]]=C[3]=1

For $1 \le i \le n, ++C[A[i]];$

i=2, A[2]=6

1	2	3	4	5	6
0	0	1	0	0	1

C[A[2]]=C[6]=1

i=3, A[3]=4

1	2	3	4	5	6
0	0	1	1	0	1

C[A[3]]=C[4]=1

⋮

i=8, A[8]=4

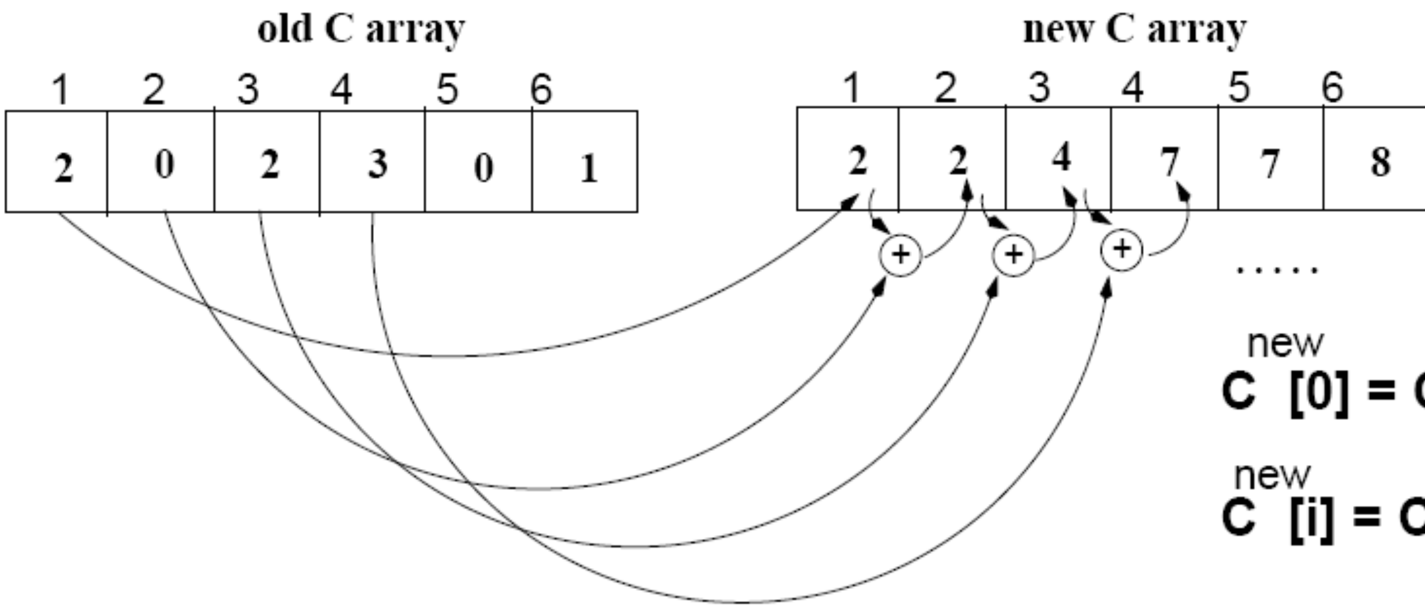
1	2	3	4	5	6
2	0	2	3	0	1

C[A[8]]=C[4]=3

C[i] = number of times element i appears in A (i.e., frequencies)

Step 2: find the no. of elements ≤ A[i]

(i.e., cumulative sums)



$$C^{new}[0] = C^{old}[0]$$

$$C^{new}[i] = C^{new}[i-1] + C^{old}[i]$$

C[i] = # elements <= i

Analysis of Counting Sort

Alg.: COUNTING-SORT(A, B, n, k)

1. **for** i ← 0 **to r**
2. **do** C[i] ← 0
3. **for** j ← 1 **to** n
4. **do** C[A[j]] ← C[A[j]] + 1

   C[i] contains the number of elements equal to i
5. **for** i ← 1 **to r**
6. **do** C[i] ← C[i] + C[i -1]

   C[i] contains the number of elements ≤ i
7. **for** j ← n **downto** 1
8. **do** B[C[A[j]]] ← A[j]
9. C[A[j]] ← C[A[j]] - 1

Example

A

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C

0	1	2	3	4	5
2	0	2	3	0	1

(frequencies)

C

0	1	2	3	4	5
2	2	4	7	7	8

(cumulative sums)

B

1	2	3	4	5	6	7	8
						3	

C

0	1	2	3	4	5
2	2	4	6	7	8

B

1	2	3	4	5	6	7	8
	0					3	

C

0	1	2	3	4	5
1	2	4	6	7	8

B

1	2	3	4	5	6	7	8
	0				3	3	

C

0	1	2	3	4	5
1	2	4	5	7	8

B

1	2	3	4	5	6	7	8
	0		2		3	3	

C

0	1	2	3	4	5
1	2	3	5	7	8

Example

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B	0	0		2		3	3	

	0	1	2	3	4	5
C	0	2	3	5	7	8

	1	2	3	4	5	6	7	8
B	0	0		2	3	3	3	5

	0	1	2	3	4	5
C	0	2	3	4	7	7

	1	2	3	4	5	6	7	8
B	0	0		2	3	3	3	

	0	1	2	3	4	5
C	0	2	3	4	7	8

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

Analysis of Counting Sort

Alg.: COUNTING-SORT(A, B, n, k)

```
for i ← 0 to r                          Θ(r)
    do C[ i ] ← 0
for j ← 1 to n                          Θ(n)
    do C[A[ j ]] ← C[A[ j ]] + 1
    C[i] contains the number of elements equal to i
for i ← 1 to r                          Θ(r)
    do C[ i ] ← C[ i ] + C[i -1]
    C[i] contains the number of elements ≤ i
for j ← n downto 1                      Θ(n)
    do B[C[A[ j ]]] ← A[ j ]
       C[A[ j ]] ← C[A[ j ]] - 1
```

Overall time: $\Theta(n + r)$

Counting Sort

- Overall time: $\Theta(n + r)$
- In practice we use COUNTING sort when $r = O(n)$

  $\Rightarrow$ running time is $\Theta(n)$
- Counting sort is stable
- Counting sort is not in place sort

Radix Sort

- Represents keys as d-digit numbers in some base-k
  - e.g., key = $x_1x_2...x_d$ where $0 \le x_i \le k-1$

- Example: key=15
  - key_{10} = 15, d=2, k=10 where $0 \le x_i \le 9$
  - key_2 = 1111, d=4, k=2 where $0 \le x_i \le 1$

Radix Sort

- Assumptions
  - $d = \Theta(1)$ and $k = O(n)$
- Sorting looks at one column at a time
  - For a d digit number, sort the least significant digit first, using a stable sort algorithm
  - Continue sorting on the next least significant digit, (stable sort) until all digits have been sorted
  - Requires only d passes through the list
- Running time: $O(d(n+k))$

326
453
608
835
751
435
704
690

Order Statistics

- 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-th order statistic
  - If *n* is even, there are 2 medians
- *How can we calculate order statistics?*
- *What is the running time?*

Order Statistics

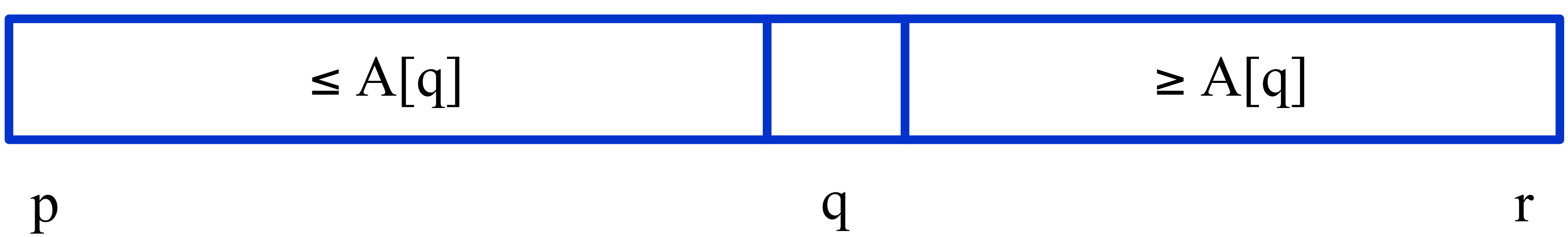
- *How many comparisons are needed to find the minimum element in a set? The maximum?*
- *Can we find the minimum and maximum with less than twice the cost?*
- Yes
  - Walk through elements by pairs
    - Compare each element in pair to the other
    - Compare the largest to maximum, smallest to minimum
  - Total cost: 3 comparisons per 2 elements = O(3n/2)

Finding Order Statistics: The Selection Problem

- A more interesting problem is *selection*: finding the *i*-th smallest element of a set
- We will show:
  - A practical randomized algorithm with O(n) expected running time
  - A cool algorithm of theoretical interest only with O(n) worst-case running time

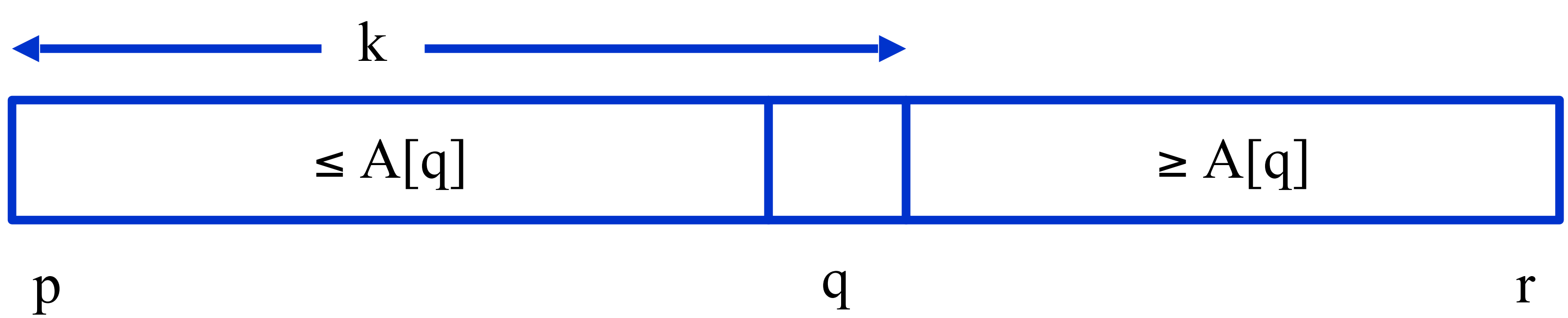
Randomized Selection

- Key idea: use partition() from quicksort
  - But, only need to examine one subarray
  - This savings shows up in running time: O(n)
  - q = RandomizedPartition(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];   // the pivot value is the answer
    if (i < k) then
        return RandomizedSelect(A, p, q-1, i);
    else
        return RandomizedSelect(A, q+1, r, i-k);
```



Analyzing Randomized Selection

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

Analyzing Randomized Selection

- Average case
  - For upper bound, assume i-th element always falls in larger side of partition:

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

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

  - Let's show that T(n) = O(n) by substitution

Analyzing Randomized Selection

- Assume T(n) ≤ cn for sufficiently large c:

$$
\begin{aligned}
T(n) &\le \frac{2}{n}\sum_{k=n/2}^{n-1} T(k) + \Theta(n) \\
&\le \frac{2}{n}\sum_{k=n/2}^{n-1} ck + \Theta(n) \\
&= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k\right) + \Theta(n) \\
&= \frac{2c}{n}\left(\frac{1}{2}(n-1)n - \frac{1}{2}\left(\frac{n}{2}-1\right)\frac{n}{2}\right) + \Theta(n) \\
&= c(n-1) - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n)
\end{aligned}
$$

Analyzing Randomized Selection

- Assume T(*n*) ≤ *cn* for sufficiently large *c*:

$$
\begin{aligned}
T(n) &\le c(n-1) - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n) \\
&= cn - c - \frac{cn}{4} + \frac{c}{2} + \Theta(n) \\
&= cn - \frac{cn}{4} - \frac{c}{2} + \Theta(n) \\
&= cn - \left(\frac{cn}{4} + \frac{c}{2} - \Theta(n)\right) \\
&\le cn \quad \text{(if c is big enough)}
\end{aligned}
$$

Worst-Case Linear-Time Selection

- Randomized algorithm works well in practice
- What follows is a worst-case linear time algorithm, really of theoretical interest only
- Basic idea: generate a good partitioning element
  - step1: divide n elements into n/5 groups
  - step2: find median of each group of 5 elements, using insertion-sorting
  - step3: recursively SELECT to find the median x of n/5 medians found in step 2
  - step 4: use x found in step 3 to partition the array, let k be one plus the number of elements in the low side of partition
  - step 5: if i=k, return x; else recursively SELECT i-th element in low side, if i<k; or SELECT (i-k)-th element in high side, if i>k.

What's next...

- Hash Tables (Chapter 11)
- Binary Search Trees (Chapter 12)