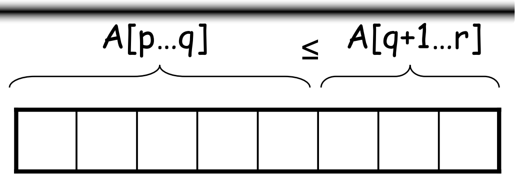
CSC 611: Analysis of Algorithms

Lecture 6

Divide and Conquer: Quick Sort

Quicksort



- Sort an array *A*[p…r]
- **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]
  - The index (pivot) *q* is computed
- **Conquer**
  - Recursively sort *A*[p..q] and *A*[q+1..r] using Quicksort
- **Combine**
  - Trivial: the arrays are sorted in place ⇒ no work needed to combine them: the entire array is now sorted

QUICKSORT

Alg.: QUICKSORT(A, p, r)

if p < r

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

QUICKSORT (A, p, q)

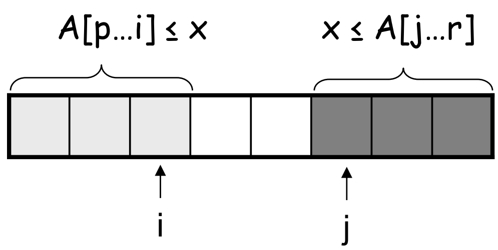
QUICKSORT (A, q+1, r)

Partitioning the Array

- Idea
  - Select a pivot element **x** around which to partition
  - Grows two regions

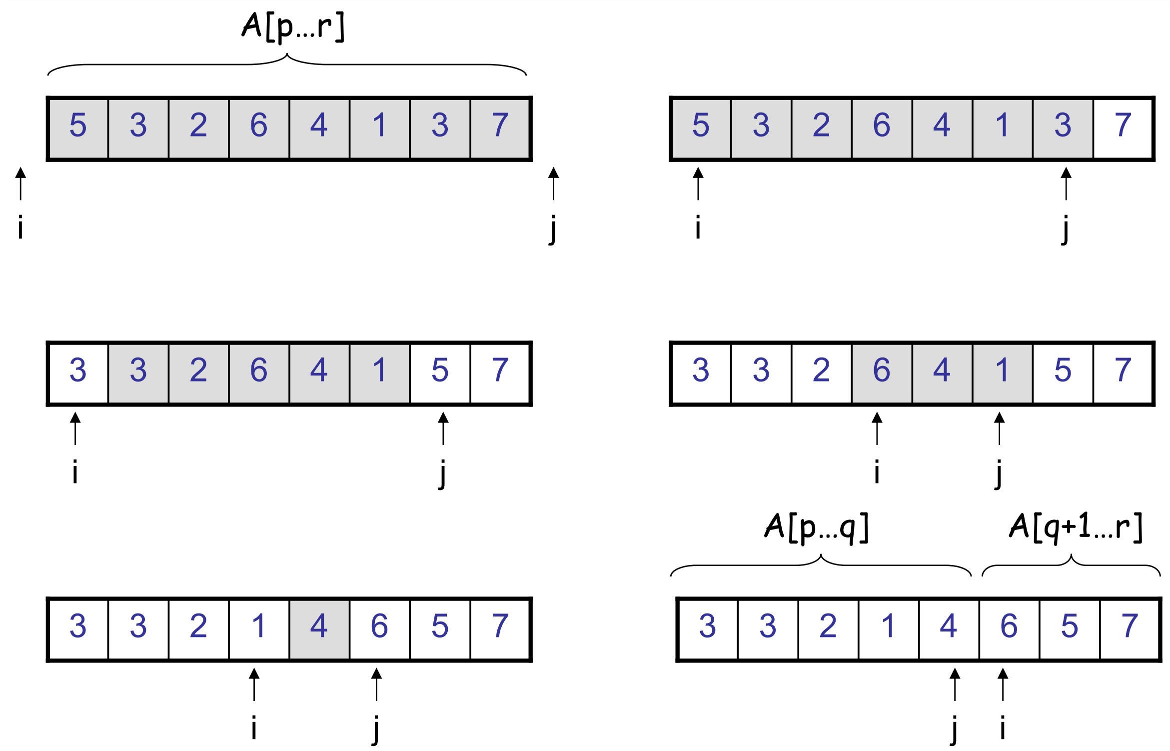
    $A[p\ldots i] \le x$

    $x \le A[j\ldots r]$



  - For now, choose the value of the first element as the pivot **x**

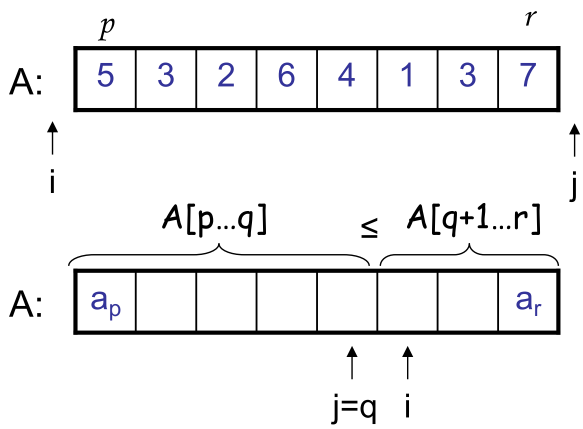
Example



Partitioning the Array

Alg. PARTITION (A, p, r)

```
x ← A[p]
i ← p - 1
j ← r + 1
while TRUE
        do repeat j ← j - 1
              until A[j] ≤ x
           repeat i ← i + 1
              until A[i] ≥ x
           if i < j
              then exchange A[i] ⇔ A[j]
           else return j
```



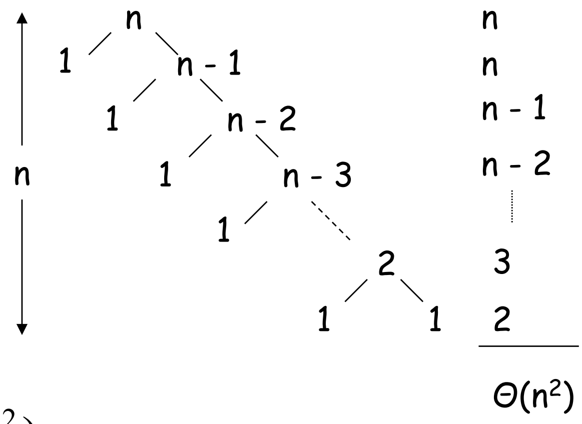
Running time: $\Theta(n)$

$n = r - p + 1$

Performance of Quicksort

- Worst-case partitioning
  - One region has 1 element and one has n – 1 elements
  - Maximally unbalanced
- Recurrence

  T(n) = T(n – 1) + T(1) + Θ(n)



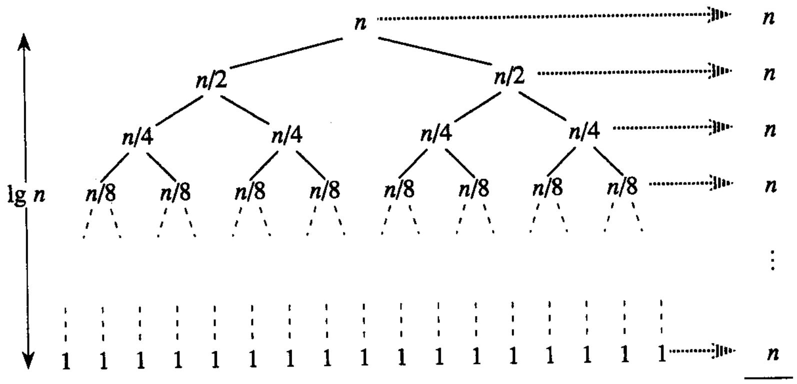
$$= \quad n + \left(\sum_{k=1}^{n} k \right) - 1 = \Theta(n^2)$$

Performance of Quicksort

- Best-case partitioning
  - Partitioning produces two regions of size **n/2**
- Recurrence

  T(n) = 2T(n/2) + Θ(n)

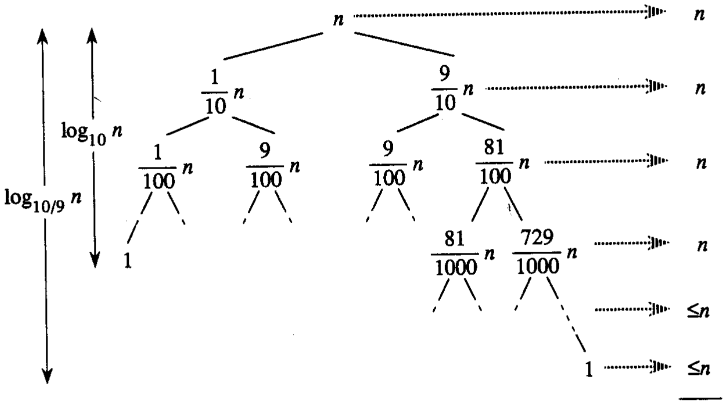
  T(n) = Θ(nlgn) (Master theorem)



Performance of Quicksort

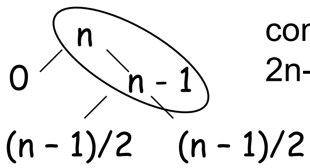
- Balanced partitioning
  - Average case is closer to best case than to worst case
  - (if partitioning always produces a **constant** split)
- E.g.: 9-to-1 proportional split

$$T(n) = T(9n/10) + T(n/10) + n$$



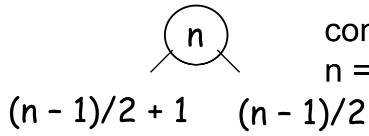
Performance of Quicksort

- Average case
  - All permutations of the input numbers are equally likely
  - On a random input array, we will have a mix of well balanced and unbalanced splits
  - Good and bad splits are randomly distributed throughout the tree



combined cost:
$2n-1 = \Theta(n)$

Alternation of a bad and a good split



combined cost:
$n = \Theta(n)$

Nearly well balanced split

- Running time of Quicksort when levels alternate between good and bad splits is O(nlgn)

Randomizing Quicksort

- Randomly permute the elements of the input array before sorting
- Modify the PARTITION procedure
  - First we exchange element $A[p]$ with an element chosen at random from $A[p…r]$
  - Now the pivot element $x = A[p]$ is equally likely to be any one of the original $r - p + 1$ elements of the subarray

Randomized Algorithms

- The behavior is determined in part by values produced by a random-number generator
  - RANDOM(a, b) returns an integer r, where $a \leq r \leq b$ and each of the $b-a+1$ possible values of r is equally likely
- Algorithm generates randomness in input
- No input can consistently elicit worst case behavior
  - Worst case occurs only if we get "unlucky" numbers from the random number generator

Randomized PARTITION

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

i ← RANDOM(p, r)

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

return PARTITION(A, p, r)

Randomized Quicksort

Alg. : 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)

Worst-Case Analysis of Quicksort

- $T(n)$ = worst-case running time
- $T(n) = \max_{1 \le q \le n-1} (T(q) + T(n-q)) + \Theta(n)$
- Use substitution method to show that the running time of Quicksort is $O(n^2)$
- Guess $T(n) = O(n^2)$
  - Induction goal: $T(n) \le cn^2$
  - Induction hypothesis: $T(k) \le ck^2$ for any $k \le n$

Worst-Case Analysis of Quicksort

- Proof of induction goal:

  $T(n) \le \max_{1 \le q \le n-1} (cq^2 + c(n-q)^2) + \Theta(n) =$

  $= c \times \max_{1 \le q \le n-1} (q^2 + (n-q)^2) + \Theta(n)$

- The expression $q^2 + (n-q)^2$ achieves a maximum over the range $1 \le q \le n-1$ at the endpoints of this interval

  The second derivative of the expression with respect to **q** *is positive*

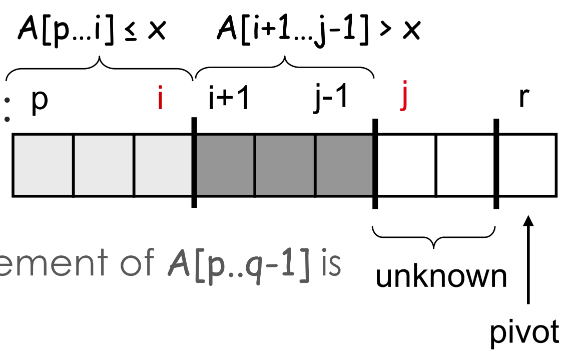
$\max_{1 \le q \le n-1} (q^2 + (n - q)^2) = 1^2 + (n - 1)^2 = n^2 - 2(n - 1)$

$T(n) \le cn^2 - 2c(n - 1) + \Theta(n)$

$\le cn^2$

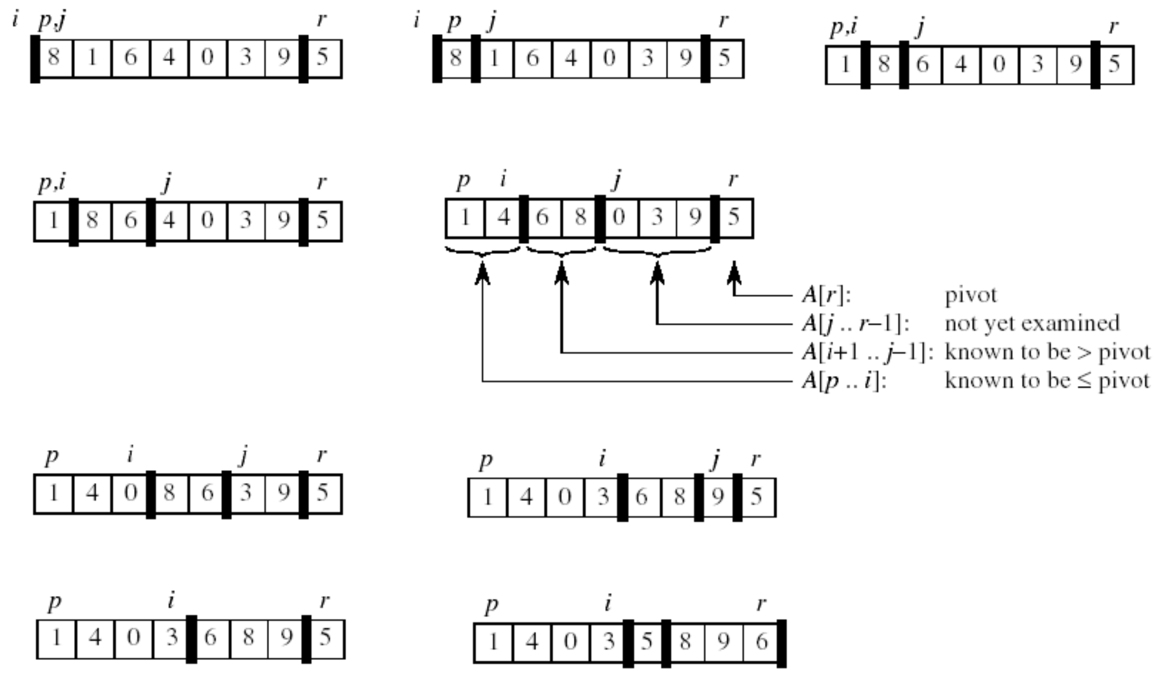
Another Way to PARTITION

- 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)



- Note: the pivot element is not included in any of the two subarrays

Example



Another Way to PARTITION

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

$x \leftarrow A[r]$

$i \leftarrow p - 1$

for $j \leftarrow p$ **to** $r - 1$

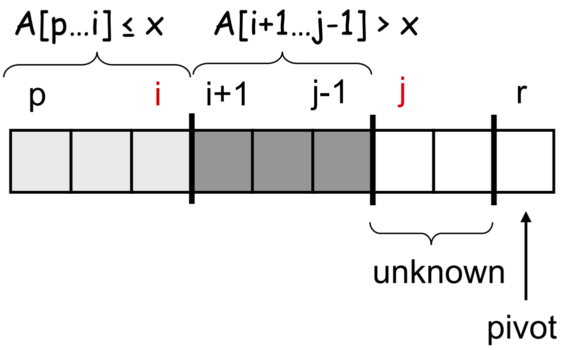
do if $A[j] \le x$

then $i \leftarrow i + 1$

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

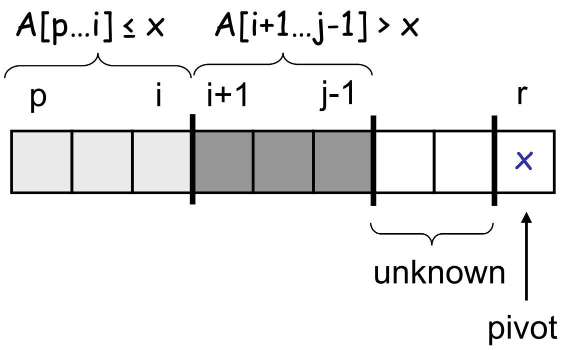
exchange $A[i + 1] \leftrightarrow A[r]$

return $i + 1$



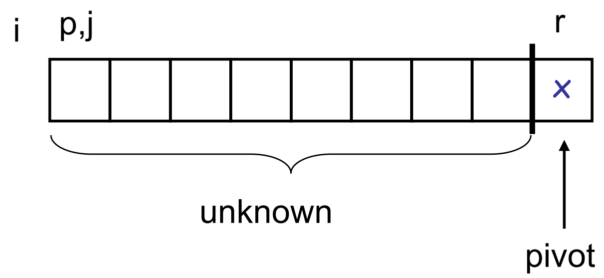
Chooses the last element of the array as a pivot
Grows a subarray [p..i] of elements ≤ x
Grows a subarray [i+1..j-1] of elements >x
Running Time: Θ(n), where n=r-p+1

Loop Invariant



1. All entries in $A[p\ ..\ i]$ are smaller than the pivot
2. All entries in $A[i + 1\ ..\ j - 1]$ are strictly larger than the pivot
3. $A[r]$ = pivot
4. $A[j\ ..\ r - 1]$ elements not yet examined

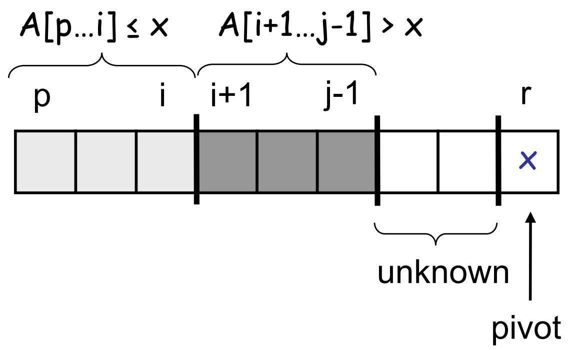
Loop Invariant



Initialization: Before the loop starts:

- $A[r]$ is the pivot
- subarrays $A[p..i]$ and $A[i+1..j-1]$ are empty
- All elements in the array are not examined

Loop Invariant



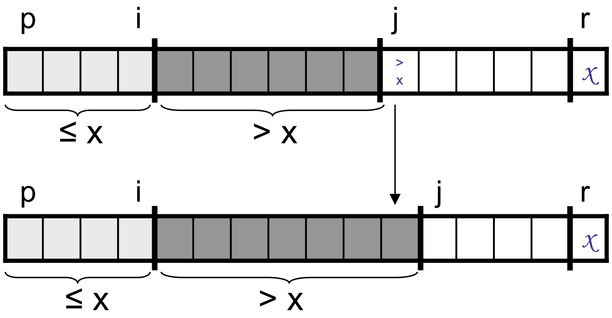
Maintenance: While the loop is running

- if $A[j] \leq$ pivot, then i is incremented, $A[j]$ and $A[i+1]$ are swapped and then j is incremented
- If $A[j] >$ **pivot**, then increment only j

Maintenance of Loop Invariant

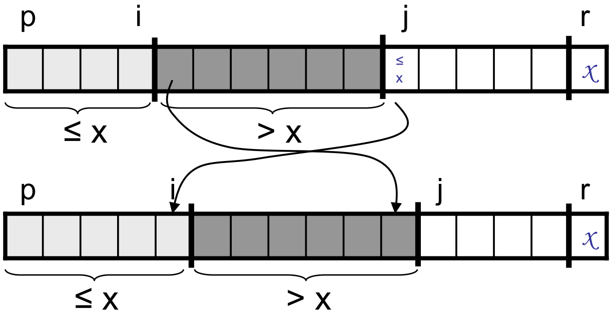
If A[j] > pivot:

- only increment j

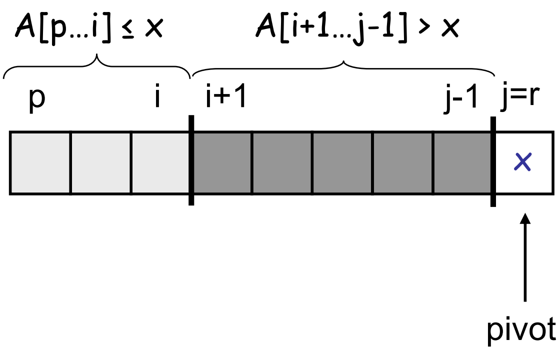


If A[j] ≤ pivot:

- i is incremented, A[j] and A[i] are swapped and then j is incremented



Loop Invariant



Termination: When the loop terminates:

- $j = r \Rightarrow$ all elements in A are partitioned into one of the three cases: $A[p \, . \, . \, i\,] \leq$ pivot, $A[i + 1 \, . \, . \, r - 1] >$ pivot, and $A[r] =$ pivot

Randomized Quicksort

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

The running time of Quicksort is dominated by PARTITION !!

then $q \leftarrow$ 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)*

$x \leftarrow A[r]$
$i \leftarrow p - 1$ } O(1) - constant
for $j \leftarrow p$ **to** $r - 1$
    do if $A[\,j\,] \le x$
        then $i \leftarrow i + 1$
            exchange $A[i] \leftrightarrow A[j]$

} Number of comparisons between the pivot and the other elements

exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$ } O(1) - constant

Need to compute the total number of comparisons performed in all calls to PARTITION

Random Variables and Expectation

*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

E.g.: X = face of one fair dice

– Possible values: {1, 2, 3, 4, 5, 6}

– Probability to take any of the values: 1/6

Random Variables and 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

E.g.: X = face of one fair dice

E[X] = 1 × 1/6 + 2 × 1/6 + 3 × 1/6 + 4 × 1/6 + 5 × 1/6 + 6 × 1/6 = 3.5

Example

E.g.: flipping two coins:

- Earn \$3 for each head, lose \$2 for each tail
- X: random variable representing your earnings
- Three possible values for variable X:
  - 2 heads ⇒ x = \$3 + \$3 = \$6, Pr{2 H's} = ¼
  - 2 tails ⇒ x = -\$2 - \$2 = -\$4, Pr{2 T's} = ¼
  - 1 head, 1 tail ⇒ x = \$3 - \$2 = \$1, Pr{1 H, 1 T} = ½
- The expected value of X is:

$$E[X] = 6 \times Pr\{2\ H's\} + 1 \times Pr\{1\ H,\ 1\ T\} - 4 \times Pr\{2\ T's\}$$

$$= 6 \times \tfrac{1}{4} + 1 \times \tfrac{1}{2} - 4 \times \tfrac{1}{4} = 1$$

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 is: $\mathbf{E[X_A] = Pr\{A\}}$

- Proof: $E[X_A] = E[I\{A\}] = 1 \times Pr\{A\} + 0 \times Pr\{\bar{A}\} = Pr\{A\}$

Example

- Determine the expected number of heads obtained when flipping a coin
  - Space of possible values: S = {H, T}
  - Random variable Y: takes on the values H and T, each with probability ½

- Indicator random variable X_H: the coin coming up heads (Y = H)
  - Counts the number of heads obtain in the flip
  - $X_H = I\{Y = H\} = \begin{cases} 1 & \text{if } Y = H \\ 0 & \text{if } Y = T \end{cases}$

- The expected number of heads obtained in one flip of the coin is:

  $E[X_H] = E[I\{Y = H\}] = 1 \times Pr\{Y = H\} + 0 \times Pr\{Y = T\} =$

  $= 1 \times ½ + 0 \times ½ = ½$

Analysis of Randomized Quicksort

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

The running time of Quicksort is dominated by PARTITION !!

if p < r

then $q \leftarrow$ 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*)

$x \leftarrow A[r]$

$i \leftarrow p - 1$

O(1) - constant

for $j \leftarrow p$ **to** $r - 1$

do if $A[\,j\,] \le x$

then $i \leftarrow i + 1$

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

Number of comparisons between the pivot and the other elements

exchange $A[i + 1] \leftrightarrow A[r]$

return $i + 1$

O(1) - constant

Need to compute the total number of comparisons performed in all calls to PARTITION

Number of Comparisons in PARTITION

- Need to compute the **total number of comparisons** performed **in all calls to PARTITION**
- $X_{ij} = I\{z_i \text{ is compared to } z_j\}$
  - For any comparison during the entire execution of the algorithm, not just during one call to PARTITION

When Do We Compare Two Elements?

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

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

- 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

When Do We Compare Elements z_i, z_j?

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} $Z_{8,10}$ = {8, 9, 10}

- If pivot x chosen such as: $z_i < x < z_j$
  - z_i and z_j will never be compared
- If 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
- Only the pivot is compared with elements in both sets

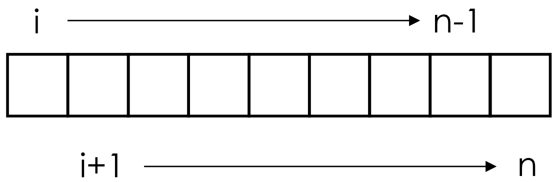
Number of Comparisons in PARTITION

- During the entire run of Quicksort each pair of elements is compared at most once
  - Elements are compared only to the pivot element
  - Since the pivot is never included in future calls to PARTITION, it is never compared to any other element

Number of Comparisons in PARTITION

- Each pair of elements can be compared at most once
  - X_{ij} = I {z_i is compared to z_j }
- Define X as the total number of comparisons performed by the algorithm

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



Number of Comparisons in PARTITION

- X is an indicator random variable
  - Compute the **expected value**

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

by linearity of expectation

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

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

Number of Comparisons in PARTITION

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:

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

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Change variable: k = j – i ⇒

$$E[X] = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

We have that: $\sum_{k=1}^{n} \frac{2}{k+1} < \sum_{k=1}^{n} \frac{2}{k}$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

We have that: $\sum_{k=1}^{n} \frac{2}{k} = O(\lg n)$

$$= \sum_{i=1}^{n-1} O(\lg n)$$

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

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

Selection

- General Selection Problem:
  - select the **i-th** smallest element form a set of **n** distinct numbers
  - that element is larger than exactly **i - 1** other elements
- The selection problem can be solved in **O(nlgn)** time
  - Sort the numbers using an **O(nlgn)**-time algorithm, such as merge sort
  - Then return the **i**-th element in the sorted array

Medians and Order Statistics

Def.: The i-th ***order statistic*** *of a set of n elements is the i-th smallest element.*

- The minimum of a set of elements:
  - The first order statistic i = 1
- The maximum of a set of elements:
  - The n-th order statistic i = n
- The median is the "halfway point" of the set
  - i = (n+1)/2, is unique when n is odd
  - $i = \lfloor (n+1)/2 \rfloor = n/2$ (lower median) and $\lceil (n+1)/2 \rceil = n/2+1$ (upper median), when n is even

Finding Minimum or Maximum

Alg.: MINIMUM(A, n)

```
min ← A[1]
for i ← 2 to n
    do if min > A[i]
        then min ← A[i]
return min
```

- How many comparisons are needed?
  - n – 1: each element, except the minimum, must be compared to a smaller element at least once
  - The same number of comparisons are needed to find the maximum
  - The algorithm is optimal with respect to the number of comparisons performed

Simultaneous Min, Max

- Find min and max independently
  - Use n – 1 comparisons for each ⇒ total of **2n – 2**
- However, we can do better: at most **3n/2** comparisons
  - Process elements in pairs
  - Maintain the minimum and maximum of elements seen so far
  - Don't compare each element to the minimum and maximum separately
  - Compare the elements of a pair to each other
  - Compare the larger element to the maximum so far, and compare the smaller element to the minimum so far
  - This leads to only 3 comparisons for every 2 elements

Analysis of Simultaneous Min, Max

- Setting up initial values:
  - n is odd: set both **min** and **max** to the first element
  - n is even: compare the first two elements, assign the smallest one to **min** and the largest one to **max**
- Total number of comparisons:
  - n is odd: we do $3(n-1)/2$ comparisons
  - n is even: we do 1 initial comparison + $3(n-2)/2$ more comparisons = $3n/2 - 2$ comparisons

Example: Simultaneous Min, Max

- n = 5 (odd), array A = {2, 7, 1, 3, 4}

1. Set **min** = **max** = 2
2. Compare elements in pairs:
   – 1 < 7 ⇒ compare 1 with **min** and 7 with **max** ⇒ **min** = 1, **max** = 7 } 3 comparisons
   – 3 < 4 ⇒ compare 3 with **min** and 4 with **max** ⇒ **min** = 1, **max** = 7 } 3 comparisons

We performed: 3(n-1)/2 = 6 comparisons

Example: Simultaneous Min, Max

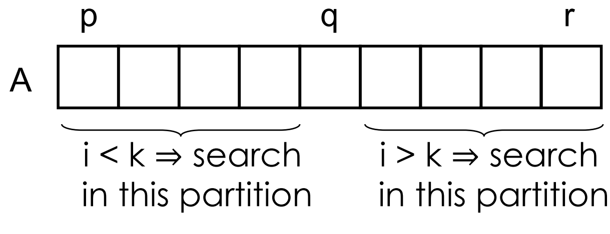
- n = 6 (even), array A = {2, 5, 3, 7, 1, 4}
  1. Compare 2 with 5: 2 < 5 — 1 comparison
  2. Set **min =** 2, **max** = 5
  3. Compare elements in pairs:
     - 3 < 7 ⇒ compare 3 with **min** and 7 with **max**
       ⇒ **min** = 2, **max** = 7 — 3 comparisons
     - 1 < 4 ⇒ compare 1 with **min** and 4 with **max**
       ⇒ **min** = 1, **max** = 7 — 3 comparisons

We performed: 3n/2 - 2 = 7 comparisons

General Selection Problem

- Select the i-th order statistic (i-th smallest element) form a set of n distinct numbers

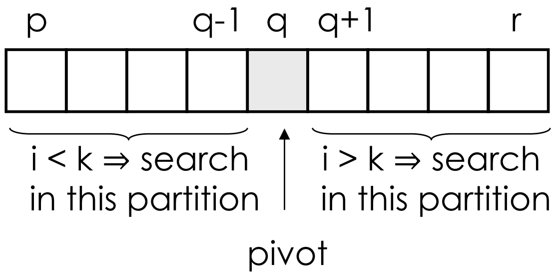
k = q – p + 1



- Idea:
  - Partition the input array similarly with the approach used for Quicksort (use RANDOMIZED-PARTITION)
  - Recurse on one side of the partition to look for the i-th element depending on where i is with respect to the pivot

- We will show that selection of the **i**-th smallest element of the array *A* can be done in $\Theta(n)$ time

Randomized Select



Alg.: RANDOMIZED-SELECT(*A*, p, r, i)

if p = r

  then return *A*[p]

q ←RANDOMIZED-PARTITION(*A*, p, r)

k ← q - p + 1

if i = k ▷pivot value is the answer

  then return *A*[q]

elseif i < k

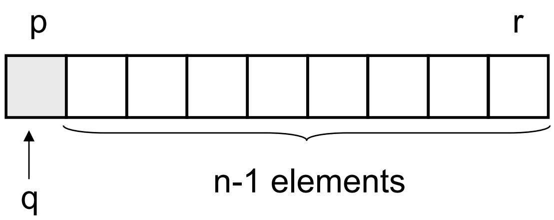
  then return RANDOMIZED-SELECT(*A*, p, q-1, i)

else return RANDOMIZED-SELECT(*A*, q + 1, r, i-k)

Analysis of Running Time

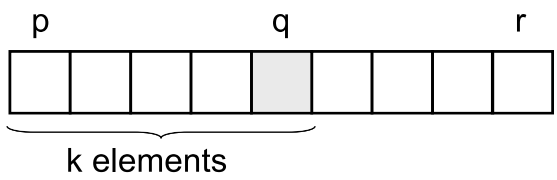
- **Worst case** running time: **$\Theta(n^2)$**
  - If we always partition around the largest/smallest remaining element
  - Partition takes $\Theta(n)$ time
  - $T(n) = \Theta(1)$ (compute k) + $\Theta(n)$ (partition) + $T(n-1)$

    $= 1 + n + T(n-1) = \Theta(n^2)$



Analysis of Running Time

- **Expected** running time (on **average**)
  - Let T(n) be a random variable denoting the running time of RANDOMIZED-SELECT



  - RANDOMIZED-PARTITION is equally likely to return any element of *A* as the pivot ⇒
  - For each k such that $1 \le k \le n$, the subarray $A[p\ ..\ q]$ has k elements (all ≤ pivot) with probability $1/n$

Analysis of Running Time

- When we call RANDOMIZED-SELECT we could have three situations:
  - The algorithm terminates with the answer ($i = k$), or
  - The algorithm recurses on the subarray $A[p..q-1]$, or
  - The algorithm recurses on the subarray $A[q+1..r]$
- The decision depends on where the i-th smallest element falls relative to $A[q]$
- To obtain an upper bound for the running time $T(n)$:
  - assume the i-th smallest element is always in the larger subarray

Analysis of Running Time (cont.)

$$E[T(n)] = \underbrace{\text{Probability that T(n) takes a value} \times \text{The value of the random variable T(n)}}_{\text{Summed over all possible values}}$$

$$E[T(n)] = \frac{1}{n}[T(\max(0,n-1))] + \frac{1}{n}[T(\max(1,n-2))] + \ldots + \frac{1}{n}[T(\max(n-1,0))] + O(n)$$

since select recurses only on the larger partition

PARTITION

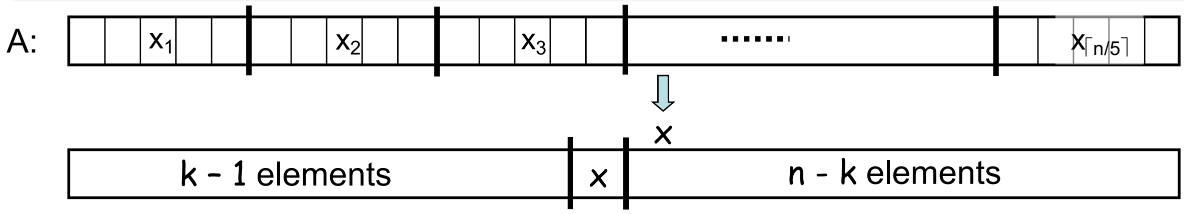
$$= \frac{1}{n}\left[T(n-1) + T(n-2) + T(n-3) \ldots + T(n/2) \ldots + T(n-3) + T(n-2) + T(n-1) \right] + O(n)$$

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

A Better Selection Algorithm

- Can perform Selection in ***O*(n)** Worst Case
- Idea: guarantee a good split on partitioning
  - Running time is influenced by how "balanced" are the resulting partitions
- Use a modified version of PARTITION
  - Takes as input the element around which to partition

Selection in *O*(*n*) Worst Case



1. Divide the **n** elements into groups of 5 ⇒ $\lceil n/5 \rceil$ groups
2. Find the median of each of the $\lceil n/5 \rceil$ groups
   - Use insertion sort, then pick the median
3. Use SELECT recursively to find the median **x** of the $\lceil n/5 \rceil$ medians
4. Partition the input array around **x**, using the modified version of PARTITION
   - There are **k-1** elements on the low side of the partition and **n-k** on the high side
5. If **i** = **k** then return **x**. Otherwise, use SELECT recursively:
   - Find the **i**-th smallest element on the low side if **i** < **k**
   - Find the **(i-k)**-th smallest element on the high side if **i** > **k**

Example

- Find the 11th smallest element in the array:

  A = {12, 34, 0, 3, 22, 4, 17, 32, 3, 28, 43, 82, 25, 27, 34, 2 ,19 ,12 ,5 ,18 ,20 ,33, 16, 33, 21, 30, 3, 47}

1. Divide the array into groups of 5 elements

12	4	43	2	20	30
34	17	82	19	33	3
0	32	25	12	16	47
3	3	27	5	33	
22	28	34	18	21	

Example (cont.)

2. Sort the groups and find their medians

0	4	25	2	20	3
3	3	27	5	16	30
12	17	34	12	21	47
34	32	43	19	33	
22	28	82	18	33	

3. Find the median of the medians

12, 12, 17, 21, 34, 30

Example (cont.)

4. Partition the array around the median of medians (17)

   First partition:

   {12, 0, 3, 4, 3, 2, 12, 5, 16, 3}

   Pivot:

   17 (position of the pivot is q = 11)

   Second partition:

   {34, 22, 32, 28, 43, 82, 25, 27, 34, 19, 18, 20, 33, 33, 21, 30, 47}

To find the 6-th smallest element we would have to recurse our search in the first partition.

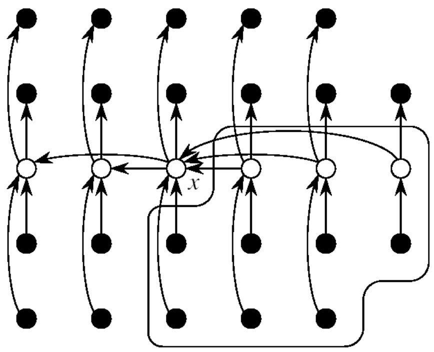
Analysis of Running Time

- Step 1: making groups of 5 elements takes $O(n)$
- Step 2: sorting $n/5$ groups in $O(1)$ time each takes $O(n)$
- Step 3: calling SELECT on $\lceil n/5 \rceil$ medians takes time $T(\lceil n/5 \rceil)$
- Step 4: partitioning the n-element array around x takes $O(n)$
- Step 5: recursion on one partition takes

depends on the size of the partition!!

Analysis of Running Time

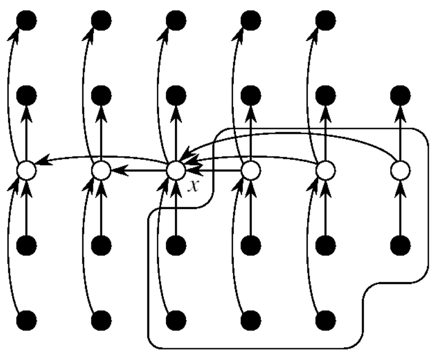
- First determine an upper bound for the sizes of the partitions
  - See how bad the split can be
- Consider the following representation
  - Each column represents one group of 5 (elements in columns are sorted)
  - Columns are sorted by their medians



Analysis of Running Time

- At least half of the medians found in step 2 are ≥ x: $\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil$
- All but two of these groups contribute 3 elements > x

$\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2$ groups with 3 elements > x



- At least $3\left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$ elements greater than x
- SELECT is called on at most $n - \left(\frac{3n}{10} - 6\right) = \frac{7n}{10} + 6$ elements

Recurrence for the Running Time

- Step 1: making groups of 5 elements takes $O(n)$
- Step 2: sorting **n/5** groups in ***O*(1)** time each takes $O(n)$
- Step 3: calling SELECT on $\lceil n/5 \rceil$ medians takes time $T(\lceil n/5 \rceil)$
- Step 4: partitioning the n-element array around **x** takes $O(n)$
- Step 5: recursion on one partition takes time ≤ $T(7n/10 + 6)$
- $T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$
- We will show that $T(n) = O(n)$

Substitution

- $T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$
  Show that $T(n) \le cn$ for some constant $c > 0$ and all $n \ge n_0$

  $$\begin{aligned} T(n) &\le c\lceil n/5 \rceil + c\,(7n/10 + 6) + an \\ &\le cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) \\ &\le cn \qquad \text{if: } -cn/10 + 7c + an \le 0 \end{aligned}$$

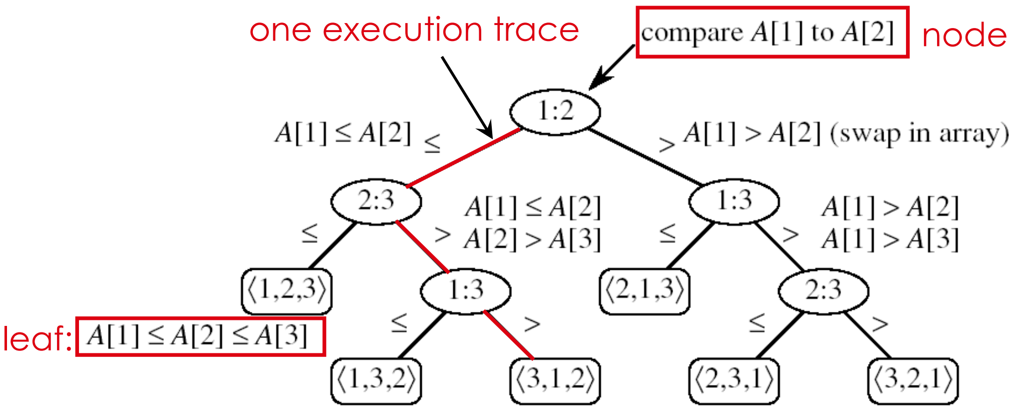
- $c \ge 10a(n/(n-70))$
  - choose $n_0 > 70$ and obtain the value of c

How Fast Can We Sort?

- Insertion sort, Bubble Sort, Selection Sort $\Theta(n^2)$
- Merge sort $\Theta(n\lg n)$
- Quicksort $\Theta(n\lg n)$
- What is common to all these algorithms?
  - These algorithms sort by making comparisons between the input elements
- To sort n elements, comparison sorts must make $\Omega(n\lg n)$ comparisons in the worst case

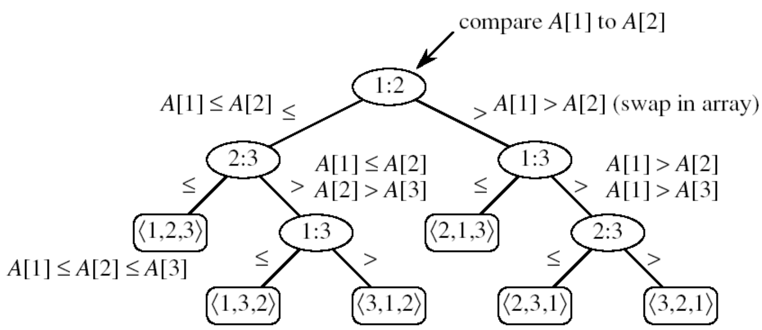
Decision Tree Model

- Represents the comparisons made by a sorting algorithm on an input of a given size: models all possible execution traces
- Control, data movement, other operations are ignored
- Count only the comparisons
- Decision tree for insertion sort on three elements:



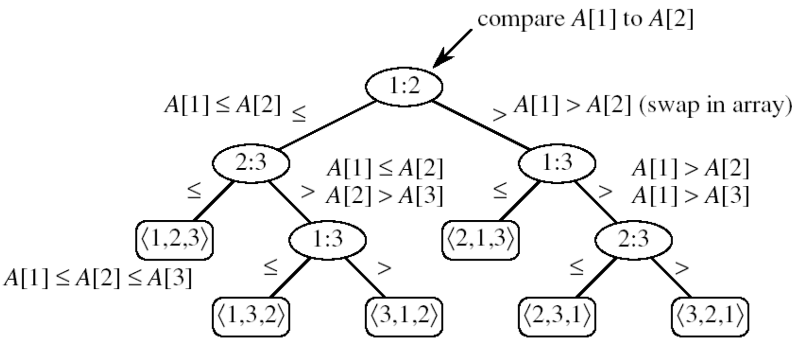
Decision Tree Model

- All permutations on n elements must appear as one of the leaves in the decision tree n! permutations
- Worst-case number of comparisons
  - the length of the longest path from the root to a leaf
  - the height of the decision tree



Decision Tree Model

- Goal: finding a lower bound on the running time on any comparison sort algorithm
  - find a lower bound on the heights of all decision trees for all algorithms



Lemma

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

Proof: induction on **h**

Basis: **h** = 0 ⇒ tree has one node, which is a leaf

$2^h = 1$

Inductive step: assume true for **h-1**

- Extend the height of the tree with one more level
- Each leaf becomes parent to two new leaves

No. of leaves for tree of height **h** =

= 2 × (no. of leaves for tree of height **h-1**)

$\leq 2 \times 2^{h-1}$

$= 2^h$

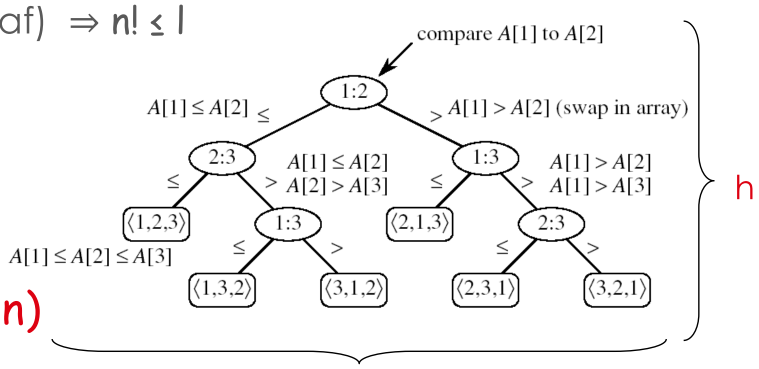
Lower Bound for Comparison Sorts

Theorem: Any comparison sort algorithm requires $\Omega(n\lg n)$ comparisons in the worst case.

Proof: How many leaves does the tree have?

– At least n! (each of the **n!** permutations of the input appears as some leaf) $\Rightarrow n! \le l$

– At most 2^h leaves



$\Rightarrow \quad n! \le l \le 2^h$

$\Rightarrow \quad h \ge \lg(n!) = \Omega(n\lg n)$

leaves l

We can beat the $\Omega(n\lg n)$ running time if we use other operations than comparisons!

Counting Sort

- Assumption:
  – The elements to be sorted are integers in the range **0** to **k**
- Idea:
  – Determine for each input element **x**, the number of elements smaller than **x**
  – Place element **x** into its correct position in the output array

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	2	4	7	7	8

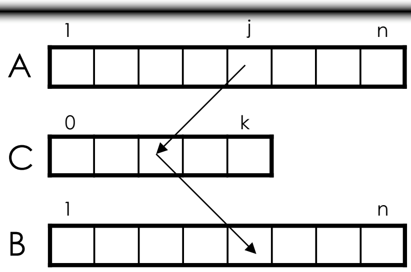
B

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

COUNTING-SORT

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

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

C

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

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 (cont.)

A

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

B

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

C

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

B

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

C

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

B

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

C

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

B

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

Analysis of Counting Sort

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

```
for i ← 0 to k                              } Θ(k)
    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 k                              } Θ(k)
    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: $\Theta(n + k)$

Analysis of Counting Sort

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

  $\Rightarrow$ running time is $\Theta(n)$
- Counting sort is **stable**
  - Numbers with the same value appear in the same order in the output array
  - Important when additional data is carried around with the sorted keys

Radix Sort

- Considers keys as numbers in a base-k number
  - A **d**-digit number will occupy a field of **d** columns
- Sorting looks at one column at a time
  - For a **d** digit number, sort the least significant digit first
  - Continue sorting on the next least significant digit, until all digits have been sorted
  - Requires only **d** passes through the list

326
453
608
835
751
435
704
690

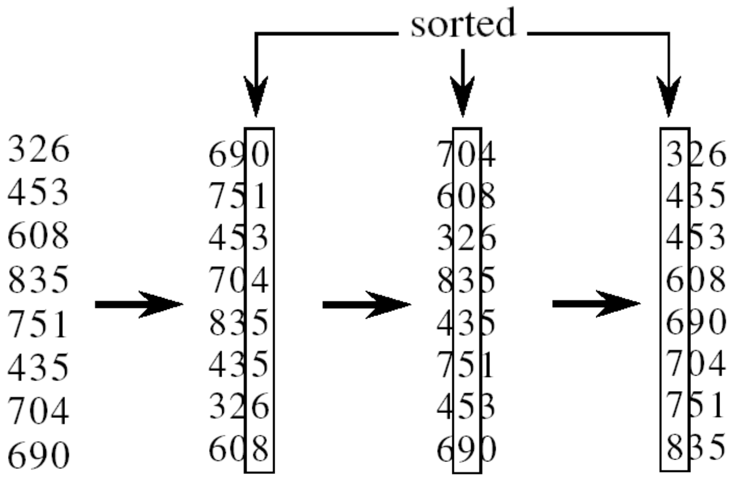
RADIX-SORT

Alg.: RADIX-SORT(*A*, d)

for *i* ← **1 to** d

  do use a stable sort to sort array ***A*** on digit ***i***

- 1 is the lowest order digit, d is the highest-order digit

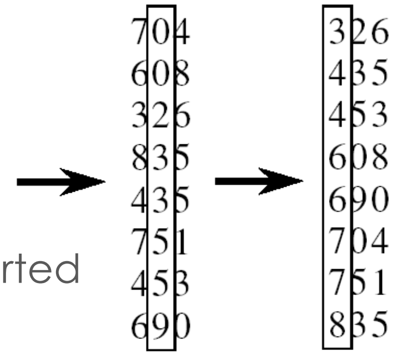


Analysis of Radix Sort

- Given **n** numbers of **d** digits each, where each digit may take up to **k** possible values, RADIX-SORT correctly sorts the numbers in $\Theta(d(n+k))$
  - One pass of sorting per digit takes $\Theta(n+k)$ assuming that we use counting sort
  - There are **d** passes (for each digit)

Correctness of Radix sort

- We use induction on the number **d** of passes through the digits
- **Basis:** If **d** = 1, there's only one digit, trivial
- **Inductive step:** assume digits 1, 2, . . . , **d-1** are sorted
  - Now sort on the **d**-th digit
  - If $a_d < b_d$, sort will put **a** before **b**: correct **a** < **b** regardless of the low-order digits
  - If $a_d > b_d$, sort will put **a** after **b**: correct **a** > **b** regardless of the low-order digits
  - If $a_d = b_d$, sort will leave **a** and **b** in the same order and **a** and **b** are already sorted on the low-order **d-1** digits



Bucket Sort

- Assumption:
  - the input is generated by a random process that distributes elements uniformly over [0, 1)
- Idea:
  - Divide [0, 1) into **n** equal-sized buckets
  - Distribute the **n** input values into the buckets
  - Sort each bucket
  - Go through the buckets in order, listing elements in each one

- **Input:** $A[1 \,..\, n]$, where $0 \le A[i] < 1$ for all i
- **Output:** elements in A sorted
- **Auxiliary array:** $B[0 \,..\, n - 1]$ of linked lists, each list initially empty

BUCKET-SORT

Alg.: BUCKET-SORT(A, n)

for i ← 1 **to** n

  do insert A[i] into list $B[\lfloor nA[i] \rfloor]$

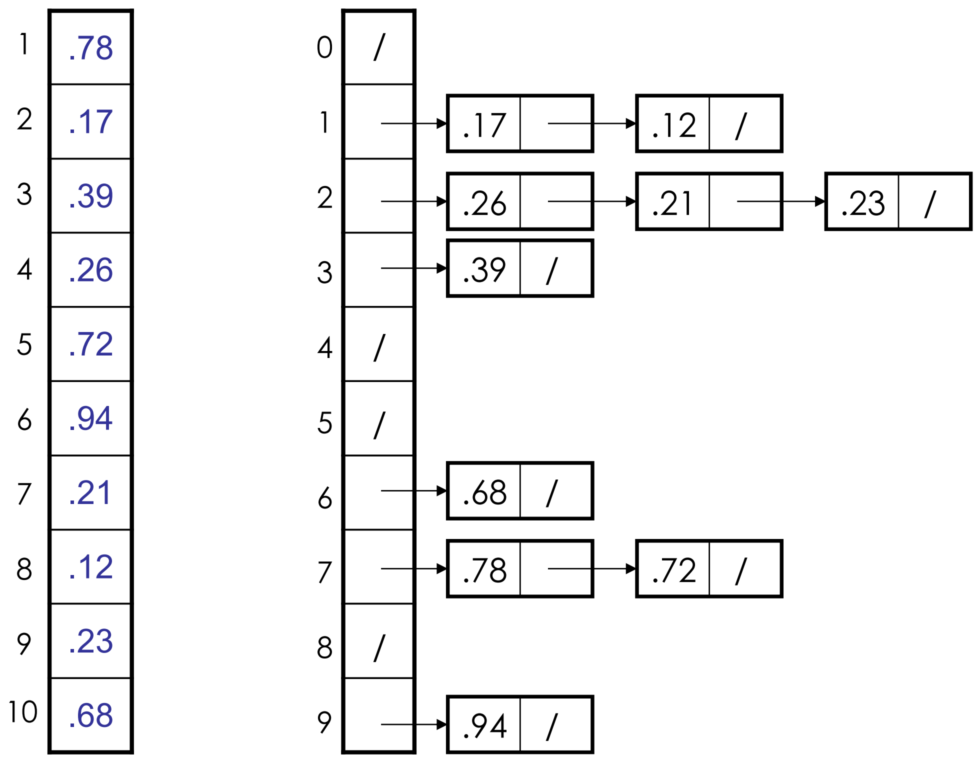
for i ← 0 **to** n - 1

  do sort list B[i] with insertion sort

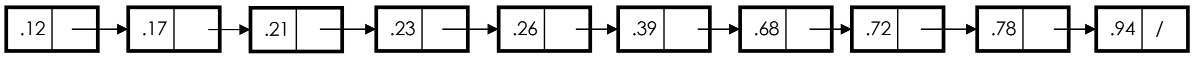
concatenate lists B[0], B[1], . . . , B[n -1] together in order

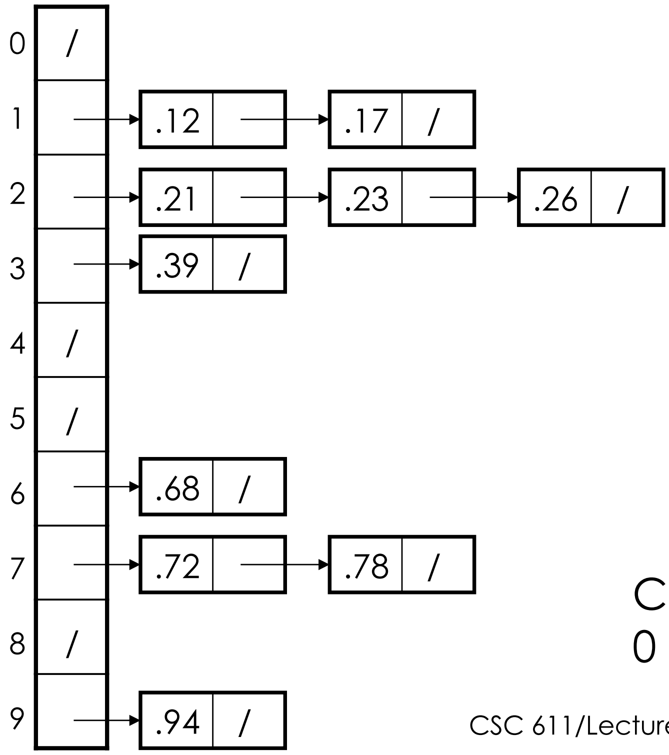
return the concatenated lists

Example - Bucket Sort



Example - Bucket Sort





Concatenate the lists from 0 to n – 1 together, in order

Correctness of Bucket Sort

- Consider two elements *A[i]*, *A[j]*
- Assume without loss of generality that *A[i]* ≤ *A[j]*
- Then $\lfloor nA[i] \rfloor \le \lfloor nA[j] \rfloor$
  - *A[i]* belongs to the same group as *A[j]* or to a group with a lower index than that of *A[j]*
- If *A[i]*, *A[j]* belong to the same bucket:
  - insertion sort puts them in the proper order
- If *A[i]*, *A[j]* are put in different buckets:
  - concatenation of the lists puts them in the proper order

Analysis of Bucket Sort

Alg.: BUCKET-SORT(A, n)

for i ← 1 **to** n

  do insert A[i] into list B[⌊nA[i]⌋] $\Big\}$ **O**(n)

for i ← 0 **to** n - 1

  do sort list B[i] with insertion sort $\Big\}$ Θ(n)

concatenate lists B[0], B[1], . . . , B[n -1] together in order $\Big\}$ **O**(n)

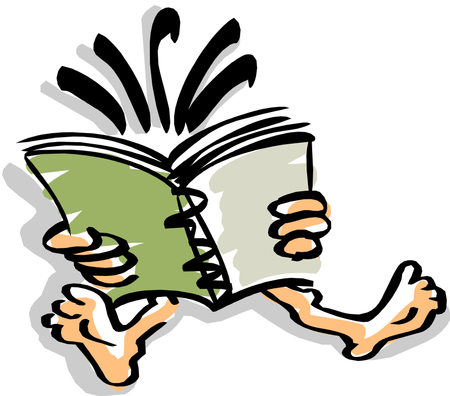
return the concatenated lists

Θ(n)

Conclusion

- Any comparison sort will take at least **nlgn** to sort an array of **n** numbers
- We can achieve a better running time for sorting if we can make certain assumptions on the input data:
  - **Counting sort:** each of the n input elements is an integer in the range **0** to **k**
  - **Radix sort:** the elements in the input are integers represented with **d** digits
  - **Bucket sort:** the numbers in the input are uniformly distributed over the interval [0, 1)

Readings



- Chapter 6, 7, 8