

# Unit 2: Sorting and Order Statistics

- **Course contents:**
  - Heapsort
  - Quicksort
  - Sorting in linear time
  - Order statistics
- **Readings:**
  - Chapters 6, 7, 8, 9

Comparison-based sorters				
Algorithm	Runtime			In-place?
	Best case	Average case	Worst case	
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
Quicksort	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$	Yes
Non-comparison-based sorters				
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	-	$O(n)$	-	No

# Types of Sorting Algorithms

---

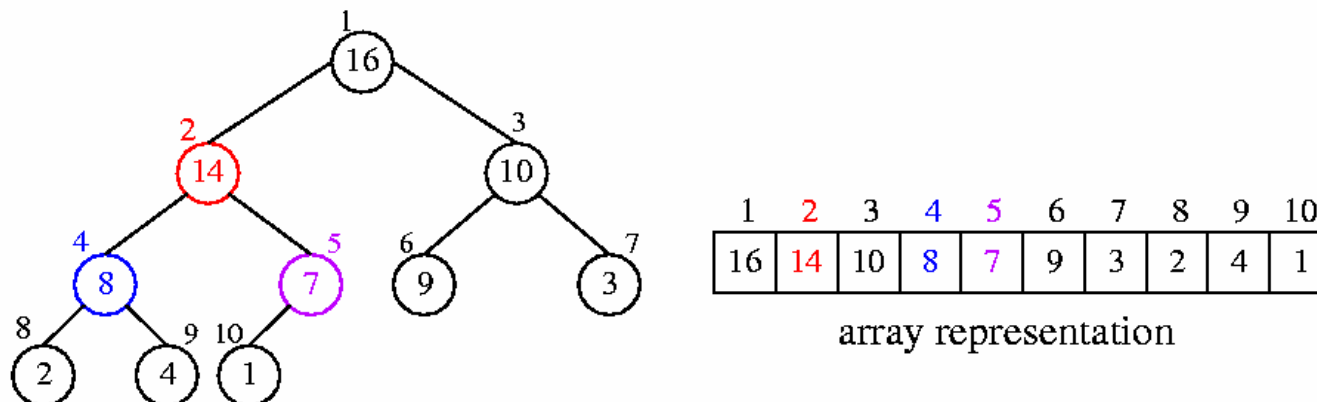
- A sorter is **in-place** if only a constant # of elements of the input are ever stored outside the array.
- A sorter is **comparison-based** if the only operation on keys is to **compare two keys**.
  - Insertion sort, merge sort, heapsort, quicksort
- The non-comparison-based sorters sort keys by looking at the values of **individual** elements.
  - **Counting sort**: Assumes keys are in  $[1..k]$  and uses array indexing to count the # of elements of each value.
  - **Radix sort**: Assumes each **integer** contains  $d$  digits, and each digit is in  $[1..k]$ .
  - **Bucket sort**: Sort data into buckets and then merge across buckets. Requires information for input **distribution**.

# Sorting Algorithm Comparisons

Comparison-based sorters				
Algorithm	Runtime			In-place?
	Best case	Average case	Worst case	
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
Quicksort	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$	Yes
Non-comparison-based sorters				
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	-	$O(n)$	-	No

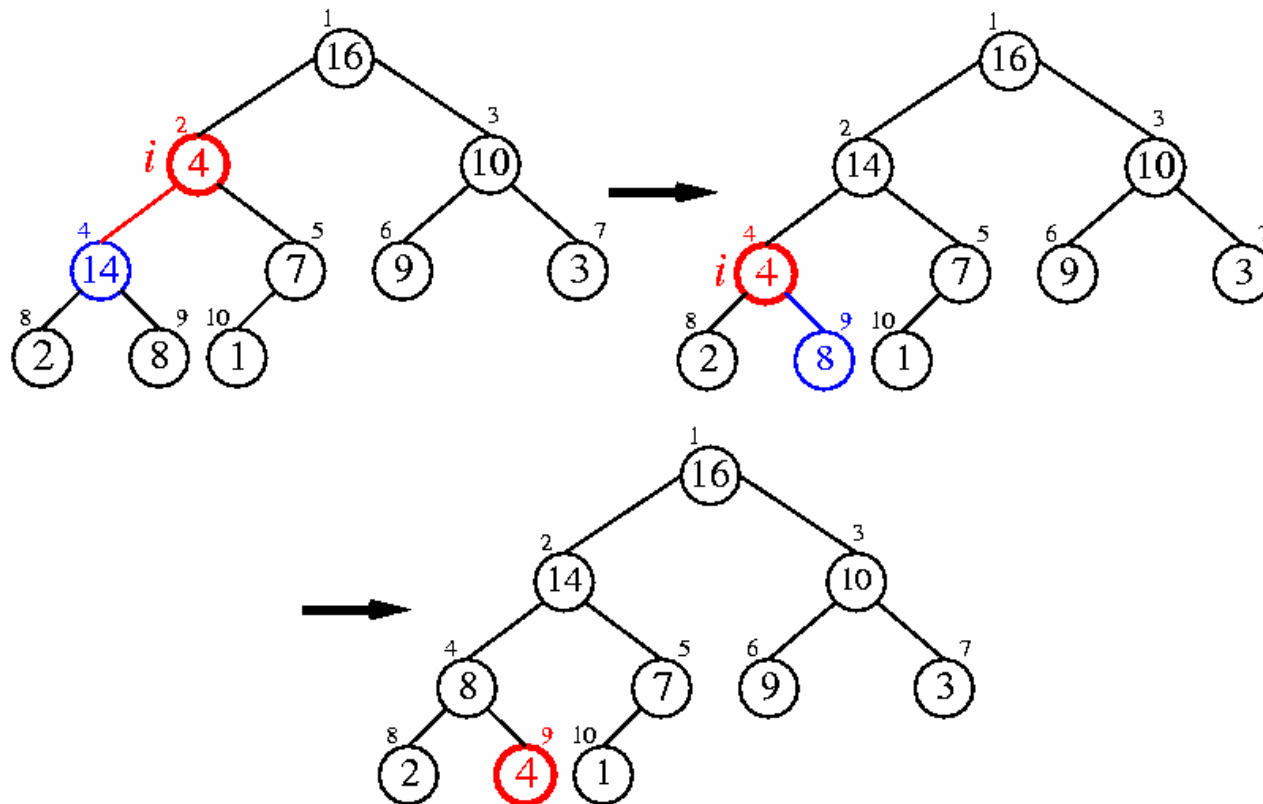
# Binary Heap

- Binary heap data structure: represented by an array  $A$ 
  - Complete binary tree, except that some rightmost leaves on the bottom level may be missing.
  - **Max-Heap property:** A node's key  $\geq$  its children's keys.
  - **Min-Heap property:** A node's key  $\leq$  its children's keys.
- Implementation
  - Root:  $A[1]$ .
  - For  $A[i]$ , LEFT child is  $A[2i]$ , RIGHT child is  $A[2i+1]$ , and PARENT is  $A[\lfloor i/2 \rfloor]$ .
  - $heap\text{-}size[A]$  (# of elements in the heap stored within  $A$ )  $\leq$   $length[A]$  (# of elements in  $A$ ).



# MAX-HEAPIFY: Maintaining the Heap Property

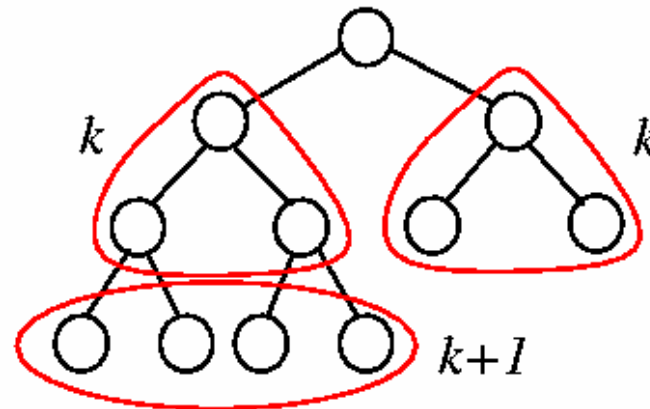
- Assume that **subtrees** indexed  $RIGHT(i)$  and  $LEFT(i)$  are **heaps**, but  $A[i]$  may be smaller than its children.
- $MAX\text{-HEAPIFY}(A, i)$  will “float down” the value at  $A[i]$  so that the subtree rooted at  $A[i]$  becomes a heap.



# MAX-HEAPIFY: Complexity

```
MAX-HEAPIFY(A, i)
1.  $l \leftarrow \text{LEFT}(i)$ 
2.  $r \leftarrow \text{RIGHT}(i)$ 
3. if  $l \leq \text{heap-size}[A]$  and  $A[l] > A[i]$ 
4.   then  $\text{largest} \leftarrow l$ 
5.   else  $\text{largest} \leftarrow i$ 
6. if  $r \leq \text{heap-size}[A]$  and  $A[r] > A[\text{largest}]$ 
7.   then  $\text{largest} \leftarrow r$ 
8. if  $\text{largest} \neq i$ 
9.   then exchange  $A[i] \leftrightarrow A[\text{largest}]$ 
10.      MAX-HEAPIFY(A,  $\text{largest}$ )
```

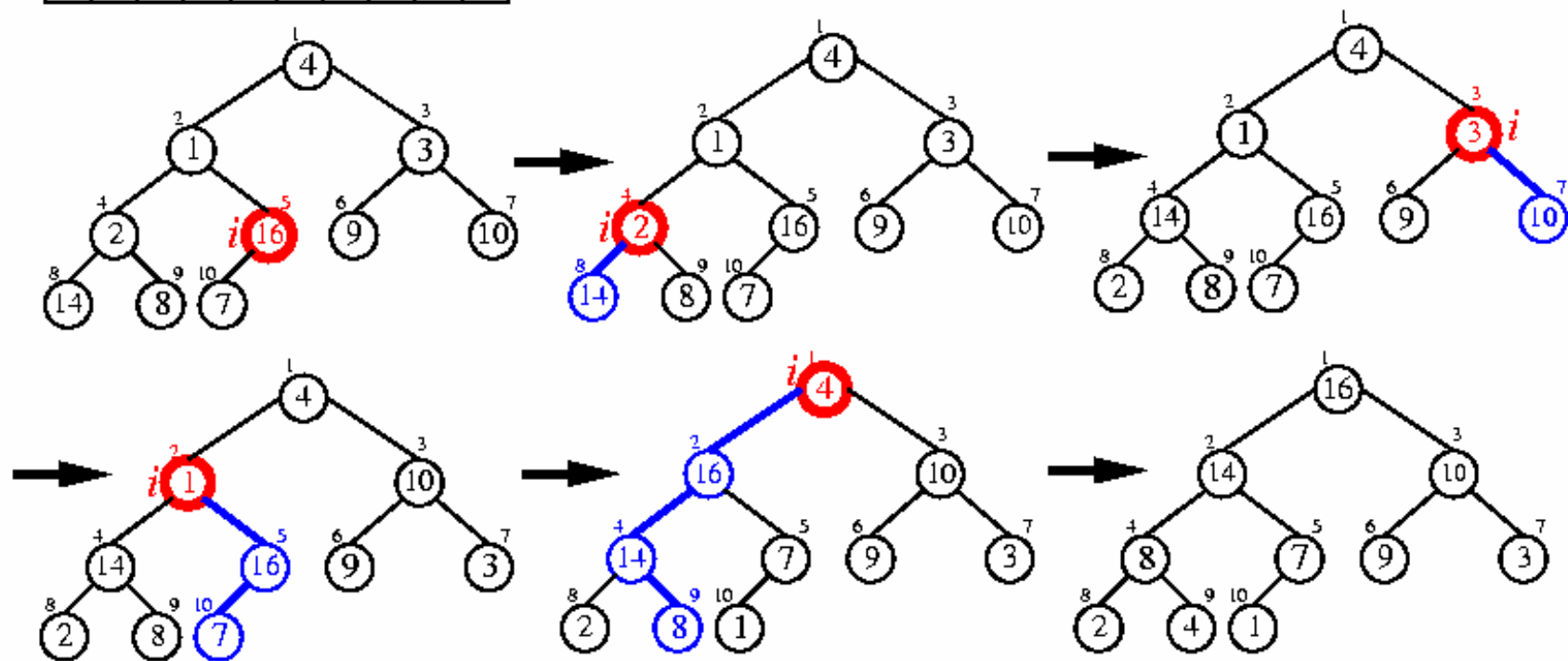
- Worst case: last row of binary tree is half empty  $\Rightarrow$  children's subtrees have size  $\leq 2n/3$ .
- Recurrence:  $T(n) \leq T(2n/3) + \theta(1) \Rightarrow T(n) = O(\lg n)$



# BUILD-MAX-HEAP: Building a Max-Heap

- Intuition: Use MAX-HEAPIFY in a bottom-up manner to convert  $A$  into a heap.
  - Leaves are already heaps, start at parents of leaves, and work upward till the root.

$A$  [ 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 ]



# BUILD-MAX-HEAP: Complexity

```
BUILD-MAX-HEAP(A)
1. heap-size[A] ← length[A]
2. for i ← ⌊length[A]/2⌋ downto 1
3.   do MAX-HEAPIFY(A,i)
```

- Naive analysis:  $O(n \lg n)$  time in total.
  - About  $n/2$  calls to HEAPIFY.
  - Each takes  $O(\lg n)$  time.
- Careful analysis:  $O(n)$  time in total.
  - Each MAX-HEAPIFY takes  $O(h)$  time ( $h$ : tree height).
  - At most  $\lceil n/2^{h+1} \rceil$  nodes of height  $h$  in an  $n$ -element array.
  - $T(n) = \sum_{h=0}^{\lfloor \lg n \rfloor} (\# \text{nodes in height } h) O(h) = \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) =$   
 $O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O(n)$
  - Note: (1) cf. **height & depth**, (2) Won't improve the overall complexity of the heap sort.

# Tree Height and Depth

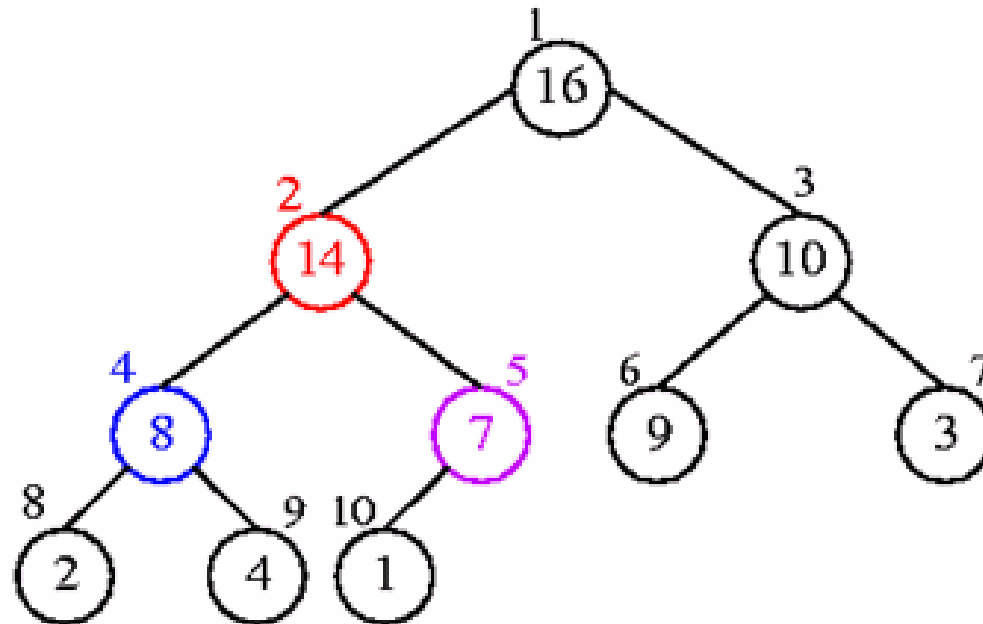
- **Height** of a node: # of edges on the longest simple downward path from the node to a leaf
- **Depth**: Length of the path from the root to a node

height = 3

height = 2

height = 1

height = 0



depth = 0

depth = 1

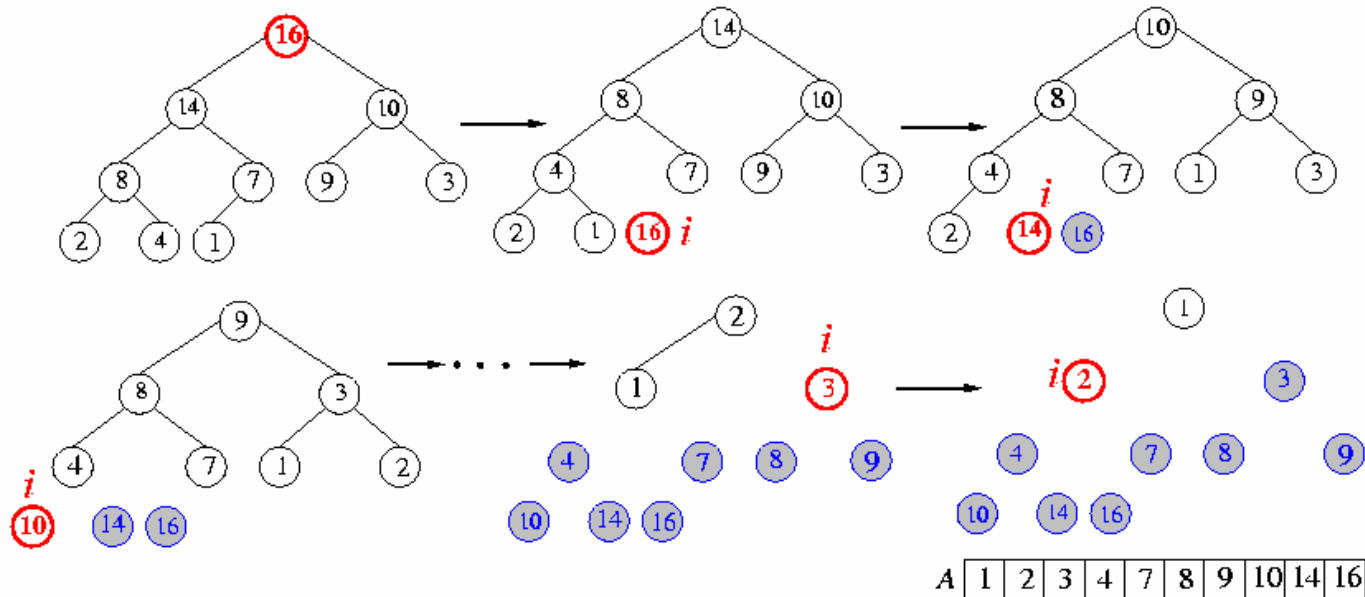
depth = 2

depth = 3

# HEAPSORT

```

HEAPSORT(A)
1. BUILD-MAX-HEAP(A)
2. for  $i \leftarrow \text{length}[A]$  downto 2
3.   do exchange  $A[1] \leftrightarrow A[i]$ 
4.    $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$ 
5.   MAX-HEAPIFY(A,1)
    
```



- Time complexity:  $O(n \lg n)$ .
- Space complexity:  $O(n)$  for array, **in-place. (Stable??)**

# Priority Queues

---

- A **priority queue** is a data structure on sets of keys; a **max-priority queue** supports the following operations:
  - INSERT( $S, x$ ): insert  $x$  into set  $S$ .
  - MAXIMUM( $S$ ): return the largest key in  $S$ .
  - EXTRACT-MAX( $S$ ): return and **remove** the largest key in  $S$ .
  - INCREASE-KEY( $S, x, k$ ): **increase** the value of element  $x$ 's key to the new value  $k$ .
- These operations can be easily supported using a heap.
  - INSERT: Insert the node at the end and fix heap in  $O(\lg n)$  time.
  - MAXIMUM: read the first element in  $O(1)$  time.
  - INCREASE-KEY: traverse a path from the target node toward the root to find a proper place for the new key in  $O(\lg n)$  time.
  - EXTRACT-MAX: delete the 1st element, replace it with the last, decrement the element counter, then heapify in  $O(\lg n)$  time.
- Compare with an array?

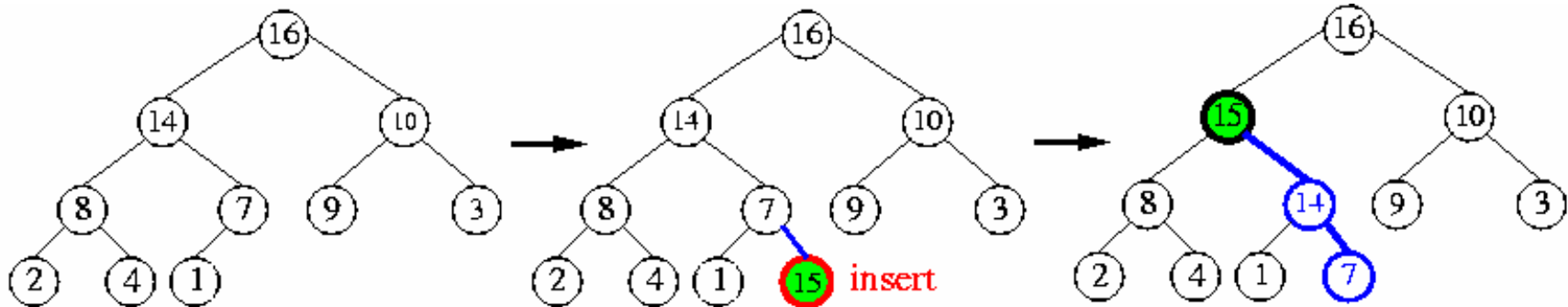
# Heap: EXTRACT-MAX and INSERT

HEAP-EXTRACT-MAX( $A$ )

1. **if**  $heap\text{-}size[A] < 1$
2.   **then error** "heap underflow"
3.  $max \leftarrow A[1]$
4.  $A[1] \leftarrow A[heap\text{-}size[A]]$
5.  $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
6. MAX-HEAPIFY( $A, 1$ )
7. **return**  $max$

MAX-HEAP-INSERT( $A, key$ )

1.  $heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
2.  $i \leftarrow heap\text{-}size[A]$
3. **while**  $i > 1$  and  $A[PARENT(i)] < key$
4.   **do**  $A[i] \leftarrow A[PARENT(i)]$
5.        $i \leftarrow PARENT(i)$
6.  $A[i] \leftarrow key$



# Quicksort

---

- A divide-and-conquer algorithm
  - **Divide:** Partition  $A[p..r]$  into  $A[p..q]$  and  $A[q+1..r]$ ; each key in  $A[p..q] \leq$  each key in  $A[q+1..r]$ .
  - **Conquer:** Recursively sort two subarrays.
  - **Combine:** Do nothing; quicksort is an in-place algorithm.

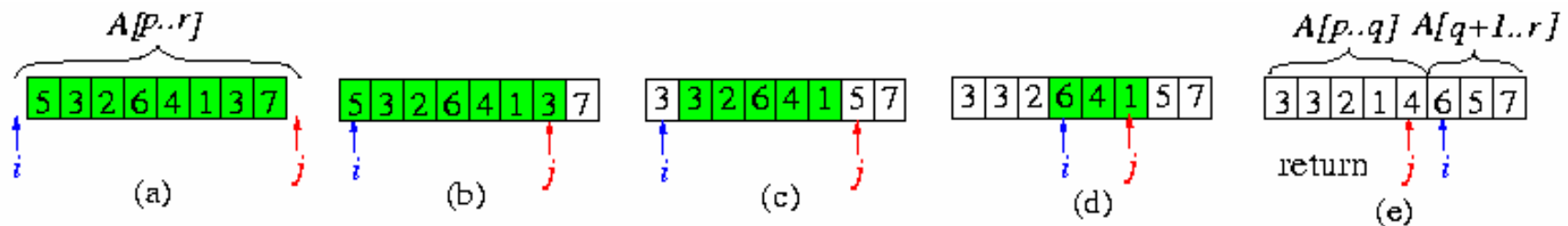
```
QUICKSORT(A, p, r)
/* Call QUICKSORT(A, 1, length[A]) to sort an entire array */
1. if  $p < r$  then
2.    $q \leftarrow$  PARTITION(A, p, r)
3.   QUICKSORT(A, p, q)
4.   QUICKSORT(A, q+1, r)
```

# Quicksort: Partition

```

PARTITION(A, p, r)
1.  $x \leftarrow A[p]$  /* break up A wrt x */
2.  $i \leftarrow p - 1$ 
3.  $j \leftarrow r + 1$ 
4. while TRUE do
5.   repeat  $j \leftarrow j - 1$ 
6.   until  $A[j] \leq x$ 
7.   repeat  $i \leftarrow i + 1$ 
8.   until  $A[i] \geq x$ 
9.   if  $i < j$ 
10.    then exchange  $A[i] \leftrightarrow A[j]$ 
11.    else return  $j$ 
    
```

- Partition A into two subarrays  $A[j] \leq x$  and  $A[i] \geq x$ .
- PARTITION runs in  $\theta(n)$  time, where  $n = r - p + 1$ .
- Ways to pick x: always pick  $A[p]$ , pick a key at random, pick the median of several keys, etc.



## Quicksort Runtime Analysis: Best Case

---

- A divide-and-conquer algorithm

$$T(n) = T(q - p + 1) + T(r - q) + \theta(n)$$

– Depends on the position of  $q$  in  $A[p..r]$ , but ???

- Best-, worst-, average-case analyses?
- **Best case:** Perfectly balanced splits---each partition gives an  $n/2 : n/2$  split.

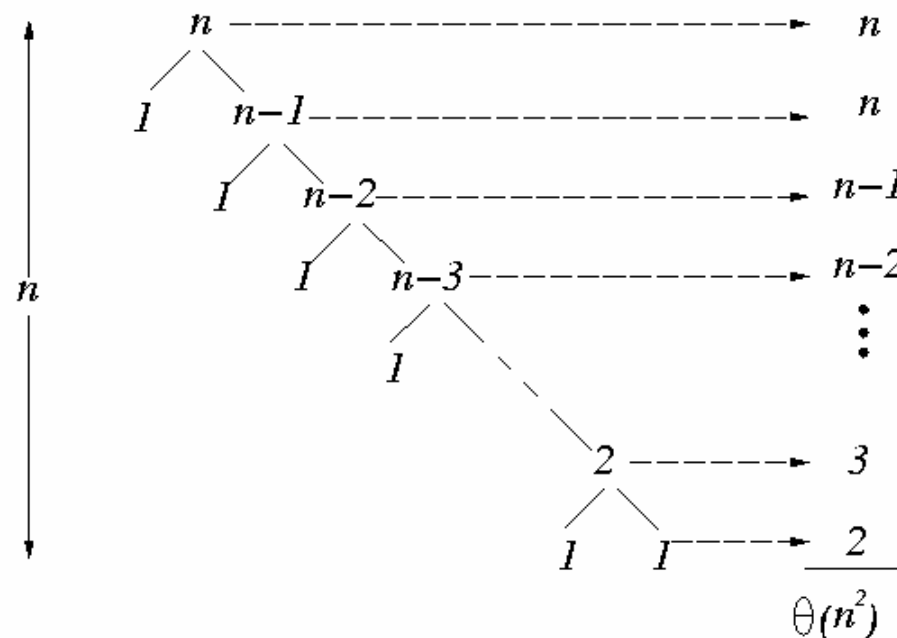
$$\begin{aligned} T(n) &= T(n/2) + T(n/2) + \theta(n) \\ &= 2T(n/2) + \theta(n) \end{aligned}$$

- Time complexity:  $\theta(n \lg n)$ 
  - Master method? Iteration? Substitution?

# Quicksort Runtime Analysis: Worst Case

- **Worst case:** Each partition gives a 1 :  $n - 1$  split.

$$\begin{aligned} T(n) &= T(1) + T(n-1) + \theta(n) \\ &= T(1) + (T(1) + T(n-2) + \theta(n-1)) + \theta(n) \\ &= \dots \\ &= nT(1) + \theta\left(\sum_{k=1}^{k=n} k\right) \\ &= \theta(n^2) \end{aligned}$$



## More on Worst-Case Analysis

---

- The **real** upperbound:

$$T(n) = \max_{1 \leq q \leq n-1} (T(q) + T(n-q) + \theta(n))$$

- Guess  $T(n) \leq cn^2$  and verify it inductively:

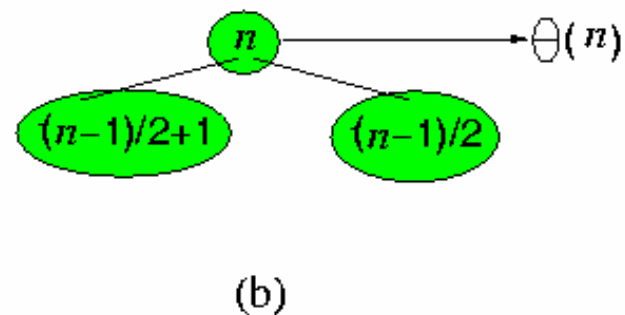
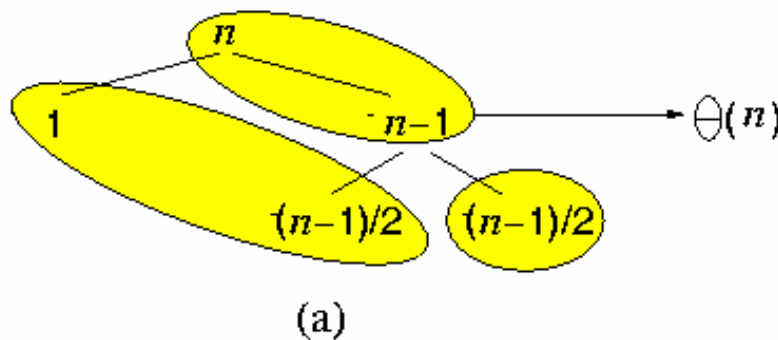
$$\begin{aligned} T(n) &\leq \max_{1 \leq q \leq n-1} (cq^2 + c(n-q)^2 + \theta(n)) \\ &= c \max_{1 \leq q \leq n-1} (q^2 + (n-q)^2) + \theta(n) \end{aligned}$$

- $q^2 + (n-q)^2$  is maximum at its endpoints:

$$\begin{aligned} T(n) &\leq c1^2 + c(n-1)^2 + \theta(n) \\ &= cn^2 - 2c(n-1) + \theta(n) \\ &\leq cn^2 \end{aligned}$$

# Quicksort: Average-Case Analysis

- **Intuition:** Some splits will be close to balanced and others imbalanced; good and bad splits will be **randomly** distributed in the recursion tree.
- **Observation:** Asymptotically bad run time occurs only when we have many bad splits **in a row**.
  - A bad split followed by a good split results in a good partitioning after one extra step!
  - Thus, we will still get  $O(n \lg n)$  run time.



# Randomized Quicksort

---

- How to modify quicksort to get good average-case behavior on **all** inputs?
- **Randomization!**
  - Randomly permute inputs, or
  - Choose the partitioning element  $x$  randomly at each iteration.

```
RANDOMIZED-PARTITION( $A, p, r$ )  
1.  $i \leftarrow \text{RANDOM}(p, r)$   
2. exchange  $A[p] \leftrightarrow A[i]$   
3. return PARTITION( $A, p, r$ )
```

```
RANDOMIZED-QUICKSORT( $A, p, r$ )  
1. if  $p < r$  then  
2.    $q \leftarrow \text{RANDOMIZED-PARTITION}(A, p, r)$   
3.   RANDOMIZED-QUICKSORT( $A, p, q$ )  
4.   RANDOMIZED-QUICKSORT( $A, q+1, r$ )
```

## Average-Case Recurrence

---

- Assume that all keys are distinct.
- Partition into lower side : upper side = 1 :  $n - 1$  with probability  $2/n$ ; others with probability  $1/n$ . **Why?**
- Partition at an index  $q$ :

$$\begin{aligned} T(n) &= \frac{1}{n} \left( 2(T(1) + T(n-1)) + \sum_{q=2}^{n-1} (T(q) + T(n-q)) \right) + \Theta(n) \\ &= \frac{1}{n} \left( T(1) + T(n-1) + \sum_{q=1}^{n-1} (T(q) + T(n-q)) \right) + \Theta(n) \\ &= \frac{1}{n} \sum_{q=1}^{n-1} (T(q) + T(n-q)) + \Theta(n) \quad / * why? * / \\ &= \frac{2}{n} \sum_{k=1}^{n-1} T(k) + \Theta(n) \quad / * why? * / \end{aligned}$$

## Average-Case Recurrence (cont'd)

- Guess  $T(n) \leq an \lg n + b$  and verify it inductively:

$$\begin{aligned} T(n) &= \frac{2}{n} \sum_{k=1}^{n-1} T(k) + \Theta(n) \\ &\leq \frac{2a}{n} \sum_{k=1}^{n-1} k \lg k + \frac{2b}{n}(n-1) + \Theta(n). \end{aligned}$$

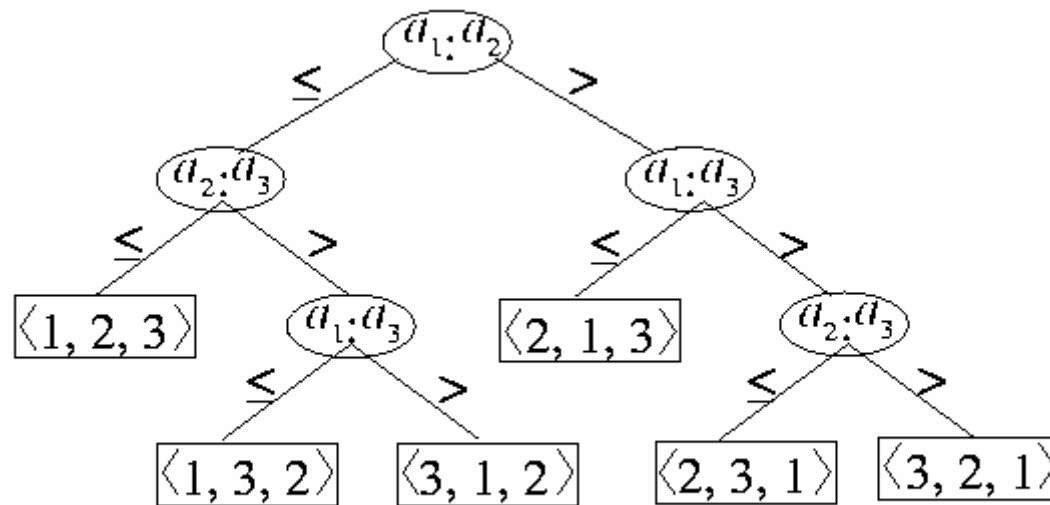
- Need to show that

$$\begin{aligned} \sum_{k=1}^{n-1} k \lg k &= \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k \\ &\leq (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k \quad /* why? */ \\ &\leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2. \end{aligned}$$

- Substituting  $\sum_{k=1}^{n-1} k \lg k$ , we have  $T(n) \leq an \lg n + b$ .
- Practically, quicksort is often 2-3 times faster than merge sort or heap sort.

## Decision-Tree Model for Comparison-Based Sorter

- Consider only the comparisons in the sorter.
- Correspond to each internal node in the tree to a comparison.
- Start at root and do the first comparison:  $\leq \Rightarrow$  go to the **left** branch;  $> \Rightarrow$  go to the **right** branch.
- Represent each leaf an ordering of the input ( $n!$  leaves!)



decision tree

## $\Omega(n \lg n)$ Lower Bound for Comparison-Based Sorters

---

- There must be  $n!$  leaves in the decision tree.
- Worst-case # of comparisons = #edges of the longest path in the tree (tree **height**).
- **Theorem:** Any decision tree that sorts  $n$  elements has height  $\Omega(n \lg n)$ .

Let  $h$  be the height of the tree  $T$ .

- $T$  has  $\geq n!$  leaves.
- $T$  is binary, so has  $\leq 2^h$  leaves.

$$2^h \geq n!$$

$$h \geq \lg n!$$

$$= \Omega(n \lg n) \quad /* \text{Stirling's approximation } n! > \left(\frac{n}{e}\right)^n */$$

- Thus, any comparison-based sorter takes  $\Omega(n \lg n)$  time in **the worst case**.
- Merge sort and heapsort are **asymptotically optimal comparison** sorts.

# Counting Sort: A Non-comparison-Based Sorter

- **Requirement:** Input integers are in known range  $[1..k]$ .
- **Idea:** For each  $x$ , find # of elements  $\leq x$  (say  $m$ , excluding  $x$ ) and put  $x$  in the  $(m+1)$ st slot.
- Runs in  $O(n+k)$  time, but needs extra  $O(n+k)$  space.
- Example: **A:** input; **B:** output; **C:** working array.

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

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

	1	2	3	4	5	6	7	8
B						4		

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

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

(a)

(b)

(c)

	1	2	3	4	5	6	7	8
B		1					4	

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

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

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

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

(d)

(e)

(f)

A: input    B: output    C: auxiliary array

# Counting Sort

---

```
COUNTING-SORT(A, B, k)
1. for  $i \leftarrow 1$  to  $k$  do
2.    $C[i] \leftarrow 0$ 
3. for  $j \leftarrow 1$  to  $\text{length}[A]$  do
4.    $C[A[j]] \leftarrow C[A[j]] + 1$ 
5. /*  $C[i]$  now contains the number of elements equal to  $i$ . */
6. for  $i \leftarrow 2$  to  $k$  do
7.    $C[i] \leftarrow C[i] + C[i-1]$ 
8. /*  $C[i]$  now contains the number of elements  $\leq i$ . */
9. for  $j \leftarrow \text{length}[A]$  downto 1 do
10.   $B[C[A[j]]] \leftarrow A[j]$ 
11.   $C[A[j]] \leftarrow C[A[j]] - 1$ 
```

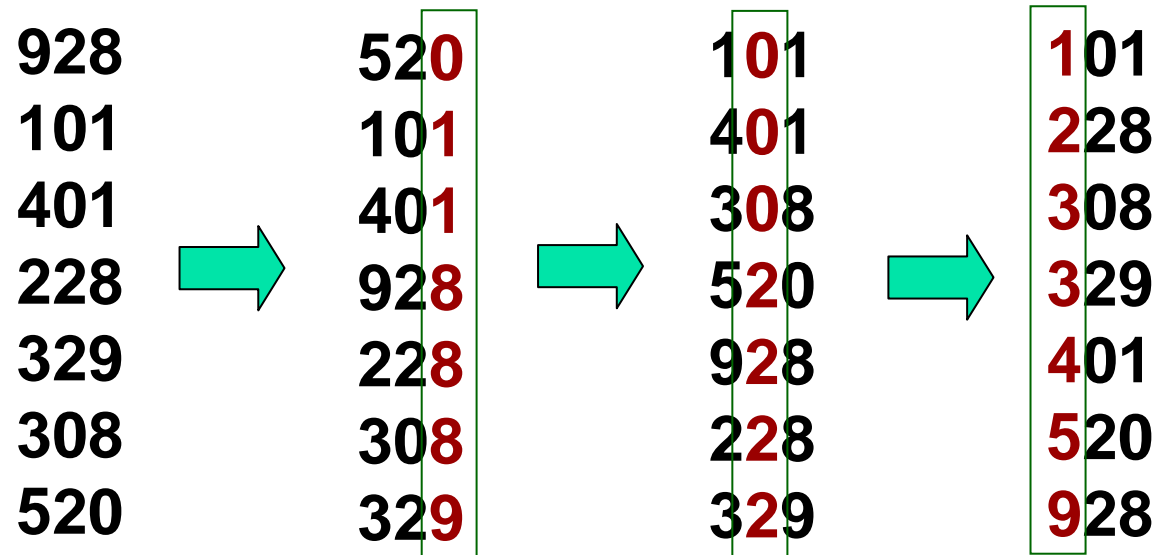
- Linear time if  $k = O(n)$ .
- **Stable** sorters: counting sort, insertion sort, merge sort.
- **Unstable** sorters: heap sort, quicksort.

# Radix Sort

RADIX-SORT( $A, d$ )

1. for  $i \leftarrow 1$  to  $d$  do

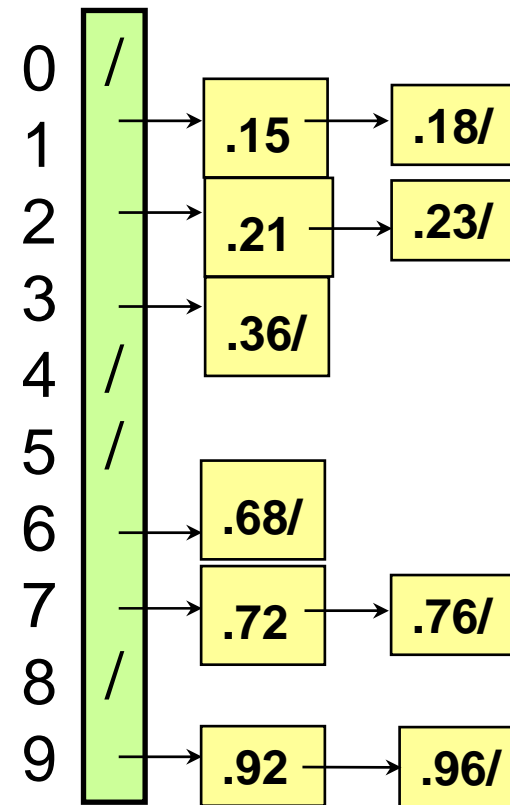
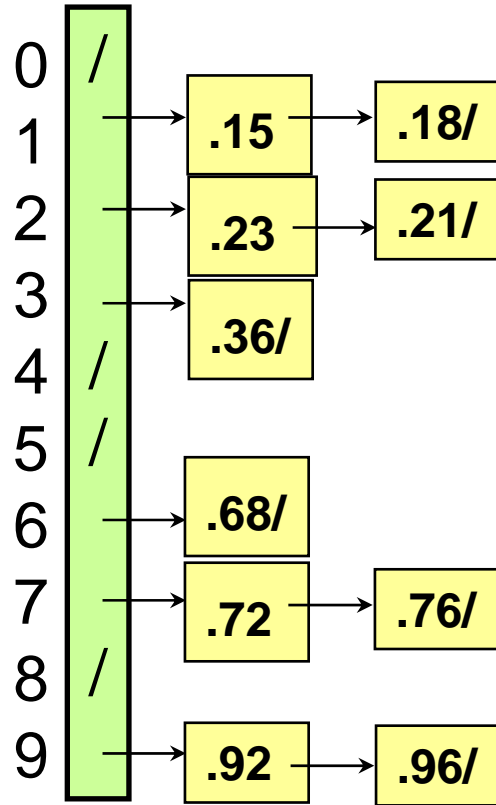
2. Use a **stable** sorter to sort array  $A$  on digit  $i$



- Time complexity:  $\Theta(d(n+k))$  for  $n$   $d$ -digit numbers in which each digit has  $k$  possible values.
  - Which sorter?

# Bucket Sort

1	.76
2	.18
3	.36
4	.96
5	.72
6	.92
7	.21
8	.15
9	.23
10	.68



**Step 1: distribute**

**Step 2: sort**

**Step 3: combine**



## Note on Sorting in Linear Time

---

Non-comparison-based sorters				
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	-	$O(n)$	-	No

- Counting sort: Linear time if  $k = O(n)$ ; pseudo-linear time, otherwise.
- Radix sort: Linear time if  $d$  is a constant and  $k' = O(n)$ ; pseudo-polynomial time, otherwise.
- Bucket sort: Expected linear time if the sum of the squares of the bucket sizes is linear in the # of elements (uniform distribution)

# Order Statistics

---

- **Def:** Let  $A$  be an ordered set containing  $n$  elements. The  **$i$ -th order statistic** is the  $i$ -th smallest element.
  - Minimum: 1st order statistic
  - Maximum:  $n$ -th order statistic
  - Median:  $\lfloor \frac{n+1}{2} \rfloor = \lceil \frac{n+1}{2} \rceil$ -th order statistic
- **The Selection Problem:** Find the  $i$ -th order statistic for a given  $i$ .
  - **Input:** A set  $A$  of  $n$  (distinct) numbers and a number  $i$ ,  $1 \leq i \leq n$ .
  - **Output:** The element  $x \in A$  that is larger than exactly  $(i-1)$  elements of  $A$ .
- Naive selection: sort  $A$  and return  $A[i]$ .
  - Time complexity:  $O(n \lg n)$ .
  - Can we do better??

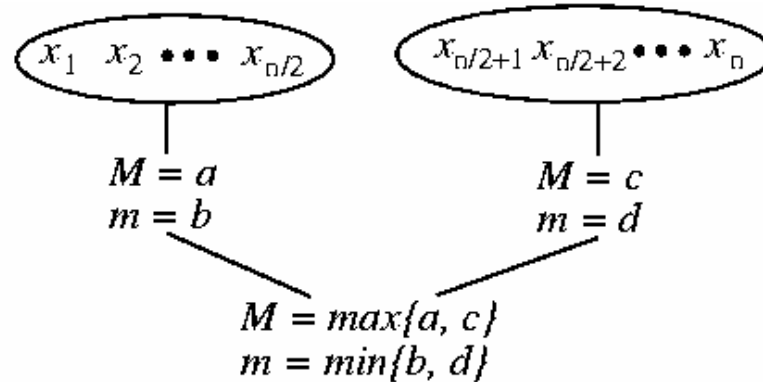
## Finding Minimum (Maximum)

---

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

- **Exactly**  $n-1$  comparisons.
  - Best possible?
  - Expected # of times executed for line 4:  $O(\lg n)$ .
- Naive simultaneous minimum and maximum:  $2n-3$  comparisons.
  - Best possible?

# Simultaneous Minimum and 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  $n = 2^k$ .

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

- This divide-and-conquer algorithm is optimal!

## Selection in Linear Expected Time

---

```
Randomized-Select( $A, p, r, i$ )
1. if  $p = r$ 
2.   then return  $A[p]$ ;
3.  $q \leftarrow$  Randomized-Partition( $A, p, r$ );
4.  $k \leftarrow q - p + 1$ ;
5. if  $i \leq k$ 
6.   then return Randomized-Select( $A, p, q, i$ );
7.   else return Randomized-Select( $A, q+1, r, i-k$ ).
```

- Randomized-Partition first swaps  $A[p]$  with a random element of  $A$  and then proceeds as in regular PARTITION.
- Randomized-Select is like Randomized-Quicksort, except that we only need to make one recursive call.
- Time complexity
  - Worst case:  $1:n-1$  partitions.
  - $T(n) = T(n-1) + \theta(n) = \theta(n^2)$
  - Best case:  $T(n) = \theta(n)$
  - Average case? Like quicksort, asymptotically close to best case.

## Selection in Linear Expected Time: Average Case

---

$$\begin{aligned} T(n) &\leq \frac{1}{n} \left( T(\max(1, n-1)) + \sum_{k=1}^{n-1} T(\max(k, n-k)) \right) + O(n) \\ &\leq \frac{1}{n} \left( T(n-1) + 2 \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) \right) + O(n) \\ &= \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + O(n) \end{aligned}$$

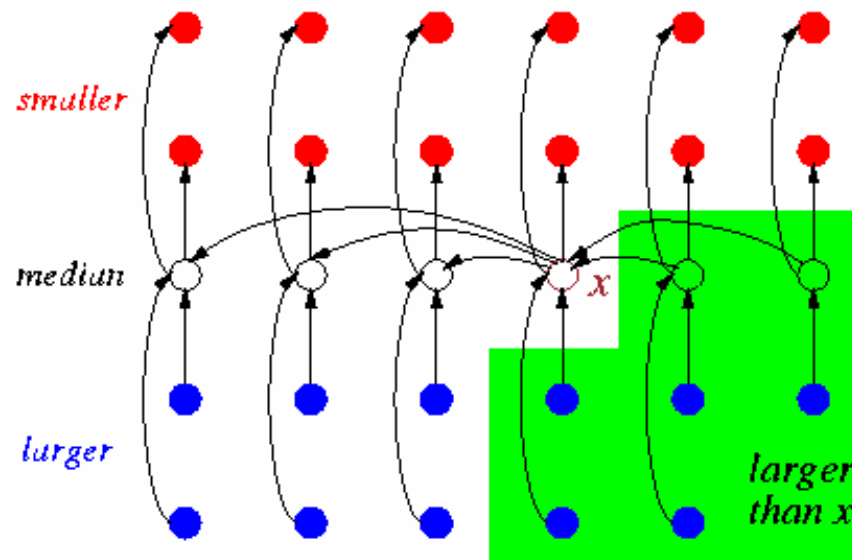
- Assume  $T(n) \leq cn$ .

$$\begin{aligned} T(n) &\leq \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} ck + O(n) \\ &\leq \frac{2c}{n} \left( \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \right) + O(n) \\ &\leq cn - c \left( \frac{n}{4} + \frac{1}{2} \right) + O(n) \\ &\leq cn. \end{aligned}$$

- Thus, on average, Randomized-Select runs in linear time.

# Selection in Worst-Case Linear Time

- **Key:** Guarantee a good split when array is partitioned.
- $\text{Select}(A, p, r, i)$ 
  1. Divide input array  $A$  into  $\lfloor n/5 \rfloor$  groups of size 5 (possibly with a leftover group of size  $< 5$ ).
  2. Find the median of each of the  $\lceil n/5 \rceil$  groups.
  3. Call  $\text{Select}$  recursively to find the median  $x$  of the  $\lceil n/5 \rceil$  medians.
  4. Partition array around  $x$ , splitting it into two arrays of  $A[p, q]$  (with  $k$  elements) and  $A[q+1, r]$  (with  $n-k$  elements).
  5. **if**  $(i \leq k)$  **then**  $\text{Select}(A, p, q, i)$  **else**  $\text{Select}(A, q + 1, r, i - k)$ .



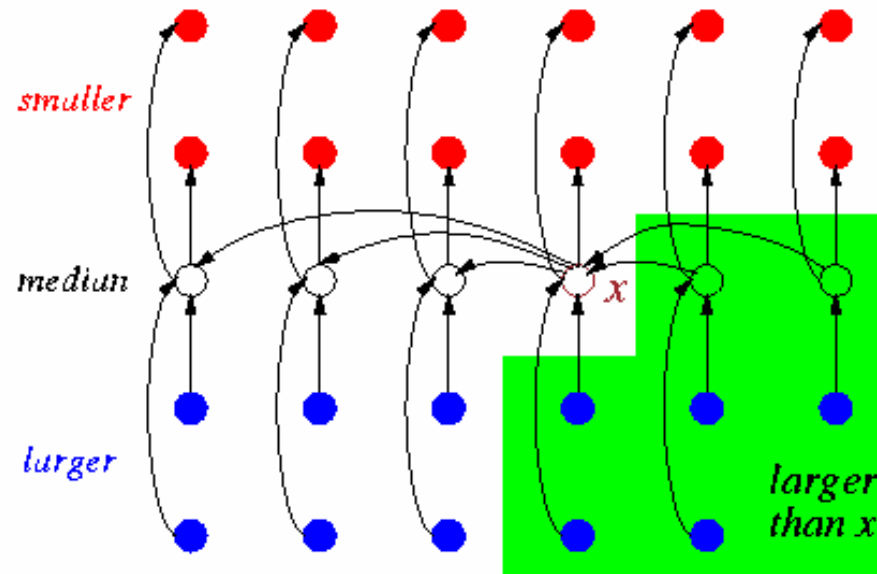
# Runtime Analysis

- Main idea: Select guarantees that  $x$  causes a good partition; 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  $> x$  (or  $< x$ )

- Worst-case split has  $7n/10 + 6$  elements in the bigger subproblem.



# Runtime Analysis

---

- 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  $> x$  (or  $< x$ )

- Worst-case split has  $7n/10 + 6$  elements in the bigger subproblem.

- Run time:  $T(n) = T(\lceil n/5 \rceil) + T(7n/10+6) + O(n)$ .
  1.  $O(n)$ : break into groups.
  2.  $O(n)$ : finding medians (constant time for 5 elements).
  3.  $T(\lceil n/5 \rceil)$ : recursive call to find median of the medians.
  4.  $O(n)$ : partition.
  5.  $T(7n/10+6)$ : searching in the bigger partition.
- Apply the substitution method to prove that  $T(n)=O(n)$ .