UNIT 2
SORTING AND ORDER STATISTICS

Iris Hui-Ru Jiang
Spring 2024

Department of Electrical Engineering
National Taiwan University

Outline

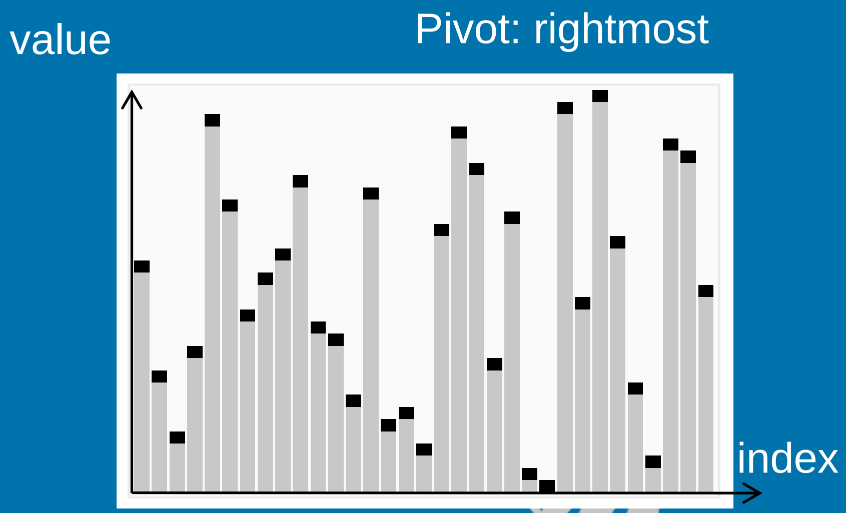
- Content:
  - Heapsort
  - Quicksort
  - Sorting in linear time
  - Order statistics
- Reading:
  - Chapters 6, 7, 8, 9

Algorithm	**Runtime**			**Properties**	
	Best case	Average case	Worst case	Stable?	In-place?
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Merge	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	Yes	No
Heap	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	No	Yes
Quicksort	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$	No	Yes
Counting	$O(n+k)$	$O(n+k)$	$O(n+k)$	Yes	No
Radix	$O(d(n+k'))$	$O(d(n+k'))$	$O(d(n+k'))$	Yes	No
Bucket	–	$O(n)$	–	Yes	No

Quicksort

C.A.R. Hoare, 1962

Top 10 algorithms in 20th century



https://en.wikipedia.org/wiki/Quicksort#/media/File:Sorting_quicksort_anim.gif

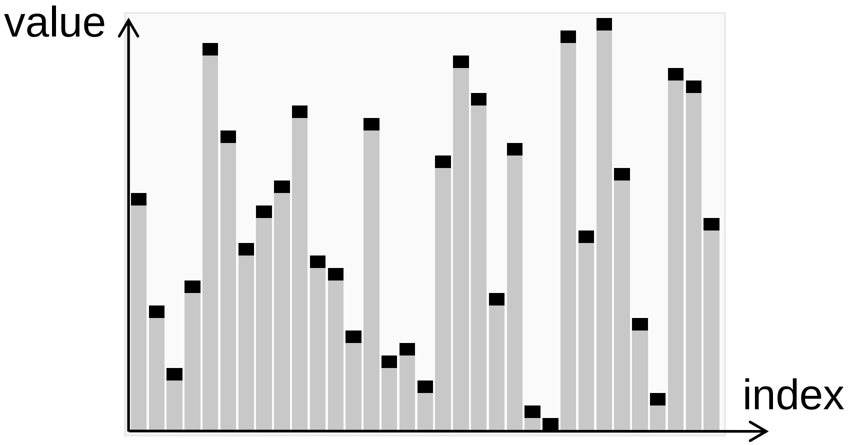
C. A. R. Hoare. Quicksort. *The Computer Journal*, Volume 5, Issue 1, 1962, Pages 10–16.

Quicksort

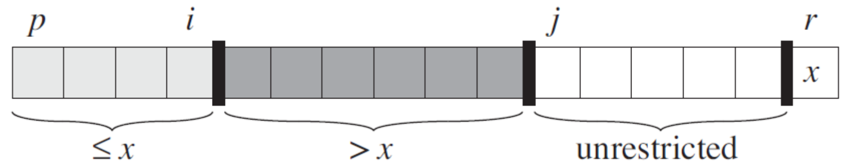
- A divide-and-conquer algorithm
  - **Divide:** Partition (rearrange) $A[p..r]$ into $A[p..q\text{-}1]$ and $A[q+1..r]$; each key in $\boldsymbol{A[p..q\text{-}1]} \leq \boldsymbol{A[q]}$ and $\boldsymbol{A[q]} <$ each key in $\boldsymbol{A[q+1..r]}$
    - Select a pivot and put it on the correct position $A[q]$
  - **Conquer:** Recursively sort two subarrays
  - **Combine:** Do nothing; each element has been at the right position

```
QUICKSORT(A, p, r)
// Call QUICKSORT(A, 1, A.length)
to sort an entire array
1. if  p < r
2.     q = PARTITION(A, p, r)
3.     QUICKSORT(A, p, q-1)
4.     QUICKSORT(A, q+1, r)
```

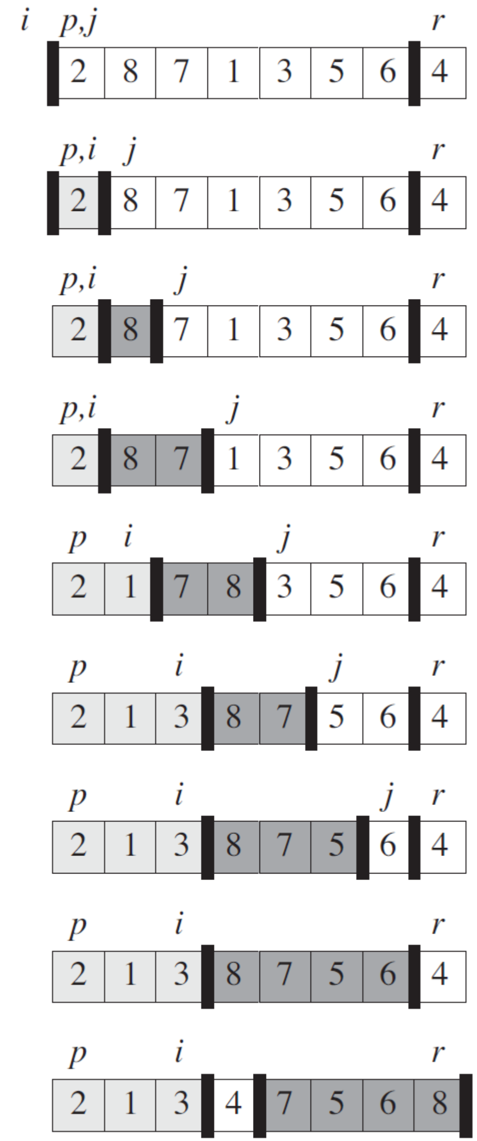
Pivot: rightmost



Quicksort: Partition

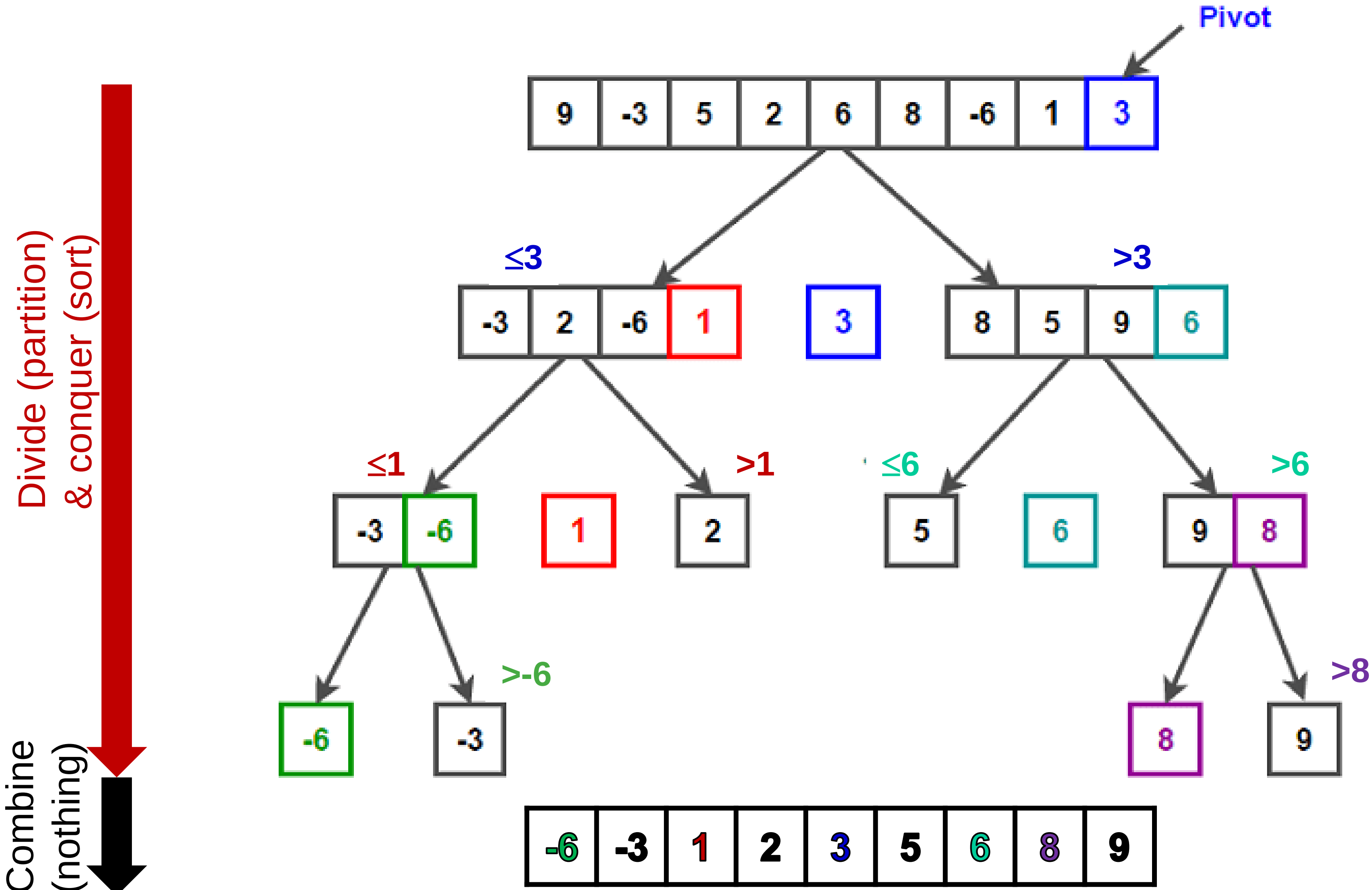


PARTITION(*A*, *p*, *r*)
1. *x* = *A*[*r*]   // break up *A* wrt pivot
2. *i* = *p* - 1
3. **for** *j* = *p* **to** *r* - 1
4.   **if** *A*[*j*] ≤ *x*
5.   *i* = *i* + 1
6.   exchange *A*[*i*] with *A*[*j*]
7. exchange *A*[*i*+1] with *A*[*r*]
9. **return** *i* + 1



- Select a pivot
- Partition *A* into subarrays *A*[..*i*] ≤ *x* and *A*[*i*+1..] > *x*
- PARTITION runs in Θ(*n*) time, where $n = r - p + 1$
- Ways to pick *x*: rightmost, random, median of 3 keys (first, last, middle)

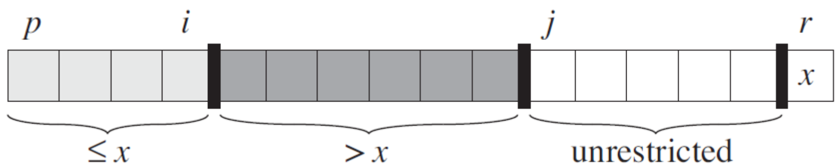
Quicksort Example



Loop Invariant of Partition

PARTITION(A, p, r)
1. $x = A[r]$ // break up A wrt pivot
2. $i = p - 1$
3. **for** $j = p$ **to** $r - 1$
4.   **if** $A[j] \leq x$
5. $i = i + 1$
6.   exchange $A[i]$ with $A[j]$
7. exchange $A[i+1]$ with $A[r]$
9. **return** $i + 1$

Four (possibly empty) regions



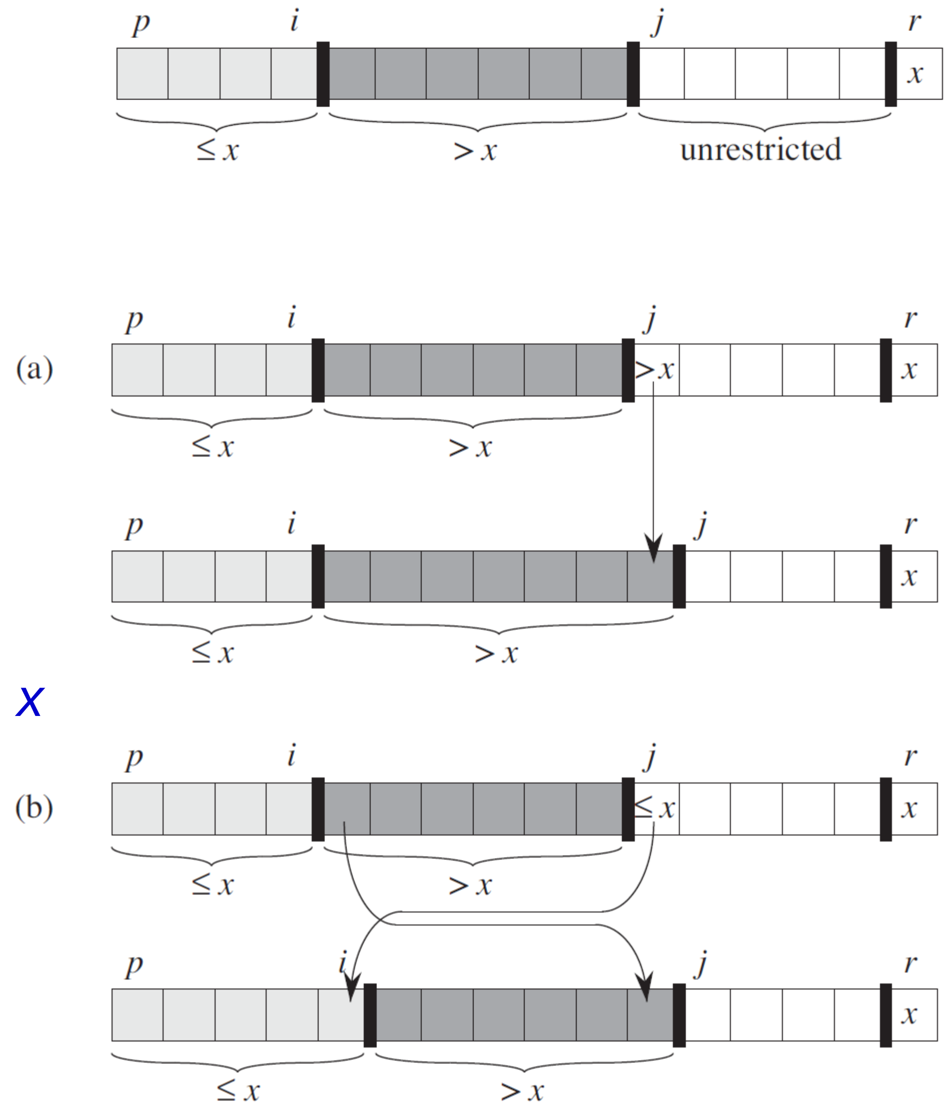
- At the beginning of each iteration of the loop of lines 3--6, for any array index k,
  - 1. if $p \leq k \leq i$, then $A[k] \leq x$
  - 2. if $i + 1 \leq k \leq j - 1$, then $A[k] > x$
  - 3. if $k = r$, then $A[k] = x$

Loop Invariant

```
PARTITION(A, p, r)
x = A[r]   // break up A wrt pivot
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i+1] with A[r]
return i + 1
```

- 1. if $p \le k \le i$, then $A[k] \le x$
- 2. if $i + 1 \le k \le j - 1$, then $A[k] > x$
- 3. if $k = r$, then $A[k] = x$

- Initialization
- Maintenance
- Termination



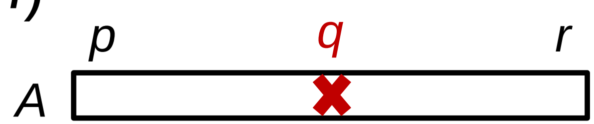
Performance of Quicksort: Best Case

Informal investigation

- The running time of quicksort depends on whether the partitioning is balanced or unbalanced
- A divide-and-conquer algorithm

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

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



- Best case: Perfectly balanced splits: each partition gives a $\lfloor n/2 \rfloor : \lceil n/2 \rceil - 1$ split

$$T(n) = 2T(n/2) + \Theta(n)$$

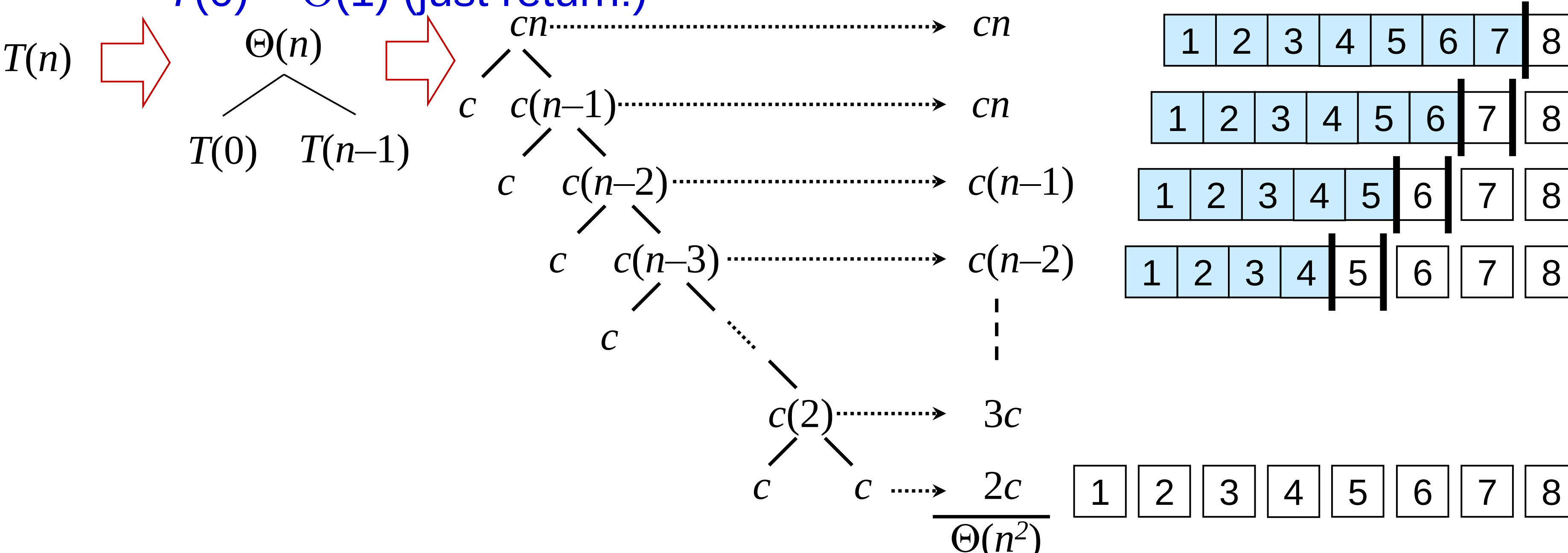
- Time complexity: $\Theta(n \lg n)$
  – Master method? Iteration? Substitution?
  – Asymptotically as fast as merge sort

Performance of Quicksort: Worst Case

- Worst case: Each partition gives a $n - 1 : 0$ split

$$T(n) = T(n\text{-}1) + T(0) + \Theta(n)$$
$$= T(n\text{-}1) + \Theta(n) = \Theta(n^2)$$

  - $T(0) = \Theta(1)$ (just return!)



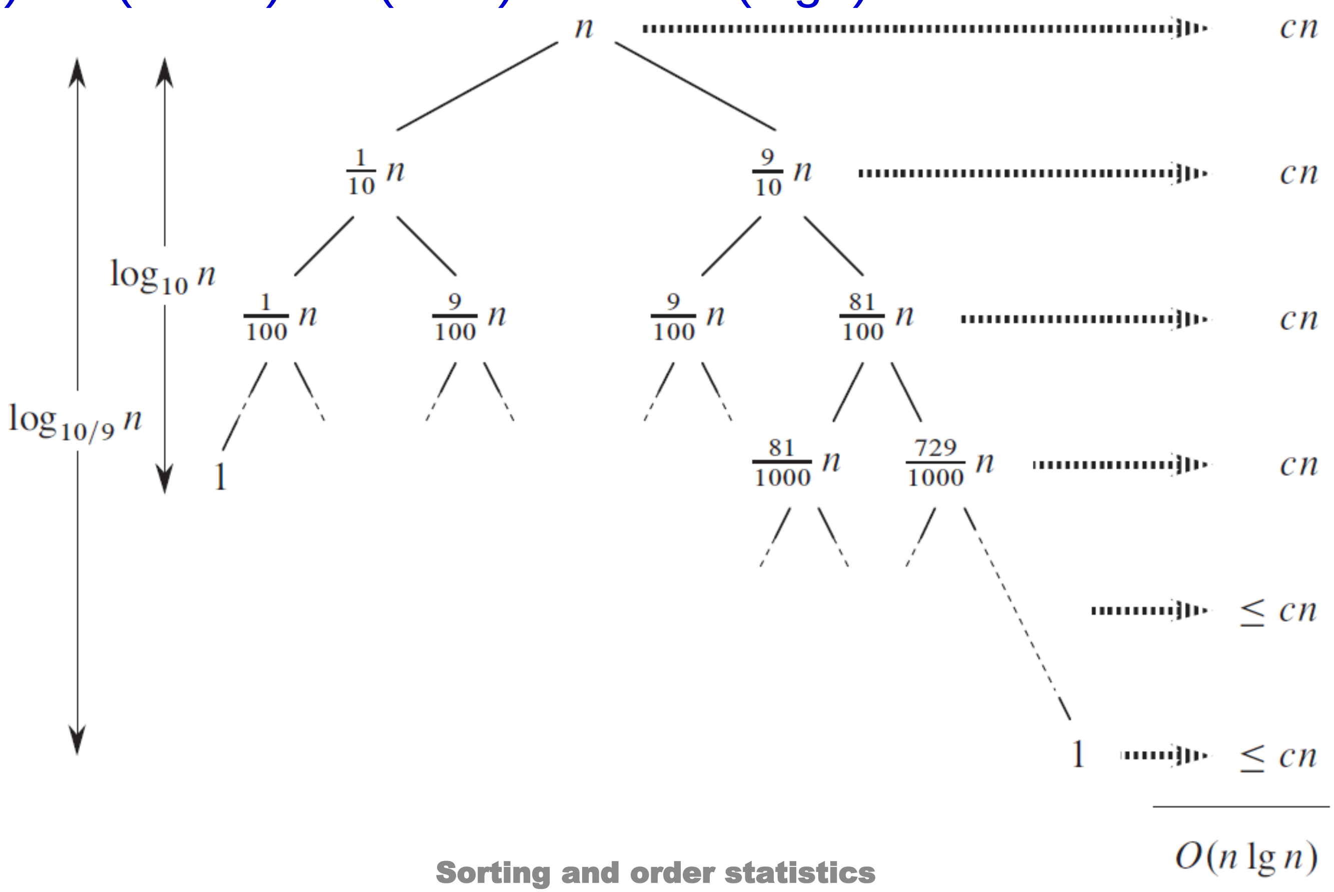
  - Asymptotically as slow as insertion sort

The worst case occurs when the array is already sorted!

(or reversely sorted)

Balanced Partitioning

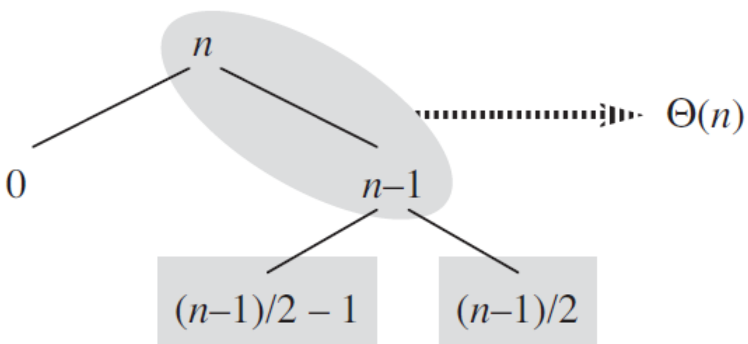
- Suppose the partitioning algorithm always produces a 9-to-1 proportional split
  - $T(n) = T(9n/10) + T(n/10) + cn = O(n\lg n)$

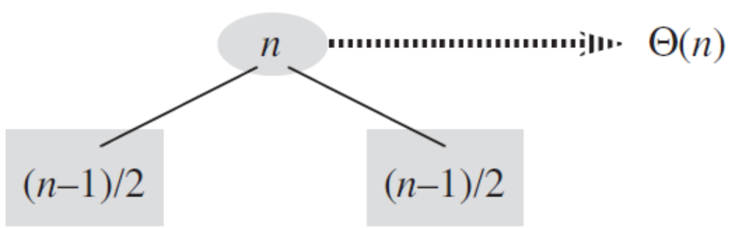


Quicksort: Average-Case Analysis

Intuition

- 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 runtime 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!
    - $\Theta(n\text{-}1)$ of the bad split can be absorbed into $\Theta(n)$ of the good split
  - Thus, we will still get $O(n \lg n)$ run time





Randomized Quicksort

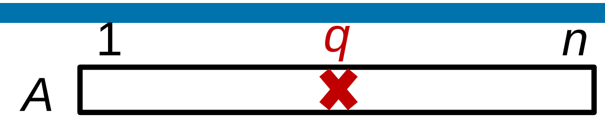
- How to modify quicksort to achieve good average-case behavior on **all** inputs?
  - Best choice: median!
- **Randomization!** Choose the pivot *x* randomly at each iteration

```
RANDOMIZED-PARTITION(A, p, r)
1. i = RANDOM(p, r)
2. exchange A[r] with A[i]
3. return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
1. if p < r
2.     q = RANDOMIZED-PARTITION(A, p, r)
3.     RANDOMIZED-QUICKSORT(A, p, q-1)
4.     RANDOMIZED-QUICKSORT(A, q+1, r)
```

Proof on Worst-Case Analysis

QUICKSORT and RANDOMIZED-QUICKSORT



- The real upperbound:

$$T(n) = \max_{1 \le q \le n} \left(T(q-1) + T(n-q) + \Theta(n) \right)$$

$$= \max_{0 \le q \le n-1} \left(T(q) + T(n-q-1) \right) + \Theta(n)$$

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

$$T(n) = \max_{0 \le q \le n-1} \left(cq^2 + c(n-q-1)^2 \right) + \Theta(n)$$

$$= c \cdot \max_{0 \le q \le n-1} \left(q^2 + (n-q-1)^2 \right) + \Theta(n)$$

- $q^2 + (n-q-1)^2$ achieves maximum at its endpoints:

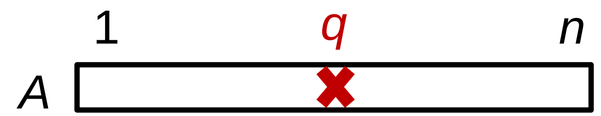
$$T(n) \le c(n-1)^2 + \Theta(n)$$

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

$$= cn^2 - c(2n-1) + an$$

$$\le cn^2$$

Expected Running Time (1/2)



Method 1

- Assume that all keys in a given array of size *n* are distinct
- Partition into lower side : upper side = *q* - 1 : *n* - *q*
- Pick any particular element as the pivot with probability 1/*n*
  - $X_i = \mathrm{I}\{i\text{th smallest element is chosen as the pivot}\}$
  - $\mathrm{E}[X_i] = {}^{1}\!/_{n}$
- Partition at an index *q*

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{q=1}^{n} X_q\big(T(q-1) + T(n-q) + \Theta(n)\big)\right]$$

$$= \sum_{q=1}^{n} \mathrm{E}[X_q\big(T(q-1) + T(n-q) + \Theta(n)\big)]$$

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

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

$$= \frac{2}{n}\sum_{q=0}^{n-1} \mathrm{E}[T(q)] + \Theta(n) = \frac{2}{n}\sum_{q=2}^{n-1} \mathrm{E}[T(q)] + \Theta(n)$$

Expected Running Time (2/2)

Method 1

- Substitution: Guess $\mathrm{E}[T(q)] \le cq \lg q$

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

$$\le \frac{2}{n}\sum_{q=2}^{n-1} cq \lg q + an \le \frac{2c}{n}\sum_{q=2}^{n-1} q \lg q + an \le \frac{2c}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + an$$

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

- Need to show

$$\sum_{q=2}^{n-1} q \lg q = \sum_{q=2}^{\lceil n/2\rceil - 1} q \lg q + \sum_{q=\lceil n/2\rceil}^{n-1} q \lg q \le (\lg n - 1)\sum_{q=2}^{\lceil n/2\rceil - 1} q + \lg n \sum_{q=\lceil n/2\rceil}^{n-1} q$$

$$= \lg n \sum_{q=2}^{n-1} q - \sum_{q=2}^{\lceil n/2\rceil - 1} q \le \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2$$

Expected Running Time (1/3)

Method 2

- Idea: How many comparisons are performed?
- Lemma: The running time of QUICKSORT is $O(n+X)$
  - n elements, X comparisons
- Pf:
  - At most n calls to PARTITION
  - Each call executes `for` loop some # of times
  - Each iteration of `for` executes line 4 (comparison)

```
PARTITION(A, p, r)
1. x = A[r]   // break up A wrt pivot
2. i = p - 1
3. for j = p to r - 1
4.     if A[j] ≤ x
5.         i = i + 1
6.         exchange A[i] with A[j]
7. exchange A[i+1] with A[r]
9. return i + 1
```

Expected Running Time (2/3)

Method 2

- How to compute X?
- Rename A as $z_1, z_2, \ldots, z_n$ in ascending order
- Define $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$
- Define $X_{ij} = \mathrm{I}\{z_i$ is compared to $z_j\}$

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

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

- Observations:
  - Only pivot in some call is compared to other elements
  - Once two elements are compared, they will not be compared again

Expected Running Time (3/3)

Method 2

$$\mathrm{E}\left[X_{ij}\right] = \Pr\left\{z_i \text{ is compared with } z_j\right\}$$
$$= \Pr\left\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\right\}$$
$$= \Pr\left\{z_i \text{ is first pivot chosen from } Z_{ij}\right\} + \Pr\left\{z_j \text{ is first pivot chosen from } Z_{ij}\right\}$$
$$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

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

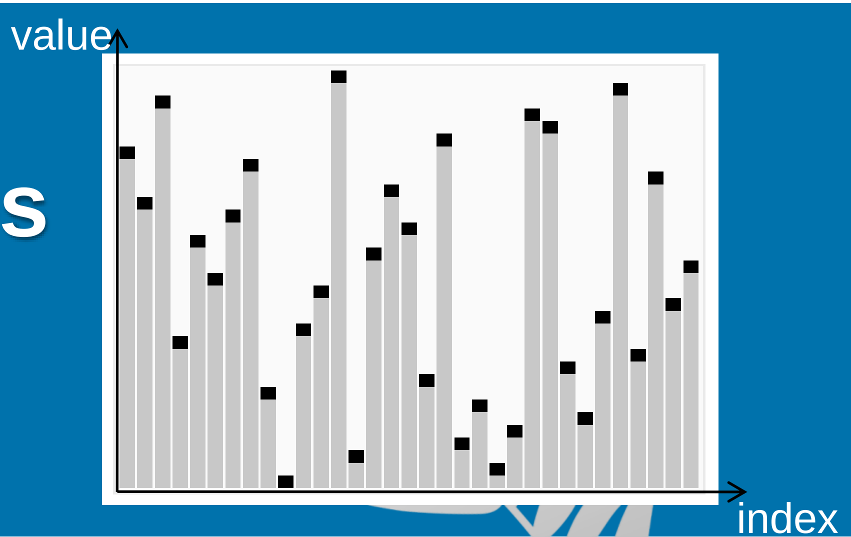
Harmonic series:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}$$
$$= \ln n + O(1)$$

Heapsort
Heaps: Priority Queues

J.W.J Williams, 1961

Binary Tree Application



https://en.wikipedia.org/wiki/Heapsort#/media/File:Sorting_heapsort_anim.gif

J. W. J. Williams, (1964), "Algorithm 232 - Heapsort", Communications of the ACM, 7 (6): 347–348

Priority Queue

- In a priority queue (PQ)
  - Each element has a priority (key)
  - Only the element with highest (or lowest) priority can be deleted
    - Max priority queue, or min priority queue
  - An element with arbitrary priority can be inserted into the queue at any time

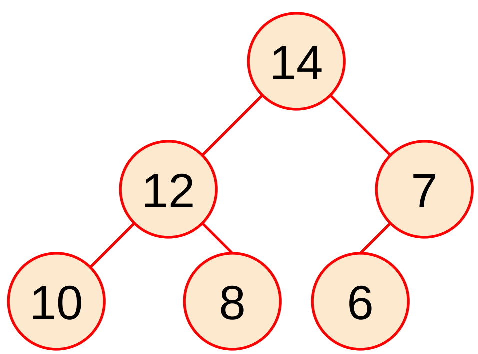
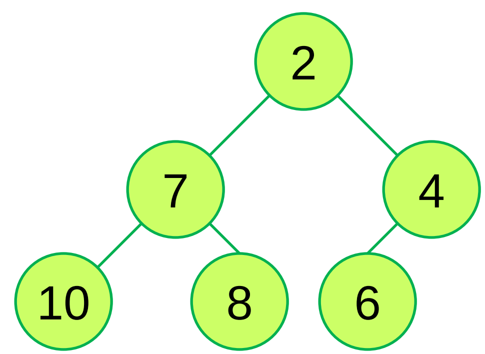
Operation	Binary heap (worst case)	Fibonacci heap (amortized)
Maximum	$\Theta(1)$	$\Theta(1)$
Extract-Max	$\Theta(\lg n)$	$O(\lg n)$
Insert	$\Theta(\lg n)$	$\Theta(1)$
Increase-Key	$\Theta(\lg n)$	$\Theta(1)$

- Compare with an array?

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein *Introduction to Algorithms, 2nd Edition. MIT Press and McGraw-Hill,* 2001.
Fredman M. L. & Tarjan R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* 34(3), pp. 596-615.

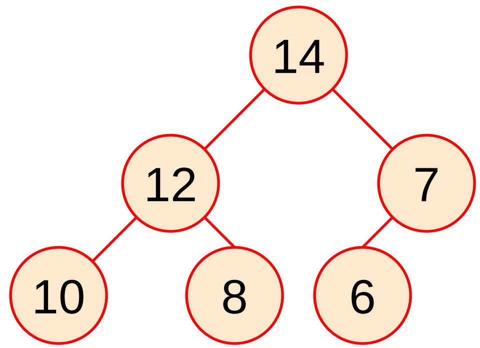
Heap

- Definition: A max (min) heap is
  - A max (min) tree: *key*[*parent*] >= (<=) *key*[*children*]
  - A complete binary tree
- Corollary: Who has the largest (smallest) key in a max (min) heap?
  - Root!
- Example

Max heap	Min heap
	

Max Heap

- Implementation?
  - Complete binary tree (except that some rightmost leaves on the bottom level may be missing) ⇒ array representation

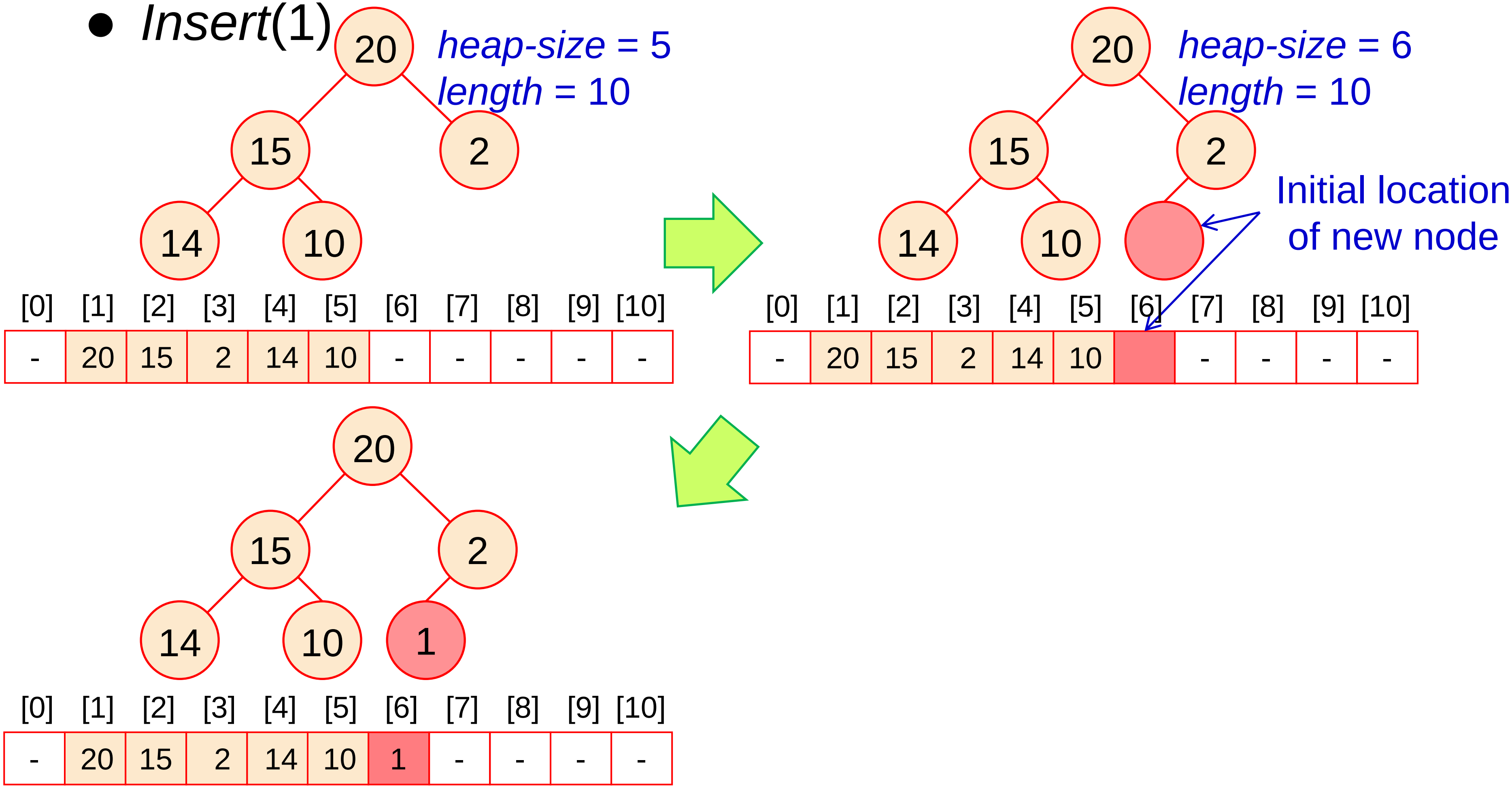


	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
heap	-	14	12	7	10	8	6	-	-	-	-

  - Root: $A[1]$
  - For $A[i]$, LEFT child is $A[2i]$, RIGHT child is $A[2i+1]$, and PARENT is $A[\lfloor i/2 \rfloor]$
  - *A.heap-size* (# of elements in the heap stored within A) ≤ *A.length* (# of elements in A)

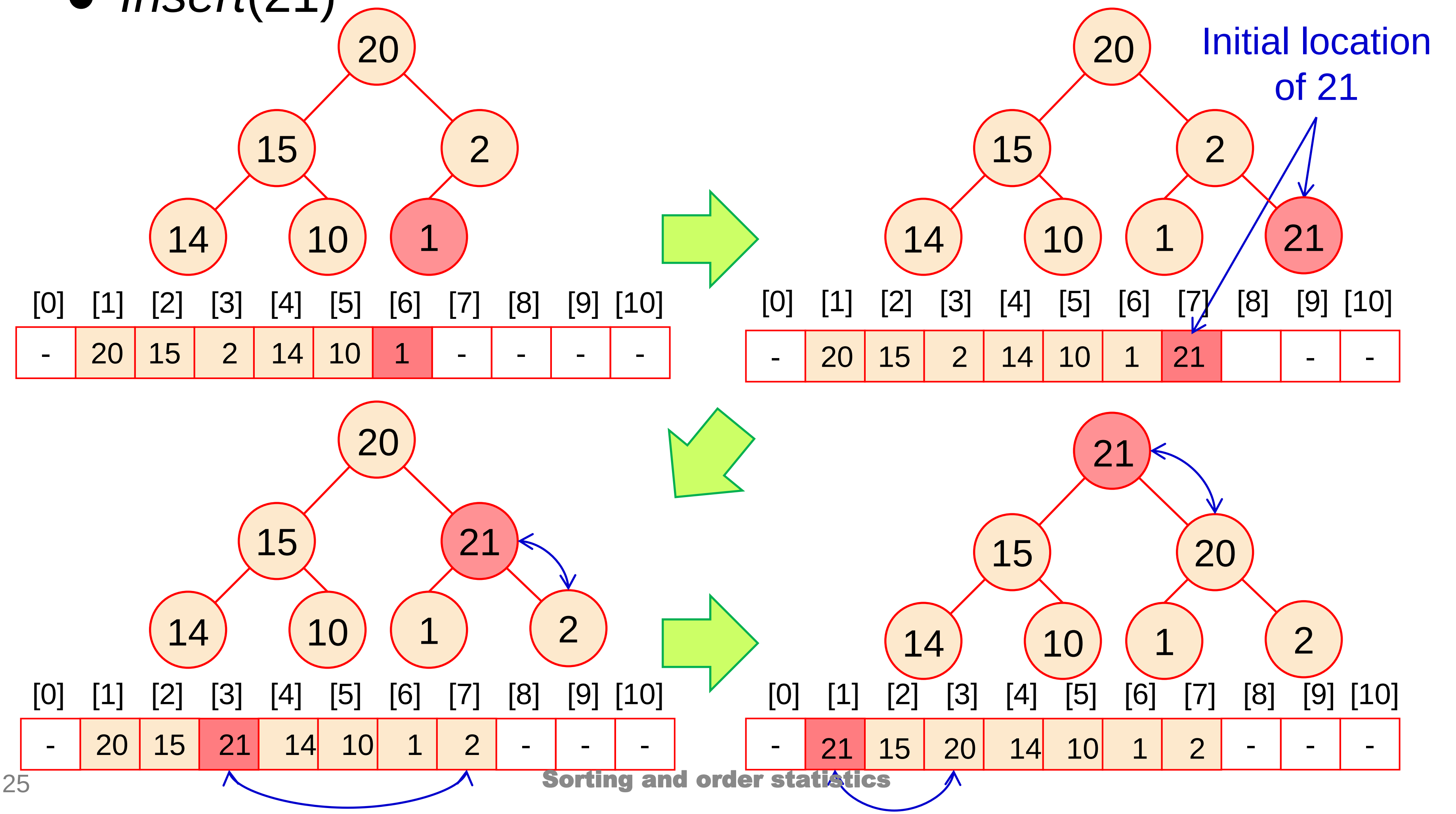
Insertion into a Max Heap (1/3)

- Maintain heap property all the times
- *Insert*(1)



Insertion into a Max Heap (2/3)

- Maintain heap ⇒ bubble up if needed!
- *Insert*(21)

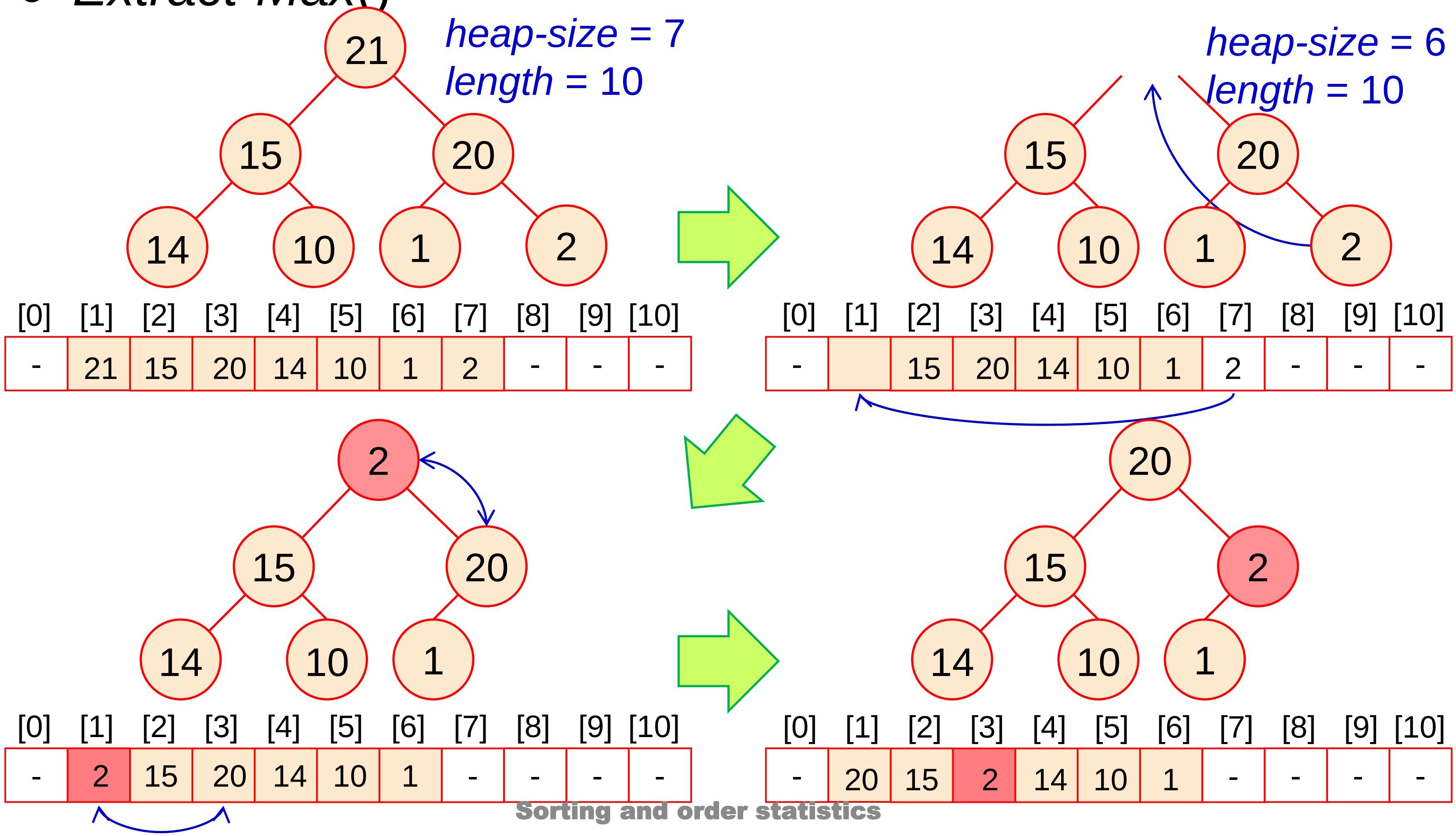


Insertion into a Max Heap (3/3)

- Time complexity?
  - How many times to bubble up in the worst case?
  - Tree height: $\Theta(\lg n)$

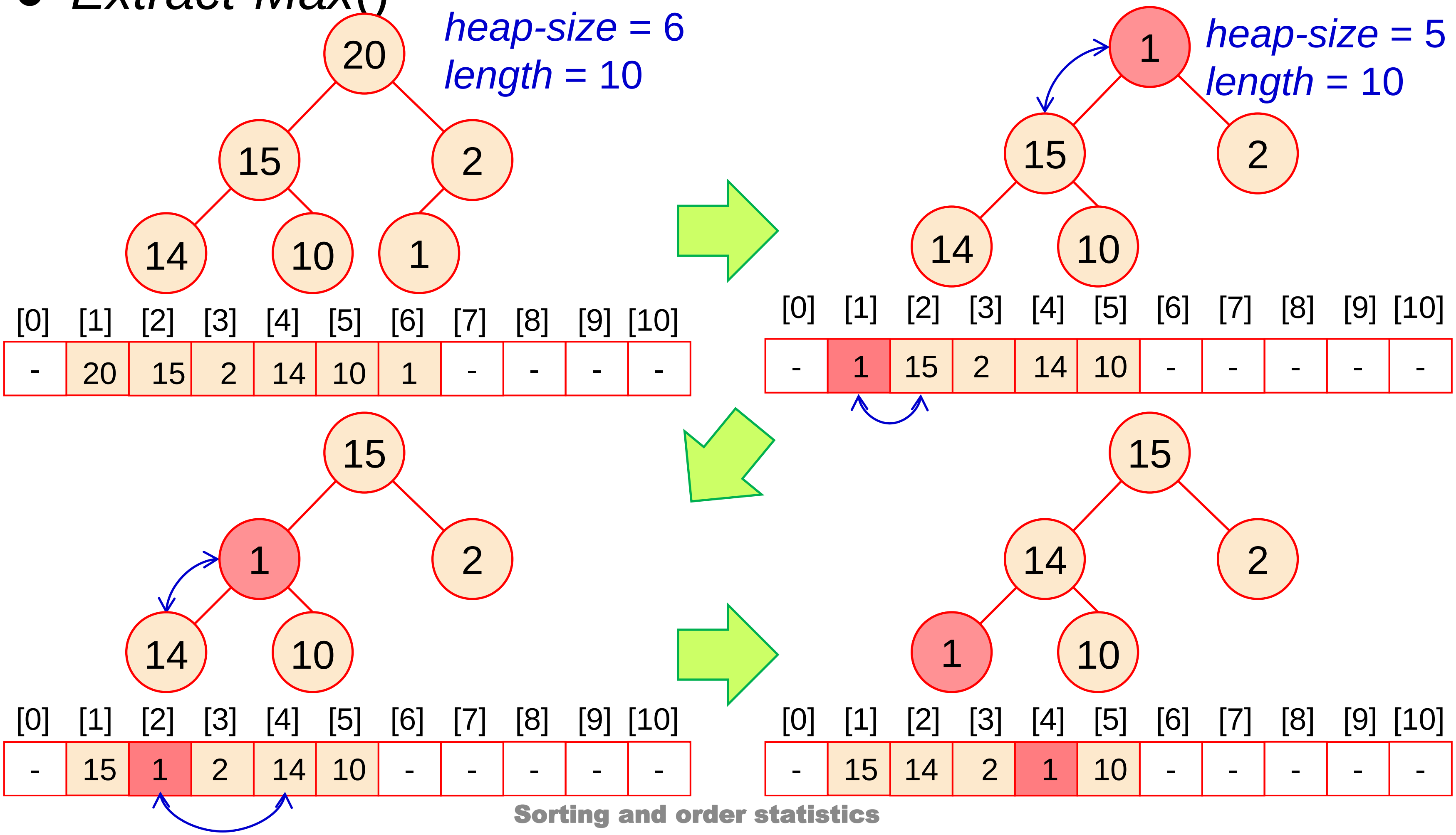
Deletion from a Max Heap (1/3)

- Maintain heap $\Rightarrow$ trickle down if needed!
- *Extract-Max*()



Deletion from a Max Heap (2/3)

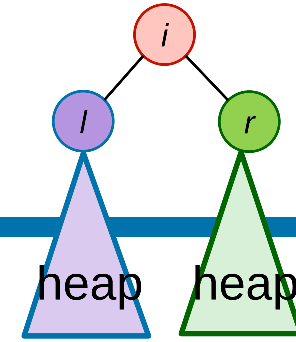
- Maintain heap ⇒ trickle down if needed!
- *Extract-Max*()



Deletion from a Max Heap (3/3)

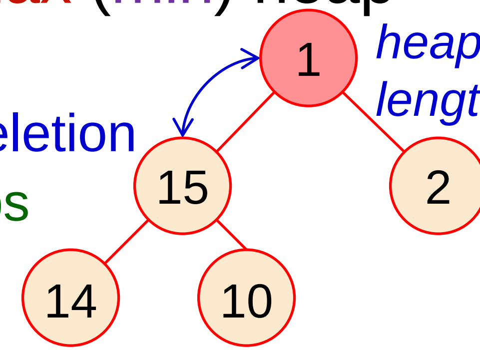
- Time complexity?
  - How many times to trickle down in the worst case? $\Theta(\lg n)$

Max Heapify ★★★★★ (1/2)

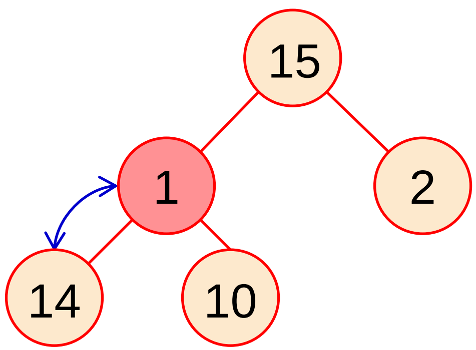


- Max (min) heapify = maintain the max (min) heap property
  - What we do to trickle down the root in deletion
  - Assume *i*'s left & right subtrees are heaps
    - But *A*[*i*] may be < (>) *A*[*children*]
  - Heapify *i* = trickle down *A*[*i*]
    ⇒ the tree rooted at *i* is a heap

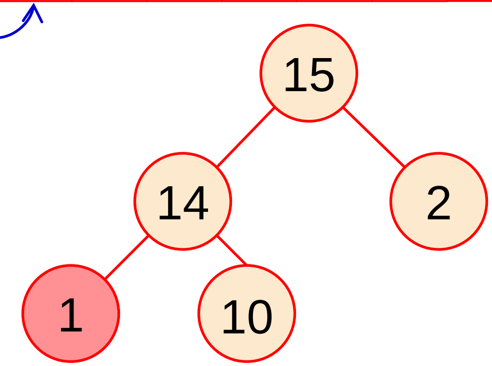
heap-size = 5
length = 10



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
-	1	15	2	14	10	-	-	-	-	-



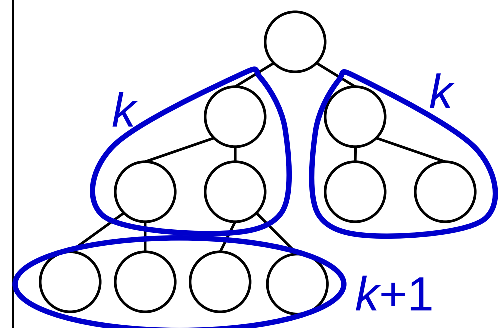
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
-	15	1	2	14	10	-	-	-	-	-



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
-	15	14	2	1	10	-	-	-	-	-

Max Heapify (2/2)

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```



- Worst case: bottom level of the tree is exactly half full ⇒ children's subtrees have size ≤ $2n/3$.
- Recurrence: $T(n) \le T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\lg n)$
- Alternatively, $O(h)$ for a node of height h
  - An n-element heap has height $\lfloor \lg n \rfloor$

Tree Height and Depth

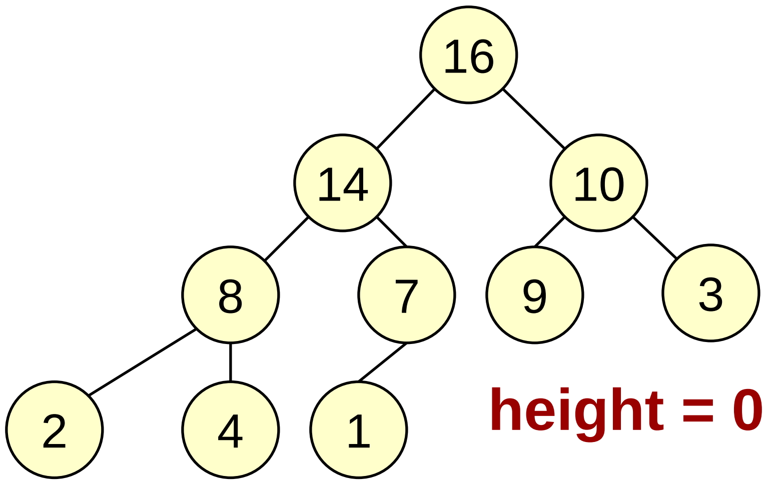
- **Height** of a node: # of edges on the longest simple downward path from the node to a leaf
- **Depth**: Length of the path from the root to a node

height = 3

height = 2

height = 1

height = 0



height = 0

depth = 0

depth = 1

depth = 2

depth = 3

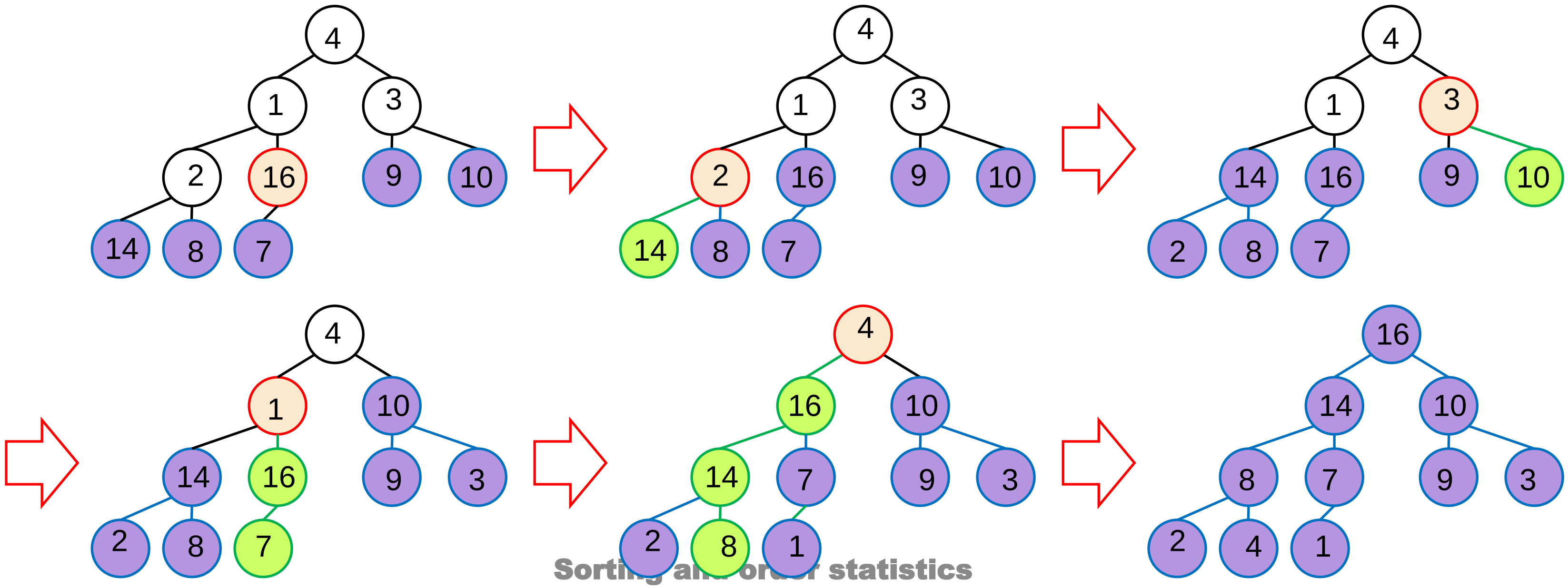
# of nodes with height 0 = 5

How to Build a Max Heap? (1/2)

Induction

- How to convert any complete binary tree to a max heap?
- Top-down manner?
- Better idea: Max heapify in a bottom-up manner
  - Induction basis: Leaves are already heaps
  - Inductive step: Start at parents of leaves, work upward till root

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
A	-	4	1	3	2	16	9	10	14	8	7



How to Build a Max Heap? (2/2)

```
BUILD-MAX-HEAP(A)
1. A.heap-size = A.length
2. for i = ⌊A.length/2⌋ downto 1
3.       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: height of a node)
  - At most $\lceil n/2^{h+1} \rceil$ nodes of height h in an n-element array
  - $T(n) = \sum_{h=0}^{\lfloor \lg n \rfloor} (\#nodes\ of\ height\ h) O(h) = \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) =$

    $cn \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} < cn \sum_{h=0}^{\infty} \frac{h}{2^h} = O(n \cdot 2) = O(n)$ Hint: $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots = 1$
  - Won't improve the overall complexity of heapsort

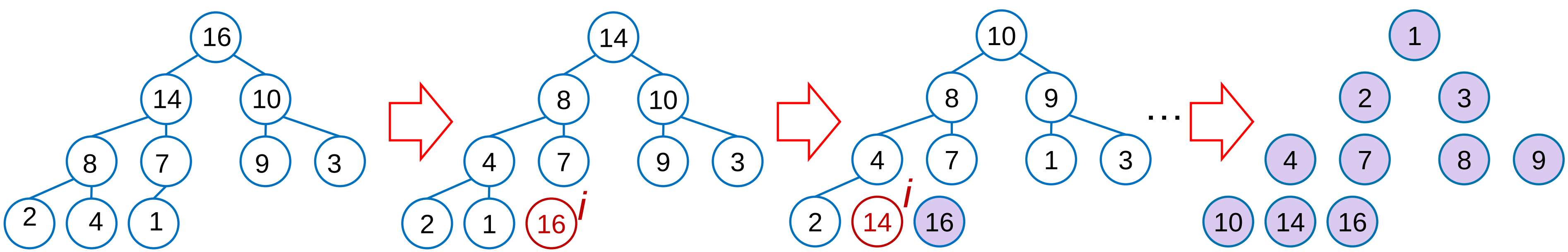
Heapsort

Heapsort is nothing but an implementation of **selection sort** using a **right data structure**

```
HEAPSORT(A)
1. BUILD-MAX-HEAP(A)                          O(n)
2. for i = A.length downto 2                  O(n)
3.     exchange A[1] with A[i]                O(1)
4.     A.heap-size = A.heap-size - 1          O(1)
5.     MAX-HEAPIFY(A,1)                       O(lgn)
```

Step 1: Convert inputs to a special data structure – heap
Step 2: Generate output based on heap property

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
A	-	16	14	10	8	7	9	3	2	4	1



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
A	-	1	2	3	4	7	8	9	10	14	16

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

Lower Bound of Sorting

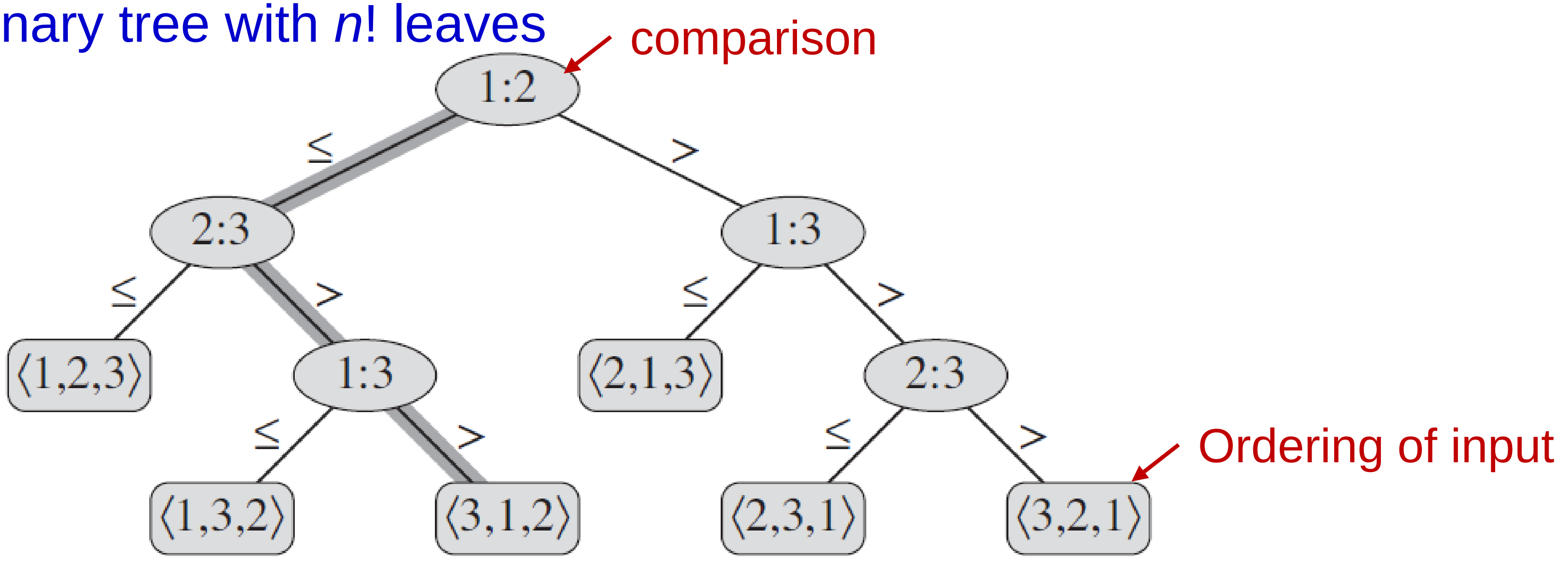
Comparison-based sorters

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 **stable** if numbers with the same value appear in the output array in the same order as they do in the input 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, radix sort, bucket sort

Decision-Tree Model for Comparison-Based Sorter

- Consider only the comparisons in the sorter
- An internal node in the tree corresponds to a comparison
- Start at root and do the first comparison: $\leq \Rightarrow$ go to the left branch; **>** $\Rightarrow$ go to the right branch
- Each leaf represents an ordering of the input (n! leaves!)
  - A binary tree with n! leaves



$\Omega(n\lg n)$ 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 binary tree T
  - T has $n!$ leaves
  - T is binary, so has $\leq 2^h$ leaves
  - $2^h \geq n!$
  - $h = \Omega(n \lg n)$ // 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

Sorting in Linear Time

Non-comparison-based sorters

Counting Sort

- **Requirement:** Input integers are in a known range [0..k]
- **Idea:** For each x, find # of elements $\leq x$ (say m, including x) and put x in the m^{th} slot
- Runs in $\Theta(n+k)$ time, but needs extra $\Theta(n+k)$ space
- Example: A: input; B: output; C: working (auxiliary) array

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

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

(a)

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

(b)

	1	2	3	4	5	6	7	8
B							3	

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

(c)

	1	2	3	4	5	6	7	8
B		0					3	

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

(d)

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

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

(e)

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

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

(f)

Counting Sort

```
COUNTING-SORT(A, B, k)
for i = 1 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the # of elements equal to i
for i = 2 to k
    C[i] = C[i] + C[i-1]
// C[i] now contains the # of elements ≤ i
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Step 1: Count

Step 2: Find out the location (how many elements at front?)

Step 3: rearrange the array

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

Radix Sort

- **Requirement:** input an array of integers, each with *d* digits
- Intuitively, one should first sort the numbers on their most significant digit, followed by the 2nd MSD, and so on
  - Problem: a lot of intermediate sets of numbers must be kept
- **Idea:** counter-intuitively, it sorts the numbers on their least significant digit first, the 2nd LSD second, and so on

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Radix Sort

RADIX-SORT(A, d)
1. **for** $i = 1$ **to** d
2. Use a **stable** sorter to sort array A on digit i

- Sort records keyed by multiple fields: year, month, day
- Time complexity: $\Theta(d(n+k))$ for n d-digit numbers in which each digit has k possible values.
  - Which sorter?
- If counting sort is used as the intermediate stable sort
  - Not in-place $\Rightarrow$ require more memory
- If insertion sort is used as the intermediate stable sort
  - $O(n^2)$

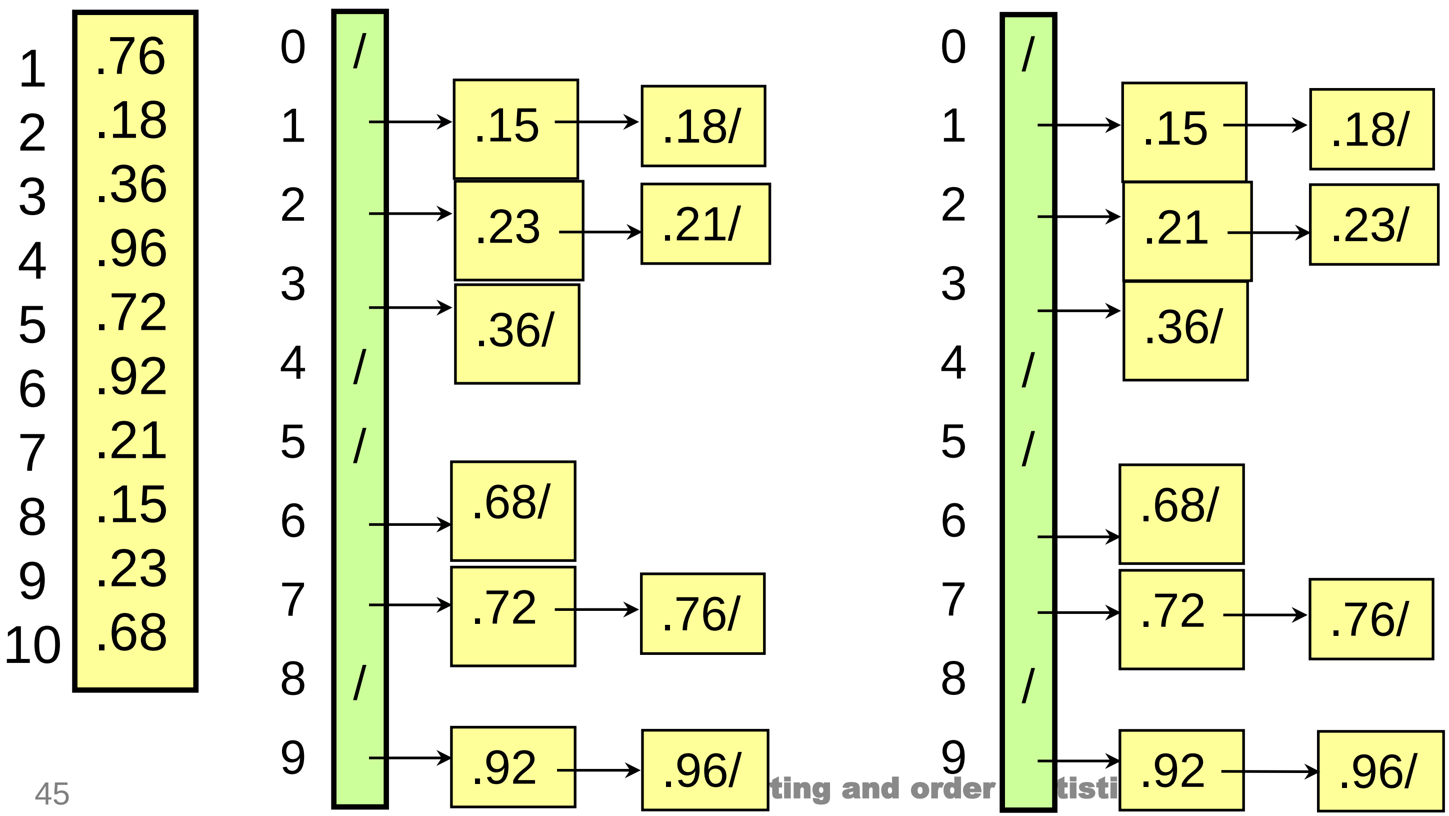
Bucket Sort

- **Requirement:** Input uniformly distributes over interval [0,1)
- Divide the interval [0,1) into *n* equal-sized buckets, and then distribute the *n* input numbers into them

Step 1: distribute

Step 2: sort

Step3: combine



Concatenate buckets

Which sorter is used in step 2?

Notes on Sorting in Linear Time

Non-comparison-based sorters

Algorithm	Runtime			Properties	
	Best case	Average case	Worst case	Stable?	In-place?
Counting	$O(n+k)$	$O(n+k)$	$O(n+k)$	Yes	No
Radix	$O(d(n+k'))$	$O(d(n+k'))$	$O(d(n+k'))$	Yes*	No
Bucket	–	$O(n)$	–	Yes	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
  - Unstable, in-place radix sort can be implemented
- Bucket sort: Expected linear time if the sum of the squares of the bucket sizes is linear in the # of elements (even if the input is not drawn from a uniform distribution)

Order Statistics

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: $\left\lfloor \frac{n+1}{2} \right\rfloor, \left\lceil \frac{n+1}{2} \right\rceil$-th order statistic

    low median high median
- **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*, $1 \le i \le n$
  - **Output:** The element $x \in A$ that is larger than exactly (i -1) other 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 A.length
3.     if  min > A[i]
4.         min = A[i]
5. return min
```

- **Exactly** *n*-1 comparisons
  - Best possible?
  - Lower bound: Every element except the winner must lose at least one match. *n*-1 comparisons are necessary to determine the minimum

Simultaneous Minimum and Maximum

- Naive simultaneous minimum and maximum: $2n$-3 comparisons.
  - Best possible?
  - 1+(n-2)*2

- Are $3\left\lfloor\frac{n}{2}\right\rfloor$ comparisons possible?
  - Idea: process elements in pairs
  - Maintain minimum and maximum
  - Compare pairs of elements from the input first with each other
  - Compare the smaller with the current minimum and the larger to the current maximum
  - Need 3 comparisons for every 2 elements
  - n is odd: $1 + 3*(n-3)/2+2 = 3\lfloor n/2\rfloor$
  - n is even: $1 + 3(n-2)/2 = 3n/2 - 2$

Selection in Expected Linear Time

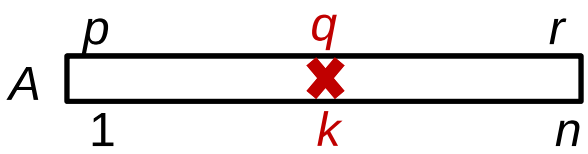
```
Randomized-Select(A, p, r, i) // Query ith order statistic
if p == r
    return A[p]
q = Randomized-Partition(A, p, r)
k = q – p + 1
if i == k // the pivot value is the answer
    return A[q]
if i < k
    return Randomized-Select(A, p, q-1, i)
else return Randomized-Select(A, q+1, r, i-k)
```

- Randomized-Partition first swaps $A[r]$ 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: 0 : n-1 partitions $\Rightarrow T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$
  - Best case: $T(n) = \Theta(n)$

Selection in Expected Linear Time (1/3)

- $X_k = \mathrm{I}\{A[p..q] \text{ has exactly } k \text{ elements}\},\ 1 \leq k \leq n$

$$\mathrm{E}[X_k] = \frac{1}{n}$$



- Three possibilities:
  - 1) terminate with the correct answer
  - 2) recurse on $A[p..q-1]$
  - 3) recurse on $A[q+1..r]$
- Assuming that $T(n)$ is monotonically increasing:

$$T(n) \leq \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n))$$

$$= \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)$$

$$\mathrm{E}[T(n)] \leq \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right]$$

Selection in Expected Linear Time (2/3)

$$\mathrm{E}[T(n)] \le \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right]$$

$$= \sum_{k=1}^{n} \mathrm{E}[X_k] \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n)$$

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

$$\mathrm{E}[T(\max(0, n-1))] + \mathrm{E}[T(\max(1, n-2))] + \cdots + \mathrm{E}[T(\max(n-2, 1))] + \mathrm{E}[T(\max(n-1, 0))]$$

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

Selection in Expected Linear Time (3/3)

- Substitution: Assume $\mathrm{E}[(T(k)] \le ck$ for $k < n$

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

$$= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k\right) + an$$

$$= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2}\right) + an$$

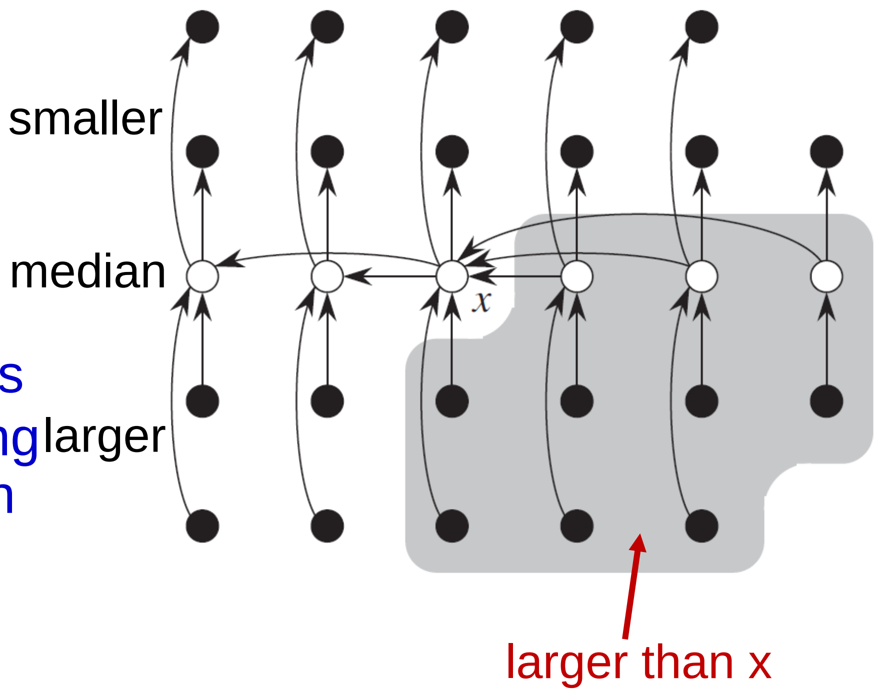
$$\le \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2}\right) + an$$

$$= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + an = c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + an$$

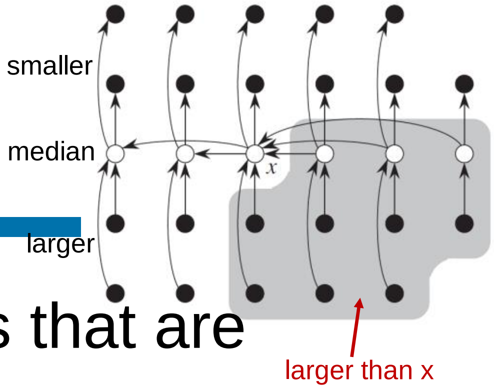
$$\le \frac{3cn}{4} + \frac{c}{2} + an = cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right) \le cn$$ Linear time!

Selection in Worst-Case Linear Time

- **Idea:** guarantee a good split upon partitioning the array
- 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 by insertion sort
  3. Call SELECT recursively to find the median *x* of the $\lceil n/5 \rceil$ medians
  4. Partition array *A* around *x*, splitting it into two arrays of $A[p, q\text{-}1]$ (with k-1 elements) and $A[q+1, r]$ (with n-k elements)
  5. If $i = k$, return *x*. Otherwise, SELECT(A, p, q-1, i) when $i < k$ or SELECT(A, q+1, r, i-k) when $i > k$



Runtime Analysis



- Determine a lower bound on # of elements that are greater than the partitioning element *x*
- SELECT guarantees *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 7*n*/10 + 6 elements in the bigger subproblem

- Running 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 $T(n)=O(n)$