Heap sort and Quick sort

This Lecture

- Sorting algorithms
- Heapsort
    - Heap *data structure* and priority queue *ADT*
- Quick sort
    - a popular algorithm, very fast on average

Sorting Algorithms so far

- Insertion sort
  - Worst-case running time $\Theta(n^2)$; in-place
- Merge sort
  - Worst-case running time $\Theta(n \log n)$; but requires additional memory

Selection Sort

```
Selection-Sort(A[1..n]):
   For i = n downto 2
A:    Find the largest element among A[1..i]
B:    Exchange it with A[i]
```

- A takes $\Theta(n)$ and B takes $\Theta(1)$: $\Theta(n^2)$ in total
- Idea for improvement: use a *data structure*, to do both A and B in $O(\lg n)$ time, balancing the work, achieving a better trade-off, and a total running time $O(n \log n)$

Heap Sort

- Combines the better attributes of merge sort and insertion sort.
  - Like merge sort, but unlike insertion sort, running time is $O(n \lg n)$.
  - Like insertion sort, but unlike merge sort, sorts in place.
- Introduces an algorithm design technique
  - Create data structure (*heap*) to manage information during the execution of an algorithm.
- The *heap* has other applications beside sorting.
  - Priority Queues
- Binary heap data structure A
  - array
  - Can be viewed as a nearly complete binary tree
    - All levels, except the lowest one are completely filled
  - The key in root is greater or equal than all its children, and the left and right subtrees are again binary heaps

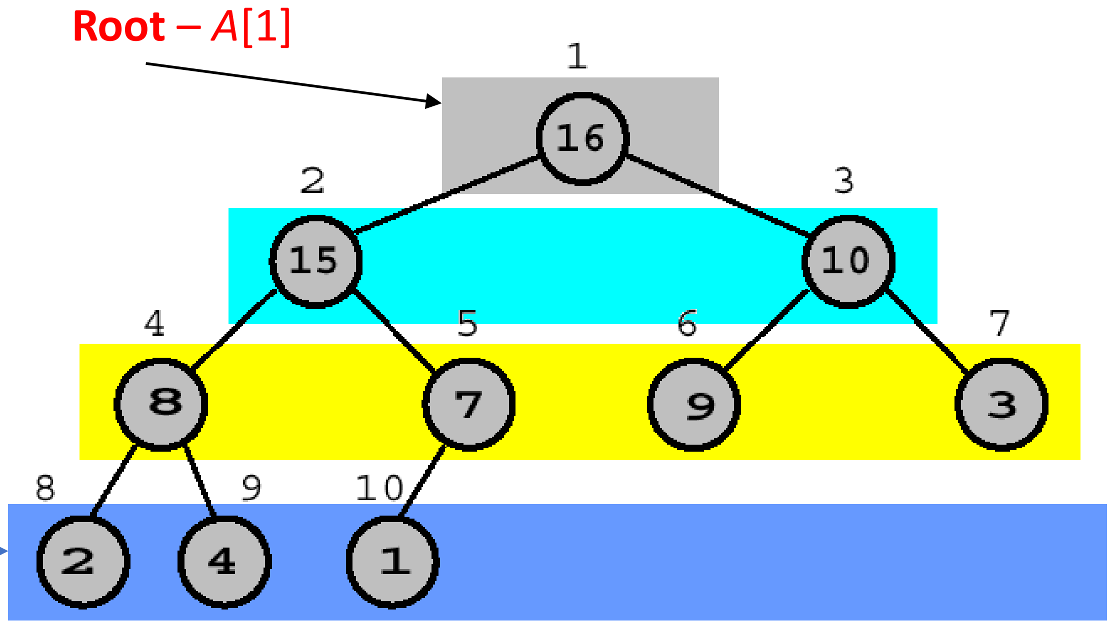
Heap Sort

Parent (i)
return $\lfloor i/2 \rfloor$

Left (i)
return $2i$

Right (i)
return $2i+1$

Last row filled from left to right.



length[A] – number of elements in A.
heap-size[A] – number of elements in heap stored in A.

heap-size[A] $\leq$ length[A]

1	2	3	4	5	6	7	8	9	10
16	15	10	8	7	9	3	2	4	1

Level: 3 2 1 0

Largest element is stored at the root.
In any subtree, no values are larger than the value stored at subtree root.

Max- heap property:
$A[\text{Parent}(i)] \geq A[i]$

Heap Sort

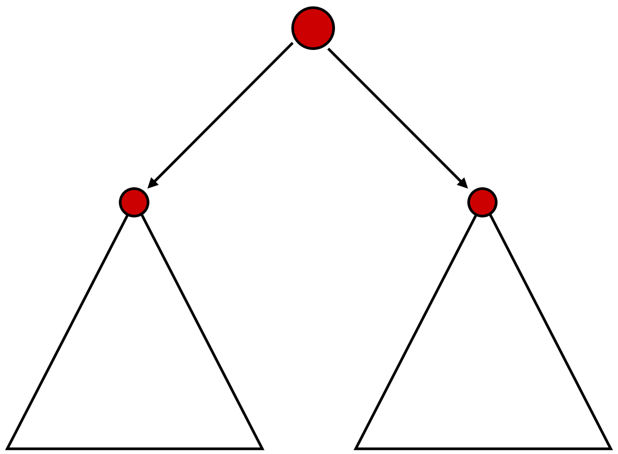
- Notice the implicit tree links; children of node i are $2i$ and $2i+1$
- *Height of a tree*: the height of the root: $\lfloor \lg n \rfloor$
- No. of *leaves* $= \lceil n/2 \rceil$
- *Height of a node in a tree*: the number of edges on the longest simple downward path from the node to a leaf.
- No. of nodes of height h $\leq \lceil n/2^{h+1} \rceil$
- Why is this useful?
  - In a binary representation, a multiplication/division by two is left/right shift
  - Adding 1 can be done by adding the lowest bit

Heap Sort (Basics)

- Use max-heaps for sorting.
- The array representation of max-heap is not sorted.
- Steps in sorting
  - Convert the given array of size n to a max-heap (*BuildMaxHeap*)
  - Swap the first and last elements of the array.
    - Now, the largest element is in the last position – where it belongs.
    - That leaves $n - 1$ elements to be placed in their appropriate locations.
    - However, the array of first $n - 1$ elements is no longer a max-heap.
    - Float the element at the root down one of its subtrees so that the array remains a max-heap (MaxHeapify)
    - Repeat step 2 until the array is sorted.

Heapify: Maintaining the heap property

- Suppose two subtrees are max-heaps, but the root violates the max-heap property.
- Fix the offending node by exchanging the value at the node with the larger of the values at its children.
  - May lead to the subtree at the child not being a heap.
- Recursively fix the children until all of them satisfy the max-heap property.

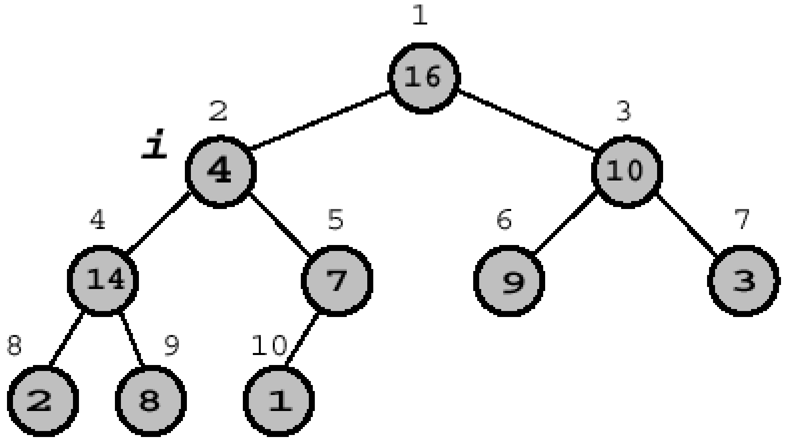


Heapify

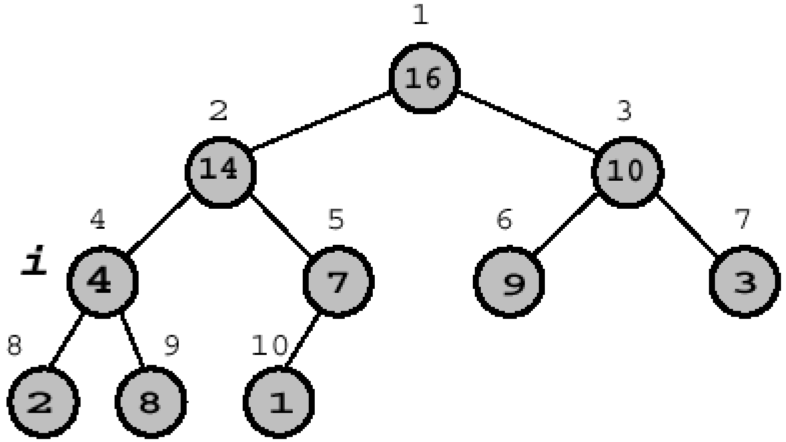
- *i* is index into the array *A*
- Binary trees rooted at Left(*i*) and Right(*i*) are heaps
- But, *A*[*i*] might be smaller than its children, thus violating the heap property
- The method **Heapify** makes *A* a heap once more by moving *A*[*i*] down the heap until the heap property is satisfied again

Heapify Example

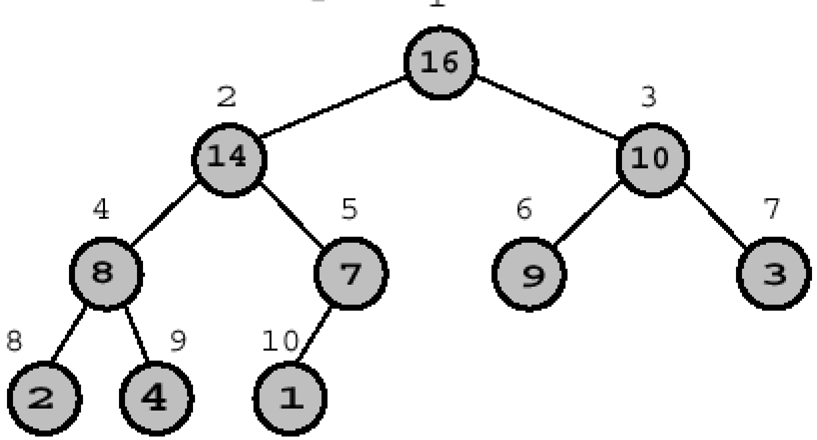
1. Call HEAPIFY(A,2)



2. Exchange A[2] with A[4] and
   recursively call HEAPIFY(A,4)



3. Exchange A[4] with A[9] and
   recursively call HEAPIFY(A,9)



4. Node 9 has no children,
   so we are done.

Heapify

<u>*MaxHeapify(A, i)*</u>

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \textit{heap-size}[A]$ and $A[l] > A[i]$
4. **then** *largest* $\leftarrow l$
5. **else** *largest* $\leftarrow i$
6. **if** $r \leq \textit{heap-size}[A]$ **and** $A[r] > A[\textit{largest}]$
7. **then** *largest* $\leftarrow r$
8. **if** $\textit{largest} \neq i$
9. **then** exchange $A[i] \leftrightarrow A[\textit{largest}]$
10. *MaxHeapify*(*A*, *largest*)

<u>Assumption:</u>
Left(i) and Right(i) are max-heaps.

Time to fix node i and its children = **$\Theta(1)$**

+

Time to fix the subtree rooted at one of i's children = T(size of subree at *largest*)

- $T(n) = T(\textit{largest}) + \Theta(1)$
- *largest* $\leq 2n/3$ (worst case occurs when the last row of tree is exactly half full)
- $T(n) \leq T(2n/3) + \Theta(1)$ $\Rightarrow T(n) = O(\lg n)$

Alternately, MaxHeapify takes $O(h)$ where h is the height of the node where MaxHeapify is applied

Building a Heap

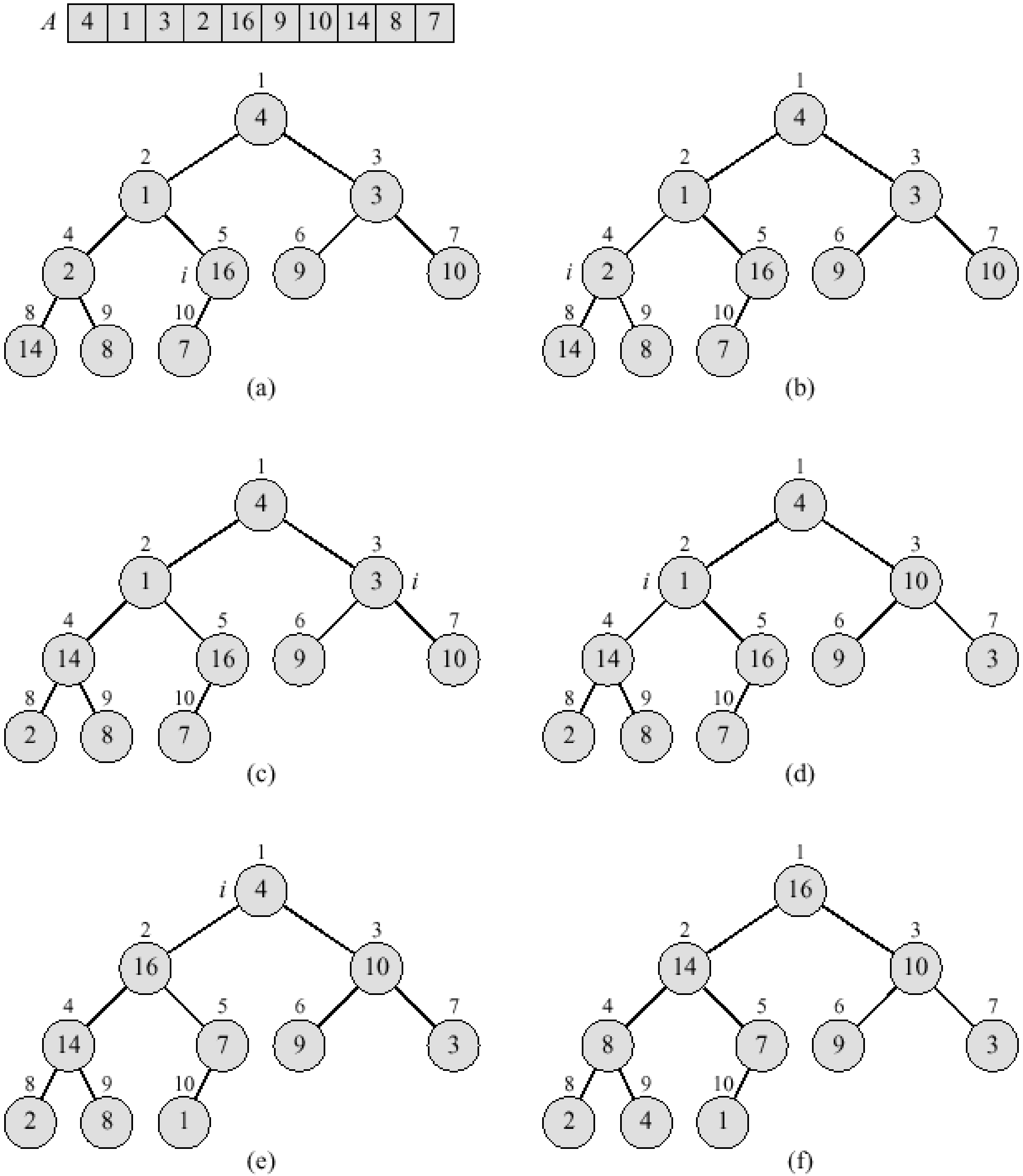
- Convert an array $A[1...n]$, where n = length[A], into a maxheap using *MaxHeapify*.
- Notice that the elements in the subarray $A[(\lfloor n/2 \rfloor + 1)...n]$ are already 1-element heaps to begin with!

<u>*BuildMaxHeap*(*A*)</u>

1. *heap-size*[*A*] $\leftarrow$ *length*[*A*]
2. **for** $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
3.   **do** *MaxHeapify*(*A*, *i*)

Why we cannot start from 1 to n/2?

Building a Heap



Building a Heap: Correctness

- <u>Loop Invariant:</u> At the start of each iteration of the **for** loop, each node $i+1$, $i+2$, …, n is the root of a max-heap.
- <u>Initialization:</u>
  - Before first iteration $i = \lfloor n/2 \rfloor$
  - Nodes $\lfloor n/2 \rfloor+1$, $\lfloor n/2 \rfloor+2$, …, n are leaves and hence roots of max-heaps.
- <u>Maintenance:</u>
  - By LI, subtrees at children of node i are max heaps.
  - Hence, MaxHeapify(i) renders node i a max heap root (while preserving the max heap root property of higher-numbered nodes).
  - Decrementing i reestablishes the loop invariant for the next iteration.

Building a Heap: Analysis

- Running time: *n* calls to Heapify = *n O(lg n) = O(n lg n)*
- Good enough for an *O(n lg n)* bound on Heapsort, but sometimes we build heaps for other reasons, would be nice to have a tight bound
  - Intuition: for most of the time Heapify works on smaller than *n* element heaps

Building a Heap: Analysis (2)

- Definitions
  - height of node: longest path from node to leaf
  - height of tree: height of root
  - time to Heapify = O(height of subtree rooted at *i*)
  - assume $n = 2^k - 1$ (a complete binary tree $k = \lfloor lg\ n \rfloor$)

<u>*BuildMaxHeap*(*A*)</u>
1. *heap-size*[*A*] ← *length*[*A*]
2. **for** *i* ← ⌊*length*[*A*]/2⌋ **downto** 1
3.   **do** *MaxHeapify*(*A*, *i*)

$$
\begin{aligned}
T(n) &= O\left(\frac{n+1}{2} + \frac{n+1}{4}\cdot 2 + \frac{n+1}{8}\cdot 3 + \ldots + 1\cdot k\right) \\
&= O\left((n+1)\cdot\sum_{i=1}^{\lfloor \lg n\rfloor}\frac{i}{2^i}\right) \text{ since } \sum_{i=1}^{\lfloor \lg n\rfloor}\frac{i}{2^i} = \frac{1/2}{(1-1/2)^2} = 2 \\
&= O(n)
\end{aligned}
$$

Building a Heap: Analysis (3)

- How? By using the following "trick"

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \text{ if } |x| < 1 \quad \text{//differentiate}$$

$$\sum_{i=1}^{\infty} i \cdot x^{i-1} = \frac{1}{(1-x)^2} \quad \text{//multiply by } x$$

$$\sum_{i=1}^{\infty} i \cdot x^{i} = \frac{x}{(1-x)^2} \quad \text{//plug in } x = \frac{1}{2}$$

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = \frac{1/2}{1/4} = 2$$

- Therefore Build-Heap time is O(*n*)

Heap Sort

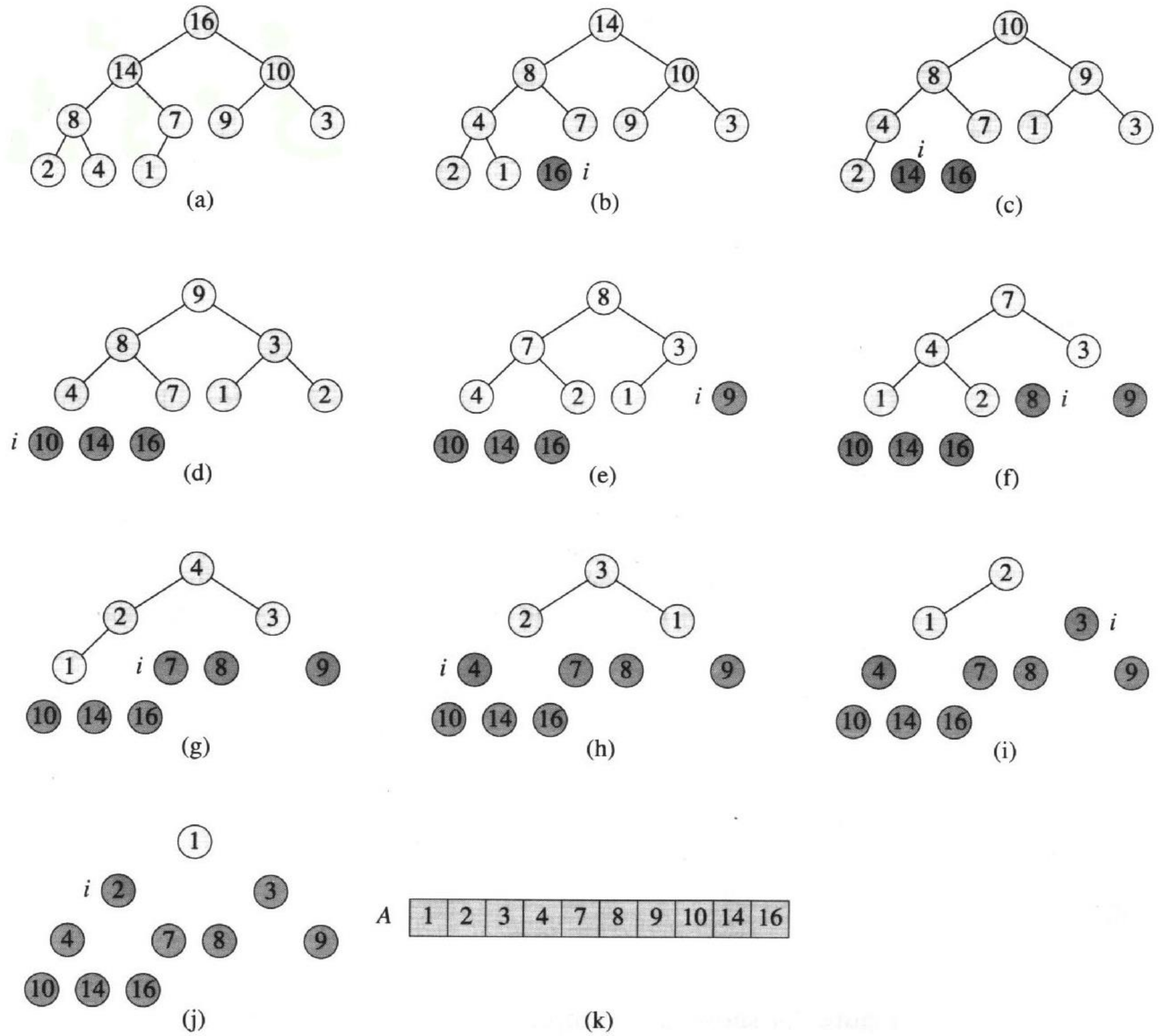
- Sort by maintaining the as yet unsorted elements as a max-heap.
- Start by building a max-heap on all elements in *A*.
  - Maximum element is in the root, *A*[1].
- Move the maximum element to its correct final position.
  - Exchange *A*[1] with *A*[*n*].
- Discard *A*[*n*] – it is now sorted.
  - Decrement heap-size[*A*].
- Restore the max-heap property on *A*[1..*n*–1].
  - Call *MaxHeapify*(*A*, 1).
- Repeat until heap-size[*A*] is reduced to 2.

<u>*HeapSort*(*A*)</u>

```
Build-Max-Heap(A)
for i ← length[A] downto 2
    do exchange A[1] ↔ A[i]
        heap-size[A] ← heap-size[A] – 1
        MaxHeapify(A, 1)
```

Build-Max-Heap takes $O(n)$ and each of the n-1 calls to Max-Heapify takes time $O(\lg n)$. Therefore, $T(n) = O(n \lg n)$

Heap Sort



Heap Sort: Summary

- Heap sort uses a heap data structure to improve selection sort and make the running time asymptotically optimal
- Running time is $O(n \log n)$ – like merge sort, but unlike selection, insertion, or bubble sorts
- Sorts in place – like insertion, selection or bubble sorts, but unlike merge sort

Priority Queues

- A priority queue is an *ADT(abstract data type)* for maintaining a set S of elements, each with an associated value called key
- A PQ supports the following operations
  - Insert(S,x) insert element x in set S ($S \leftarrow S \cup \{x\}$)
  - Maximum(S) returns the element of S with the largest key
  - Extract-Max(S) returns and removes the element of S with the largest key
  - Increase-Key(S, x, k) – increases the value of element x's key to the new value k.

Priority Queues

- Applications:
  - job scheduling shared computing resources (Unix)
  - Event simulation
  - As a building block for other algorithms
- A Heap can be used to implement a PQ

Priority Queues

<u>*Heap-Extract-Max*(*A*)</u>

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

- Removal of max takes constant time,
- Then, MaxHeapify is applied.
- Running time : Dominated by the running time of MaxHeapify , which is $O(\lg n)$

Priority Queues

```
Heap-Insert(A, key)
heap-size[A] ← heap-size[A] + 1
    i ← heap-size[A]
while i > 1 and A[Parent(i)] < key
      do A[i] ← A[Parent(i)]
            i ← Parent(i)
A[i] ← key
```

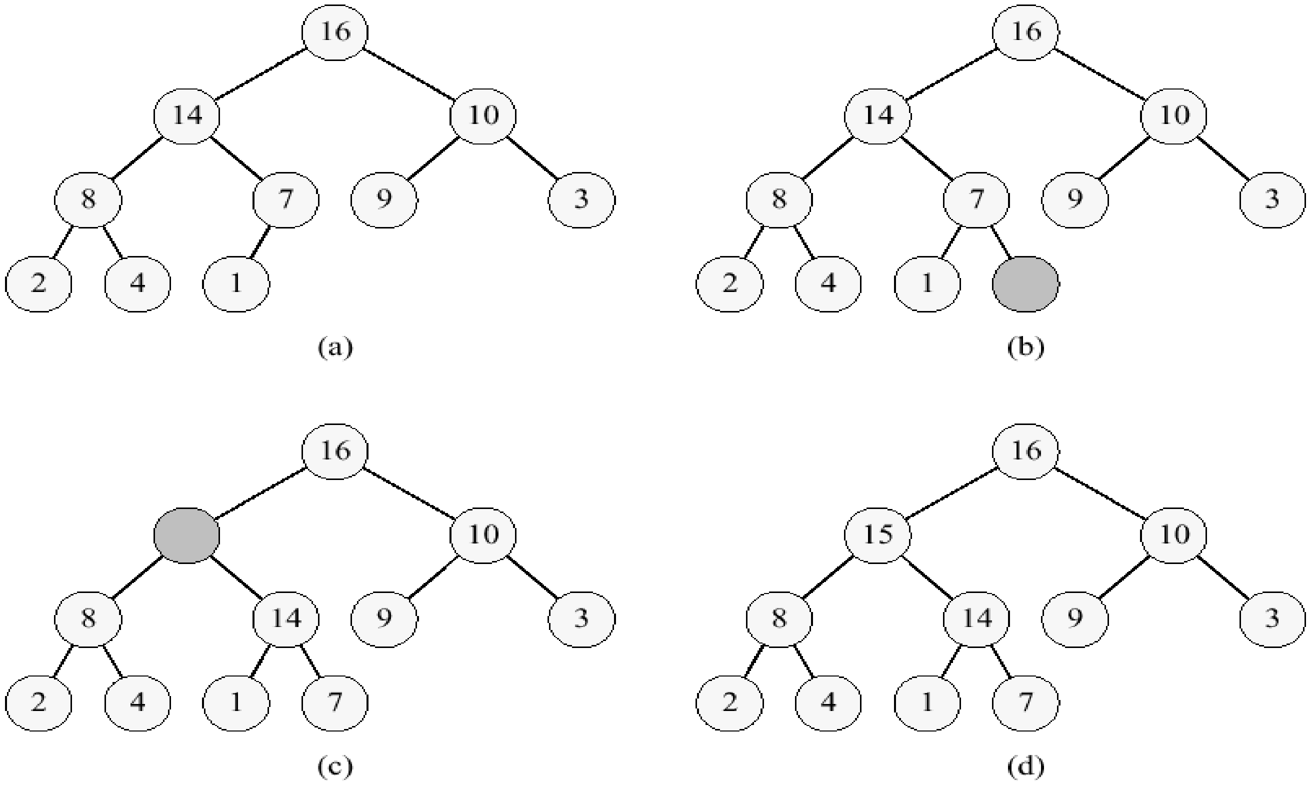
- Insertion of a new element
  - enlarge the PQ and propagate the new element from last place ”up” the PQ
  - tree is of height *lg n*, running time: $\Theta(\lg n)$

Priority Queues

```
Heap-Increase-Key(A, i, key)
If key < A[i]
    then error "new key is smaller than the current key"
A[i] ← key
while i > 1 and  A[Parent[i]] < A[i]
        do exchange A[i] ↔ A[Parent[i]]
                i ← Parent[i]
```

- Increasing the key of A[i]
  - enlarge the PQ and propagate the new key from its location to"up" the PQ
  - tree is of height *lg n*, running time: $\Theta(\lg n)$

Priority Queues



Inserting key =15

Quick Sort

- Characteristics
  - sorts in "place," i.e., does not require an additional array
  - like insertion sort, unlike merge sort
  - very practical, average sort performance O(n log n), but worst case O(n^2)

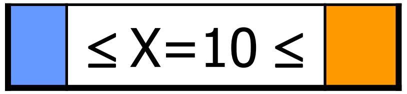
Quick Sort – the Principle

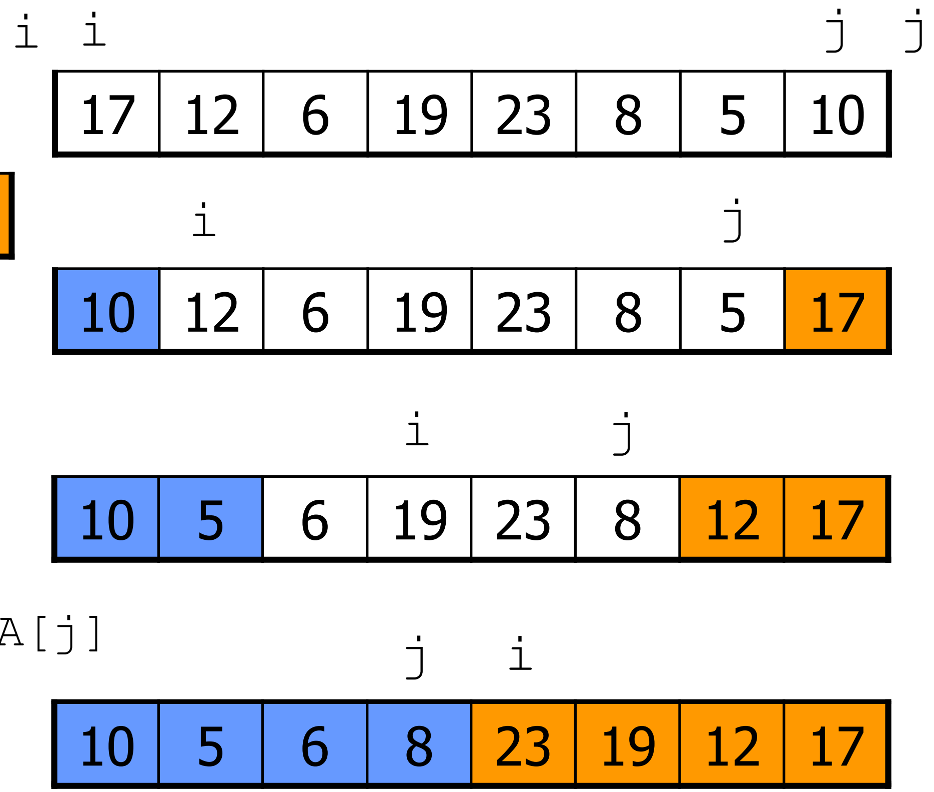
- To understand quick-sort, let’s look at a high-level description of the algorithm
- A divide-and-conquer algorithm
  - **Divide**: partition array into 2 subarrays such that elements in the lower part <= elements in the higher part
  - **Conquer**: recursively sort the 2 subarrays
  - **Combine**: trivial since sorting is done in place

Partitioning

- Linear time partitioning procedure

```
Partition(A,p,r)
x←A[r]
i←p-1
j←r+1
while TRUE
    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
```

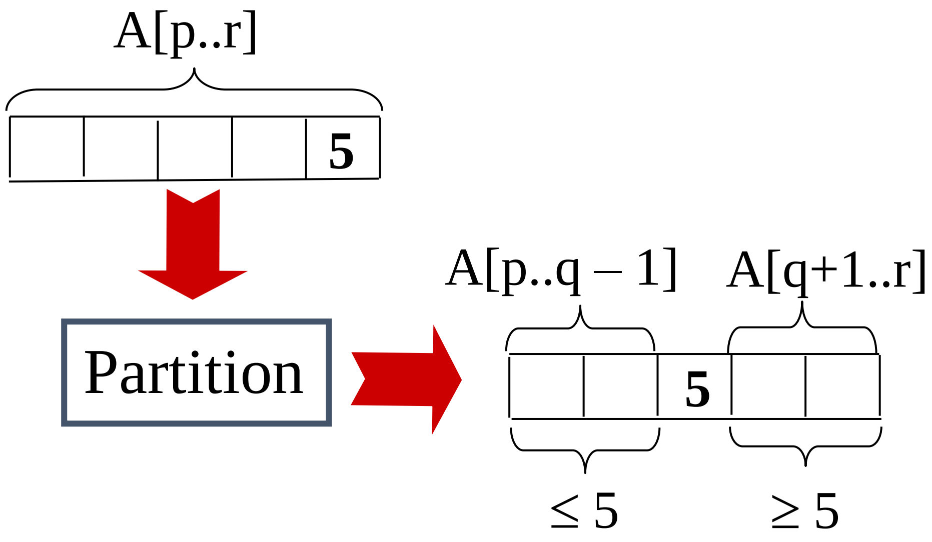




Quick Sort Algorithm

- Initial call **Quicksort(A, 1, length[A])**

```
Quicksort(A,p,r)
if p<r
   then q←Partition(A,p,r)
        Quicksort(A,p,q)
        Quicksort(A,q+1,r)
```



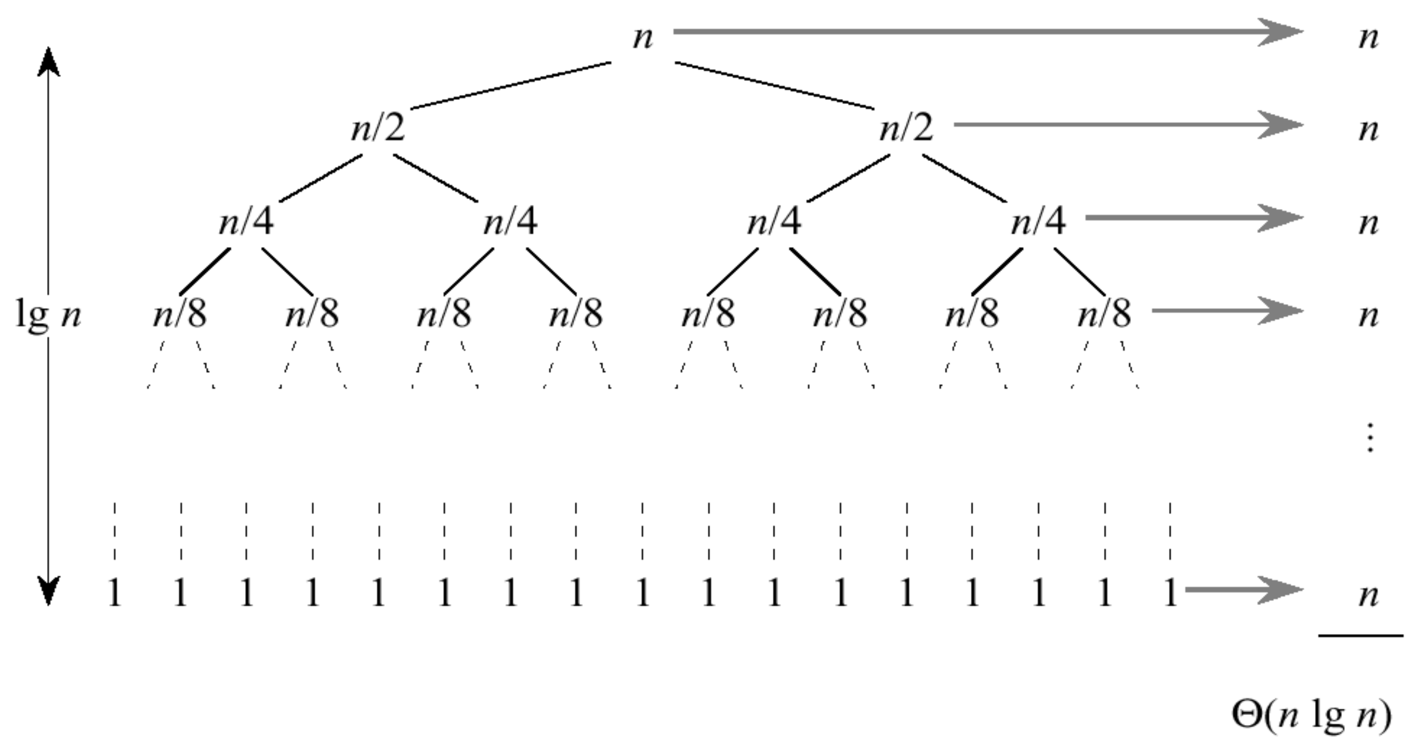
Analysis of Quicksort

- Assume that all input elements are distinct
- The running time depends on the distribution of splits

Best Case

- If we are lucky, Partition splits the array evenly

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

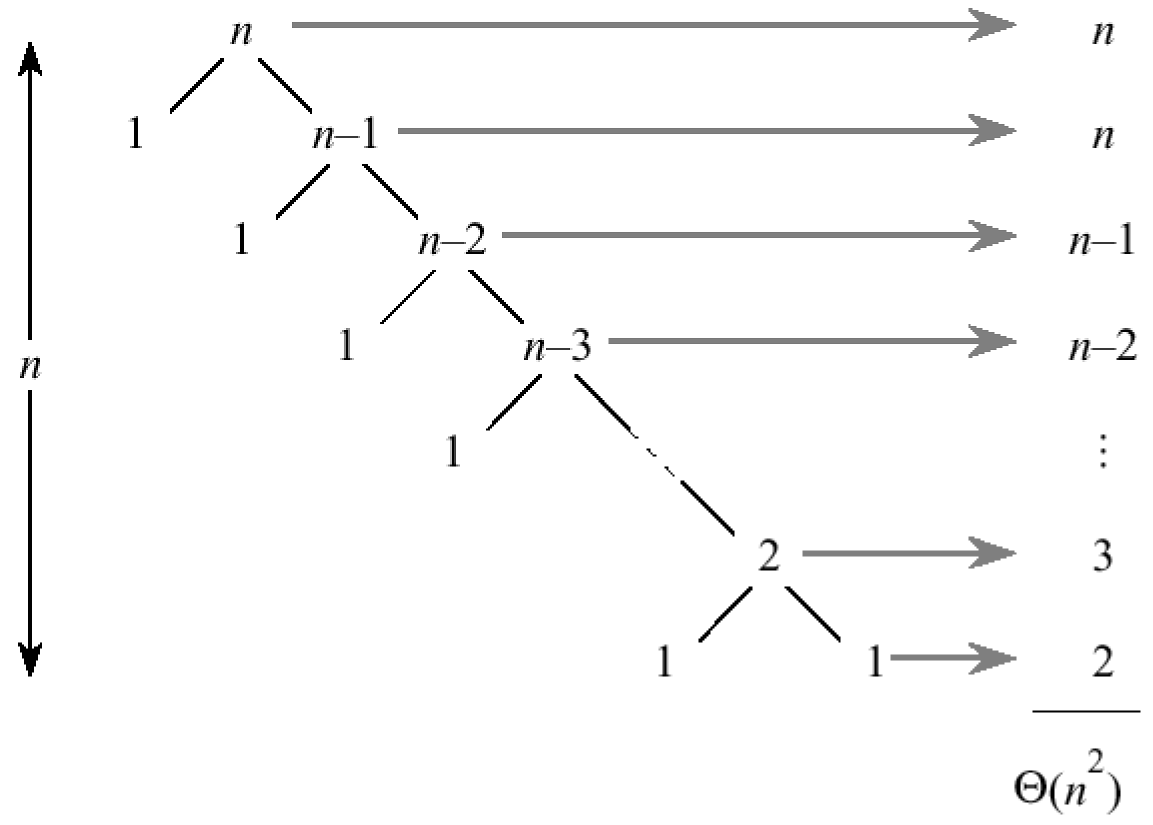


Worst Case

- What is the worst case?
- One side of the parition has only one element

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

Worst Case (2)



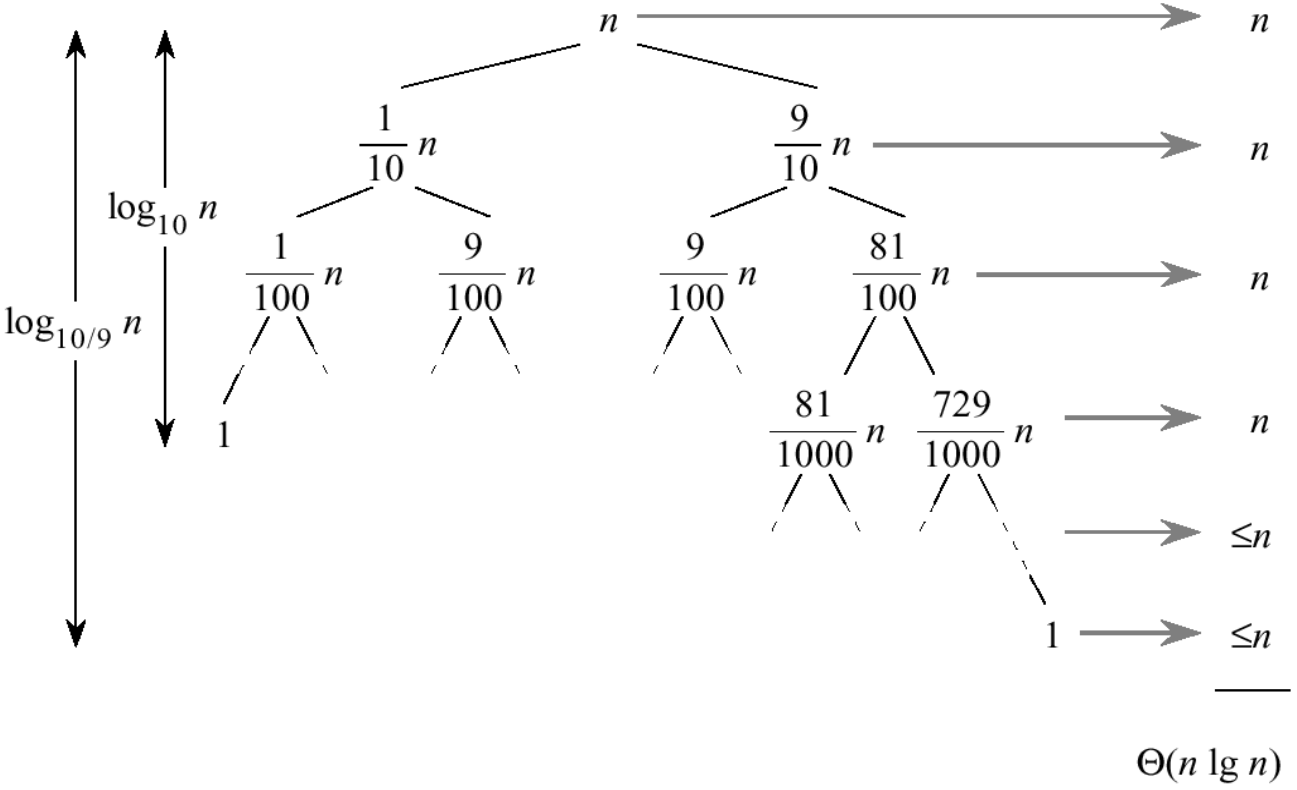
Worst Case (3)

- When does the worst case appear?
  - input is sorted
  - input reverse sorted
- Same recurrence for the worst case of insertion sort
- However, sorted input yields the best case for insertion sort!

Analysis of Quicksort

- Suppose the split is 1/10 : 9/10

$$T(n) = T(n/10) + T(9n/10) + \Theta(n) = \Theta(n \log n)$$

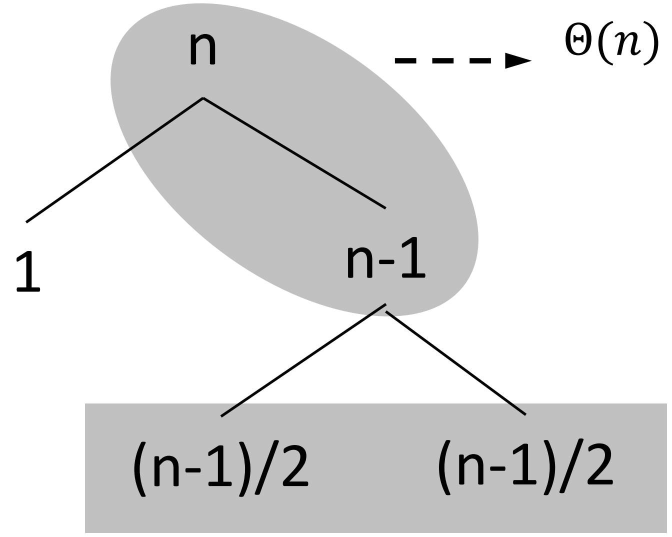


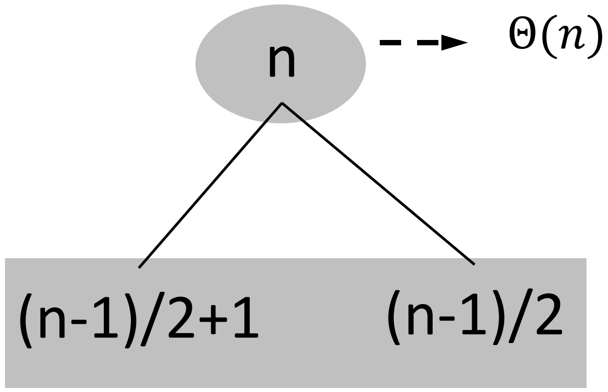
An Average Case Scenario

- Suppose, we alternate lucky and unlucky cases to get an average behavior

$L(n) = 2U(n/2) + \Theta(n)$ lucky
$U(n) = L(n-1) + \Theta(n)$ unlucky
we consequently get

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$





An Average Case Scenario (2)

- How can we make sure that we are usually lucky?
  - Partition around the ”middle” (n/2th) element?
  - Partition around a random element (works well in practice)
- Randomized algorithm
  - running time is independent of the input ordering
  - no specific input triggers worst-case behavior
  - the worst-case is only determined by the output of the random-number generator

Randomized Quicksort

- Assume all elements are distinct
- Partition around a random element
- Consequently, all splits (1:n-1, 2:n-2, ..., n-1:1) are equally likely with probability 1/n

- Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity

Randomized Quicksort (2)

```
Randomized-Partition(A,p,r)
i←Random(p,r)
exchange A[r]↔A[i]
return Partition(A,p,r)
```

```
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)
```