CSE 3101: Introduction to the Design and Analysis of Algorithms

Instructor: Suprakash Datta (datta[at]cse.yorku.ca) ext 77875

Lectures: Tues, BC 215, 7–10 PM

Office hours: Wed 4-6 pm (CSEB 3043), or by appointment.

Textbook: Cormen, Leiserson, Rivest, Stein. Introduction to Algorithms (3rd Edition)

Next…

1. Covered basics of a simple design technique (Divide-and-conquer) – Ch. 2 of the text.
2. Next, more sorting algorithms.

Sorting

Switch from design paradigms to applications.
Sorting and order statistics (Ch 6 – 9).

First:

Heapsort

–Heap *data structure* and priority queue *ADT*

Quicksort

–a popular algorithm, very fast on average

Why Sorting?

"When in doubt, sort" – one of the principles of algorithm design. Sorting used as a subroutine in many of the algorithms:

- Searching in databases: we can do binary search on sorted data
- A large number of computer graphics and computational geometry problems
- Closest pair, element uniqueness

- A large number of sorting algorithms are developed representing different algorithm design techniques.
- A lower bound for sorting $\Omega(n \log n)$ is used to prove lower bounds of other problems.

Sorting algorithms so far

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

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

- 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
- Two attributes
  - length[*A*]
  - heap-size[*A*]

Heap sort

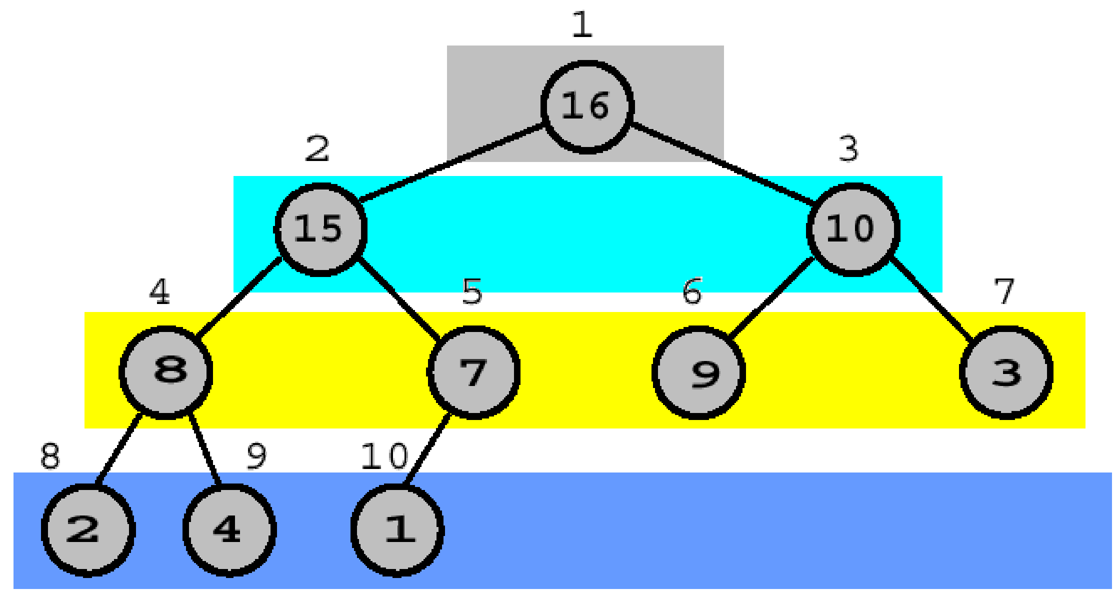
Parent (*i*)
return $\lfloor i/2 \rfloor$

Left (*i*)
return 2*i*

Right (*i*)
return 2*i*+1

Heap property:

$A[\text{Parent}(i)] \geq A[i]$



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

Heap sort

- Notice the implicit tree links; children of node *i* are 2*i* and 2*i*+1
- 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

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

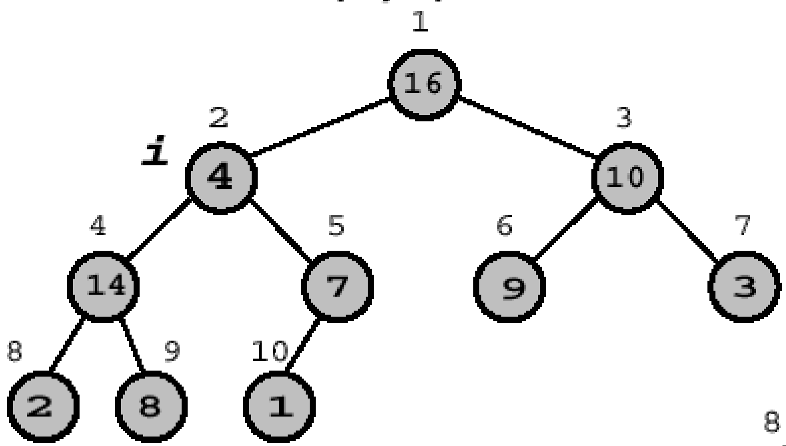
Heapify

n is total number of elements

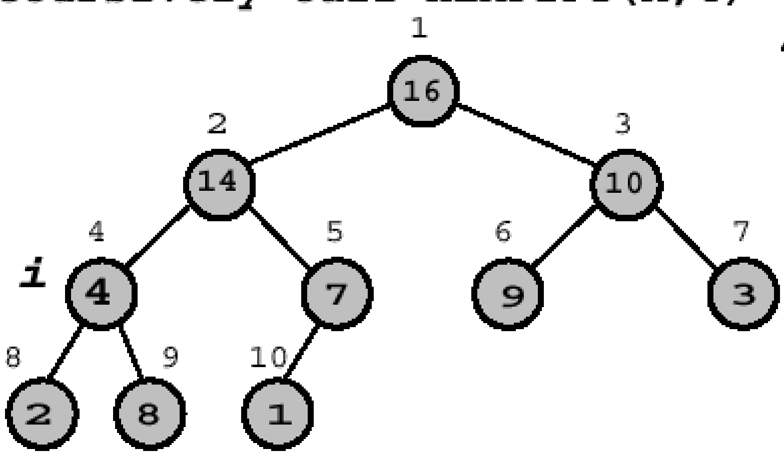
```
HEAPIFY(A, i)
  ▷ Left & Right subtrees of i are heaps.
  ▷ Makes subtree rooted at i a heap.
  l ← LEFT(i)          ▷ l = 2i
  r ← RIGHT(i)         ▷ r = 2i + 1
  if l ≤ n and A[l] > A[i]
     then largest ← l
     else largest ← i
  if r ≤ n and A[r] > A[largest]
     then largest ← r
  if largest ≠ i
     then exchange A[i] ↔ A[largest]
          HEAPIFY(A, largest)
```

Heapify Example

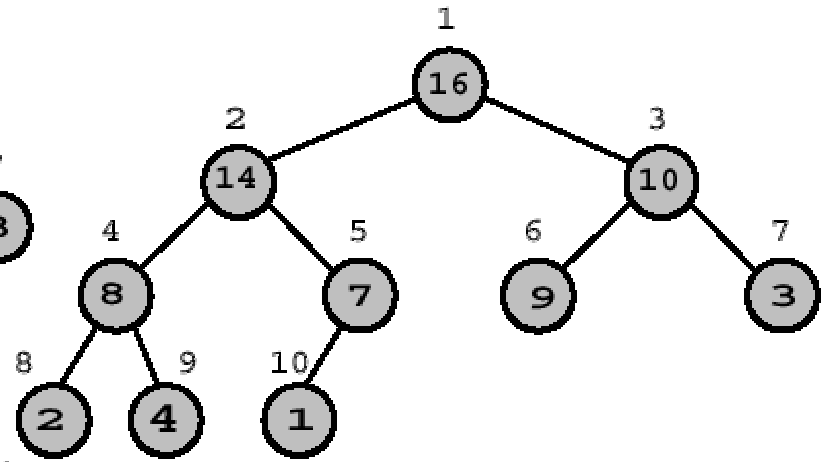
1. Call HEAPIFY(A,2)



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: Running time

- The running time of Heapify on a subtree of size n rooted at node i is
  - determining the relationship between elements: $\Theta(1)$
  - plus the time to run Heapify on a subtree rooted at one of the children of i, where 2n/3 is the worst-case size of this subtree.
  - Alternatively
    - Running time on a node of height h: O(h)

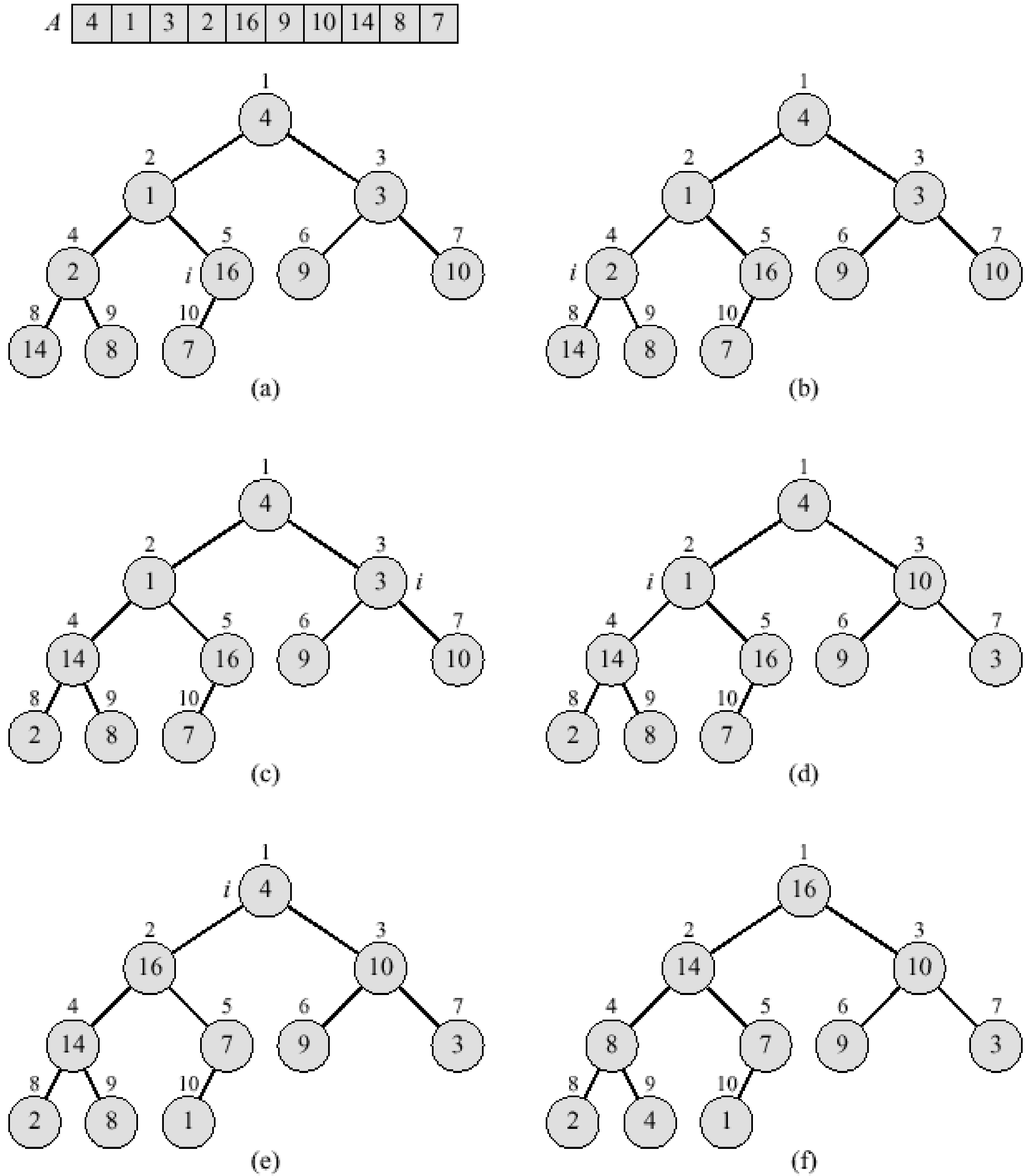
$$T(n) \leq T(2n/3) + \Theta(1) \quad \Rightarrow \quad T(n) = O(\log n)$$

Building a Heap

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

$$
\begin{array}{l}
\textsc{Build-Heap}(A) \\
\quad 1\ \textbf{for}\ i \leftarrow \lfloor n/2 \rfloor\ \textbf{downto}\ 1 \\
\quad 2\ \qquad \textbf{do}\ \textsc{Heapify}(A, i)
\end{array}
$$

Building a heap



Building a Heap: Analysis

- Correctness: induction on i, all trees rooted at m > i are heaps
- Running time: less than 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

```
BUILD-HEAP(A)
1 for i ← ⌊n/2⌋ downto 1
2      do HEAPIFY(A, i)
```

  - time to Heapify = O(height of subtree rooted at i)
  - assume n = $2^k - 1$ (a complete binary tree k = $\lfloor lg\ n \rfloor$)

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

Building a Heap: Analysis (3)

- How? By using the following "trick"

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

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

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

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

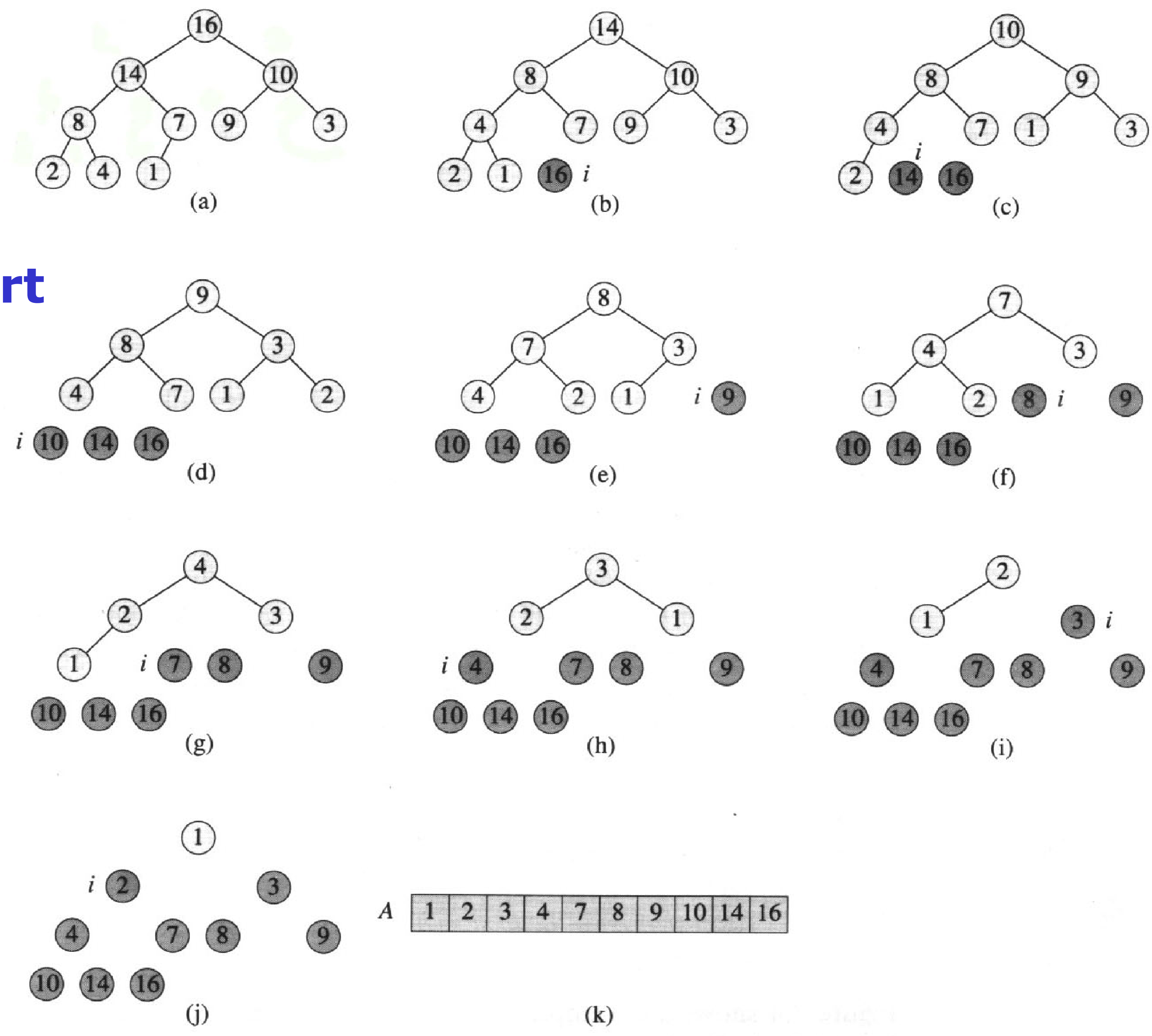
- Therefore Build-Heap time is O(n)

Heap sort

HEAPSORT(A)	**Analysis**
BUILD-HEAP(A)	?? O(n)
for $i \leftarrow n$ **downto** 2	n times
do exchange $A[1] \leftrightarrow A[i]$	O(1)
$n \leftarrow n - 1$	O(1)
HEAPIFY($A, 1$)	O($\lg n$)

The total running time of heap sort is O(n lg n) + Build-Heap(A) time, which is O(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

Priority Queues (2)

- 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(3)

- Removal of max takes constant time on top of Heapify $\Theta(\lg n)$

```
HEAP-EXTRACT-MAX(A)
  ▷ Removes and returns largest element of A
  max ← A[1]
  A[1] ← A[n]
  n ← n − 1
  HEAPIFY(A, 1)              ▷ Remakes heap
  return max
```

Priority Queues(4)

- 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: $\Theta(\lg n)$

$\text{Heap-Insert}(A, key)$

$heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$

$i \leftarrow heap\text{-}size[A]$

while $i > 1$ and $A[\text{Parent}(i)] < key$

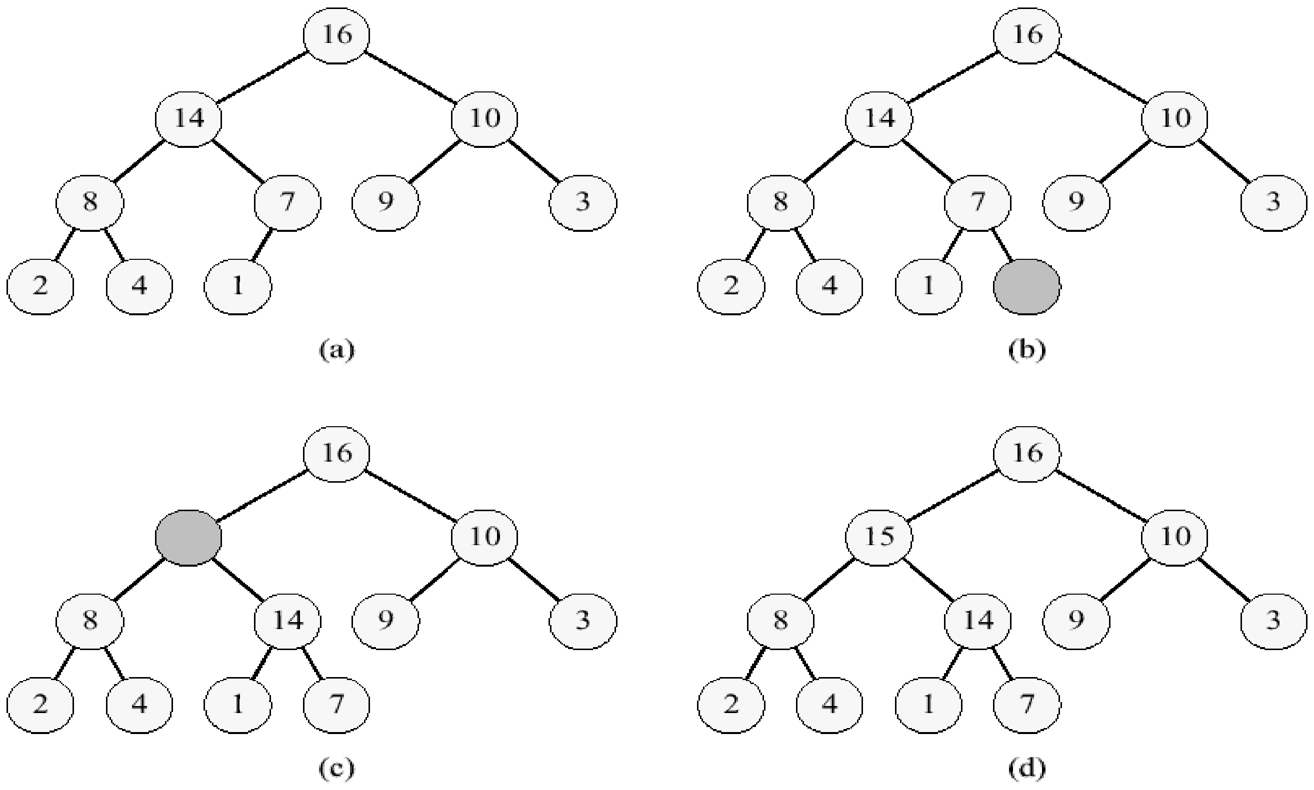
  do $A[i] \leftarrow A[\text{Parent}(i)]$

    $i \leftarrow \text{Parent}(i)$

$A[i] \leftarrow key$

Priority Queues(5)

Insert a new element: 15



Quick Sort

- Characteristics
  - sorts ”almost” in place, i.e., does not require an additional array, like insertion sort
  - Divide-and-conquer, like merge sort
  - very practical, average sort performance O(n log n) (with small constant factors), but worst case O(n^2) [CAVEAT: this is true for the CLRS version]

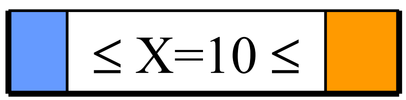
Quick Sort – the main idea

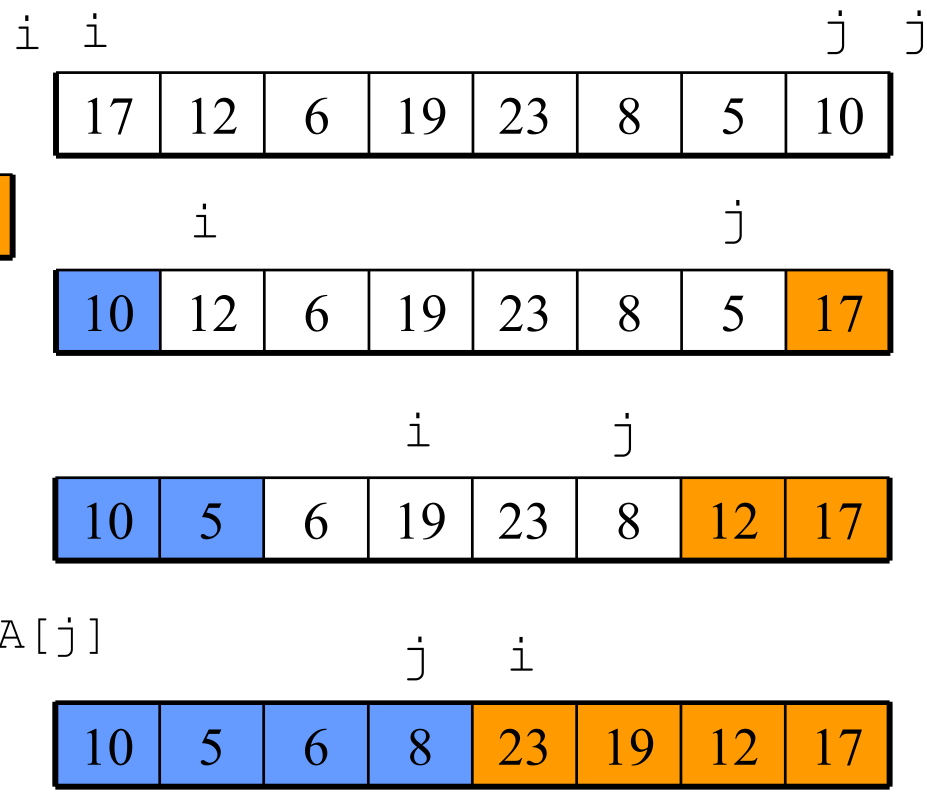
- To understand quick-sort, let's look at a high-level description of the algorithm
- A divide-and-conquer algorithm
  - Divide: partition array into 2 subarrays such that elements in the lower part <= elements in the higher part
  - Conquer: recursively sort the 2 subarrays
  - Combine: trivial since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

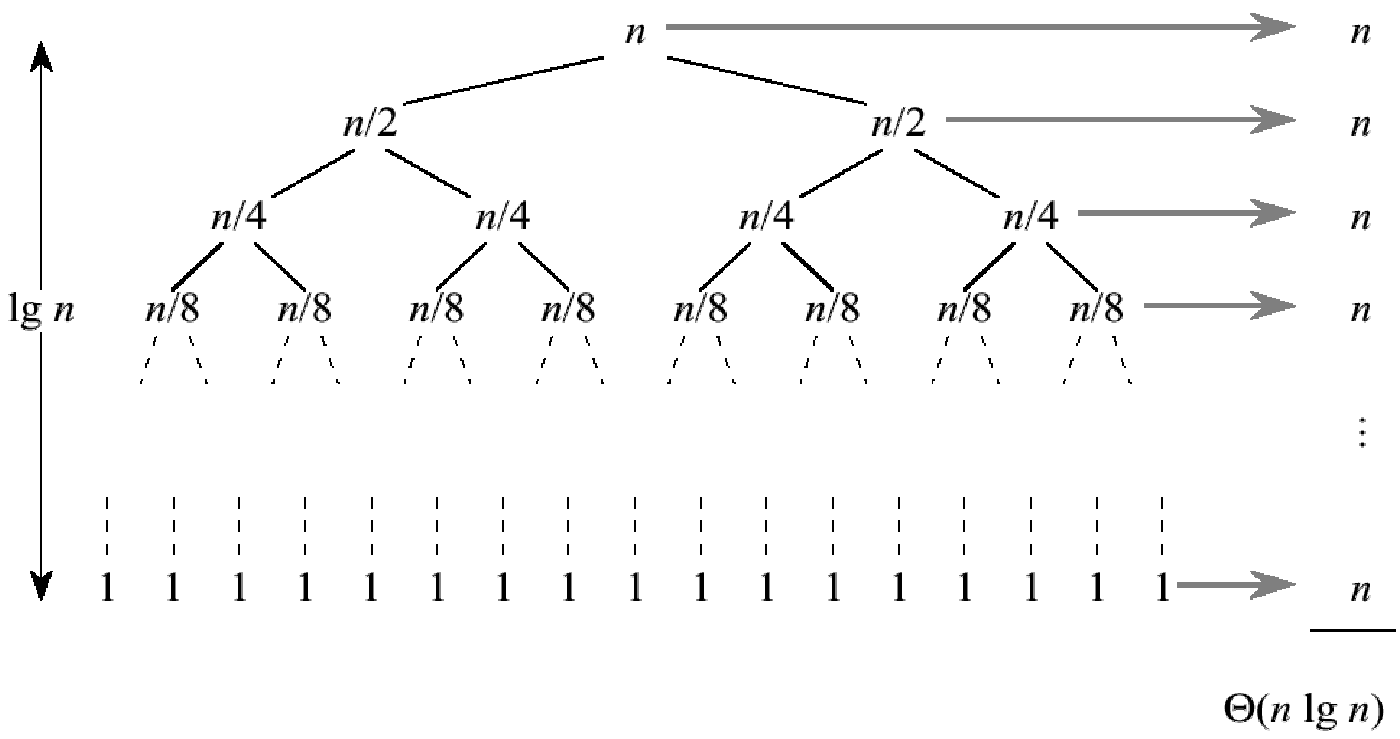
```
Quicksort(A,p,r)
if p<r
    then q←Partition(A,p,r)
         Quicksort(A,p,q)
         Quicksort(A,q+1,r)
```

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



Using the median as a pivot

- The recurrence in the previous slide works out, BUT……

Q: Can we find the median in linear-time?

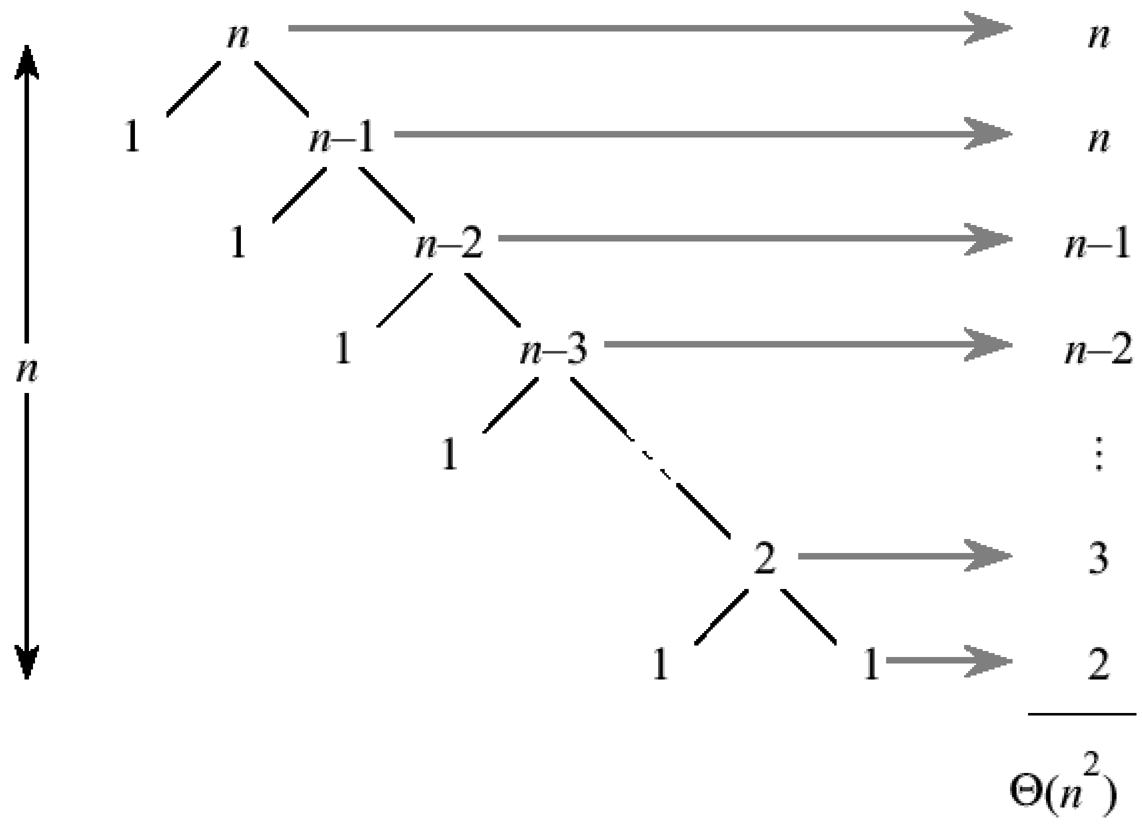
A: YES! But we need to wait until we get to Chapter 8…..

Worst Case

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

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

Worst Case (2)



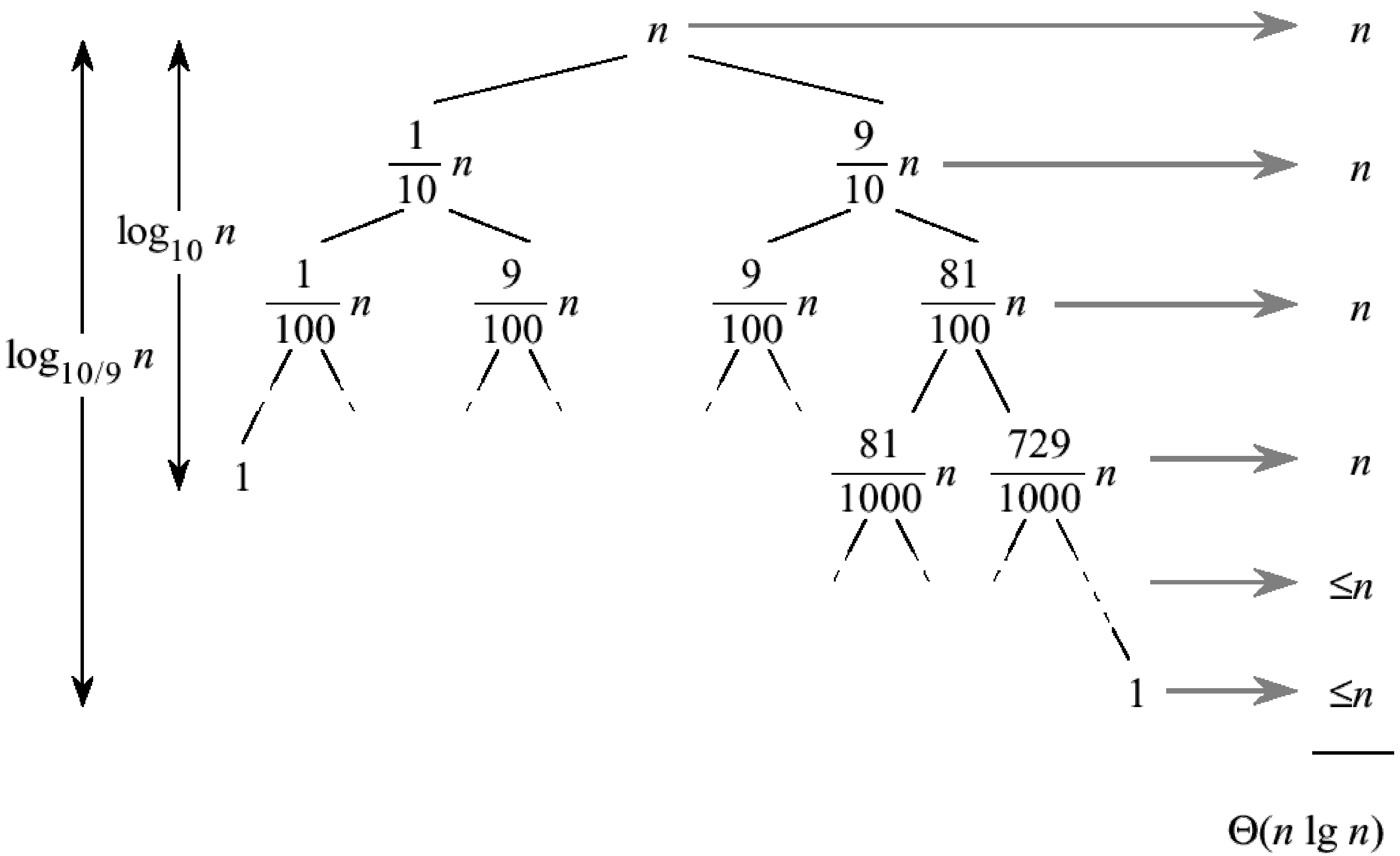
Worst Case (3)

- 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

$$L(n) = 2U(n/2) + \Theta(n) \quad \text{lucky}$$

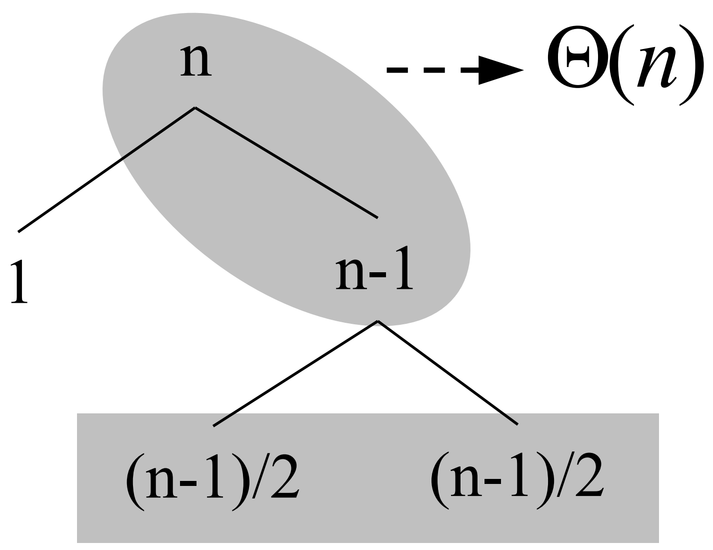
$$U(n) = L(n-1) + \Theta(n) \quad \text{unlucky}$$

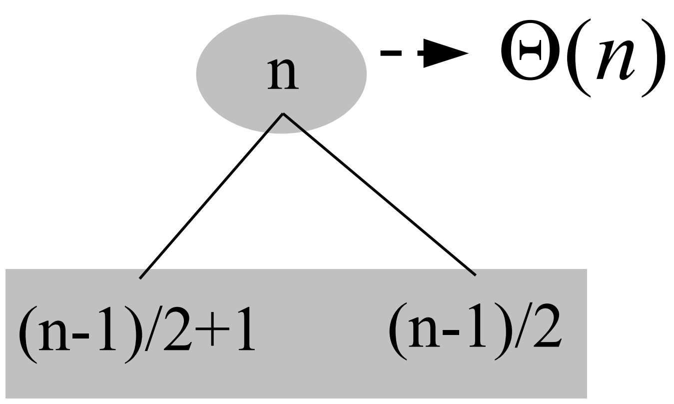
we consequently get

$$L(n) = 2(L(n/2-1) + \Theta(n/2)) + \Theta(n)$$

$$= \quad 2L(n/2-1) + \Theta(n)$$

$$= \quad \Theta(n \log n)$$





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

Next: Lower bounds

Q: Can we beat the Ω(n log n) lower bound for sorting?

A: In general no, but in some special cases YES!

Ch 7: Sorting in linear time

Let's prove the Ω(n log n) lower bound.

Lower bounds

- What are we counting?

  Running time? Memory? Number of times a specific operation is used?
- What (if any) are the assumptions?
- Is the model general enough?

Here we are interested in lower bounds for the WORST CASE. So we will prove (directly or indirectly):

for any algorithm for a given problem, for each $n>0$, there exists an input that make the algorithm take $\Omega(f(n))$ time. Then $f(n)$ is a lower bound on the worst case running time.

Comparison-based algorithms

Finished looking at comparison-based sorts.

Crucial observation: All the sorts work for any set of elements – numbers, records, objects,……

Only require a comparator for two elements.

```
#include <stdlib.h>
```

void qsort(void *base, size_t *nmemb*, size_t *size*, int(**compar*)(const void *, const void *));

DESCRIPTION: The qsort() function sorts an array with *nmemb* elements of *size* size. The base argument points to the start of the array.

The contents of the array are sorted in ascending order according to a comparison function pointed to by *compar*, which is called with two arguments that point to the objects being compared.

Comparison-based algorithms

- The algorithm only uses the results of comparisons, not values of elements (*).
- Very general – does not assume much about what type of data is being sorted.
- However, other kinds of algorithms are possible!
- In this model, it is reasonable to count #comparisons.
- Note that the #comparisons is a **lower bound** on the running time of an algorithm.

(*) If values are used, lower bounds proved in this model are not lower bounds on the running time.

Lower bound for a simpler problem

Let's start with a simple problem.

Minimum of n numbers

Minimum (A)

1. min = A[1]
2. for i = 2 to length[A]
3.   do if min >= A[i]
4.   then min = A[i]
5. return min

Can we do this with fewer comparisons?

We have seen very different algorithms for this problem. How can we show that we cannot do better by being smarter?

Lower bounds for the minimum

Claim: Any comparison-based algorithm for finding the minimum of n keys must use at least n-1 comparisons.

Proof: If x,y are compared and x > y, call x the winner.

Any key that is not the minimum must have won at least one comparison. WHY?

Each comparison produces exactly one winner and at most one NEW winner.

$\Rightarrow$ at least n-1 comparisons have to be made.

Points to note

Crucial observations: We proved a claim about ANY algorithm that only uses comparisons to find the minimum. Specifically, we made no assumptions about

1. Nature of algorithm.
2. Order or number of comparisons.
3. Optimality of algorithm
4. Whether the algorithm is reasonable – e.g. it could be a very wasteful algorithm, repeating the same comparisons.

On lower bound techniques

Unfortunate facts:

Lower bounds are usually hard to prove.

Virtually no known general techniques – must try ad hoc methods for each problem.

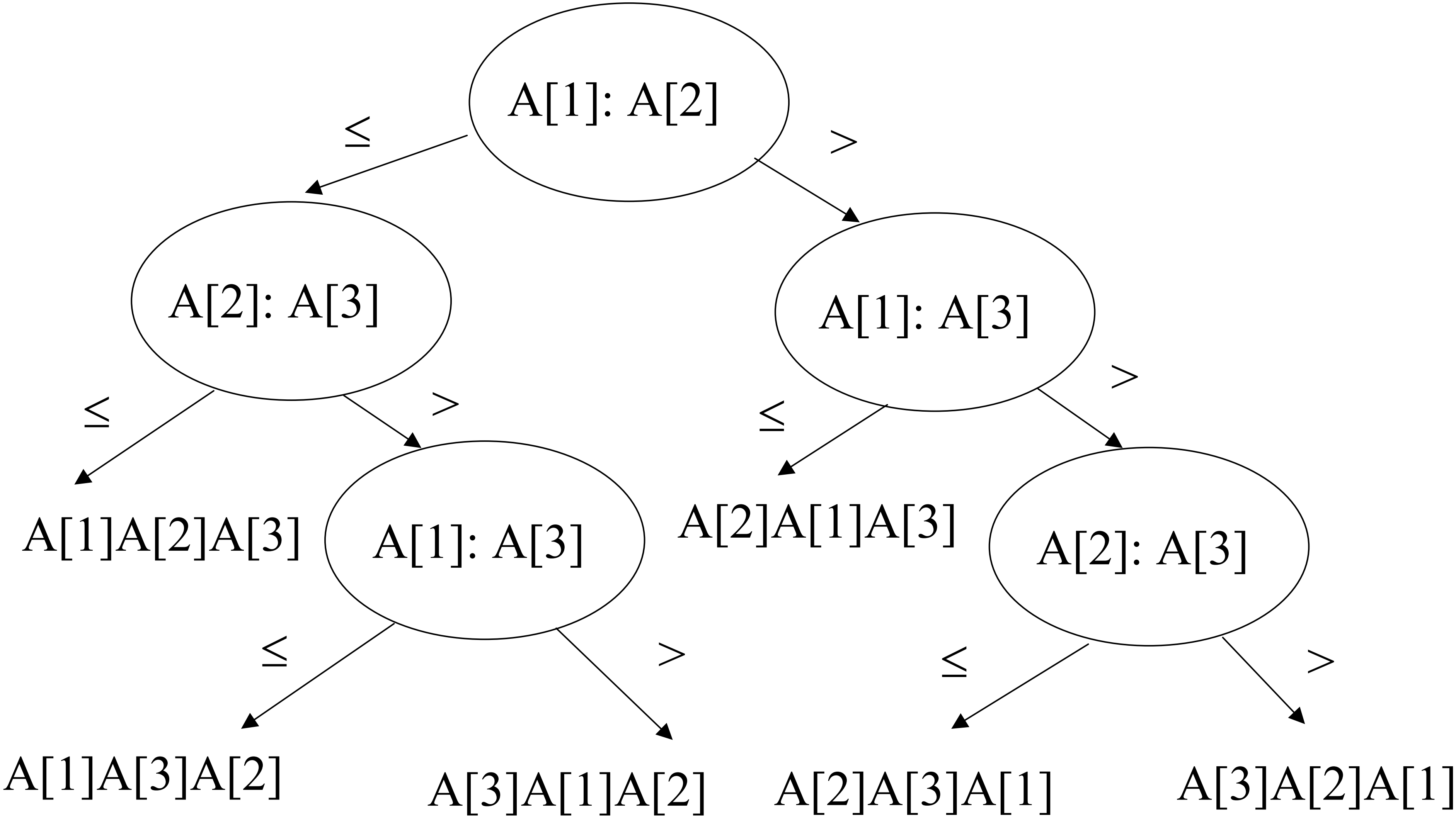
Lower bounds for comparison-based sorting

- Trivial: $\Omega(n)$ – every element must take part in a comparison.
- Best possible result – $\Omega(n \log n)$ comparisons, since we already know several O(n log n) sorting algorithms.
- Proof is non-trivial: how do we reason about all possible comparison-based sorting algorithms?

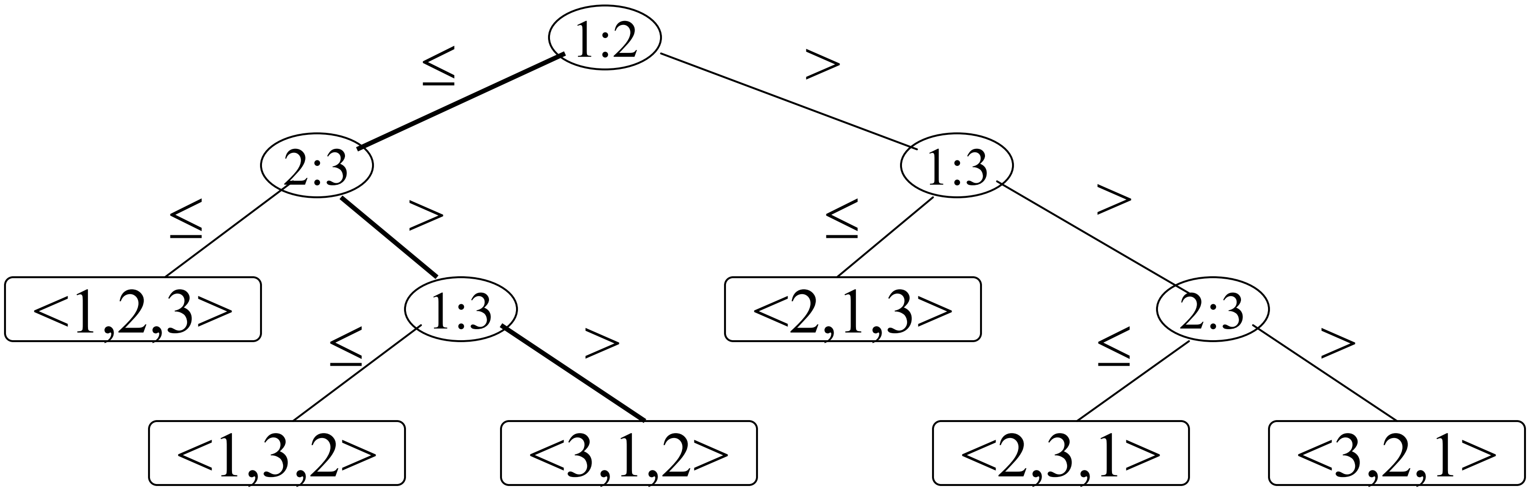
The Decision Tree Model

- Assumptions:
  - All numbers are distinct (so no use for $a_i = a_j$)
  - All comparisons have form $a_i \leq a_j$ (since $a_i \leq a_j$, $a_i \geq a_j$, $a_i < a_j$, $a_i > a_j$ are equivalent).
- Decision tree model
  - Full binary tree
  - Ignore control, movement, and all other operations, just use comparisons.
  - suppose three elements $< a_1, a_2, a_3>$ with instance <6,8,5>.

Example: insertion sort (n=3)



The Decision Tree Model



Internal node i:j indicates comparison between a_i and a_j.
Leaf node $<\pi(1), \pi(2), \pi(3)>$ indicates ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq a_{\pi(3)}$.
Path of bold lines indicates sorting path for <6,8,5>.
There are total 3!=6 possible permutations (paths).

Summary

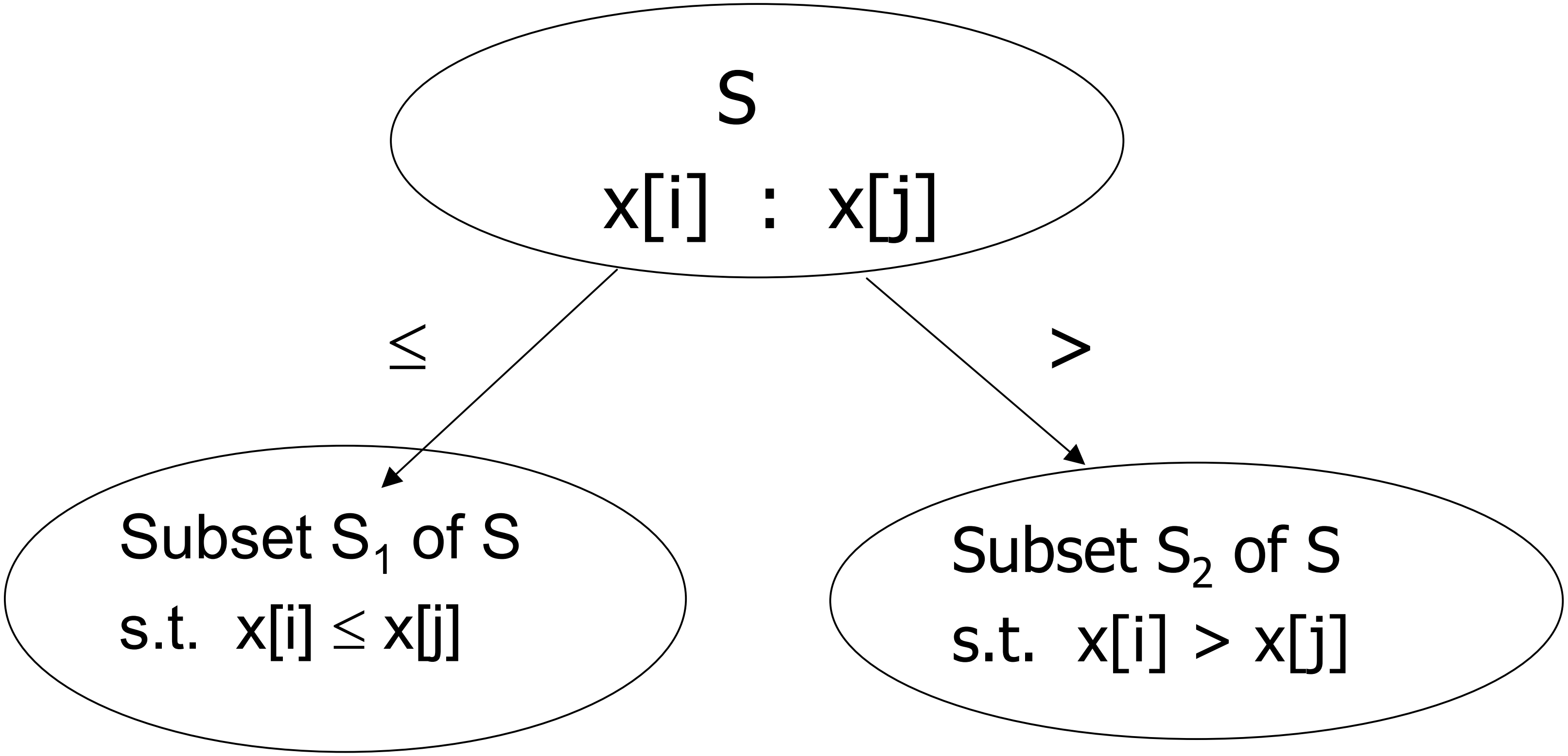
- Only consider comparisons
- Each internal node = 1 comparison
- Start at root, make the first comparison

  - if the outcome is $\leq$ take the LEFT branch

  - if the outcome is > - take the RIGHT branch
- Repeat at each internal node
- Each LEAF represents ONE correct ordering

Intuitive idea

S is a set of permutations



Lower bound for the worst case

- Claim: The decision tree must have at least n! leaves. WHY?
- worst case number of comparisons= the height of the decision tree.
- Claim: Any comparison sort in the worst case needs Ω(n log n) comparisons.
- Suppose height of a decision tree is h, number of paths (i,e,, permutations) is n!.
- Since a binary tree of height h has at most 2^h leaves,

$$n! \le 2^h \text{ , so } h \ge \lg (n!) \ge \Omega(n \lg n)$$

Lower bounds: check your understanding

Can you prove that any algorithm that searches for an element in a sorted array of size n must have running time $\Omega(\lg n)$?