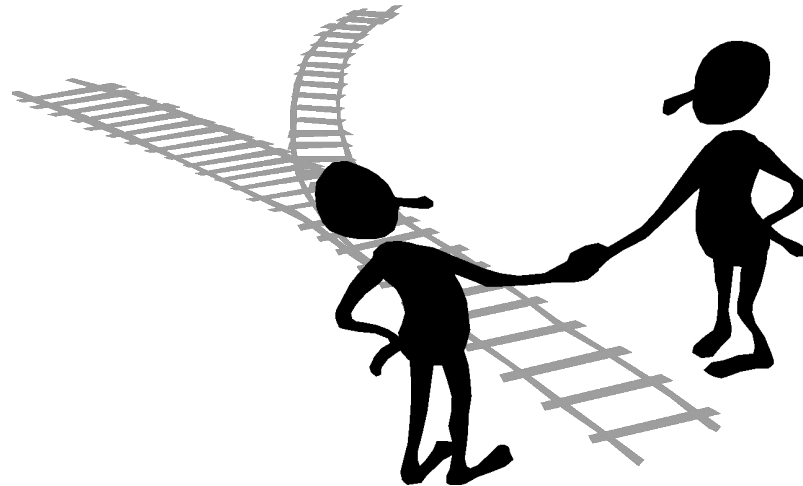


Computer Algorithms

Segment 2

Divide and Conquer Approach Sorting & Heaps



Divide and Conquer

- A common approach to solving a problem is to **partition** the problem into smaller parts, find solutions for the parts, and then combine the solutions for the parts into a solution for the whole.
- This approach, especially when used recursively, often **yields efficient solutions to problems** in which the sub-problems are smaller versions of the original
- ~~problem~~ divide the problem into a number of subproblems
- **conquer** the subproblems (solve them)
- **combine** the subproblem solutions to get the solution to the original problem
- Note: often the “**conquer**” step is done recursively

Recursive algorithm

- To solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems.
 - usually the subproblems are smaller in size than the 'parent' problem
 - divide-and-conquer algorithms are often recursive

Algorithm for General Divide and Conquer Sorting

//Algorithm for General Divide and Conquer Sorting

Begin Algorithm

Start Sort(L)

If L has length greater than 1 then

Begin

Partition the list into two lists, high and low

Start Sort(high)

Start Sort(low)

Combine high and low

End

End Algorithm

Analyzing Divide-and-Conquer Algorithms

- When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence equation which describes the overall running time on a problem of size n in terms of the running time on smaller inputs.
- For divide-and-conquer algorithms, we get recurrences that look like

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Analyzing Divide-and-Conquer Algorithms

where

a is the number of subproblems we break the problem into

n/b is the size of the subproblems (in terms of n)

$D(n)$ is the time to divide the problem of size n into the subproblems

$C(n)$ is the time to combine the subproblem solutions to get the answer for the problem of size n


Recurrence Relations

- A recurrence relation is an equation which is defined in terms of itself.
- Why are recurrences good things?
 - Many natural functions are easily expressed as recurrences:
 - $a_n = a_{n-1} + 1, a_1 = 1 \rightarrow a_n = n$ (polynomial)
 - $a_n = 2a_{n-1}, a_1 = 1 \rightarrow a_n = 2^n$ (exponential)
 - $a_n = na_{n-1}, a_1 = 1 \rightarrow a_n = n!$ (weird function)
- It is often easy to find a recurrence as the solution of a counting problem

Recurrence Equations

- A recurrence equation defines a function, say $T(n)$. The function is defined recursively, that is, the function $T(\cdot)$ appear in its definition. (recall recursive function call). The recurrence equation should has a base case.
- For example:

$$T(n) = \begin{cases} T(n-1)+T(n-2), & \text{if } n > 1 \\ 1, & \text{if } n=1 \text{ or } n=0. \end{cases}$$

base case 

for convenient, we sometime write the recurrence equation

as: $T(n) = T(n-1)+T(n-2)$

$$T(0) = T(1) = 1.$$

More Recurrence equations

$$T(n) = 2 * T(n/2) + 1,$$

$$T(1) = 1.$$

Base case;
initial condition.

$$T(n) = T(n-1) + n,$$

$$T(1) = 1.$$

Selection Sort

$$T(n) = 2 * T(n/2) + n,$$

$$T(1) = 1.$$

Merge Sort

Quick Sort

$$T(n) = 2 * T(n/2) + \log n,$$

$$T(1) = 1.$$

Heap Construction

$$T(n) = T(n/2) + 1,$$

$$T(1) = 0.$$

Binary search

Example

- As we will see, induction provides a useful tool to solve recurrences - guess a solution and prove it by induction.
- $T_n = 2T_{n-1} + 1, T_0 = 0$

n	0	1	2	3	4	5	6	7
Tn	0	1	3	7	15	31	63	127

- Guess what the solution is?

Solve Recurrence Relation by Induction Method

- Prove $T_n = 2^n - 1$ by induction:
 - Show that the basis is true: $T_0 = 2^0 - 1$
 - Now assume true for T_{n-1}
 - Using this assumption show:
- $T_n = 2T_{n-1} + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 1$

Solving Recurrences

There are 4 general methods for solving recurrences

1. **Substitution:** Guess & Verify: guess a solution and verify it is correct with an inductive proof
2. **Iteration:** Convert to Summation: convert the recurrence into a summation (by expanding some terms) and then bound the summation.
3. **Recursion-tree method**
4. Apply “**Master Theorem**”: if the recurrence has the form

$$T(n) = aT(n/b) + f(n)$$

then there is a formula that can (often) be applied.

Solving Recurrences: Substitution

(guess and verify)

- This method involves guessing form of solution
- Use mathematical induction to find the constants and verify solution
- Use to find an upper or a lower bound (do both to obtain a tight bound)

Solving Recurrences: Substitution (guess and verify)

Example: $T(n) = 4T(n/2) + n$ (upper bound)

guess $T(n) = O(n^3)$ and try to show $T(n) \leq cn^3$ for some $c > 0$
(we'll have to find c)

basis ?

assume $T(k) \leq ck^3$ for $k < n$, and prove $T(n) \leq cn^3$

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\leq 4(c(n/2)^3) + n \\ &= c/2 n^3 + n \\ &= cn^3 - (c/2 n^3 - n) \\ &\leq cn^3 \end{aligned}$$

where the last step holds if $c > 2$ and $n > 1$

We find values of c and n_0 by determining when $c/2n^3 - n \geq 0$

More Example: Solving Recurrences by Guessing

- Guess the form of the answer, then use induction to find the constants and show that solution works
- Examples:
 - $T(n) = 2T(n/2) + \Theta(n) \quad \square \quad T(n) = \Theta(n \lg n)$
 - $T(n) = 2T(\lfloor n/2 \rfloor) + n \quad \square \quad T(n) = \Theta(n \lg n)$
 - $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n \quad \square \quad \Theta(n \lg n)$

Solving Recurrences: Iteration (convert to summation)

- Expand the recurrence
- Work some algebra to express as a summation
- Evaluate the summation
- We will show several examples

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

Solving Recurrences: Iteration (Example)

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

- $s(n) =$
 $c + s(n-1)$
 $c + c + s(n-2)$
 $2c + s(n-2)$
 $2c + c + s(n-3)$
 $3c + s(n-3)$

.....
 $kc + s(n-k) = ck + s(n-k)$
- So far for $n \geq k$ we have
 - $s(n) = ck + s(n-k)$
- What if $k = n$?
 - $s(n) = cn + s(0) = cn$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

- $s(n)$

$$= n + s(n-1)$$

$$= n + n-1 + s(n-2)$$

$$= n + n-1 + n-2 + s(n-3)$$

$$= n + n-1 + n-2 + n-3 + s(n-4)$$

$$= \dots$$

$$= n + n-1 + n-2 + n-3 + \dots + n-(k-1) + s(n-k)$$

$$=$$

$$\sum_{i=n-k+1}^n i + s(n-k)$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

- So far for $n \geq k$ we have

$$\sum_{i=n-k+1}^n i + s(n-k)$$

- What if $k = n$?

$$\sum_{i=1}^n i + s(0) = \sum_{i=1}^n i + 0 = n \frac{n+1}{2}$$

- Thus in general

$$s(n) = n \frac{n+1}{2}$$

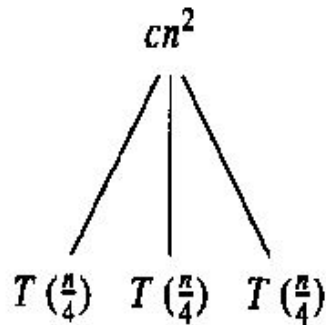
Recurrence Trees

- Allow you to visualize the process of iterating the recurrence
- Allows you make a good guess for the substitution method
- Or to organize the bookkeeping for iterating the recurrence
- Convert the recurrence into a tree:
 - Each node represents the cost of a single subproblem in the set of recursive function invocation.
 - Sum up the costs within each level of the tree to obtain a set of pre-level costs.
 - Then sum all the pre-level costs to determine the total of all levels of the recursion.

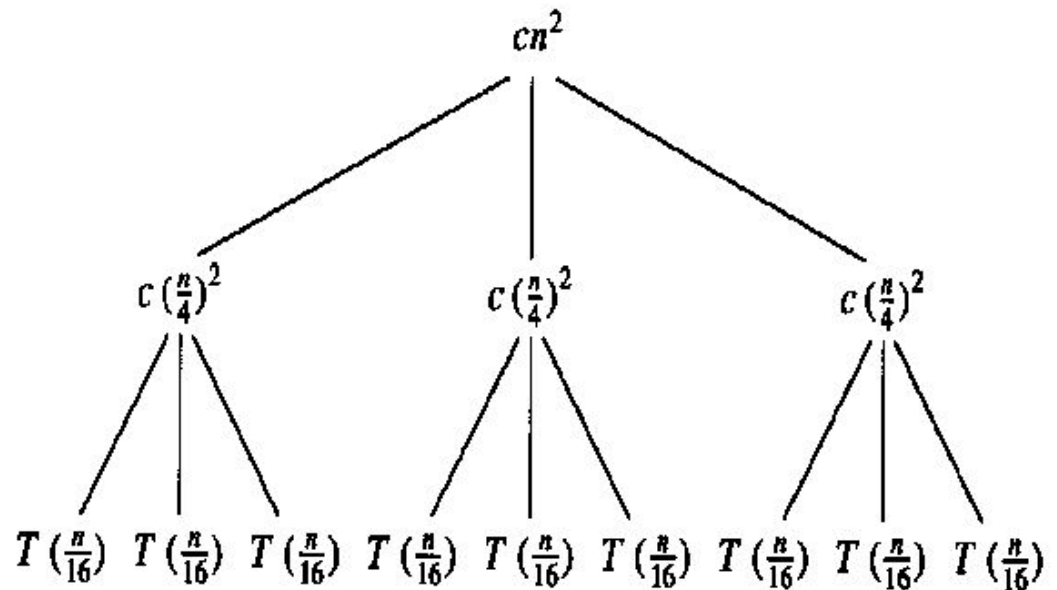
Recursion-tree: Example 1

$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$

$T(n)$

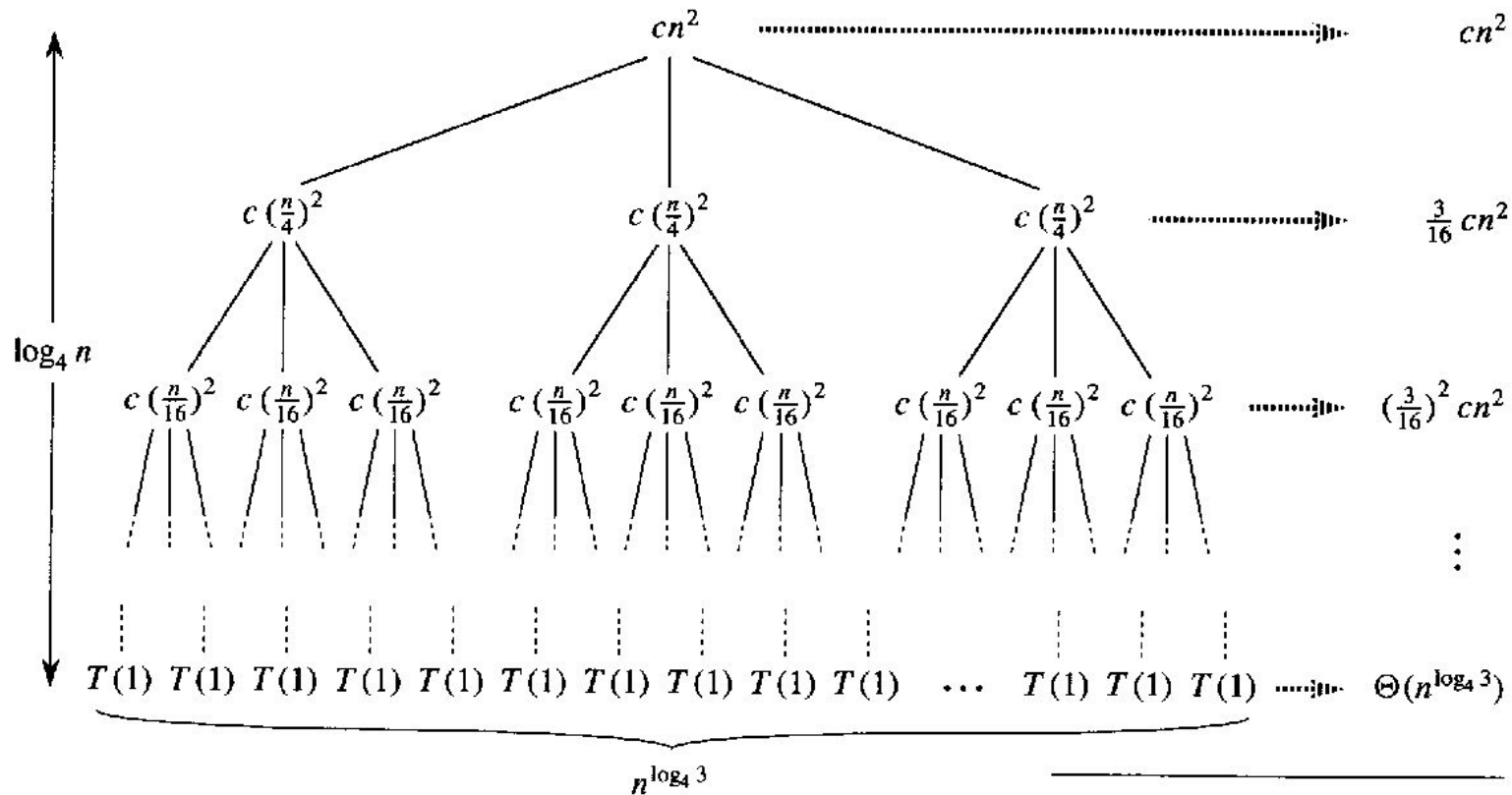


(a)



(c)

Recursion-tree method



(d)

Total: $O(n^2)$

Recursion-tree method

- Subproblem size for a node at depth i , $\frac{n}{4^i}$
- Subproblem size hits $n=1$ when $\frac{n}{4^i}=1$ or when $i = \log_4 n$
- Total level of tree $\log_4 n + 1$ ($0, 1, 2, \dots, \log_4 n$.)
- Number of nodes at depth i , 3^i
- Cost of each node at depth i , $c(\frac{n}{4^i})^2$
- Total cost over all nodes at depth i , $3^i c(\frac{n}{4^i})^2 = (\frac{3}{16})^i cn^2$
- Last level, at depth $\log_4 n$, has $3^{\log_4 n} = n^{\log_4 3}$ nodes

and cost is $n^{\log_4 3} T(1)$, which is, $\Theta(n^{\log_4 3})$

Recursion-tree method

Prove of $3^{\log_4 n} = n^{\log_4 3}$

$$\begin{aligned}\log_4 3^{\log_4 n} &= (\log_4 n)(\log_4 3) \\ &= (\log_4 3)(\log_4 n) \\ &= \log_4 n^{\log_4 3}\end{aligned}$$

We conclude, $3^{\log_4 n} = n^{\log_4 3}$

The cost of the entire tree

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}). \end{aligned}$$

The cost of the entire tree

Backing up one step and applying equation, $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16} \right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$< \sum_{i=0}^{\infty} \left(\frac{3}{16} \right)^i cn^2 + \Theta(n^{\log_4 3})$$

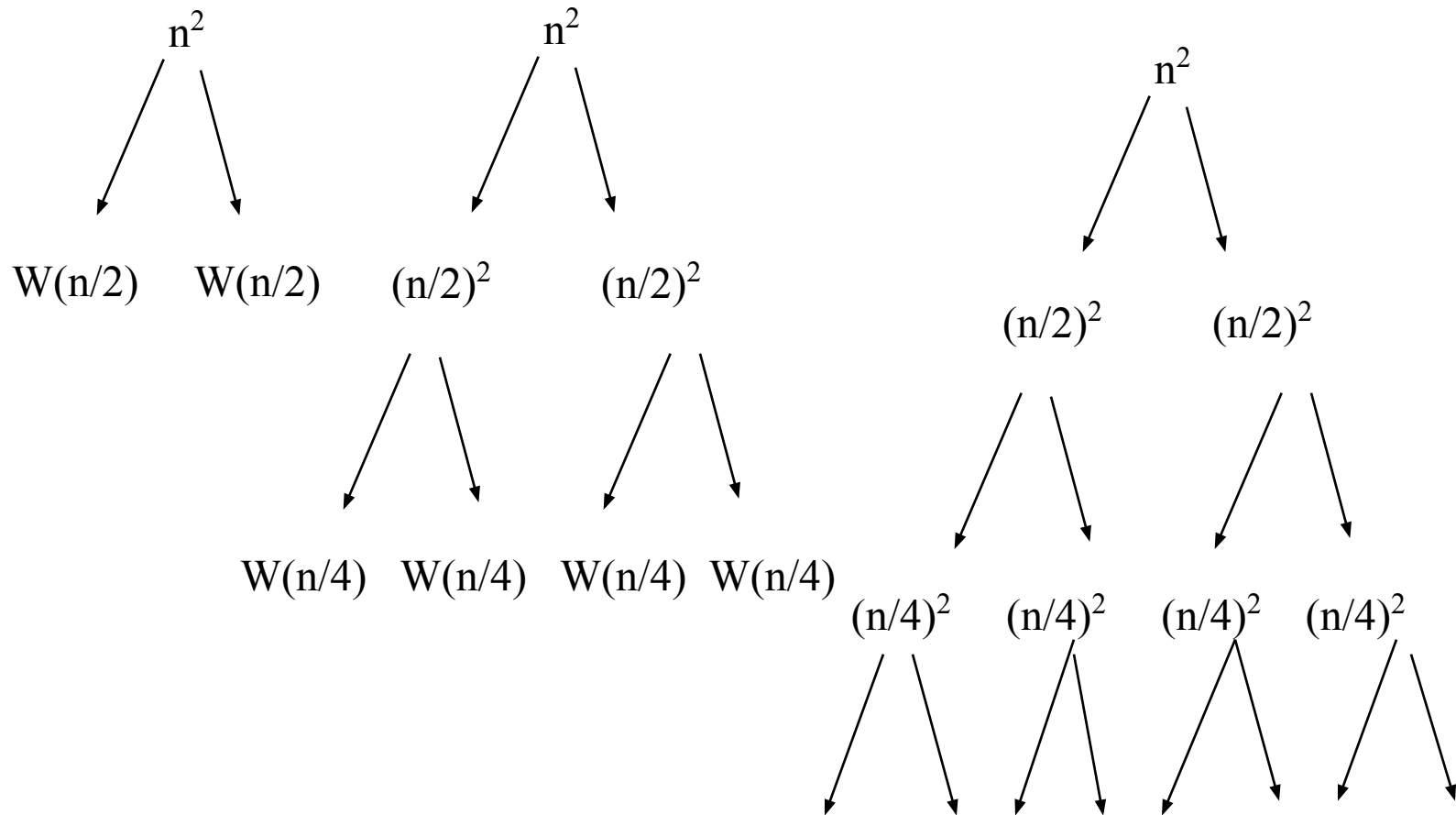
$$= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3})$$

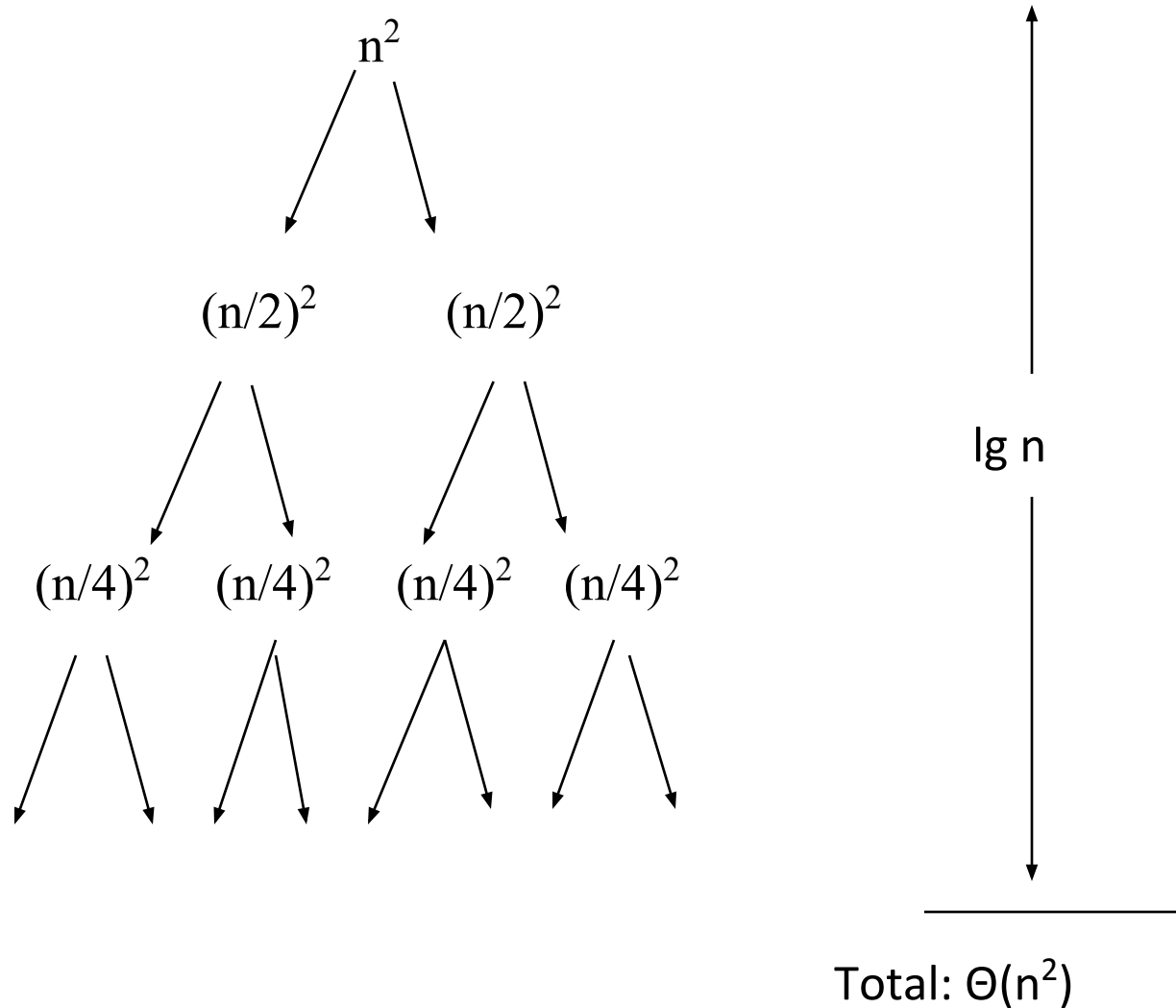
$$= O(n^2)$$

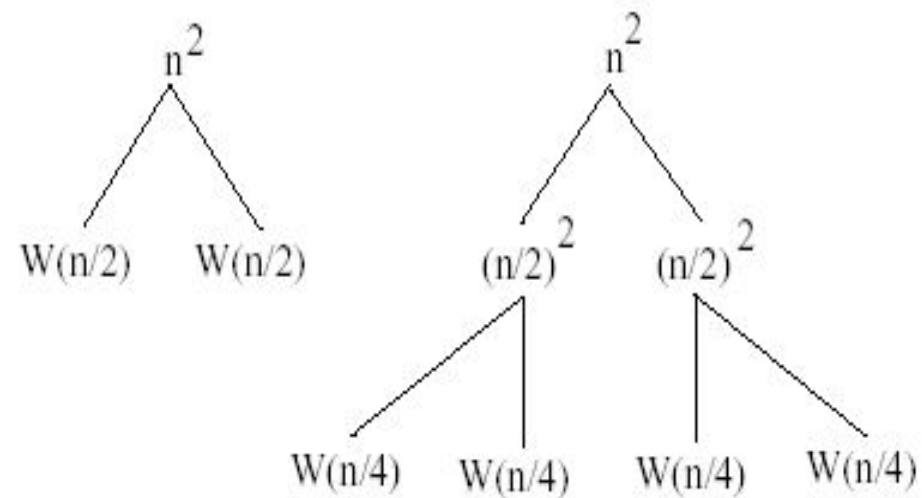
Recurrence Tree: Example 2

$$W(n) = 2W(n/2) + n^2$$



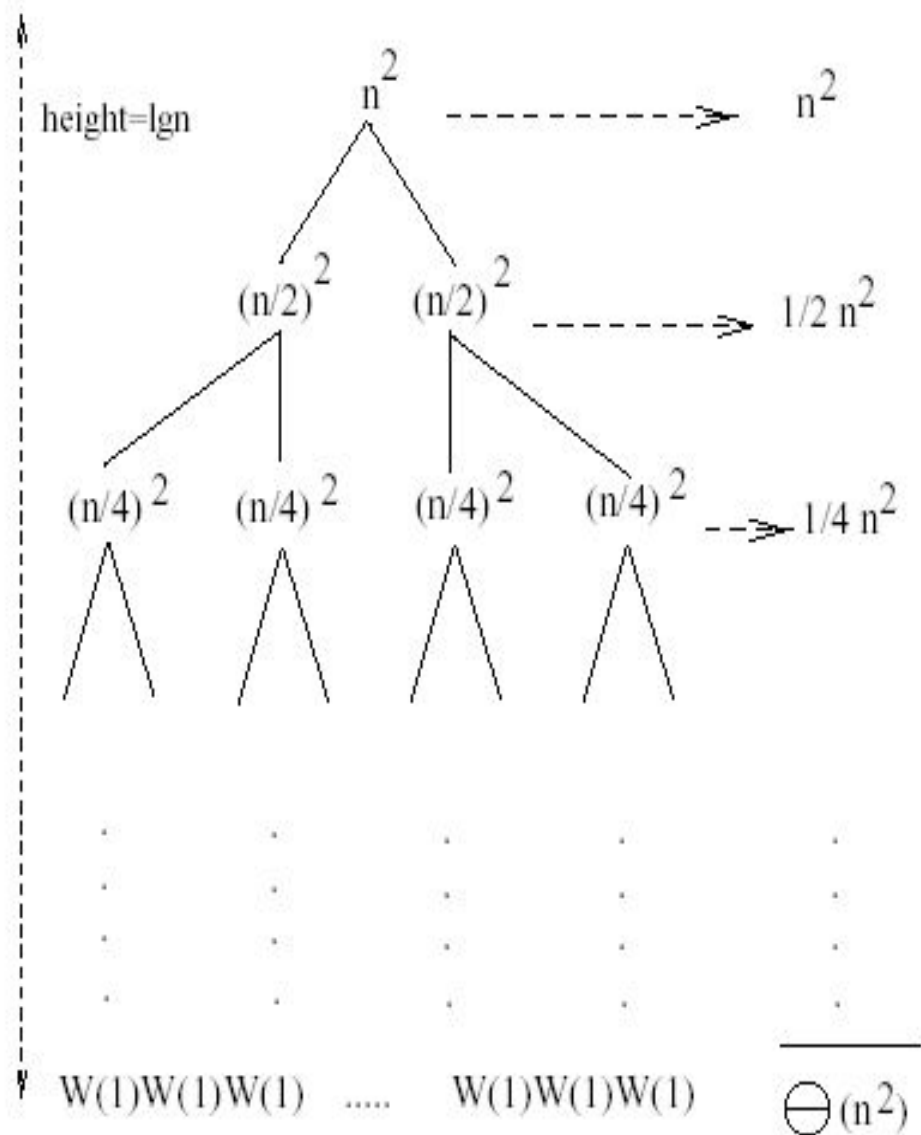
Recurrence Tree Example





$$W(n/2) = 2W(n/4) + (n/2)^2$$

$$W(n/4) = 2W(n/8) + (n/4)^2$$



The cost of the entire tree

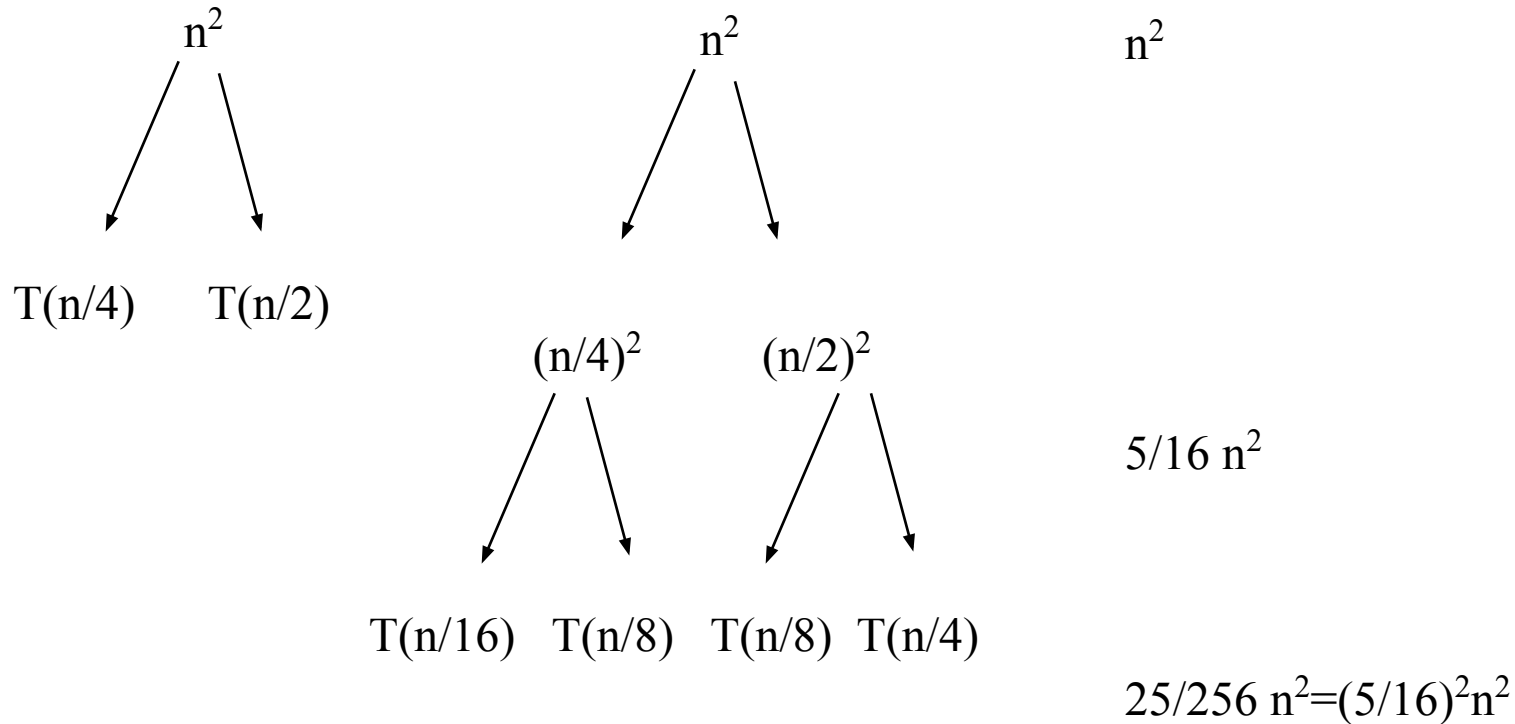
- Subproblem size at level i is: $n/2^i$
- Subproblem size hits 1 when $n/2^i=1 \Rightarrow i = \lg n$
- Total level of tree , $\lg n+1$ (0,1,2,... $\lg n$.)
- Cost of the problem at level $i = (n/2^i)^2$
- No. of nodes at level $i = 2^i$
- Total cost:

$$W(n) = \sum_{i=0}^{\lg n-1} \frac{n^2}{2^i} + 2^{\lg n} W(1) = n^2 \sum_{i=0}^{\lg n-1} \left(\frac{1}{2}\right)^i + n \leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i + O(n) = n^2 \frac{1}{1-\frac{1}{2}} + O(n) = 2n^2$$

$$\Rightarrow W(n) = O(n^2)$$

Example 3

$$T(n) = T(n/4) + T(n/2) + n^2$$

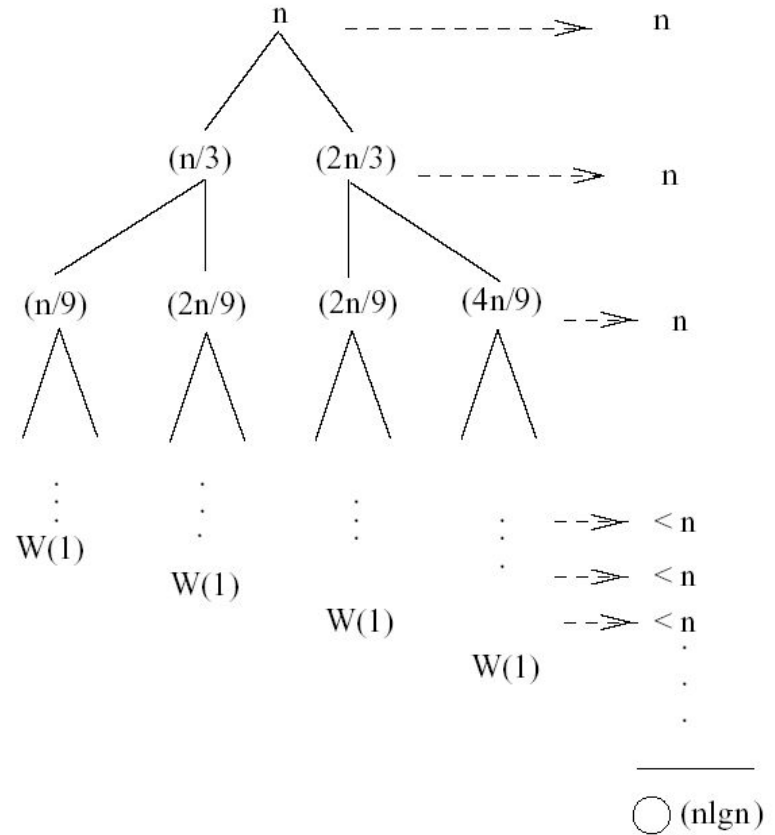


Since the values decrease geometrically, the total is at most a constant factor more than the largest term and hence the solution is $\Theta(n^2)$

Example 4 (simpler proof)

$$W(n) = W(n/3) + W(2n/3) + n$$

- The longest path from the root to a leaf is:
 $n \rightarrow (2/3)n \rightarrow (2/3)^2 n \rightarrow \dots \rightarrow 1$
- Subproblem size hits 1 when $1 = (2/3)^i n \Leftrightarrow i = \log_{3/2} n$
- Cost of the problem at level $i = n$
- Total cost:



$$W(n) < n + n + \dots = n(\log_{3/2} n) = n \frac{\lg n}{\lg \frac{3}{2}} = O(n \lg n)$$

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

The Master Theorem

- Given: a *divide and conquer* algorithm
 - An algorithm that divides the problem of size n into a subproblems, each of size n/b
 - Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function $f(n)$
- Then, the Master Theorem gives us a cookbook for the algorithm's running time:

Solving Recurrences: The Master Method

- **Master Theorem:** Let $a > 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on nonnegative integers as:

$$T(n) = aT(n/b) + f(n),$$

Then, $T(n)$ can be bounded asymptotically as follows:

1. $T(n) = \Theta(n^{\log_b a})$ If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$
2. $T(n) = \Theta(n^{\log_b a} \log n)$ If $f(n) = \Theta(n^{\log_b a})$
3. $T(n) = \Theta(f(n))$ If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n .

The Master Theorem

- if $T(n) = aT(n/b) + f(n)$ then

$$T(n) = \left\{ \begin{array}{ll} \Theta(n^{\log_b a}) & f(n) = O(n^{\log_b a - \varepsilon}) \\ \Theta(n^{\log_b a} \log n) & f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & f(n) = \Omega(n^{\log_b a + \varepsilon}) \text{ AND} \\ & af(n/b) < cf(n) \text{ for large } n \end{array} \right\} \begin{array}{l} \varepsilon > 0 \\ c < 1 \end{array}$$

Case I

Example: $T(n) = 9T(n/3) + n$

- $a = 9, b = 3, f(n) = n, n^{\log_b a} = n^{\log_3 9} = n^2$
- compare $f(n) = n$ with $n^{\log_b a} = n^2$
- $n = O(n^{2-\epsilon})$ ($f(n)$ is polynomially smaller than $n^{\log_b a}$)
- case 1 applies: $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

Case II

Example: $T(n) = T(2n/3) + 1$

- $a = 1, b = 3/2, f(n) = 1, n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$
- compare $f(n) = 1$ with $n^{\log_b a} = 1$
- $1 = \Theta(1)$ ($f(n)$ is asymptotically equal to $n^{\log_b a}$)
- case 2 applies: $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(\log n)$

Case III

Example: $T(n) = 3T(n/4) + n \log n$

- $a = 3, b = 4, f(n) = n \log n, n^{\log_b a} = n^{\log_4 3} = n^{0.793}$
- compare $f(n) = n \log n$ with $n^{\log_b a} = n^{0.793}$
- $n \log n = \Omega(n^{0.793-\epsilon})$ is polynomially larger than $n^{\log_b a}$
- case 3 **might** apply: need to check 'regularity' of $f(n)$
 - find $c < 1$ s.t. $af(n/b) \leq cf(n)$ for large enough n
 - ie. $\frac{3n}{4} \log \frac{n}{4} \leq cn \log n$ Which is true for $c = 3/4$
- case 3 applies: $T(n) = \Theta(f(n)) = \Theta(n \log n)$

Example: Merge Sort

- divide the n -element sequence to be sorted into two $n/2$ element sequences
- conquer: sort the subproblems, recursively using merge sort
- combine: merge the resulting two sorted $n/2$ element sequences

Example: Merge Sort Analysis

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ 2T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

$a = 2$ (two subproblems)

$n/b = n/2$ (each subproblem has size approx $n/2$)

$D(n) = \Theta(1)$ (just compute midpoint of array)

$C(n) = \Theta(n)$ (merging can be done by scanning sorted subarrays)

Merge Sort

- Apply divide-and-conquer to sorting problem
- Problem: Given n elements, sort elements into non-decreasing order
- Divide-and-Conquer:
 - If $n=1$ terminate (every one-element list is already sorted)
 - If $n>1$, partition elements into two or more sub-collections; sort each; combine into a single sorted list
- How do we partition?

Merge Sort

- Idea:
 - Take the array you would like to sort and divide it in half to create 2 unsorted subarrays.
 - Next, sort each of the 2 subarrays.
 - Finally, merge the 2 sorted subarrays into 1 sorted array.
- Efficiency: $O(n \log_2 n)$

Merge Sort

theArray:

8	1	4	3	2
---	---	---	---	---

Divide the array in half

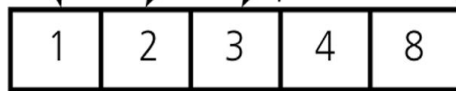


Sort the halves

Merge the halves:

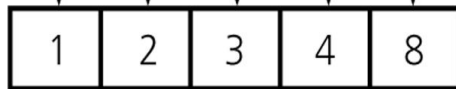
- a. $1 < 2$, so move 1 from left half to tempArray
- b. $4 > 2$, so move 2 from right half to tempArray
- c. $4 > 3$, so move 3 from right half to tempArray
- d. Right half is finished, so move rest of left half to tempArray

Temporary array
tempArray:



Copy temporary array back into
original array

theArray:



Merge Sort

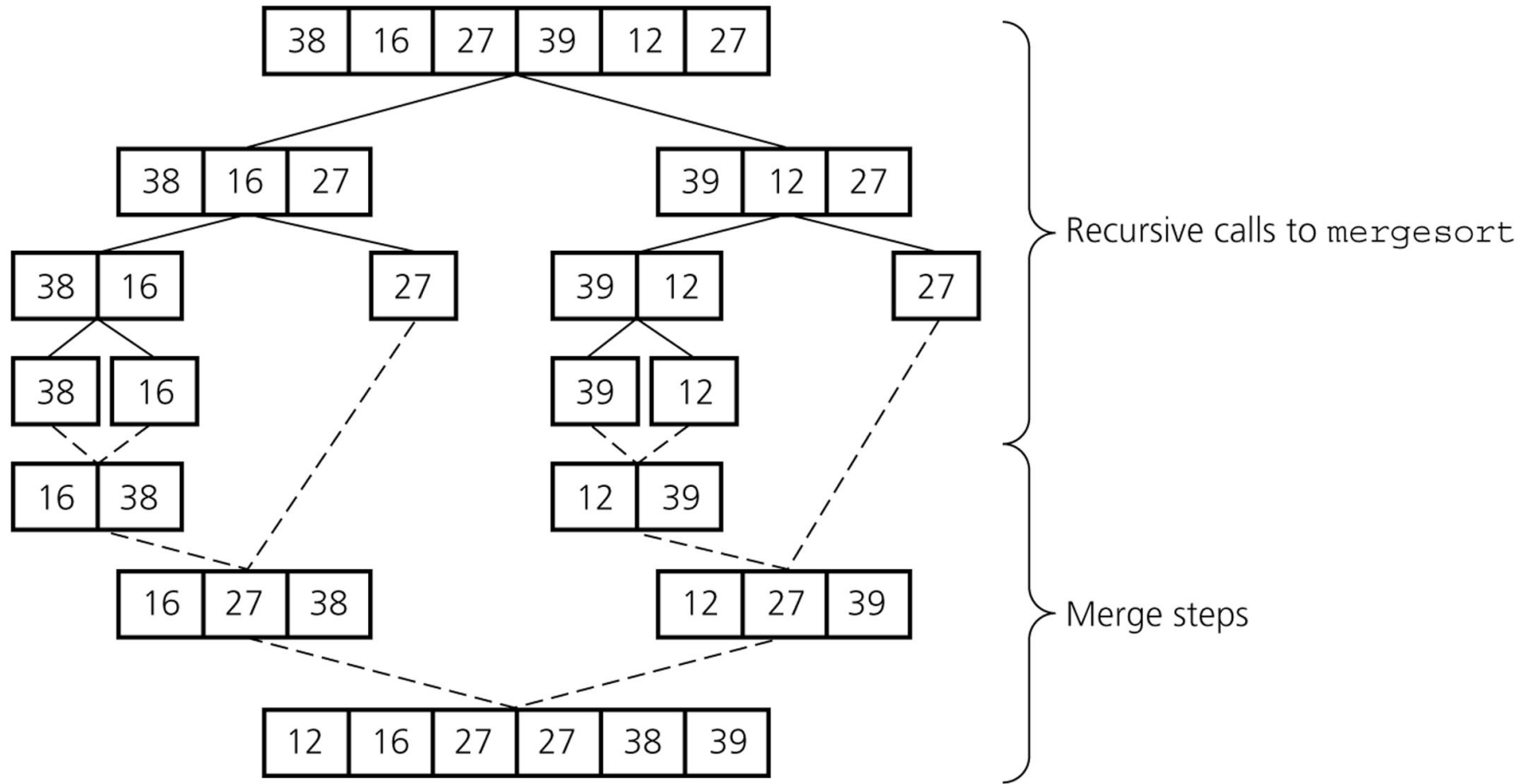
- Although the merge step produces a sorted array, we have overlooked a very important step.
- **How did we sort the 2 halves before performing the merge step?**

We used merge sort!

Merge Sort

- By continually calling the merge sort algorithm, we eventually get a subarray of size 1.
- Since an array with only 1 element is clearly sorted, we can back out and merge 2 arrays of size 1.

Merge Sort



Merge Sort

- The basic merging algorithm consists of:
 - 2 input arrays (arrayA and arrayB)
 - An output array (arrayC)
 - 3 position holders (indexA, indexB, indexC), which are initially set to the beginning of their respective arrays.

Merge Sort

- The smaller of arrayA[indexA] and arrayB[indexB] is copied into arrayC[indexC] and the appropriate position holders are advanced.
- When either input list is exhausted, the remainder of the other list is copied into arrayC.

Merge Sort

arrayA

1	13	24	26
indexA			

arrayB

2	15	27	38
indexB			

arrayC

indexC							

We compare arrayA[indexA] with arrayB[indexB].
Whichever value is smaller is placed into arrayC[indexC].

$1 < 2$ so we insert
arrayA[indexA] into
arrayC[indexC]

Merge Sort

arrayA

1	13	24	26
	indexA		

arrayB

2	15	27	38
indexB			

arrayC

1							
	indexC						

2 < 13 so we insert
arrayB[indexB] into
arrayC[indexC]

Merge Sort

arrayA

1	13	24	26
	indexA		

arrayB

2	15	27	38
	indexB		

arrayC

1	2						
		indexC					

13 < 15 so we insert
arrayA[indexA] into
arrayC[indexC]

Merge Sort

arrayA

1	13	24	26
		indexA	

arrayB

2	15	27	38
	indexB		

arrayC

1	2	13					
			indexC				

15 < 24 so we insert
arrayB[indexB] into
arrayC[indexC]

Merge Sort

arrayA

1	13	24	26
		indexA	

arrayB

2	15	27	38
		indexB	

arrayC

1	2	13	15				
				indexC			

24 < 27 so we insert
arrayA[indexA] into
arrayC[indexC]

Merge Sort

arrayA

1	13	24	26
			indexA

arrayB

2	15	27	38
		indexB	

arrayC

1	2	13	15	24			
					indexC		

26 < 27 so we insert
arrayA[indexA] into
arrayC[indexC]

Merge Sort

arrayA

1	13	24	26

arrayB

2	15	27	38
		indexB	

Since we have exhausted one of the arrays, arrayA, we simply copy the remaining items from the other array, arrayB, into arrayC

arrayC

1	2	13	15	24	26		
						indexC	

Merge Sort

arrayA

1	13	24	26

arrayB

2	15	27	38

arrayC

1	2	13	15	24	26	27	38

Merge Sort Pseudocode

```
1.  Algorithm MergeSort(low, high)
2.  {
3.      if (low<high) then
4.      {
5.          mid:=floor((low+high)/2); //Divide the array into subproblems
6.          MergeSort(low, mid); // Solve the subproblems.
7.          MergeSort(mid+1, high);
8.          Merge(low, mid, high); //Combine the solutions.
9.      }
10. }
```

//Reference : Page 146, Sahni

Merge Sort Pseudocode (cont)

```
1.  Algorithm Merge(low, mid, high)
2.  {
3.      h := low; i := low; j := mid + 1;
4.      while ((h ≤ mid) and (j ≤ high)) do
5.          {
6.              if (a[h] ≤ a[j]) then
7.                  { b[i] := a[h]; h := h + 1; }
8.              else
9.                  { b[i] := a[j]; j := j + 1; }
10.             i := i + 1;
11.         }
```

//Reference : Page 147, Sahni

Merge Sort Pseudocode (cont)

```
12.    if ( $h > mid$ ) then
13.        for  $k := j$  to  $high$  do
14.            {
15.                 $b[i] := a[k]; i := i + 1;$ 
16.            }
17.        else
18.            for  $k := h$  to  $mid$  do
19.                {
20.                     $b[i] := a[k]; i := i + 1;$ 
21.                }
22.    for  $k := low$  to  $high$  do  $a[k] := b[k];$ 
23. }
```

//Reference : Page 147, Sahni

Merge Sort Example

Consider the array of ten elements $a[1:10] = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520)$

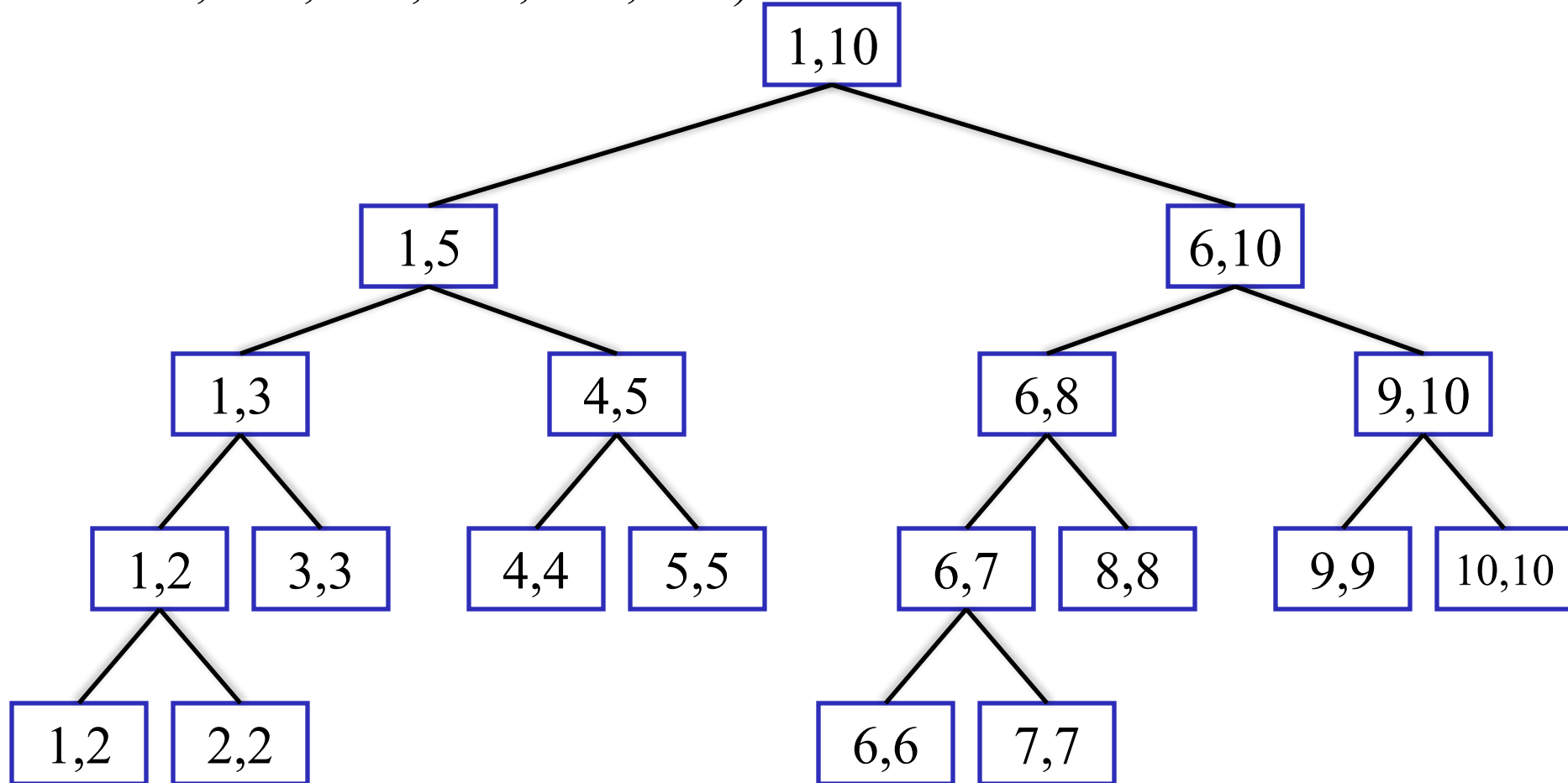


Fig: Tree of calls of MergeSort(1,10), Reference: Page 148;
Sahni

Merge Sort Example (cont.)

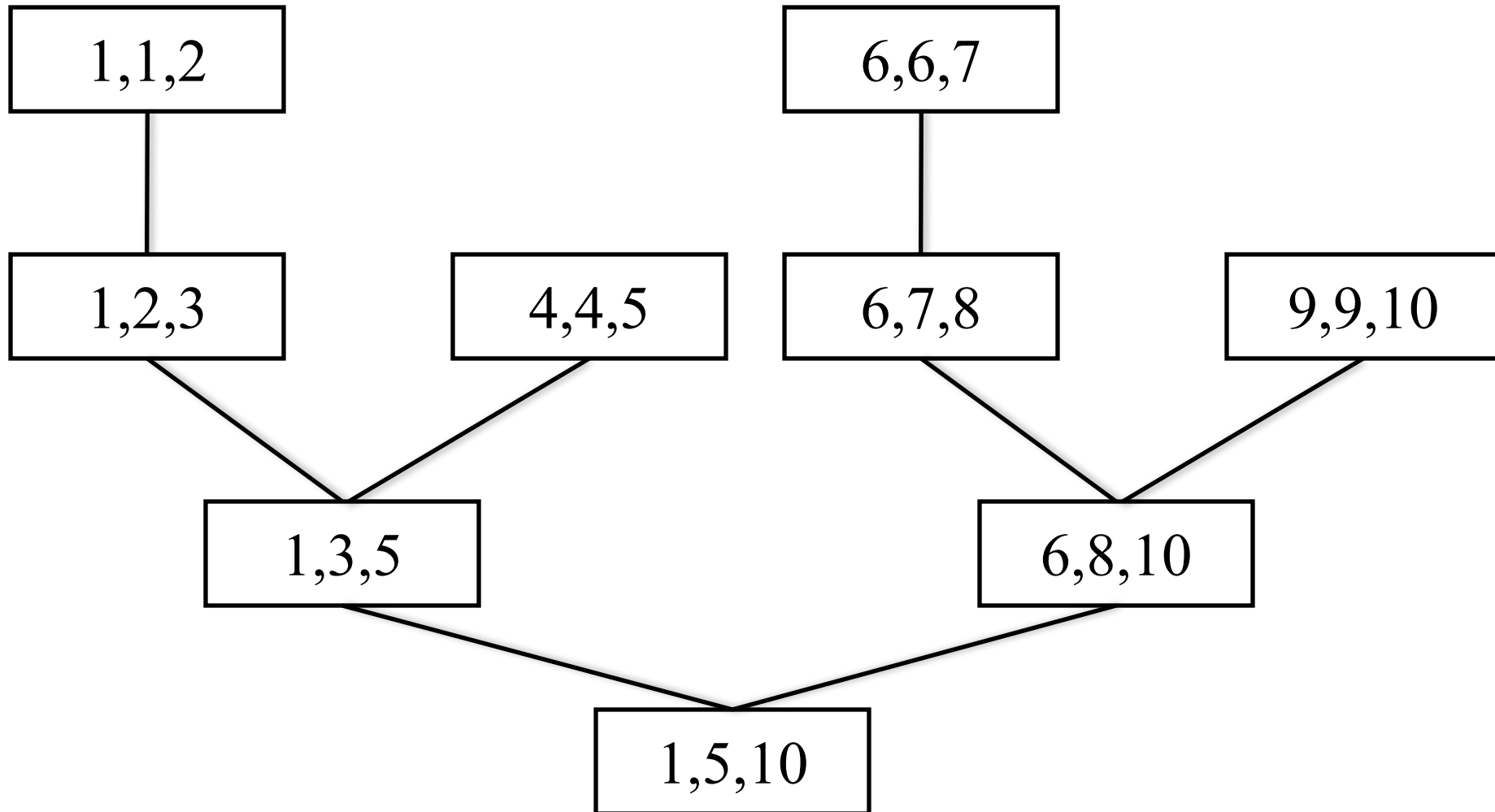


Fig: Tree of calls of Merge Reference: Page 149; Sahni

Merge Sort Analysis

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ 2T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

$a = 2$ (two subproblems)

$n/b = n/2$ (each subproblem has size approx $n/2$)

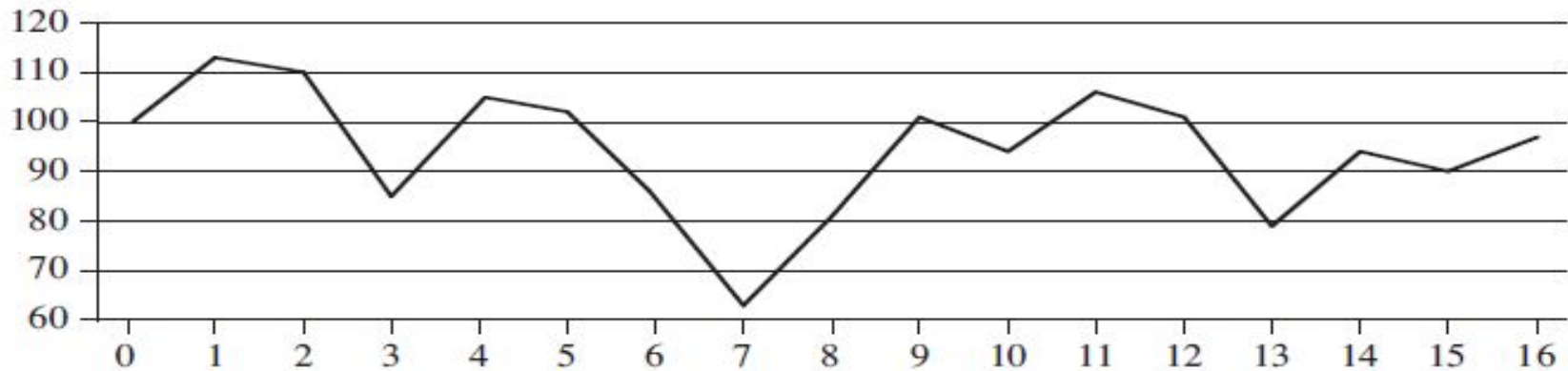
$D(n) = \Theta(1)$ (just compute midpoint of array)

$C(n) = \Theta(n)$ (merging can be done by scanning sorted subarrays)

Using case 2 of the master theorem has the solution $T(n) = O(n \log n)$.

Maximum-Subarray Problem

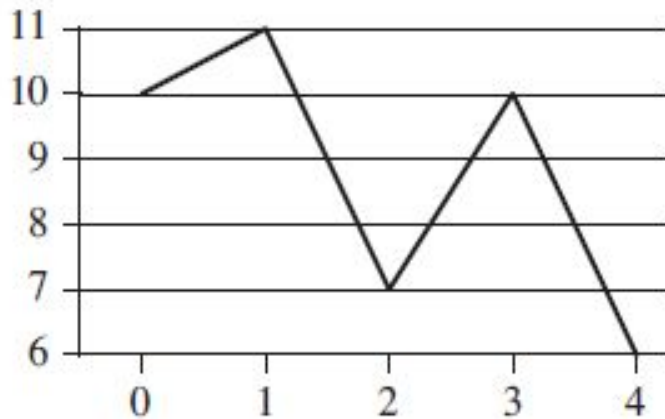
- Allowed to buy one unit of stock only one time and then sell it at a later date, buying and selling after the close of trading for the day.
- Allowed to learn what the price of the stock will be in the future.
- Goal is to maximize the profit.



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

Figure 4.1 Information about the price of stock in the Volatile Chemical Corporation after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.

Maximum-Subarray Problem



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	-4	3	-4

Figure 4.2 An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of \$3 per share would be earned by buying after day 2 and selling after day 3. The price of \$7 after day 2 is not the lowest price overall, and the price of \$10 after day 3 is not the highest price overall.

Maximum-Subarray Problem


A brute-force solution

Can easily devise a brute-force solution to this problem: just try every possible pair of buy and sell dates in which the buy date precedes the sell date. $\Theta(n^2)$

Maximum subarray

Contiguous subarray of **A** whose values have the largest sum. We call this contiguous subarray the *maximum subarray*. *For example, the maximum subarray of A[1 ... 16] is A[8...11], with the sum 43.*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7


maximum subarray

Maximum-Subarray Problem

A solution using divide-and-conquer

Let's think about how we might solve the maximum-subarray problem using the divide-and-conquer technique. Suppose we want to find a maximum subarray of the subarray $A[\text{low} \dots \text{high}]$. Divide-and-conquer suggests that we divide the subarray into two subarrays of as equal size as possible. That is, we find the midpoint, say mid , of the subarray, and consider the subarrays $A[\text{low} \dots \text{mid}]$ and $A[\text{mid} + 1 \dots \text{high}]$. As Figure 4.4(a) shows, any contiguous subarray $A[i \dots j]$ of $A[\text{low} \dots \text{high}]$ must lie in exactly one of the following places:

- entirely in the subarray $A[\text{low} \dots \text{mid}]$, so that $\text{low} \leq i \leq j \leq \text{mid}$,
- entirely in the subarray $A[\text{mid} + 1 \dots \text{high}]$, so that $\text{mid} < i \leq j \leq \text{high}$, or
- crossing the midpoint, so that $\text{low} \leq i \leq \text{mid} < j \leq \text{high}$.

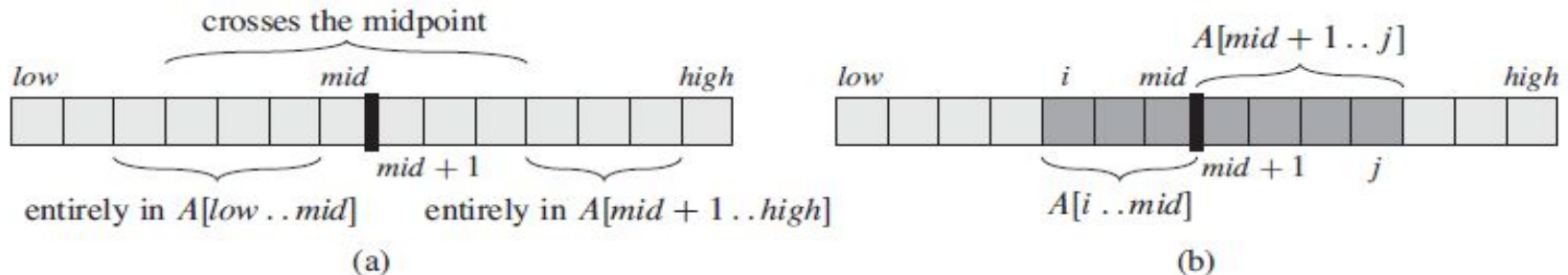


Figure 4.4 (a) Possible locations of subarrays of $A[\text{low} \dots \text{high}]$: entirely in $A[\text{low} \dots \text{mid}]$, entirely in $A[\text{mid} + 1 \dots \text{high}]$, or crossing the midpoint mid . (b) Any subarray of $A[\text{low} \dots \text{high}]$ crossing the midpoint comprises two subarrays $A[i \dots \text{mid}]$ and $A[\text{mid} + 1 \dots j]$, where $\text{low} \leq i \leq \text{mid}$ and $\text{mid} < j \leq \text{high}$.

A solution using divide-and-conquer

We can easily find a maximum subarray crossing the midpoint in time linear in the size of the subarray $A[\text{low} \dots \text{high}]$. *This problem is not a smaller instance* of our original problem, because it has the added restriction that the subarray it chooses must cross the midpoint. As Figure 4.4(b) shows, any subarray crossing the midpoint is itself made of two subarrays $A[i \dots \text{mid}]$ and $A[\text{mid}+1 \dots j]$, where $\text{low} \leq i \leq \text{mid}$ and $\text{mid} < j \leq \text{high}$. Therefore, we just need to find maximum subarrays of the form $A[i \dots \text{mid}]$ and $A[\text{mid}+1 \dots j]$ and then combine them. The procedure FIND-MAX-CROSSING-SUBARRAY takes as input the array A and the indices low , mid , and high , and it returns a tuple containing the indices demarcating a maximum subarray that crosses the midpoint, along with the sum of the values in a maximum subarray.

FIND-MAX-CROSSING-SUBARRAY($A, \text{low}, \text{mid}, \text{high}$)

```
1  left-sum =  $-\infty$ 
2  sum = 0
3  for i = mid downto low
4      sum = sum + A[i]
5      if sum > left-sum
6          left-sum = sum
7          max-left = i
8  right-sum =  $-\infty$ 
9  sum = 0
10 for j = mid + 1 to high
11     sum = sum + A[j]
12     if sum > right-sum
13         right-sum = sum
14         max-right = j
15 return (max-left, max-right, left-sum + right-sum)
```

If the subarray $A[\text{low} \dots \text{high}]$ contains n entries (so that $n = \text{high} - \text{low} + 1$)

Total number of iterations is:

$$(\text{Mid} - \text{low} + 1) + (\text{high} - \text{mid}) = \text{high} - \text{low} + 1 = n$$

A solution using divide-and-conquer

With a linear-time FIND-MAX-CROSSING-SUBARRAY procedure in hand, we can write pseudocode for a divide-and-conquer algorithm to solve the maximum-subarray problem:

FIND-MAXIMUM-SUBARRAY($A, low, high$)

```
1  if  $high == low$ 
2      return ( $low, high, A[low]$ )           // base case: only one element
3  else  $mid = \lfloor (low + high)/2 \rfloor$ 
4      ( $left-low, left-high, left-sum$ ) =
        FIND-MAXIMUM-SUBARRAY( $A, low, mid$ )
5      ( $right-low, right-high, right-sum$ ) =
        FIND-MAXIMUM-SUBARRAY( $A, mid + 1, high$ )
6      ( $cross-low, cross-high, cross-sum$ ) =
        FIND-MAX-CROSSING-SUBARRAY( $A, low, mid, high$ )
7      if  $left-sum \geq right-sum$  and  $left-sum \geq cross-sum$ 
8          return ( $left-low, left-high, left-sum$ )
9      elseif  $right-sum \geq left-sum$  and  $right-sum \geq cross-sum$ 
10         return ( $right-low, right-high, right-sum$ )
11     else return ( $cross-low, cross-high, cross-sum$ )
```

The initial call FIND-MAXIMUM-SUBARRAY($A, 1, A.length$) will find a maximum subarray of $A[1..n]$.

Analyzing the divide-and-conquer algorithm

Next we set up a recurrence that describes the running time of the recursive FIND-MAXIMUM-SUBARRAY procedure. As we did when we analyzed merge sort in Section 2.3.2, we make the simplifying assumption that the original problem size is a power of 2, so that all subproblem sizes are integers. We denote by $T(n)$ the running time of FIND-MAXIMUM-SUBARRAY on a subarray of n elements. For starters, line 1 takes constant time. The base case, when $n = 1$, is easy: line 2 takes constant time, and so

$$T(1) = \Theta(1) . \tag{4.5}$$

The recursive case occurs when $n > 1$. Lines 1 and 3 take constant time. Each of the subproblems solved in lines 4 and 5 is on a subarray of $n/2$ elements (our assumption that the original problem size is a power of 2 ensures that $n/2$ is an integer), and so we spend $T(n/2)$ time solving each of them. Because we have to solve two subproblems—for the left subarray and for the right subarray—the contribution to the running time from lines 4 and 5 comes to $2T(n/2)$. As we have

already seen, the call to FIND-MAX-CROSSING-SUBARRAY in line 6 takes $\Theta(n)$ time. Lines 7–11 take only $\Theta(1)$ time. For the recursive case, therefore, we have

$$\begin{aligned} T(n) &= \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\ &= 2T(n/2) + \Theta(n) . \end{aligned} \tag{4.6}$$

Combining equations (4.5) and (4.6) gives us a recurrence for the running time $T(n)$ of FIND-MAXIMUM-SUBARRAY:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases} \tag{4.7}$$

Quick Sort

- Quicksort, like mergesort, is based on the divide-and-conquer paradigm. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p \dots r]$.

Divide: Partition the array $A[p..r]$ into two non-empty subarrays $A[p..q-1]$ and $A[q+1..r]$ such that each element of $A[p..q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1..r]$. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.

Combine: Since the subarrays are sorted in place, no work is needed to combine them, the entire array $A[p..r]$

Quick Sort Example

(a) The initial array and variable settings

i	<i>p j</i>							<i>r</i>
(a)	2	8	7	1	3	5	6	4

(b) The value 2 is swapped with itself and put in the partition of smaller values.

	<i>p i</i>	<i>j</i>						<i>r</i>
(b)	2	8	7	1	3	5	6	4

(c) The value 8 is added to the partition of larger values.

	<i>p i</i>		<i>j</i>					<i>r</i>
(c)	2	8	7	1	3	5	6	4

Quick Sort Example (cont.)

(d) The value 7 is added to the partition of larger values.

(d)	p	i			j			r
	2	8	7	1	3	5	6	4

(e) The values 1 and 8 are swapped, and the smaller portion grows

(e)	p	i			j			r
	2	1	7	8	3	5	6	4

(f) The values 3 and 7 are swapped, and the smaller portion grows

(e)	p		i			j		r
	2	1	3	8	7	5	6	4

Quick Sort Example (cont.)

(g) The values 5 and 6 are added to the partition of larger values and loop terminates.

(e)	p		i					r
	2	1	3	8	7	5	6	4

(h) The pivot element is swapped so that it lies between the two partitions. Hence, the pivot element has been placed in its final sorted position.

(h)	p		i					r
	2	1	3	4	7	5	6	8

- Now repeat the process using the sub-arrays to the left and right of the pivot.

Quick Sort Pseudocode

```
1. Quicksort ( $A, p, r$ )
2. {
3.   if  $p < r$  then
4.   {
5.      $q := \text{Partition}(A, p, r);$ 
6.     Quicksort ( $A, p, q-1$ );
7.     Quicksort ( $A, q+1, r$ );
8.   }
9. }
```

- To sort an entire array A , the initial call is Quicksort ($A, 1, \text{length}[A]$).

//Reference: Page 146, Corman

Quick Sort Pseudocode

```
1. Partition( $A, p, r$ )
2. {
3.      $x := A[r]$ ;
4.      $i := p-1$ ;
5.     for  $j := p$  to  $r-1$  do
6.         if ( $A[j] \leq x$ ) then
7.             {
8.                  $i := i+1$ ;
9.                 exchange  $A[i] \leftrightarrow A[j]$ ;
10.            }
11.     exchange  $A[i + 1] \leftrightarrow A[r]$ 
12.     return  $i+1$ 
13. }
```

//Reference: Page 146, Corman

A randomized version of quicksort

RANDOMIZED-PARTITION(A, p, r)

- 1 $i = \text{RANDOM}(p, r)$
- 2 exchange $A[r]$ with $A[i]$
- 3 **return** PARTITION(A, p, r)

The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

RANDOMIZED-QUICKSORT(A, p, r)

- 1 **if** $p < r$
- 2 $q = \text{RANDOMIZED-PARTITION}(A, p, r)$
- 3 RANDOMIZED-QUICKSORT($A, p, q - 1$)
- 4 RANDOMIZED-QUICKSORT($A, q + 1, r$)

Worst Case Analysis

- occurs when pivot is smallest or largest element (one sublist has $n - 1$ entries and the other is empty)
- also if list is pre-sorted or is in reverse order: all elements go into S1 or S2
 - since $C(0) = 0$, we have $C(n) = n - 1 + C(n - 1)$
 - we can solve this recurrence relation by starting at the bottom:

Worst Case Analysis (cont.)

$$C(1) = 0$$

$$C(2) = 1 + C(1) = 1$$

$$C(3) = 2 + C(2) = 2 + 1$$

$$C(4) = 3 + C(3) = 3 + 2 + 1$$

... ..

$$C(n) = n - 1 + C(n - 1)$$

$$= (n - 1) + (n - 2) + \dots + 2 + 1$$

$$= (n - 1)n/2$$

$$= 1/2 n^2 - 1/2 n \text{ (sum of integers from 1 to } n-1)$$

$$= O(n^2)$$

Worst Case Analysis (cont.)

- Selection sort makes approx. $\frac{1}{2}n^2 - \frac{1}{n}$ key comparisons, or $O(n^2)$, in worst case. So, in its worst case, quicksort is as bad as the worst case of selection sort.
- Number of swaps made by quicksort is about 3 times as many as worst case of insertion sort

Average Case Analysis

- Average behavior of quicksort, when applied to lists in random order
 - assume all possible orderings of S_1 equally likely, hence has probability of $1/n$
 - pivot is equally likely to be any element
 - number of key comparisons made is $n-1$, as in worst case

Average Case Analysis (cont.)

- Let $C(n)$ be average number of comparisons done by quicksort on list of length n and $C(n,p)$ be average number of comparisons on list of length n where the pivot for the first partition is p .
- Recursion makes $C(p-1)$ comparisons for the sublist of entries less than the pivot and $C(n-p)$ comparisons for the sublist of entries greater than the pivot. Thus, for $n \geq 2$,
$$C(n,p) = n - 1 + C(p - 1) + C(n - p)$$

Average Case Analysis (cont.)

- To find $C(n)$, take the average of these expressions, since p is random, by adding them from $p=1$ to $p=n$ and dividing by n . This yields the recurrence relation for the number of comparisons done in the average case:

$$C(n) = n - 1 + 2/n(C(0) + C(1) + \dots + C(n-1))$$

Average Case Analysis (cont.)

- To solve: note that if sorting a list of length $n-1$, we would have the same expression with n replaced by $n-1$, as long as $n > 2$:

$$C(n-1) = n - 2 + 2/(n-1)(C(0) + C(1) + \dots + C(n-2))$$

- Multiply the first expression by n , the second by $n-1$ and subtract:

$$nC(n) - (n-1)C(n-1) = n(n-1) - (n-1)(n-2) + 2C(n-1)$$

Average Case Analysis (cont.)

Rearrange:

$$(C(n)-2)/(n+1) = (C(n-1) - 2)/n + 2/(n+1)$$

Now let $S(n)$ be defined as $(C(n)-2)/(n+1)$

Then $S(n) = S(n-1) + 2/(n+1)$

Average Case Analysis (cont.)

- Solve by working down from n , each time replacing expression by a smaller case until we reach $n=1$:

$$S(n) = 2/(n+1) + S(n-1)$$

$$S(n-1) = 2/n + S(n-2)$$

$$S(n-2) = 2/(n-1) + S(n-3)$$

...

$$S(n) = 2/(n+1) + 2/n + 2/(n-1) \dots + 2/2$$

...

$$= 2 (1/(n+1) + 1/n + 1/(n-1) + \dots + 1)$$

$$= 2 H_n + 1$$

Average Case Analysis (cont.)

- Now we can write:

$$(C(n)-2)/(n+1) = 2 H_n + 1$$

$$(C(n)-2)/(n+1) = 2 \ln n + O(1)$$

$$\begin{aligned} C(n) &= (n+1) * 2 \ln n + (n+1) * O(1) \\ &= 2n \ln n + O(n) \end{aligned}$$

- In its average case, quicksort performs $C(n) = 2n \ln n + O(n)$ comparisons of keys in sorting a list of n entries

$\ln n = (\ln 2)(\lg n)$ and $\ln 2 = 0.693$,

so that $C(n) = 1.386n \lg n + O(n) = O(n \log n)$

Best case Analysis

- pivot is middle element; two subfiles are each half the size of original

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

which by case 2 of the master theorem has the solution $T(n)=O(n \log n)$.

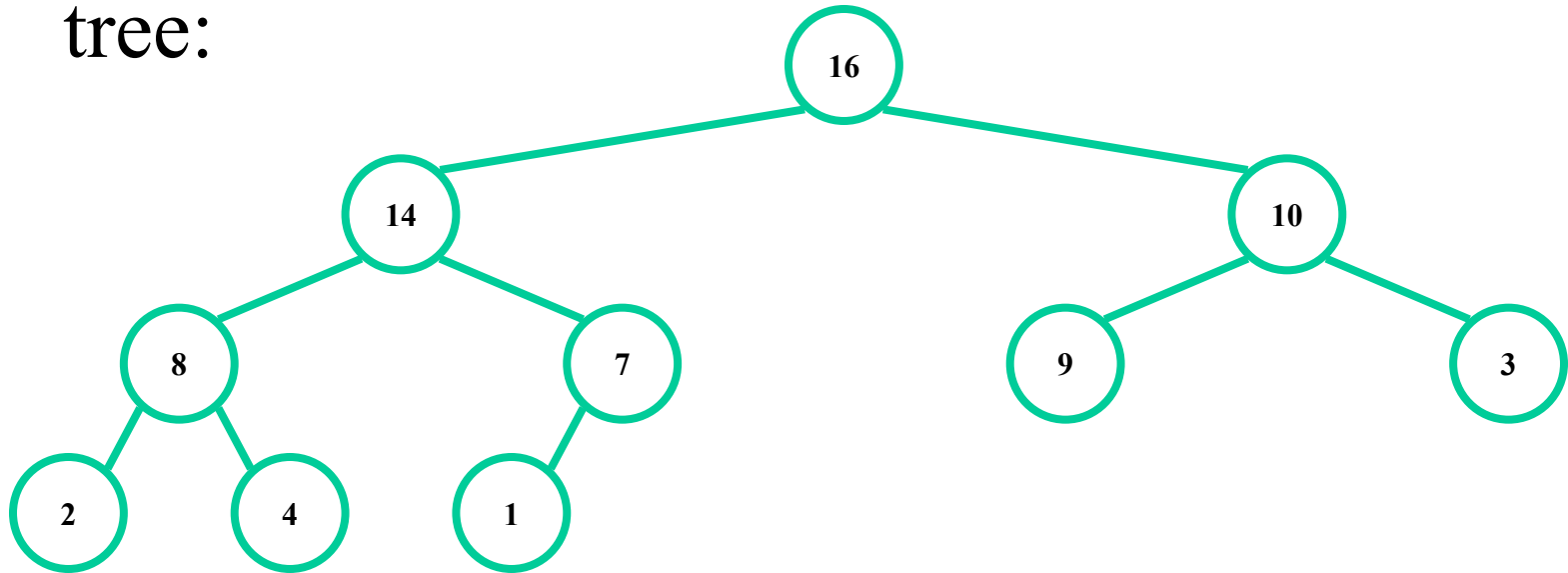
Heapsort

First, heapsort must build a heap

- A ‘heap’ is a tree wherein each node is of a value equal to or less than its parent.
- Additionally, a ‘binary tree’ is used, which adds the constraint that no parent may have more than two nodes; this allows heapsort to run within its own array, without additional space complexity.
- Each limb is passed through in reverse order, if the largest child is larger than the parent, they are swapped

Heaps

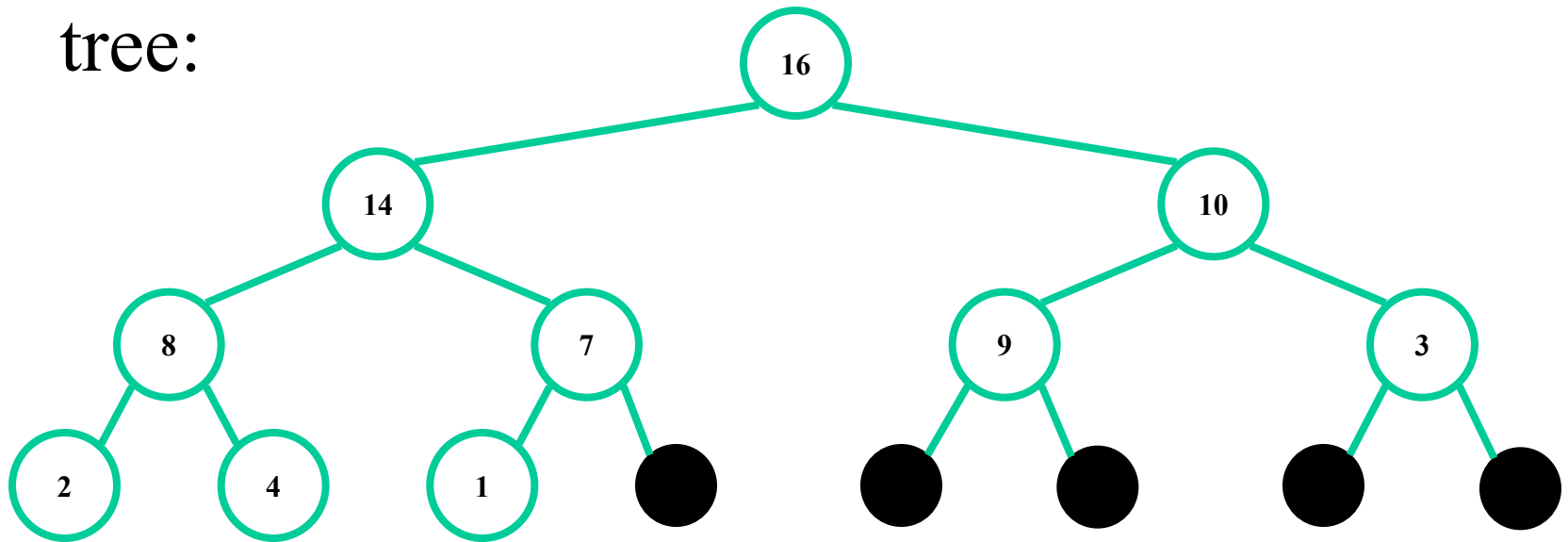
- A *heap* can be seen as a complete binary tree:



- *What makes a binary tree complete?*
- *Is the example above complete?*

Heaps

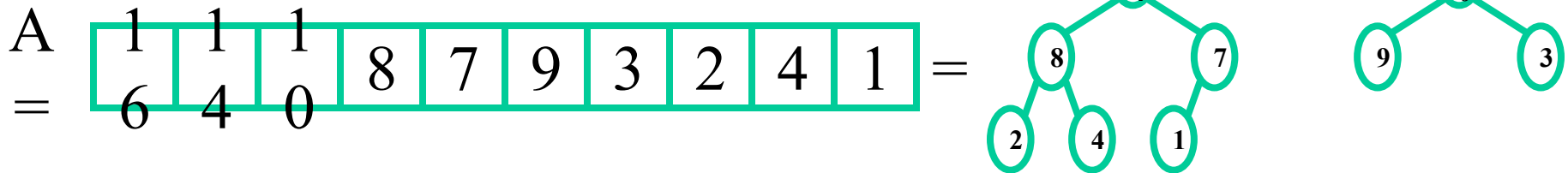
- A *heap* can be seen as a complete binary tree:



- The book calls them “nearly complete” binary trees; can think of unfilled slots as null pointers

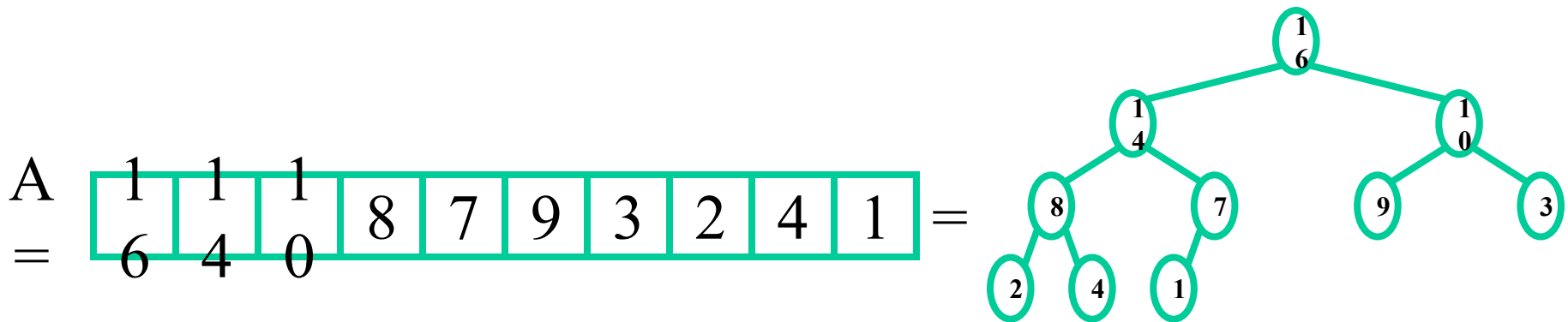
Heaps

- In practice, heaps are usually implemented as arrays:



Heaps

- To represent a complete binary tree as an array:
 - The root node is $A[1]$
 - Node i is $A[i]$
 - The parent of node i is $A[i/2]$ (note: integer divide)
 - The left child of node i is $A[2i]$
 - The right child of node i is $A[2i + 1]$



Referencing Heap Elements

- So...

```
Parent(i)
```

```
{ return [i/2]; }
```

```
Left(i)
```

```
{ return 2*i; }
```

```
Right(i)
```

```
{ return 2*i + 1; }
```

//Reference: Page 128, Cormen

The Heap Property

- Heaps also satisfy the *heap property (max-heap)*:

$$A[\text{Parent}(i)] \geq A[i] \quad \text{for all nodes } i > 1$$

- In other words, the value of a node is at most the value of its parent

- *Where is the largest element in a heap stored?*

- Heaps also satisfy the *heap property (min-heap)*:

$$A[\text{Parent}(i)] \leq A[i] \quad \text{for all nodes } i > 1$$

- In other words, the smallest element in a min-heap is at root.

- *Where is the smallest element in a heap stored?*

Heap Height

- Definitions:
 - The *height* of a node in the heap = the number of edges on the longest downward path from the node to a leaf
 - The height of a heap = the height of its root
- *What is the height of an n -element heap? Why? ($\Theta(\log n)$)*
- This is nice: basic heap operations take at most time proportional to the height of the heap
- Is an array that is in sorted order a min-heap?

Heap Operations: Max-Heapify()

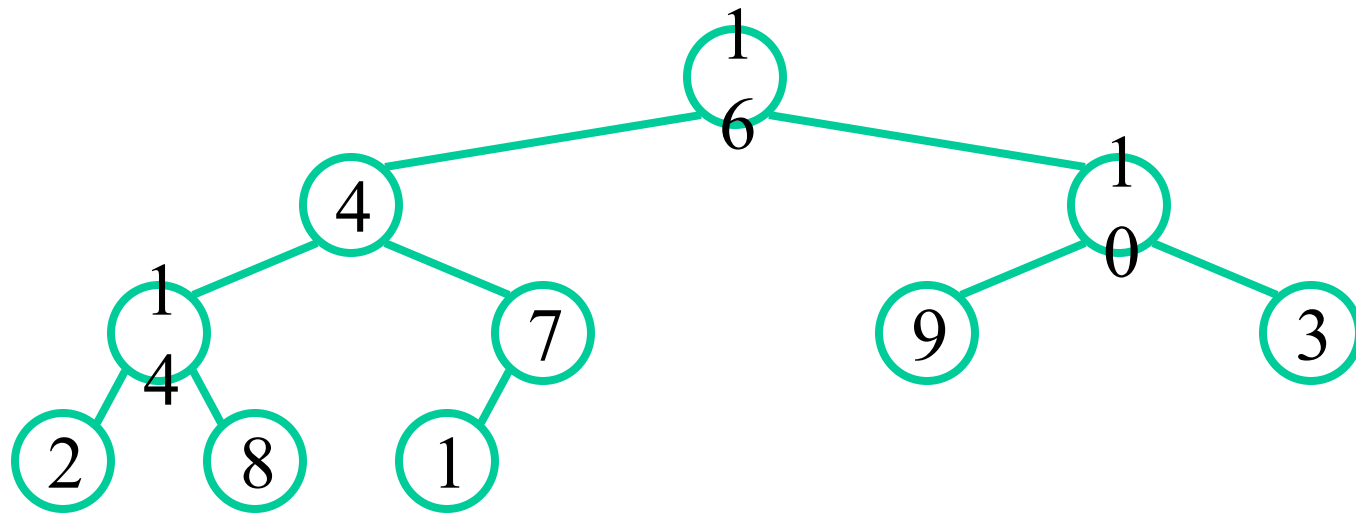
- **Max-Heapify ()** : maintain the heap property
 - Given: a node i in the heap with children l and r
 - Given: two subtrees rooted at l and r , assumed to be heaps
 - Problem: The subtree rooted at i may violate the heap property (*How?*)
 - Action: let the value of the parent node “float down” so subtree at i satisfies the heap property
 - *What do you suppose will be the basic operation between i , l , and r ?*

Heap Operations: Max-Heapify()

```
Max-Heapify(A, i)
{
    l = Left(i);
    r = Right(i);
    if (l <= heap_size(A) && A[l] > A[i])
        largest = l;
    else
        largest = i;
    if (r <= heap_size(A) && A[r] > A[largest])
        largest = r;
    if (largest != i)
        exchange A[i] ↔ A[largest];
    Max-Heapify(A, largest);
}
```

//Reference: Page 130, Cormen

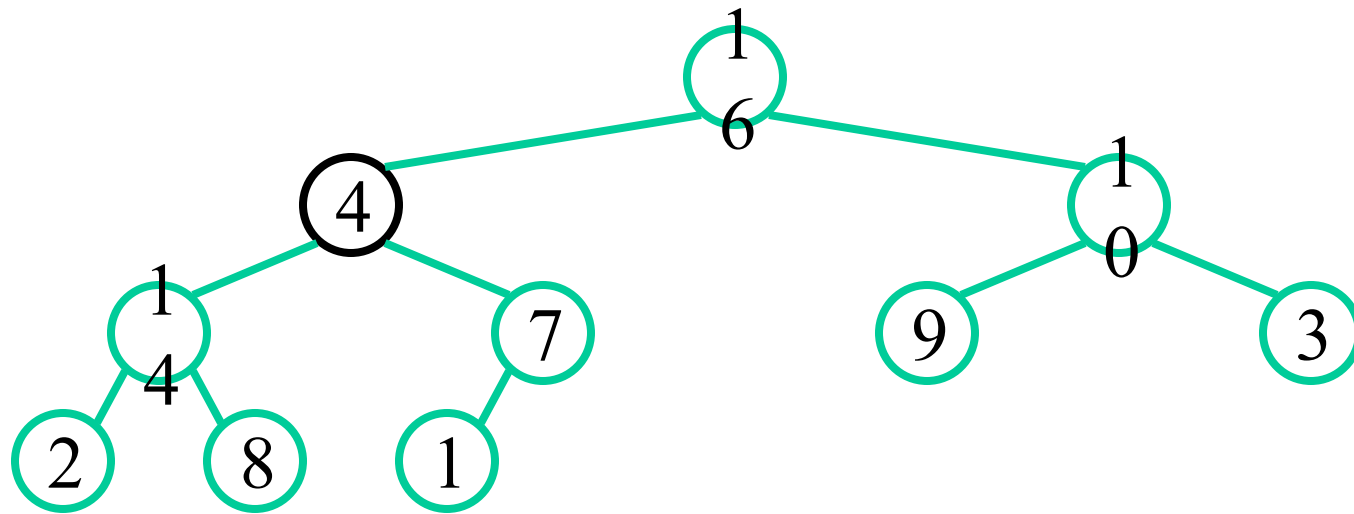
Max-Heapify() Example



A =

1	4	1	1	7	9	3	2	8	1
6		0	4						

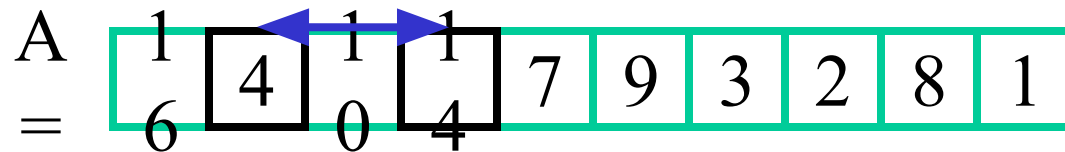
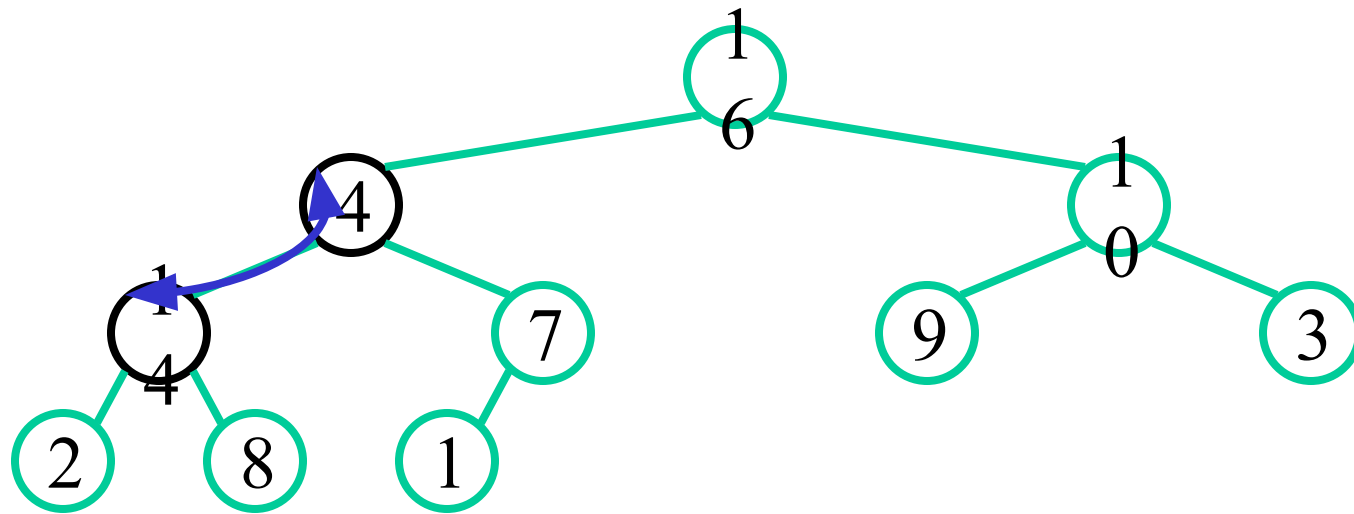
Max-Heapify() Example



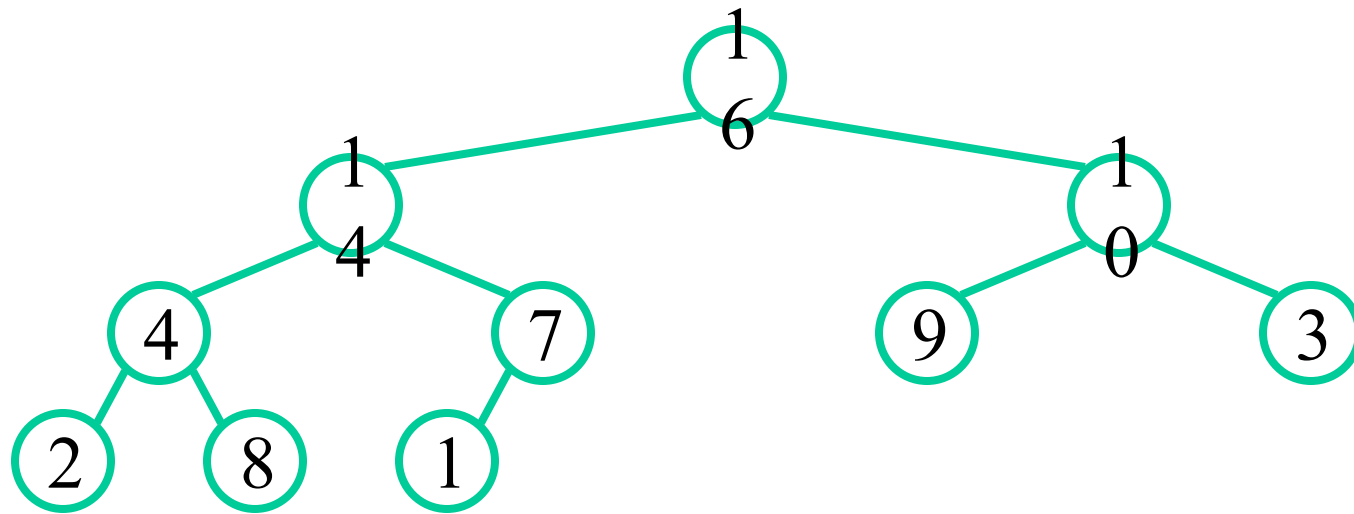
A =

1	4	1	1	7	9	3	2	8	1
6		0	4						

Max-Heapify() Example



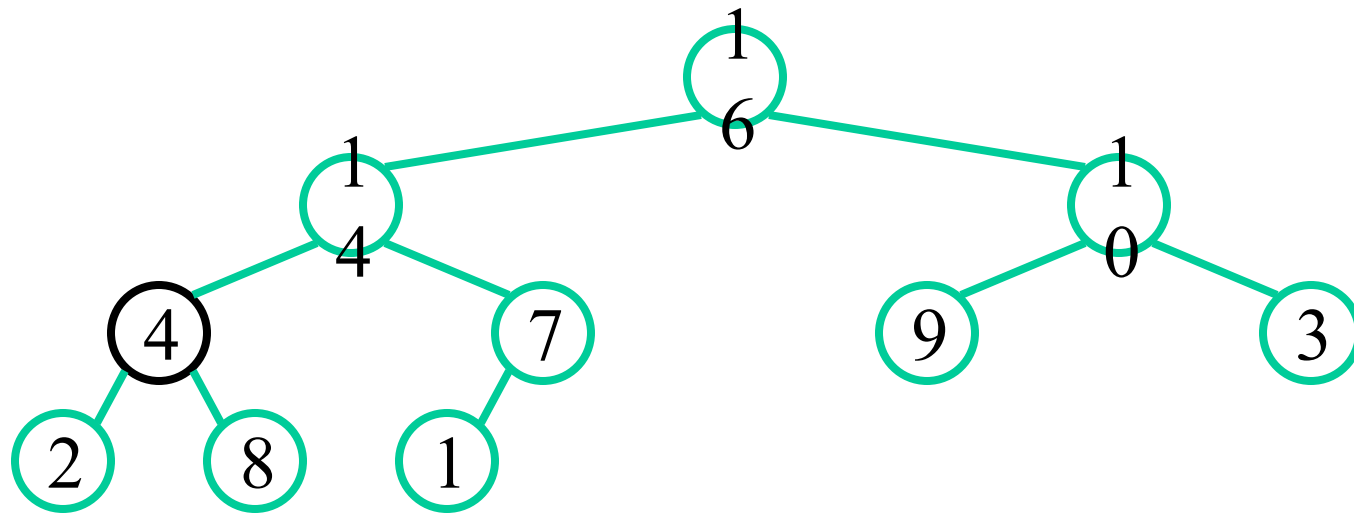
Max-Heapify() Example



A =

1	1	1	4	7	9	3	2	8	1
6	4	0							

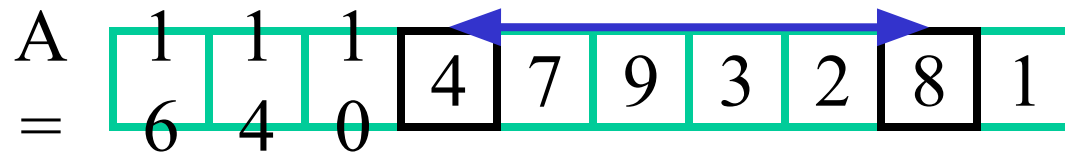
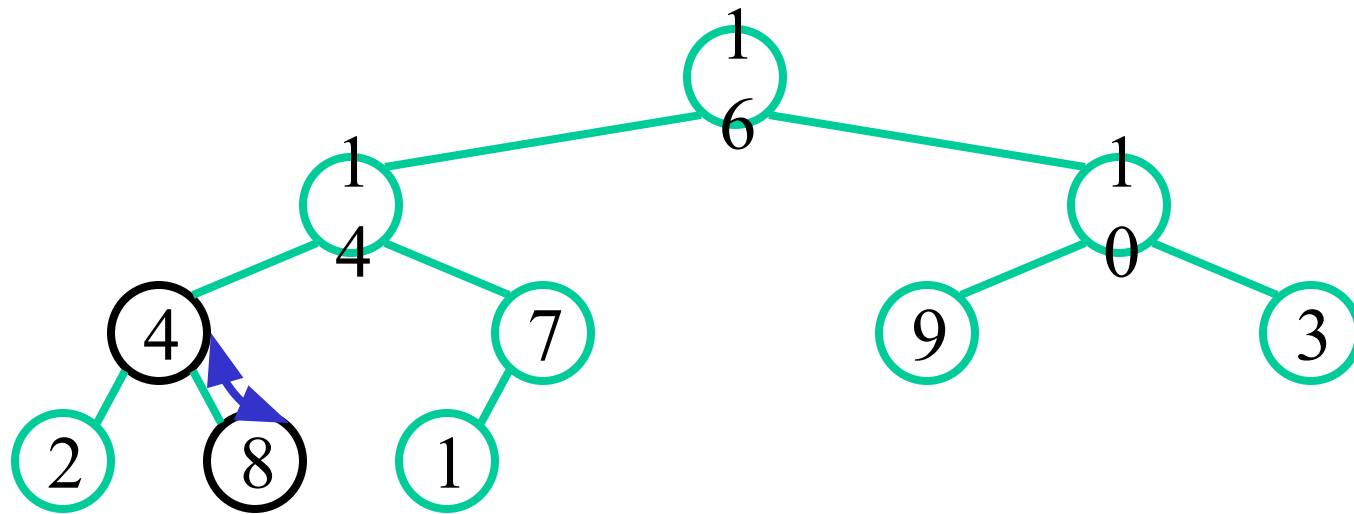
Max-Heapify() Example



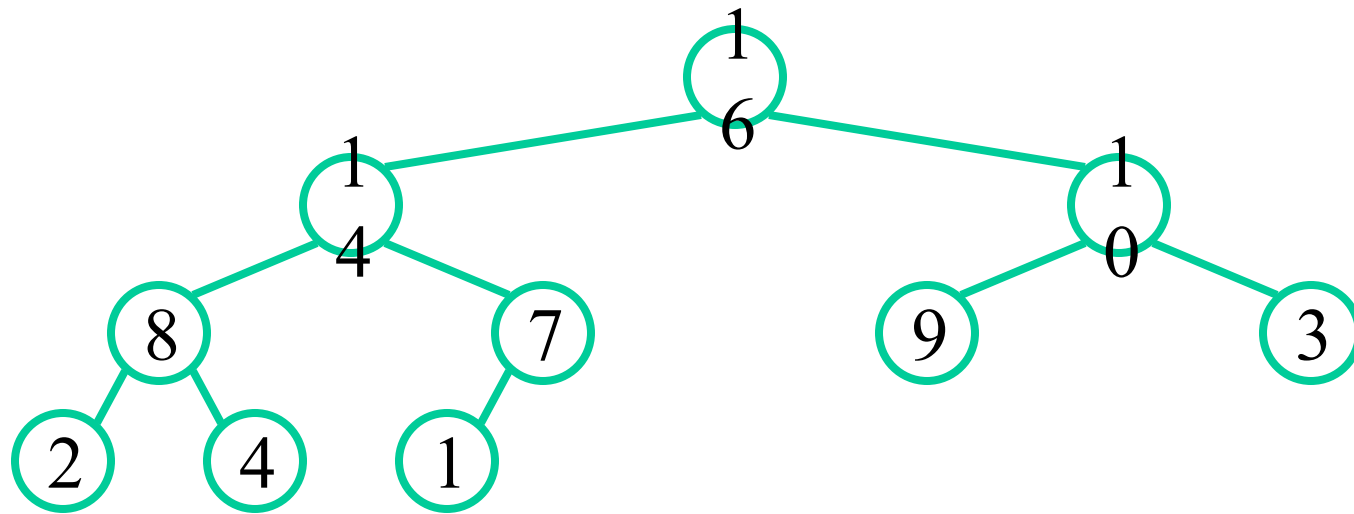
A =

1	1	1	4	7	9	3	2	8	1
6	4	0							

Max-Heapify() Example



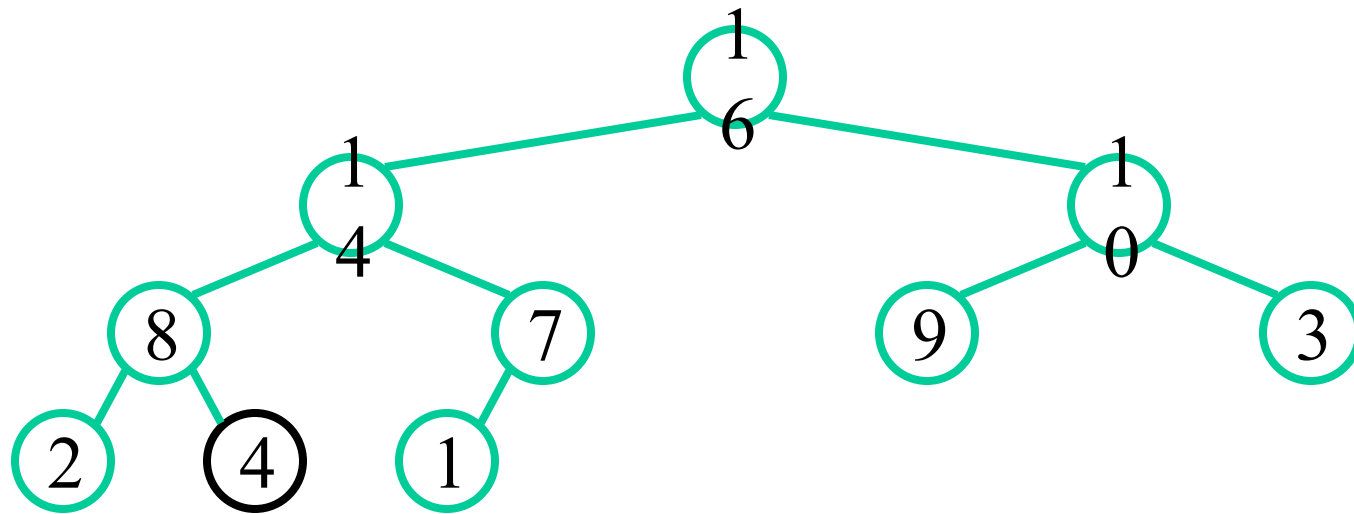
Max-Heapify() Example



A =

1	1	1	8	7	9	3	2	4	1
6	4	0							

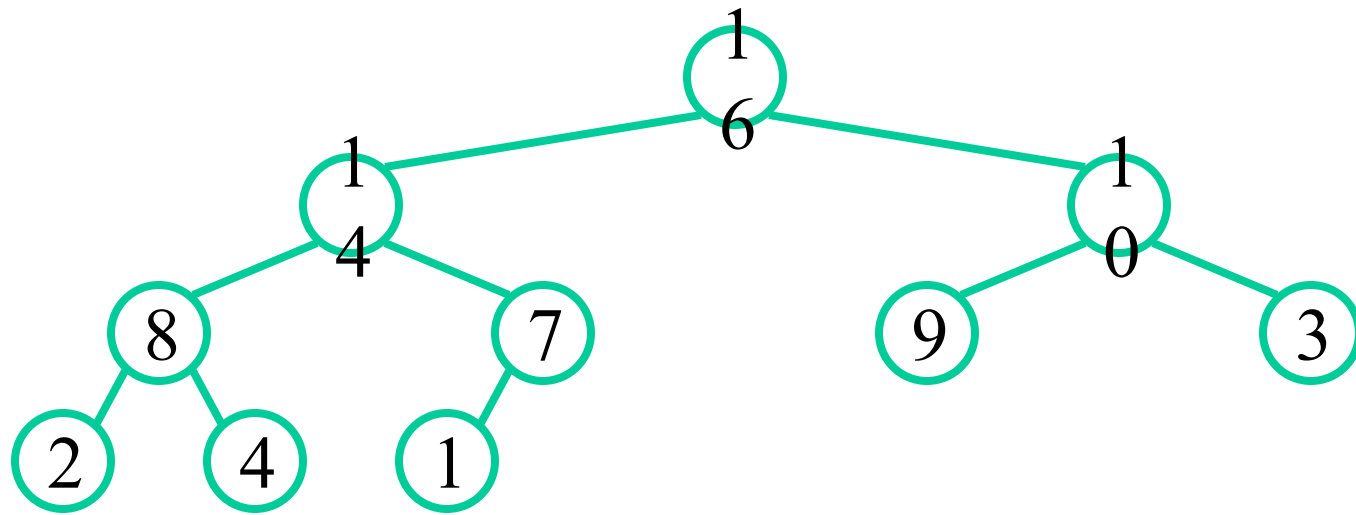
Max-Heapify() Example



A =

1	1	1	8	7	9	3	2	4	1
6	4	0							

Max-Heapify() Example



A =

1	1	1	8	7	9	3	2	4	1
6	4	0							

Analyzing Max-Heapify(): Informal

- *Aside from the recursive call, what is the running time of **Max-Heapify()**?*
- *How many times can **Max-Heapify()** recursively call itself?*
- *What is the worst-case running time of **Max-Heapify()** on a heap of size n ?*

AnalyzingMax-Heapify(): Formal

- Fixing up relationships between i , l , and r takes $\Theta(1)$ time
- *If the heap at i has n elements, how many elements can the subtrees at l or r have?*
 - Draw it
- Answer: $2n/3$ (worst case: bottom row 1/2 full)
- So time taken by **Max-Heapify()** is given by

$$T(n) \leq T(2n/3) + \Theta(1)$$

Analyzing Max-Heapify(): Formal

- So we have

$$T(n) \leq T(2n/3) + \Theta(1)$$

- By case 2 of the Master Theorem,

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

- Thus, **Max-Heapify()** takes logarithmic time

Heap Operations: BuildHeap()

- We can build a heap in a bottom-up manner by running **Max-Heapify()** on successive subarrays
 - Fact: for array of length n , all elements in range $A[\lfloor n/2 \rfloor + 1 .. n]$ are heaps (*Why?*)
 - So:
 - Walk backwards through the array from $n/2$ to 1, calling **Max-Heapify()** on each node.
 - Order of processing guarantees that the children of node i are heaps when i is processed

Build-Max-Heap()

// given an unsorted array A, make A a heap

Build-Max-Heap(A)

{

 heap_size(A) = length(A) ;

 for (i = $\lfloor \text{length}[A]/2 \rfloor$ downto 1)

 Max-Heapify(A, i) ;

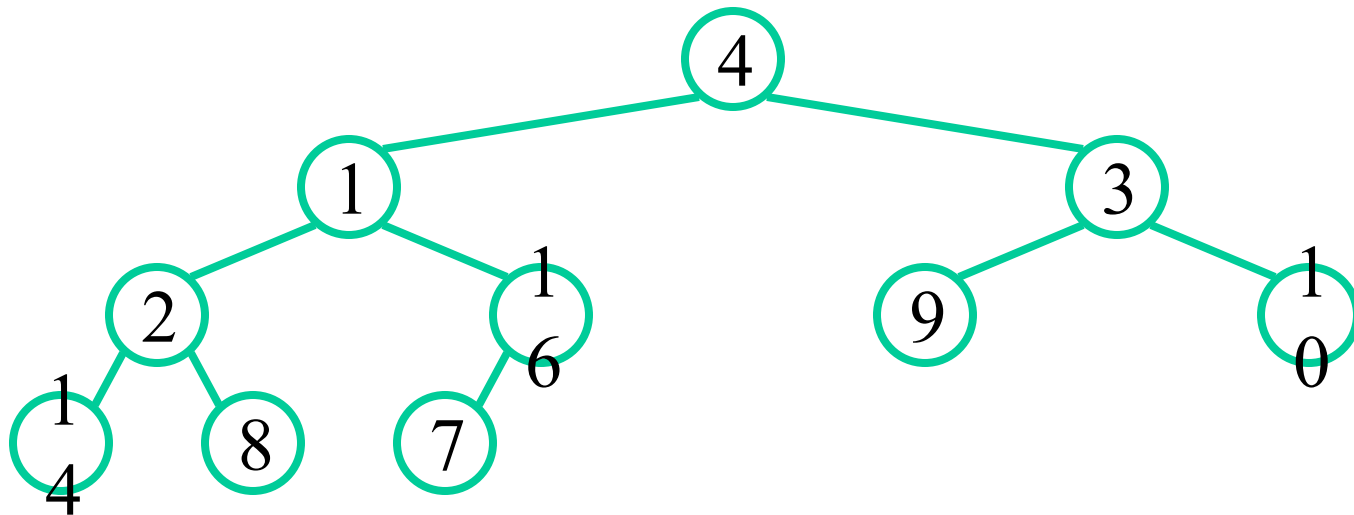
}

//Reference: Page 133, Cormen

Build-Max-Heap() Example

- Work through example

$A = \{4, 1, 3, 2, 16, 9, 10, 14, 8, 7\}$



Analyzing Build-Max-Heap()

- Each call to **Max-Heapify()** takes $O(\lg n)$ time
- There are $O(n)$ such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is $O(n \lg n)$
 - *Is this a correct asymptotic upper bound?*
 - *Is this an asymptotically tight bound?*
- A tighter bound is $O(n)$
 - *How can this be? Is there a flaw in the above reasoning?*

Analyzing Build-Max-Heap(): Tight

- To **Max-Heapify()** a subtree takes $O(h)$ time where h is the height of the subtree
 - $h = O(\lg m)$, $m = \#$ nodes in subtree
 - The height of most subtrees is small
- Fact: an n -element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height h
- The time required by Max-Heapify() when called on a node of height is $O(h)$, so we can express the total cost of **Max-BuildHeap()** as

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n) \quad // \text{See page 135 Cormen}$$

- **Max-BuildHeap()** takes $O(n)$ time

Heapsort

- Given **Build-Max-Heap()**, an in-place sorting algorithm is easily constructed:
 - Maximum element is at $A[1]$
 - Discard by swapping with element at $A[n]$
 - Decrement $\text{heap_size}[A]$
 - $A[n]$ now contains correct value
 - Restore heap property at $A[1]$ by calling **Max-Heapify()**
 - Repeat, always swapping $A[1]$ for $A[\text{heap_size}(A)]$

Heapsort

```
Heapsort (A)
{
    Build-Max-Heap (A) ;
    for (i = length[A] downto 2)
    {
        exchange (A[1] ↔ A[i]) ;
        heap_size (A) -= 1 ;
        Max-Heapify (A, 1) ;
    }
}
```

//Reference: Page 136, Cormen

Analyzing Heapsort

- The call to **Build-Max-Heap()** takes $O(n)$ time
- Each of the $n - 1$ calls to **Max-Heapify()** takes $O(\lg n)$ time
- Thus the total time taken by **HeapSort()**
 $= O(n) + (n - 1) O(\lg n)$
 $= O(n) + O(n \lg n)$
 $= O(n \lg n)$

Priority Queues

- Heapsort is a nice algorithm, but in practice Quicksort
- But the heap data structure is incredibly useful for implementing *priority queues*
 - A data structure for maintaining a set S of elements, each with an associated value or *key*
 - Supports the operations **Insert()**, **Maximum()**, and **ExtractMax()**
 - *What might a priority queue be useful for?*

Priority Queue Operations

- **Insert(S, x)** inserts the element x into set S
- **Maximum(S)** returns the element of S with the maximum key
- **ExtractMax(S)** removes and returns the element of S with the maximum key
- *How could we implement these operations using a heap? (Home work: see pages 138-140, Cormen)*

Sorting in Linear Time

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

We have now introduced several algorithms that can sort n numbers in $O(n \lg n)$ time. Merge sort and heap sort achieve this upper bound in the worst case; quicksort achieves it on average. Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in $\Omega(n \lg n)$ time.

These algorithms share an interesting property: *the sorted order they determine is based only on comparisons between the input elements. We call such sorting algorithms **comparison sorts**. All the sorting algorithms introduced thus far are comparison sorts.*

Lower bounds for sorting

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \dots, a_n \rangle$. That is, given two elements a_i and a_j , we perform one of the tests $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$ to determine their relative order. We may not inspect the values of the elements or gain order information about them in any other way.

In this section, we assume without loss of generality that all the input elements are distinct. Given this assumption, comparisons of the form $a_i = a_j$ are useless, so we can assume that no comparisons of this form are made. We also note that the comparisons $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$, and $a_i < a_j$ are all equivalent in that they yield identical information about the relative order of a_i and a_j . We therefore assume that all comparisons have the form $a_i \leq a_j$.

Lower bounds for sorting

The decision-tree model

We can view comparison sorts abstractly in terms of decision trees. A *decision tree* is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored. Figure 8.1 shows the decision tree corresponding to the insertion sort algorithm from Section 2.1 operating on an input sequence of three elements.

Lower bounds for sorting

The decision-tree model

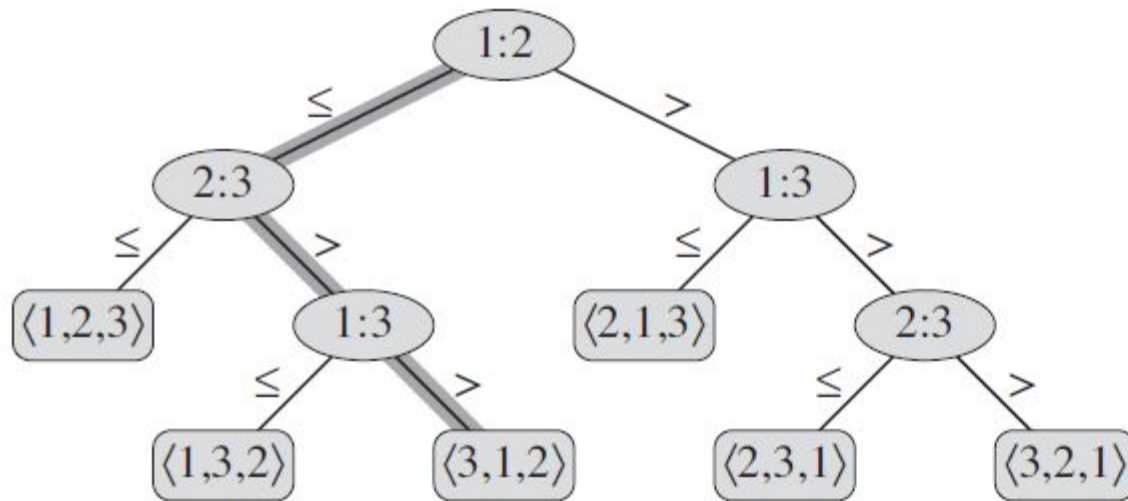


Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i:j$ indicates a comparison between a_i and a_j . A leaf annotated by the permutation $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, and so the decision tree must have at least 6 leaves.

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

Counting sort

Counting sort assumes that each of the n input elements is an integer in the range 0 to k , for some integer k . When $k = O(n)$, the sort runs in $\Theta(n)$ time.

Counting sort determines, for each input element x , the number of elements less than x . It uses this information to place element x directly into its position in the output array. For example, if 17 elements are less than x , then x belongs in output position 18. We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want to put them all in the same position.

In the code for counting sort, we assume that the input is an array $A[1..n]$, and thus $A.length = n$. We require two other arrays: the array $B[1..n]$ holds the sorted output, and the array $C[0..k]$ provides temporary working storage.

Counting sort

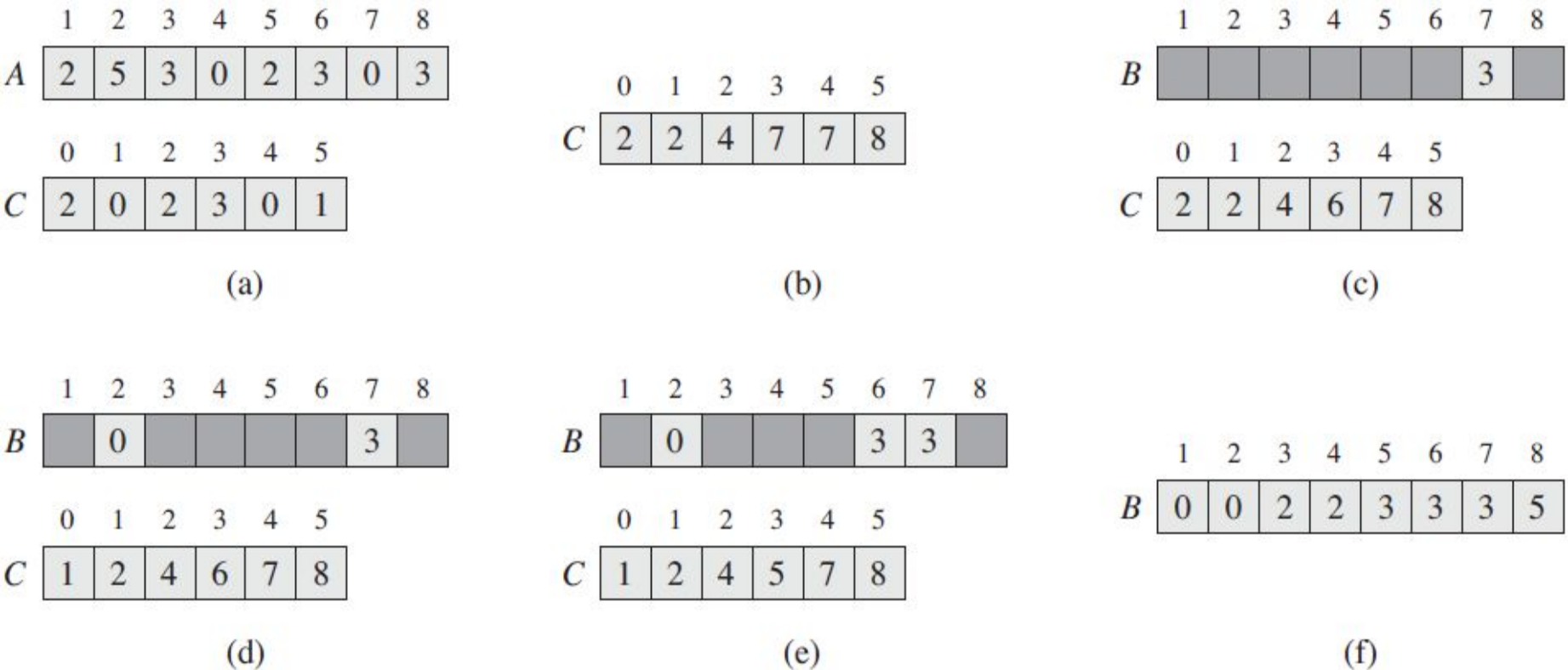


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. (a) The array A and the auxiliary array C after line 5. (b) The array C after line 8. (c)–(e) The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array B have been filled in. (f) The final sorted output array B .

Counting sort

COUNTING-SORT(A, B, k)

```
1  let  $C[0..k]$  be a new array
2  for  $i = 0$  to  $k$ 
3       $C[i] = 0$ 
4  for  $j = 1$  to  $A.length$ 
5       $C[A[j]] = C[A[j]] + 1$ 
6  //  $C[i]$  now contains the number of elements equal to  $i$ .
7  for  $i = 1$  to  $k$ 
8       $C[i] = C[i] + C[i - 1]$ 
9  //  $C[i]$  now contains the number of elements less than or equal to  $i$ .
10 for  $j = A.length$  downto 1
11      $B[C[A[j]]] = A[j]$ 
12      $C[A[j]] = C[A[j]] - 1$ 
```

How much time does counting sort require? The **for** loop of lines 2–3 takes time $\Theta(k)$, the **for** loop of lines 4–5 takes time $\Theta(n)$, the **for** loop of lines 7–8 takes time $\Theta(k)$, and the **for** loop of lines 10–12 takes time $\Theta(n)$. Thus, the overall time is $\Theta(k + n)$. In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$.

8.2-1

Using Figure 8.2 as a model, illustrate the operation of COUNTING-SORT on the array $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$.

Radix sort

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

Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

RADIX-SORT(A, d)

- 1 **for** $i = 1$ **to** d
- 2 use a stable sort to sort array A on digit i

Given n d -digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d(n + k))$ time if the stable sort it uses takes $\Theta(n + k)$ time.

When d is constant and $k = O(n)$, we can make radix sort run in linear time. More generally, we have some flexibility in how to break each key into digits.