CSC 323 Algorithm Design and Analysis
Instructor: Dr. Natarajan Meghanathan

Sample Questions for Module 2 - Classical Algorithm Design Techniques

2.1 Brute-Force

1) Determine the number of character comparisons made by the brute-force algorithm in searching for the pattern GOAT in the text below of length 47 characters.

```
THERE_IS_MORE_TO_LIFE_THAN_INCREASING_ITS_SPEED
```

The Search String is of length 47 characters; the Search Pattern is of length 4 characters. There will be 47-4+1 = 44 iterations of the algorithm. For 43 iterations, the first character comparison itself will fail, as the first character in the Search Pattern (character 'G') appears only once in the Search String. However, for the iteration starting with character 'G', the second character comparison will fail. Hence, there will be a total of 43*1 + 1*2 = 45 character comparisons.

2) How many comparisons (both successful and unsuccessful) are made by the brute-force string-matching algorithm in searching for each of the following patterns in the binary text of 1000 zeros?

a) 00001

The Search String (0000….000) is of length 1000; the Search Pattern (00001) is of length 5. Hence, there will be 1000-5+1 = 996 iterations. In each of these iterations, the first 4 comparisons would be successful and the last comparison will be unsuccessful. Hence, there will be 996*4 = 3,984 successful comparisons and 996*1 = 996 unsuccessful comparisons. Total comparisons = 3984 + 96 = 4980 comparisons.

b) 10000

There will be a total of 1000-5+1 = 996 iterations. In each of these iterations, the first comparison would itself be unsuccessful. Hence, there will be 996*1 = 996 unsuccessful comparisons and there will not be any successful comparisons. Total comparisons = 996.

c) 01010

There will be a total of 1000-5+1 = 996 iterations. In each of these iterations, the first comparison would be successful and the second comparison would be unsuccessful. Hence, there will be 996*1 = 996 successful comparisons and another 996*1 = 996 unsuccessful comparisons. Total comparisons = 1992.

3) Consider the problem of counting, in a given text, the number of substrings that start with an A and end with a B. (For example, there are 9 such substrings in DAAXBABAGBD). Design a $\Theta(n)$ algorithm to count such substrings.

Note that the number of desired substrings that end with a B at a given position i ($0 < i \leq n-1$) in the text is equal to the number of A's to the left of that position. This leads to the following algorithm:

Initialize the number of A's encountered and the number of desired substrings encountered to 0. Scan the text from left to right. When an A is encountered, increment the number of A's encountered. When a B is encountered, increment the number of desired substrings encountered by the current value of the number of A's encountered. When the text is exhausted, return the last value of the number of substrings encountered. Since, we do a linear pass on the text and spends constant time on each of its characters, the algorithm is linear.

		D	A	A	X	B	A	B	A	G	B	D
# A's	0	0	1	2	2	2	3	3	4	4	4	4
# desired substrings	0	0	0	0	0	2	2	5	5	5	9	**9**

2.2 Decrease and Conquer

1) Apply Insertion Sort to sort the list A L G O R I T H M S in alphabetical order.

A L G O R I T H M S
A | L G O R I T H M S
A L | G O R I T H M S
A G L | O R I T H M S
A G L O | R I T H M S
A G L O R | I T H M S
A G I L O R | T H M S
A G I L O R T | H M S
A G H I L O R T | M S
A G H I L M O R T | S
A G H I L M O R S T

2) Analyze the best-case, worst-case and average-case time complexity of Insertion sort.

Best Case (if the array is already sorted): the element v at A[i] will be just compared with A[i-1] and since A[i-1] ≤ A[i] = v, we retain v at A[i] itself and do not scan the rest of the sequence A[0…i-1]. There is only one comparison for each value of index i. $\sum_{i=1}^{n-1} 1 = n = \Theta(n)$

Worst Case (if the array is reverse-sorted): the element v at A[i] has to be moved all the way to index 0, by scanning through the entire sequence A[0…i-1].

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i-1}^{0} 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$$

Average Case: On average for a random input sequence, we would be visiting half of the sorted sequence A[0…i-1] to put A[i] at the proper position.

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i-1}^{(i-1)/2} 1 = \sum_{i=1}^{n-1} \frac{(i-1)}{2} + 1 = \sum_{i=1}^{n-1} \frac{(i+1)}{2} = \Theta(n^2)$$

Overall Time Complexity: $O(n^2)$

$$\lim_{n->\infty} \frac{Best-case}{Worst-case} = \lim_{n->\infty} \frac{n-1}{\left(\frac{n(n-1)}{2}\right)} = \lim_{n->\infty} \frac{2}{n} = 0$$

2.3 Divide and Conquer

1) Solve the following recurrence relations (using Master Theorem):

a) $T(n) = 3T(n/2) + n^2$
$a = 3; b = 2; d = 2$
$b^d = 2^2 = 4$
$a = 3 < b^d = 4$
Hence, $T(n) = \Theta(n^2)$

b) $T(n) = 3T(n/3) + \sqrt{n}$
$T(n) = 3T(n/3) + n^{(1/2)}$
$a = 3; b = 3; d = 1/2$
$b^d = 3^{1/2} = 1.732$
$a = 3 > b^d = 1.732$
$T(n) = \Theta(n^{\log_3 3}) = \Theta(n)$

c) $T(n) = 4T(n/2) + \log n$
$a = 4; b = 2; d < 1$, because $\log n < n^1$
$b^d = 2^{<1} < 2$
$a > b^d$
$T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$

d) $T(n) = 6T(n/3) + n^2 \log n$
$a = 6; b = 3; 2 < d < 3$, because $\log n < n$ and hence $n^2 \log n < n^3$
$b^d = 3^{2<d<3} > 9 > a$
$a < b^d$
Hence, $T(n) = \Theta(n^d) = \Theta(n^2 \log n)$

2) Consider the merge sort algorithm. Solve for the worst-case number of comparisons in the algorithm.

The recurrence relation for the number of key comparisons C(n) is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, C(1) = 0$$

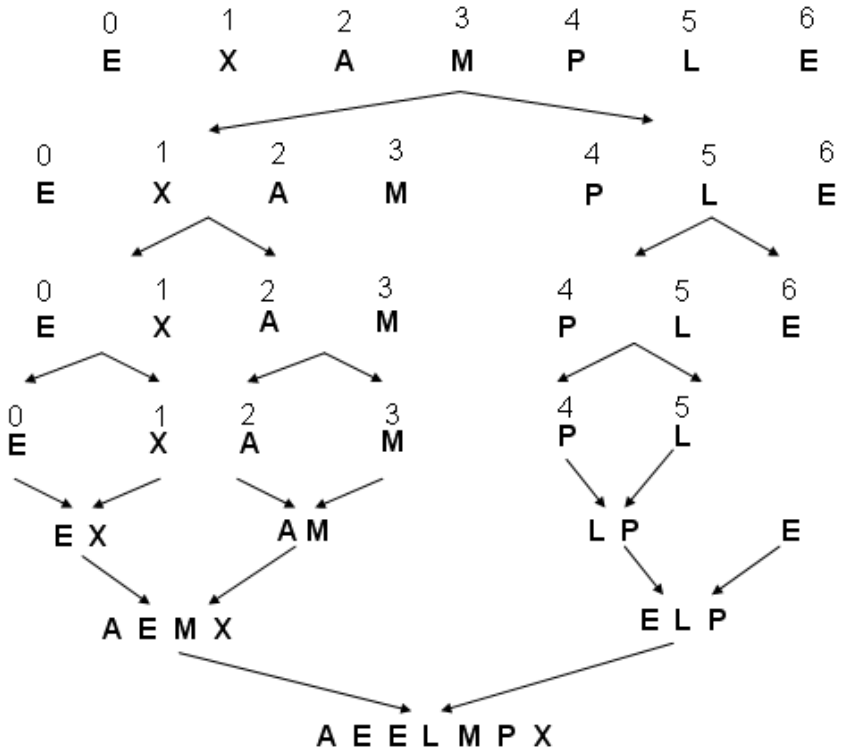
At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needed to be processed is reduced by one. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays). Hence, the worst case of $C_{merge}(n) = n - 1$ for $n > 1$.

$$C(n) = 2C(n/2) + (n-1) \qquad f(n) = n - 1 = \Theta(n)$$

Hence, according to Master Theorem,

$$a = 2, b = 2, d = 1 \qquad C(n) \in \Theta(n \log n)$$
$$a = b^d$$

3) Apply Merge sort to the list E, X, A, M, P, L, E to sort the list in alphabetical order.



4) Develop a divide and conquer algorithm to find the position (index) of the largest element in an array of *n* integers. Illustrate the working of your algorithm by executing it on the following array. If there is more than one occurrence of the largest element in your array, where is the index of the largest element returned by your algorithm located?

0	1	2	3	4	5	6	7	8	9
30	40	10	50	23	60	12	33	21	60

The idea is to divide an array of size n into two sub-arrays of size n/2 each and find the index of the maximum element within the sub-arrays using a recursive approach. To conquer the solution, we compare the values of the elements that are the largest in the two sub-arrays and the larger of the two is the largest element within the composite array of the two sub-arrays. The pseudo code of a divide-and-conquer algorithm based on the above idea is given below:

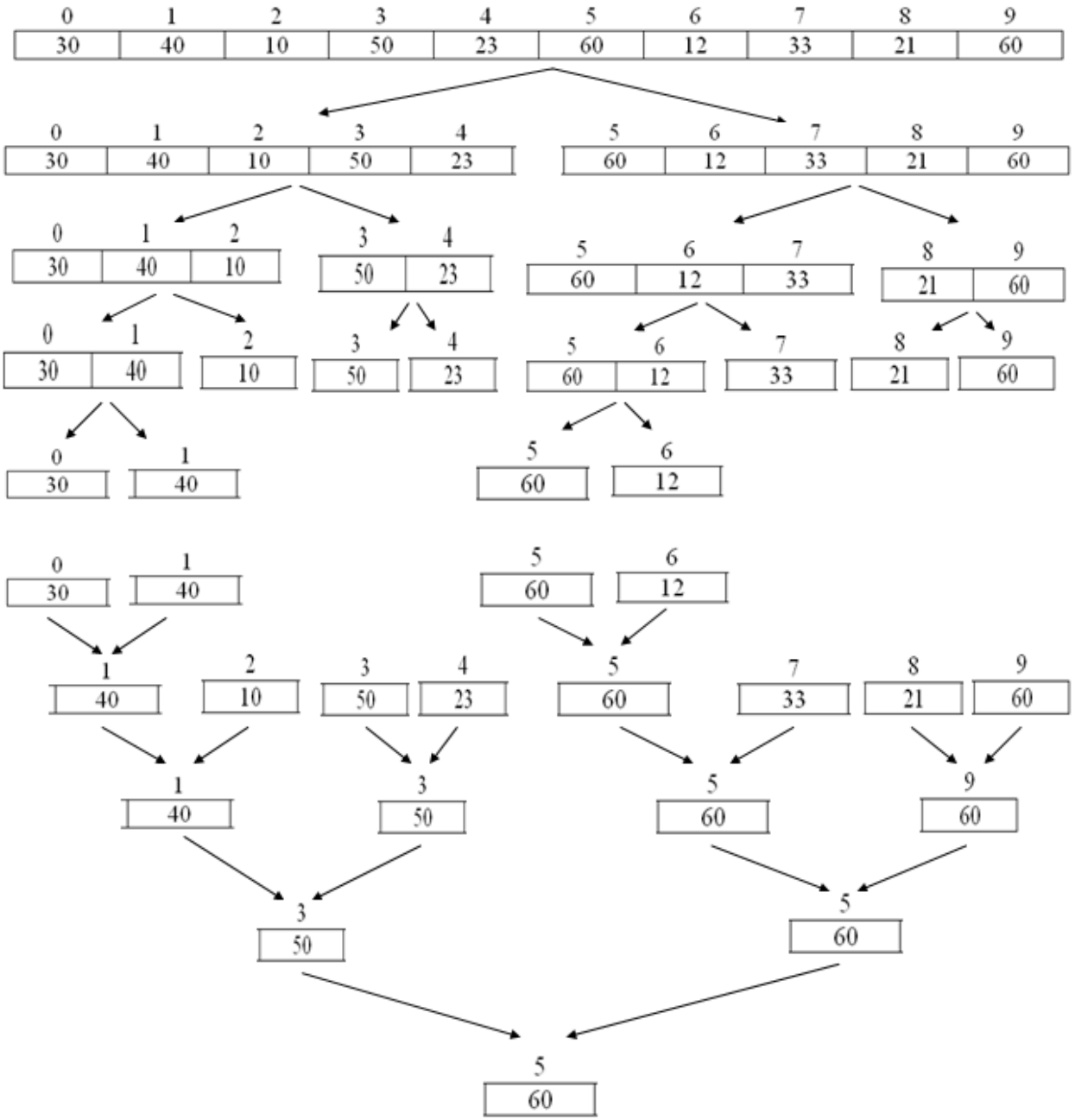
Call **Algorithm** $MaxIndex(A, 0, n-1)$ where

Algorithm $MaxIndex(A, l, r)$
//Input: A portion of array $A[0..n-1]$ between indices l and r $(l \le r)$
//Output: The index of the largest element in $A[l..r]$
if $l = r$ **return** l
else $temp1 \leftarrow MaxIndex(A, l, \lfloor (l+r)/2 \rfloor)$
  $temp2 \leftarrow MaxIndex(A, \lfloor (l+r)/2 \rfloor + 1, r)$
  if $A[temp1] \ge A[temp2]$
    return $temp1$
  else return $temp2$

Comparison of the elements in the different indices is the basic operation. The number of comparisons needed to find the index with the largest element in an array of size n is the number of comparisons needed to find the index with the largest element in the two sub-arrays, each of size n/2 and 1 more comparison to divide the larger among the two elements corresponding to the largest element in the two sub-arrays. Hence, the recursion is: $C(n) = 2\,C(n/2) + 1$ for $n > 1$ and $C(1) = 0$

Solving the above recurrence using Master Theorem, $a = 2$; $b = 2$; $d = 0$ ➔ $a > b^d$. Hence, $C(n) = \Theta(n^{\log_2 2}) = \Theta(n)$.

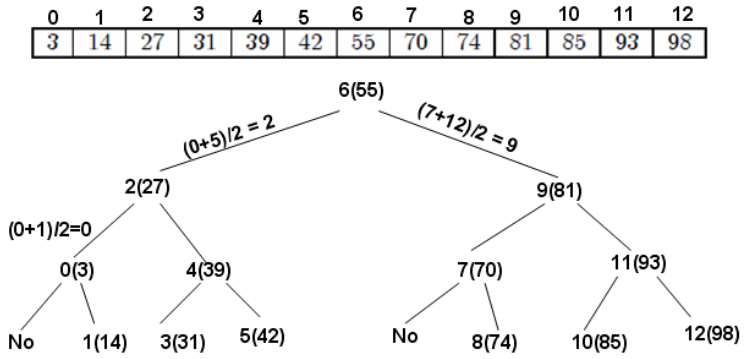
Example



5) Consider the following 13-element sorted array.

3	14	27	31	39	42	55	70	74	81	85	93	98

a) List all the keys of this array that will require the largest number of key comparisons when searched for by binary search.
b) Find the average number of key comparisons made by binary search in a successful search in this array. Assume that each key is searched with the same probability.
c) Find the average number of key comparisons made by binary search in an unsuccessful search in this array. Assume that searches for keys in each of the 14 intervals formed by the array's elements are equally likely.



a) The keys that will require the maximum number of comparisons (4) are the ones at positions (index values) 1, 3, 5, 8, 10 and 12.
b) There will be:
1 comparison for key at position 6,
2 comparisons for keys at positions 2 and 9,
3 comparisons for keys at positions 0, 4, 7 and 11
4 comparisons for keys at positions 1, 3, 5, 8, 10 and 12
Hence, the average number of key comparisons for a successful search is 1* (1/13) + 2 * (2/13) + 3 * (4/13) + 4 * (6/13) = 41/13 = 3.15.
c) Three comparisons are required to do an unsuccessful search for a key that is less than the value of the key at position 0 and to search for a key that is in between the values of the keys at positions 6 and 7. For the remaining 12 of the 14 intervals, there will be 4 comparisons incurred for an unsuccessful key search. Hence, the average number of key comparisons for an unsuccessful search is 3*(2/14) + 4*(12/14) = 54/14 = 3.86.

6) Analyze the space-time tradeoff between Insertion Sort and Merge Sort.

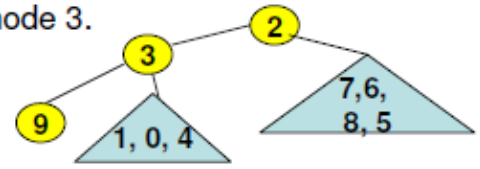
	Overall Time Complexity	Overall Space Complexity
Insertion Sort	$O(n^2)$	$\Theta(1)$
Merge Sort	$O(n\log n)$	$\Theta(n)$

Note that Insertion Sort is "in-place" because it requires only a constant amount of additional memory space for its execution, irrespective of the size of the input array to be sorted. On the other hand, Merge Sort is "not in-place" because its requirement for memory space grows with the size of the input array. In other words, if we want to sort a 100-element integer array, Merge Sort requires an additional memory space for 100 integers (in addition to the memory need to store the input array).

7) Draw a binary tree with 10 nodes labeled 0, 1, …, in such a way that the in-order and post-order traversals of the tree yield the following lists: 9, 3, 1, 0, 4, 2, 7, 6, 8, 5 (in-order) and 9, 1, 4, 0, 3, 6, 7, 5, 8, 2 (post-order).

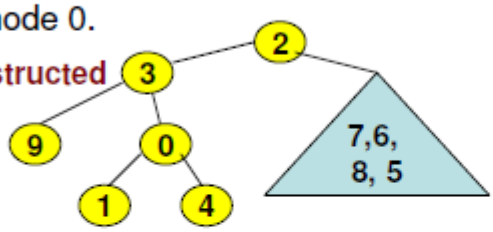
- Solution: Note that the post-order traversal always lists the root node of the binary tree as the last node. Hence node '2' is the root node of the binary tree. The in-order traversal lists nodes 9, 3, 1, 0, 4 as the nodes before node '2'. Hence these nodes are in the left sub tree of node 2 and nodes 7, 6, 8, 5 are in the right sub tree of node 2.
- Applying the above logic recursively to the left and right sub trees, we find that the post-order traversal lists the nodes (9, 3, 1, 0, 4) of the left sub tree in the order 9, 1, 4, 0, 3. Hence node 3 is the root node among these nodes. The in-order traversal lists nodes 1, 0, 4 after node 3. Hence, these three nodes constitute the right sub tree of node 3. And node 9 is in the left sub tree of node 3.

Tree constructed so far:

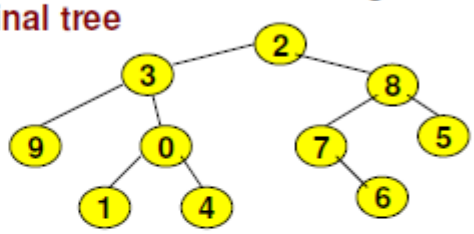


- The nodes (1, 0, 4) in the right sub tree of node 3 are listed in the post-order traversal as 1, 4, 0. Hence node 0 is the root of this sub tree. Node 0 is in between nodes 1 and 4 in the in-order list. Hence node 1 should be the left of node 0 and node 4 should be to the right of node 0.

Tree constructed so far:



Final tree



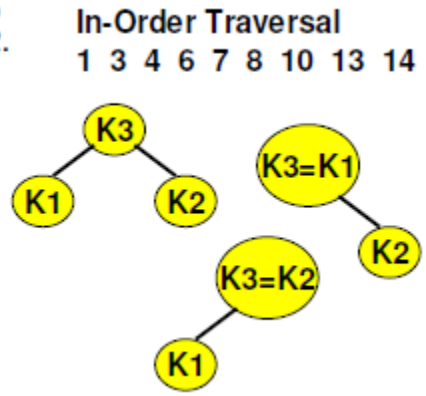
- Continuing our analysis on the right sub tree with nodes (7, 6, 8, 5), we notice that these nodes are listed in the post-order traversal as 6, 7, 5, 8. Hence node 8 should be the root. The position of node 8 in the above in-order list implies that nodes 7, 6 are in the left sub tree of node 8 and node 5 is to the right of node 8.
- Nodes (7, 6) in the left sub tree of node 8 are listed in the post-order traversal as 6, 7. Hence, node 7 should be the root node of this sub tree and according to the in-order list, node 6 should be to the right of node 7.

8) Prove that an in-order traversal of a binary search tree lists the keys of the nodes in the tree in a sorted order.

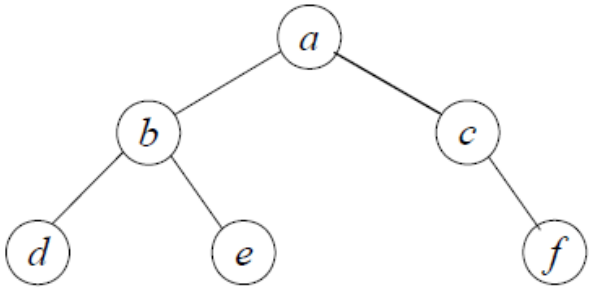
Proof: Let there be two keys K1 and K2 at two different nodes of a BST such that K1 < K2. Let K3 be the key located at their nearest common ancestor.

If K3 is different from K1 and K2, then the definition of the BST ensures that K1 and K2 are located in the left and right sub trees of K3 and that K1 is visited before visiting K2.

If K3 coincides with K1, then K2 is in the right sub tree of K1. Likewise, if K3 coincides with K2, then K1 is in the left sub tree of K1. Either way, an in-order traversal visits K1 before K2.



9) Traverse the following binary tree in (i) pre-order, (ii) in-order and (iii) post-order



Pre-order: a, b, d, e, c, f
In-order: d, b, e, a, c, f
Post-order: d, e, b, f, c, a

2.4 Transform and Conquer

1) Prove Euclid's GCD Formula: For any two integers m, n such that m > n, GCD(m, n) = GCD(n, m mod n).

Proof

- To prove GCD(m, n) = GCD(n, m mod n)
- Let *d* be an integer that divides both *m* and *n*.
- We need to prove that *d* also divides *n* and *m* mod *n*.
- Since *d* divides *m* and *n*, *d* also divides *m* – *n*;
  - Why? Let *m* = *q1*d*; *n* = *q2*d* for some integers *q1* and *q2*. So, *q1* – *q2* must also be an integer.
  - Then, *m* – *n* = (*q1* – *q2*) * *d* ➔ *d* divides (*m* – *n*).
- In fact, *d* divides (*m* – *q*n*) for any integer *q*.
  - Why? If *d* divides *n*; then *d* also divides *q*n* for some integer *q*.
  - From the above argument, if *d* divides *m* and *d* divides *q*n*, then *d* divides *m* – *q*n*.

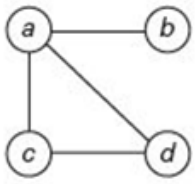
- The division of *m* by *n* can be represented as *m* = *q*n* + (*m* mod *n*) where (*m* mod *n*) is the remainder when *m* is divided by *n* and *q* is the quotient.

- From the above, since *d* divides (*m* – *q*n*), *d* also divides (*m* mod *n*).
- Hence *d* divides both *n* and (*m* mod *n*).
- Thus, any integer that divides both *m* and *n* also divides (*m* mod *n*). Thus, the GCD(*m*, *n*) = GCD(*n*, *m* mod *n*).

2) Find the GCD of 90 and 48 using the Euclid's formula. Using this GCD, also find the LCM of 90 and 48.

GCD(90, 48) = GCD(48, 42) = GCD(42, 6) = GCD(6, 0) = 6.
LCM(90, 48) = 90*48/GCD(90, 48) = 90*48/6 = 720.

3) Find the number of paths of length 2 between any two vertices in the following graph.

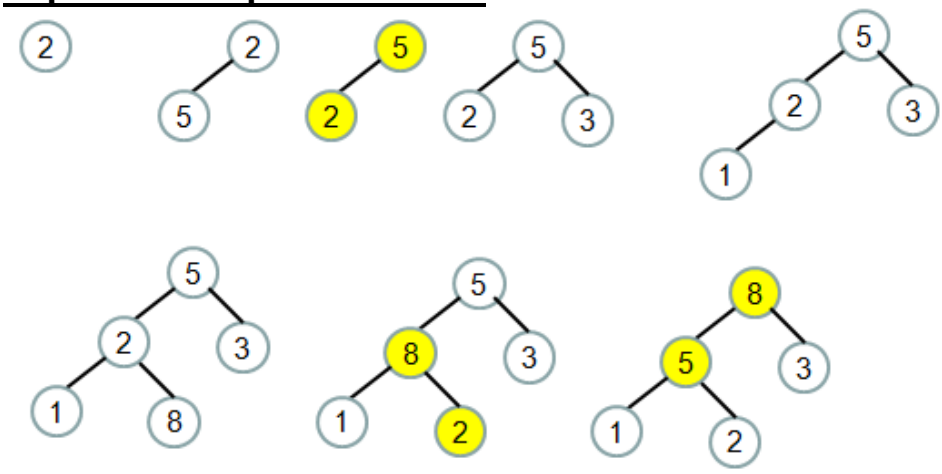


Solution:

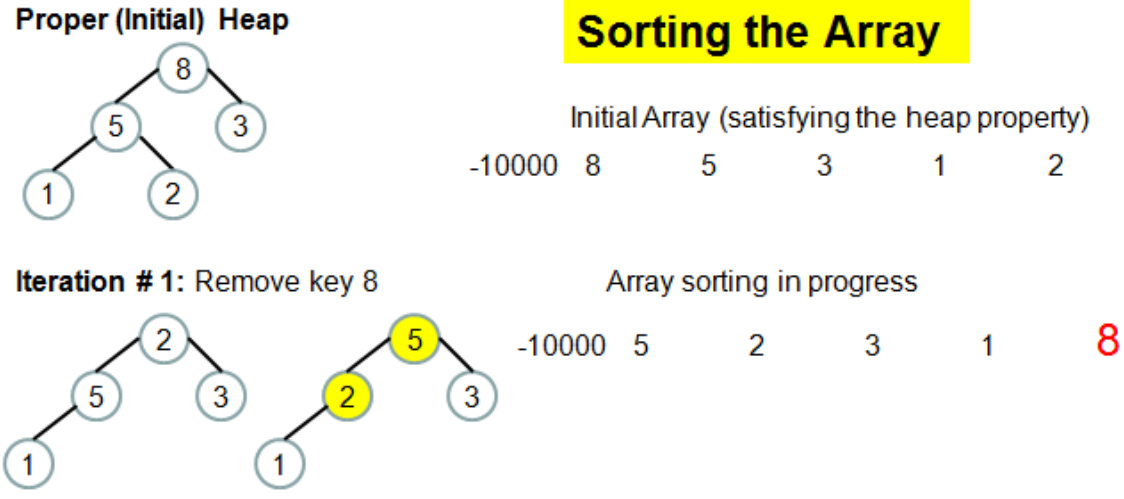
$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array} \qquad A^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix} \end{array}$$

4) Construct a heap for the array [2, 5, 3, 1, 8] using the top-down approach and sort the array using repeated key removal operations of the element at the top of the heap. Show all the steps

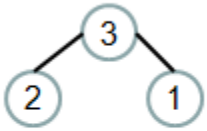
Top-Down Heap Construction



Sorting the Array



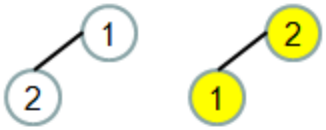
Iteration # 2: Remove key 5



Array sorting in progress

-10000 3 2 1 5 8

Iteration # 3: Remove key 3



Array sorting in progress

-10000 2 1 3 5 8

Iteration # 4: Remove key 2

Array sorting in progress

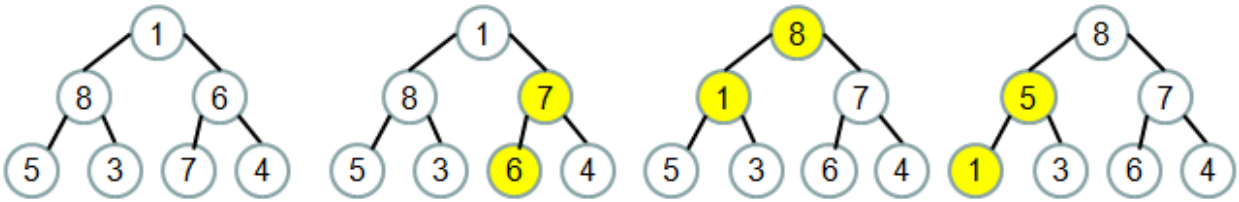
-10000 1 2 3 5 8

Iteration # 5: Remove key 1

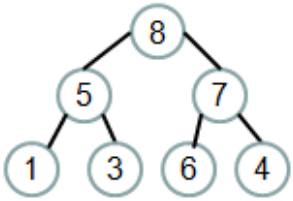
Final sorted array

-10000 1 2 3 5 8

5) Construct a heap for the array [1, 8, 6, 5, 3, 7, 4] using the bottm-up approach and sort the array using repeated key removal operations of the element at the top of the heap. Show all the steps



Proper (Initial) Heap

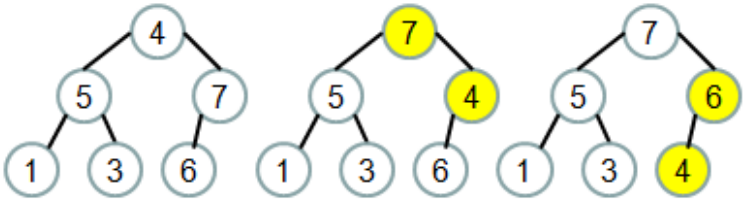


Sorting the Array

Initial Array (satisfying the heap property)

-10000 8 5 7 1 3 6 4

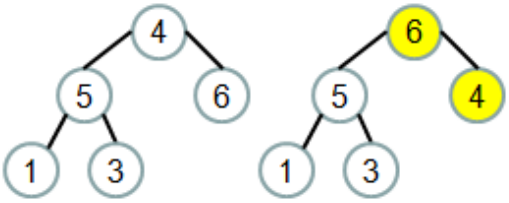
Iteration # 1: Remove key 8



Array sorting in progress

-10000 7 5 6 1 3 4 8

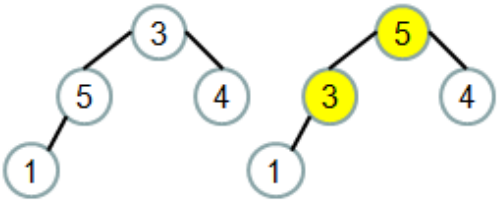
Iteration # 2: Remove key 7



Array sorting in progress

-10000 6 5 4 1 3 7 8

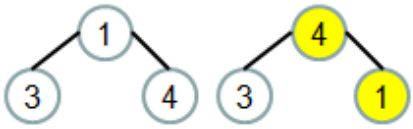
Iteration # 3: Remove key 6



Array sorting in progress

-10000 5 3 4 1 6 7 8

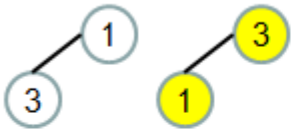
Iteration # 4: Remove key 5



Array sorting in progress

-10000 4 3 1 5 6 7 8

Iteration # 5: Remove key 4



Array sorting in progress

-10000 3 1 4 5 6 7 8

Iteration # 6: Remove key 3

Array sorting in progress

-10000 1 3 4 5 6 7 8

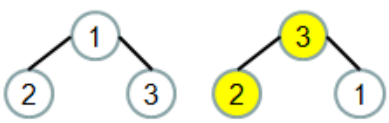
Iteration # 7: Remove key 1

Array sorting in progress

-10000 1 3 4 5 6 7 8

6) Show using a simple example that the heap constructed using the bottom-up approach and the top-down approach need not be the same.

Example
1, 2, 3



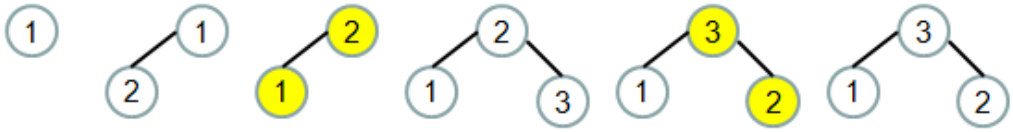
Bottom-Up Construction

Array (satisfying the heap property)

-10000 3 2 1

1, 2, 3

Top-Down Construction



Array (satisfying the heap property)

-10000	3	1	2

Thus, for a given input sequence, the arrays (satisfying the heap property) that are constructed using the bottom-up approach and the top-down approach need not always be the same, as observed in the above example.

2.5 Space-Time Tradeoffs

1) Compare the working principle and time complexity of the top-down and bottom-up heap construction strategies and when would you use each of them?

The **top-down heap construction** strategy works to construct a heap of a list of elements whose contents are not known a priori (i.e. not known beforehand). Whenever a new element is to be inserted to an existing heap, we add that new element as the rightmost element in the bottom-most level of the heap and re-heapify the binary tree (i.e., move the newly inserted element higher up in the tree if the element is greater than its parental node until the parent-child heap property is retained). The height of a binary tree forming the heap of n nodes is $\Theta(\log n)$. Hence, at the worst-case, a newly inserted node has to be moved up all the way from the leaf node level (bottom-most level) to the root of the tree (i.e. incur $\log n$ swappings). For an n-element list, it would then take $\Theta(n*\log n)$ time to construct the heap. The worst-case scenario would be if the input array to be sorted is already-sorted 1, 2, 3, 4, … Every new element to be inserted would incur a $\Theta(\log n)$ time to restore the parent-child node heap property.

The **bottom-up heap construction** strategy works by constructing a binary tree of the initial array in the top-down, right-to-left at each level fashion. Each parental/internal node of the binary tree (i.e., half of the nodes in the binary tree) is then visited to ensure that its value is greater than or equal to its two children. If the value of a parental node of a sub tree is lower than at least one of its two immediate children, then the parental node is swapped with the larger of its two children and this operation is recursively operated until the heap property of the sub tree rooted is restored. The above procedure is conducted at every parental node of the binary tree. The worst-case time complexity of the bottom-up heap construction strategy is $\Theta(n)$ time. Since the bottom-up heap construction strategy is more time-efficient compared to the top-down heap construction strategy, it is preferable when the contents of the array to be sorted are known a priori (i.e. known beforehand).

2) For the input sequence 30, 20, 56, 75, 31, 19 and hash function h(K) = K mod 11,

a. Construct the open hash table
b. Find the largest number of key comparisons in a successful search in this table
c. Find the average number of key comparisons in a successful search in this table

a.

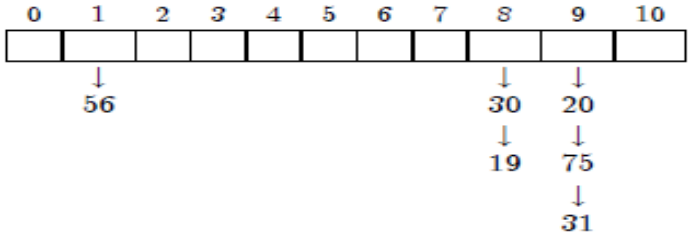
The list of keys: 30, 20, 56, 75, 31, 19

The hash function: $h(K) = K \bmod 11$

The hash addresses:

K	30	20	56	75	31	19
$h(K)$	8	9	1	9	9	8

The open hash table:



b. The largest number of key comparisons in a successful search in this table is 3 (in searching for $K = 31$).

c. The average number of key comparisons in a successful search in this table, assuming that a search for each of the six keys is equally likely, is

$$\frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 2 = \frac{10}{6} \approx 1.7.$$

3) **For the input sequence 30, 20, 56, 75, 31, 19 and hash function h(K) = K mod 11,**

d. Construct the closed hash table

e. Find the largest number of key comparisons in a successful search in this table

f. Find the average number of key comparisons in a successful search in this table

a.

The list of keys: 30, 20, 56, 75, 31, 19

The hash function: $h(K) = K \bmod 11$

The hash addresses:

K	30	20	56	75	31	19
$h(K)$	8	9	1	9	9	8

0	1	2	3	4	5	6	7	8	9	10
								30		
								30	20	
	56							30	20	
	56							30	20	75
31	56							30	20	75
31	56	19						30	20	75

b. The largest number of key comparisons in a successful search is 6 (when searching for $K = 19$).

c. The average number of key comparisons in a successful search in this table, assuming that a search for each of the six keys is equally likely, is

$$\frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 6 = \frac{14}{6} \approx 2.3.$$

4) Find the probability of all *n* keys being hashed to the same cell of a hash table of size *m* if the hash function distributes keys evenly among all the cells of the table.

The probability of all n keys to be hashed to a particular address is equal to $\left(\frac{1}{m}\right)^n$. Since there are m different addresses, the answer is $\left(\frac{1}{m}\right)^n m = \frac{1}{m^{n-1}}$.

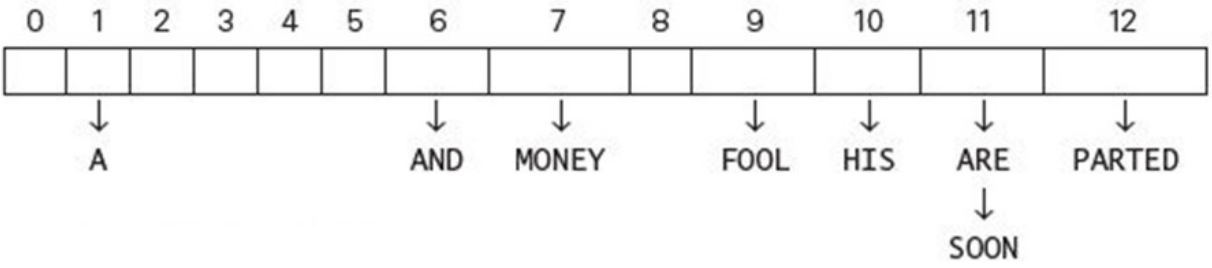
5) Consider the following table that defines the numerical equivalence of each character in the set of alphabets A – Z. Determine an open hashing table and a closed hashing table for the sequence of words given after the table

A – 1	D – 4	G – 7	J – 10	M – 13	P – 16	S – 19	V – 22	Y – 25
B – 2	E – 5	H – 8	K – 11	N – 14	Q – 17	T – 20	W – 23	Z – 26
C – 3	F – 6	I – 9	L – 12	O – 15	R – 18	U – 21	X – 24	

A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

Open Hashing Table

keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12



Closed Hashing Table

keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A								FOOL			
	A					AND			FOOL			
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	
	A					AND	MONEY		FOOL	HIS	ARE	SOON
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON

6) Mention one critical advantage of open hashing over closed hashing. What is the significance of a load factor in open hashing?

With open hashing, the number of elements (n) in the input sequence can exceed the size of the hash table (m). With closed hashing, n <= m. The load factor in open hashing is a measure of the length of the linked list for a particular entry in the hash table. The average number of key comparisons for a successful search of an element in the hash table is the (load factor)/2. The worst-case number of comparisons for an unsuccessful search is (load factor).

7) How would you use the hash table approach to determine whether every element in an input sequence is unique or not? What is the worst-case time complexity of the approach?

Determine the hash values of the elements in the input sequence, one followed by the other. If the hash value of two different elements is the same, then we have to resolve the collision by going through the linked list and insert the new element at the end of the list (in the case of open hashing) or going through the adjacent cells in the table and insert the new element at the first empty cell that is encountered starting

from the cell to which we hashed to (in the case of closed hashing). During the traversal of the linked list in a open hash table or the adjacent cells in a closed hash table, if we encounter an element that is equal to the new element that we are trying to insert, then the input sequence does not have unique elements. At the worst case, if all the elements in the input sequence hash to the same value, we will have to $\Theta(n)$ key comparisons for each element and for an *n*-element sequence, the worst-case time complexity would be $\Theta(n^2)$ key comparisons.

8) What is lazy deletion? Why is it needed for closed hashing?

With deletions in a Closed Hashing Table, if we simply delete a key, then we may not be able to successfully search for a key that has the same hash value as that of the key being deleted. With Lazy Deletion, the previously occupied locations of the deleted keys (these locations are available for new insertions) can be marked by a special symbol to distinguish them from