Chapter 2

Sorting Algorithms

LEARNING OBJECTIVES

- *Sorting algorithms*
- *Merge sort*
- *Bubble sort*
- *Insertion sort*
- *Selection sort*
- *Selection sort algorithm*
- *Binary search trees*
- *Heap sort*
- *Sorting–performing delete max operations*
- *Max-heap property*
- *Min-heap property*
- *Priority queues*

Sorting Algorithms

Purpose of sorting

Sorting is a technique which reduces problem complexity and search complexity.

- Insertion sort takes $\theta(n^2)$ time in the worst case. It is a fast inplace sorting algorithm for small input sizes.
- Merge sort has a better asymptotic running time $\theta(n \log n)$, but it does not operate in inplace.
- Heap sort, sorts 'n' numbers inplace in $\theta(n \log n)$ time, it uses a data structure called heap, with which we can also implement a priority queue.
- Quick sort also sorts 'n' numbers in place, but its worst – case running time is $\theta(n^2)$. Its average case is $\theta(n \log n)$. The constant factor in quick sort's running time is small, This algorithm performs better for large input arrays.
- Insertion sort, merge sort, heap sort, and quick sort are all comparison based sorts; they determine the sorted order of an inputarray by comparing elements.
- We can beat the lower bound of Ω ($n \log n$) if we can gather information about the sorted order of the input by means other than comparing elements.
- The counting sort algorithm, assumes that the input numbers are in the set $\{1, 2, \ldots. k\}$. By using array indexing as a tool for determining relative order, counting sort can sort n numbers in $\theta(k + n)$ time. Thus counting sort runs in time that is linear in size of the input array.
- Radix sort can be used to extend the range of counting sort. If there are 'n' integers to sort, each integer has 'd' digits, and each digit is in the set $\{1, 2, \ldots k\}$, then radix sort can sort the numbers in $\theta(d(n + k))$ time. Where 'd' is constant. Radix sort runs in linear time.
- Bucket sort, requires knowledge of the probabilistic distribution of numbers in the input array.

Merge Sort

Suppose that our division of the problem yields 'a' sub problems, each of which is $\left(\frac{1}{b}\right)$th size of the original problem. For merge sort, both a and b are 2, but sometimes $a \neq b$. If we take $D(n)$ time to divide the problem into sub problems and $C(n)$ time to combine the solutions of the sub problems into the solution to the original problem. The recurrence relation for merge sort is

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

Running time is broken down as follows:

Divide: This step computes the middle of the sub array, which takes constant time $\theta(1)$.

Conquer: We solve 2 sub problems of size $(n/2)$ each recursively which takes $2T(n/2)$ time.

Combine: Merge sort procedure on an n-element sub array takes time $\theta(n)$.

- Worst case running time $T(n)$ of merge sort

$$T(n) = \begin{cases} 0(1) \text{ if } & n \le 1 \\ aT(n/2) + \theta(n) \text{ if } & n > 1 \end{cases}$$

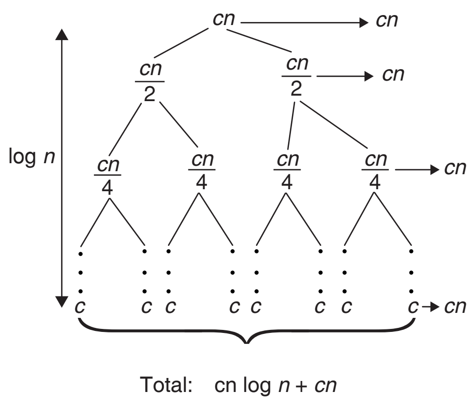


Figure 1 Recurrence tree

The top level has total cost 'cn', the next level has total cost $c(n/2) + c(n/2) = cn$ and the next level has total cost $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$ and so on. The ith level has total cost $2^i\ c\ (n/2^i) = cn$. At the bottom level, there are 'n' nodes, each contributing a cost of c, for a total cost of 'cn'. The total number of levels of the 'recursion tree' is $\log n + 1$.

There are $\log n + 1$ levels, each costing cn, for a total cost of $cn\ (\log n + 1) = cn \log n + cn$ ignoring the low–order term and the constant c, gives the desired result of $\theta(n \log n)$.

Bubble Sort

Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items, and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements 'bubble' to the top of the list.

Example: Take the array of numbers '5 1 4 2 8'and sort the array from lowest number to greatest number using bubble sort algorithm. In each step, elements underlined are being compared.

First pass:

$(\underline{5\quad 1}\quad 4\quad 2\quad 8) \rightarrow (1\quad 5\quad 4\quad 2\quad 8)$, here algorithm compares the first 2 elements and swaps them
$(1\quad \underline{5\quad 4}\quad 2\quad 8) \rightarrow (1\quad 4\quad 5\quad 2\quad 8)$, swap $(5 > 4)$
$(1\quad 4\quad \underline{5\quad 2}\quad 8) \rightarrow (1\quad 4\quad 2\quad 5\quad 8)$, swap $(5 > 2)$
$(1\quad 4\quad 2\quad \underline{5\quad 8}) \rightarrow (1\quad 4\quad 2\quad 5\quad 8)$, since these elements are already in order, algorithm does not swap them.

Second pass:

$(\underline{1\quad 4}\quad 2\quad 5\quad 8) \rightarrow (1\quad 4\quad 2\quad 5\quad 8)$
$(1\quad \underline{4\quad 2}\quad 5\quad 8) \rightarrow (1\quad 2\quad 4\quad 5\quad 8)$, swap since $(4 > 2)$
$(1\quad 2\quad \underline{4\quad 5}\quad 8) \rightarrow (1\quad 2\quad 4\quad 5\quad 8)$
$(1\quad 2\quad 4\quad \underline{5\quad 8}) \rightarrow (1\quad 2\quad 4\quad 5\quad 8)$

The array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third pass:

$(\underline{1\quad 2}\quad 4\quad 5\quad 8) \rightarrow (1\quad 2\quad 4\quad 5\quad 8)$
$(1\quad \underline{2\quad 4}\quad 5\quad 8) \rightarrow (1\quad 2\quad 4\quad 5\quad 8)$
$(1\quad 2\quad \underline{4\quad 5}\quad 8) \rightarrow (1\quad 2\quad 4\quad 5\quad 8)$
$(1\quad 2\quad 4\quad \underline{5\quad 8}) \rightarrow (1\quad 2\quad 4\quad 5\quad 8)$

Finally the array is sorted, and the algorithm can terminate.

Algorithm

```
void bubblesort (int a [ ], int n)
{
  int i, j, temp;
  for (i=0; i < n-1; i++)
  {
    for (j=0; j < n - 1 - i; j++)
    if (a [j] > a [j + 1])
    {
      temp = a [j + 1];
      a [j + 1] = a [j];
      a [j] = temp;
    }
  }
}
```

Insertion Sort

Insertion sort is a comparison sort in which the sorted array is built one entry at a time. It is much less efficient on large lists than more advanced algorithms such a quick sort, heap sort, (or) merge sort. Insertion sort provides several advantages.

- Efficient for small data sets.
- Adaptive, i.e., efficient for data set that are already substantially sorted. The complexity is $O(n + d)$, where d is the number of inversions.
- More efficient in practice than most other simple quadratic, i.e., $O(n^2)$ algorithms such as selection sort (or) bubble sort, the best case is $O(n)$.
- Stable, i.e., does not change the relative order of elements with equal keys.
- In-place i.e., only requires a constant amount $O(1)$ of additional memory space.
- Online, i.e., can sort a list as it receives it.

Algorithm

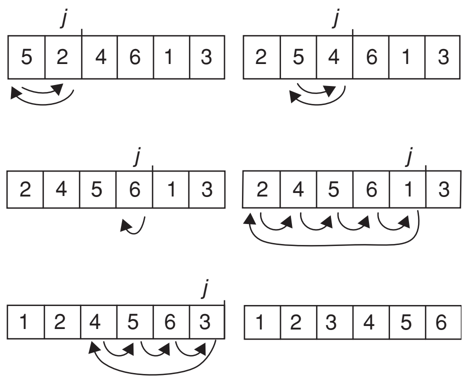
```
Insertion sort (A)
For (j ← 2) to length [A]
Do key ← A [j]
i ←j - 1;
While i > 0 and A [i] > key
{
Do A [i + 1] ← A [i]
i ← i - 1
}
A [i + 1] ← key
```

Every repetition of insertion sort removes an element from the input data, inserting it into the correct position in the already sorted list, until no input element remains. Sorting is typically done in–place. The resulting array after K iterations has the property where the first $k + 1$ entries are sorted. In each iteration the first remaining entry of the input is removed, inserted into the result at the correct position, with each element greater than X copied to the right as it is compared against. X.

Performance

- The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., $\theta(n)$).
- The worst case input is an array sorted in reverse order. In this case every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. For this case insertion sort has a quadratic running time ($O(n^2)$).
- The average case is also quadratic, which makes insertion sort impractical for sorting large arrays, however, insertion sort is one of the fastest algorithms for sorting very small arrays even faster than quick sort.

Example: Following figure shows the operation of insertion sort on the array $A = (5, 2, 4, 6, 1, 3)$. Each part shows what happens for a particular iteration with the value of j indicated. j indexes the 'Current card' being inserted.



Read the figure row by row. Elements to the left of $A[j]$ that are greater than $A[j]$ move one position to the right and $A[j]$ moves into the evacuated position.

Selection Sort

Selection sort is a sorting algorithm, specifically an in-place comparison sort. It has $O(n^2)$ complexity, making it inefficient on large lists.
The algorithm works as follows:

1. Find the minimum value in the list.
2. Swap it with the value in the first position.
3. Repeat the steps above for the remainder of the list (starting at the second position and advancing each time).

Analysis

Selection sort is not difficult to analyze compared to other sorting algorithms, since none of the loops depend on the data in the array selecting the lowest element requires scanning all n elements (this takes $n - 1$ comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining $n - 1$ elements and so on, for $(n - 1) + (n - 2) + \cdots + 2 + 1 = n(n - 1)/2 \in \theta(n^2)$ comparisons.

Each of these scans requires one swap for $n - 1$ elements (the final element is already in place).

Selection sort Algorithm

First, the minimum value in the list is found. Then, the first element (with an index of 0) is swapped with this value. Lastly, the steps mentioned are repeated for rest of the array (starting at the 2nd position).

Example 1: Here's a step by step example to illustrate the selection sort algorithm using numbers.

Original array: 6 3 5 4 9 2 7
1st pass $\rightarrow$ 2 3 5 4 9 6 7 (2 and 6 were swapped)
2nd pass $\rightarrow$ 2 3 5 4 9 6 7 (no swap)
3rd pass $\rightarrow$ 2 3 4 5 9 6 7 (4 and 5 were swapped)
4th pass $\rightarrow$ 2 3 4 5 6 9 7 (6 and 9 were swapped)
5th pass $\rightarrow$ 2 3 4 5 6 7 9 (7 and 9 were swapped)
6th pass $\rightarrow$ 2 3 4 5 6 7 9 (no swap)

Note: There are 7 keys in the list and thus 6 passes were required. However, only 4 swaps took place.

Example 2: Original array: LU, KU, HU, LO, SU, PU
1st pass $\rightarrow$ HU, KU, LU, LO, SU, PU
2nd pass $\rightarrow$ HU, KU, LU, LO, SU, PU
3rd pass $\rightarrow$ HU, KU, LO, LU, SU, PU
4th pass $\rightarrow$ HU, KU, LO, LU, SU, PU
5th pass $\rightarrow$ HU, KU, LO, LU, PU, SU

Note: There were 6 elements in the list and thus 5 passes were required. However, only 3 swaps took place.

Binary Search Trees

Search trees are data structures that support many dynamic, set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE. A search tree can be used as a dictionary and as a priority Queue. Operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with 'n' nodes, basic operations run in $\theta(\log n)$ worst-case time. If the tree is a linear chain of 'n' nodes, the basic operations take $\theta(n)$ worst-case time.

A binary search tree is organized, in a binary tree such a tree can be represented by a linked data structure in which each node is an object. In addition to key field, each node contains fields left, right and P that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If the child (or) parent is missing, the appropriate field contains the value NIL. The root node is the only node in the tree whose parent field is NIL.

Binary search tree property

The keys in a binary search tree are always stored in such a way as to satisfy the binary search tree property.

Let ‘*a*’ be a node in a binary search tree. If ‘*b*’ is a node in the left sub tree of ‘*a*’, key [*b*] ≤ key [*a*]

If ‘*b*’ is a node in the right sub tree of ‘*a*’ then key [*a*] ≤ key [*b*].

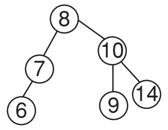


Figure 2 Binary search tree.

The binary search tree property allows us to print out all keys in a binary search tree in sorted order by a simple recursive algorithm called an inorder tree.

Algorithm

```
INORDER-TREE-WALK (root [T])
    INORDER-TREE-WALK (a)
    1. If $a \neq$ NIL
    2. Then INORDER-TREE-WALK (left [a])
    3. Print key [a]
    4. INORDER-TREE-WALK (right [a])
```

It takes $\theta(n)$ time to walk an n-node binary search tree, since after the initial call, the procedure is called recursively twice for each node in the tree.

Let $T(n)$ denote the time taken by IN-ORDER-TREE-WALK, when it is called on the root of an n-node subtree.

INORDER-TREE-WALK takes a small, constant amount of time on an empty sub-tree (for the test $x \neq$ NIL).

So $T(1) = C$ for some positive constant C.

For $n > 0$, suppose that INORDER-TREE-WALK is called on a node ‘*a*’ whose left subtree has k nodes and whose right subtree has $n - k - 1$ nodes.

The time to perform in order traversal is $T(n) = T(k) + T(n - k - 1) + d$.

For some positive constant ‘*d*’ that reflects the time to execute in-order (a), exclusive of the time spent in recursive calls $T(n) = (c + d)\,n + c$.

For $n = 0$, we have $(c + d)\,0 + c = T(0)$,

For $n > 0$,

$$\begin{aligned} T(n) &= T(k) + T(n - k - 1) + d \\ &= ((c + d)(k + c) + ((c + d)\,(n - k - 1) + c) + d \\ &= (c + d)\,n + c - (c + d) + c + d = (c + d)n + c \end{aligned}$$

HEAP SORT

Heap sort begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the partially sorted array. After removing the largest item, it reconstructs heap, removes the largest remaining item, and places, it in the next open position from the end of the partially sorted array. This is repeated until there are no items left in the heap and the sorted array is full. Elementary implementations require two arrays one to hold the heap and the other to hold the sorted elements.

- Heap sort inserts the input list elements into a binary heap data structure. The largest value (in a max-heap) or the smallest value (in a min-heap) is extracted until none remain, the value having been extracted in sorted order.

Example: Given an array of 6 elements: 15, 19, 10, 7, 17, 16, sort them in ascending order using heap sort.

Steps:

1. Consider the values of the elements as priorities and build the heap tree.
2. Start delete Max operations, storing each deleted element at the end of the heap array.

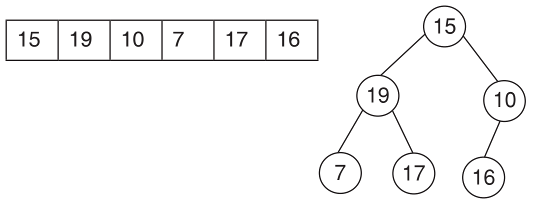
If we want the elements to be sorted in ascending order, we need to build the heap tree in descending order-the greatest element will have the highest priority.

1. Note that we use only array, treating its parts differently,
2. When building the heap-tree, part of the array will be considered as the heap, and the rest part-the original array.
3. When sorting, part of the array will be the heap and the rest part-the sorted array.

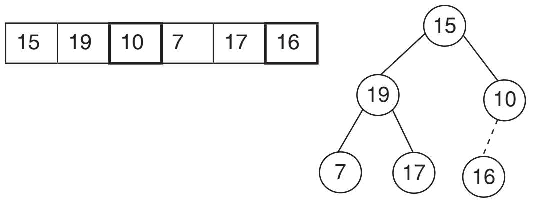
Here is the array: 15, 19, 10, 7, 17, 6.

Building the Heap Tree

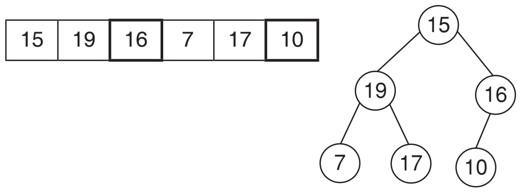
The array represented as a tree, which is complete but not ordered.



Start with the right most node at height 1 – the node at position 3 = size/2. It has one greater child and has to be percolated down.

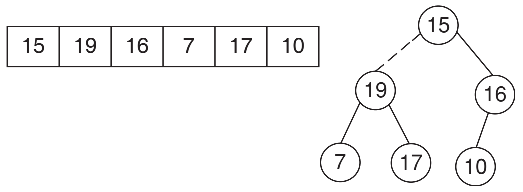


After processing array [3] the situation is:

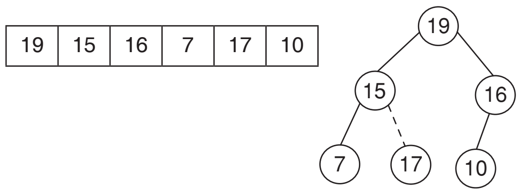


Next comes array [2]. Its children are smaller, so no percolation is needed.

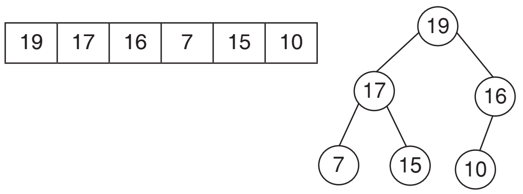
The last node to be processed is array[1]. Its left child is the greater of the children. The item at array [1] has to be percolated down to the left, swapped with array [2].



As a result:



The children of array [2] are greater and item 15 has to be moved down further, swapped with array [5].

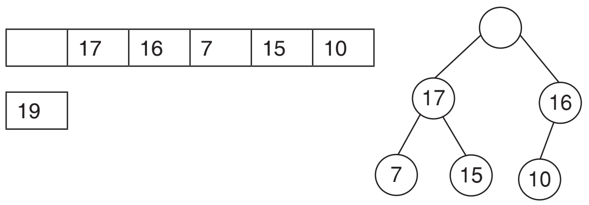


Now the tree is ordered, and the binary heap is built.

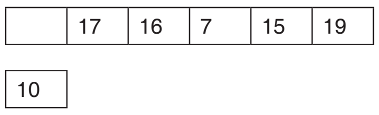
Sorting-performing Delete Max Operations

Delete the top element

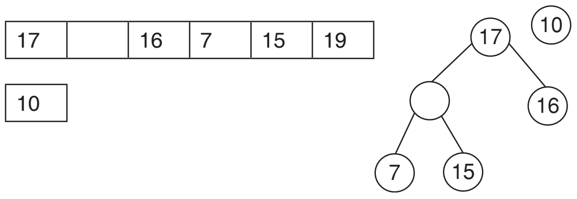
Store 19 in a temporary place, a hole is created at the top.



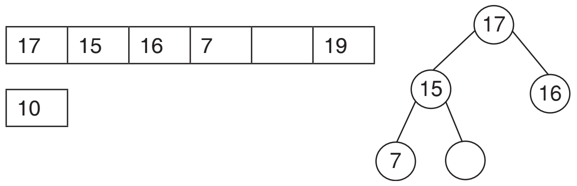
Swap 19 with the last element of the heap. As 10 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array



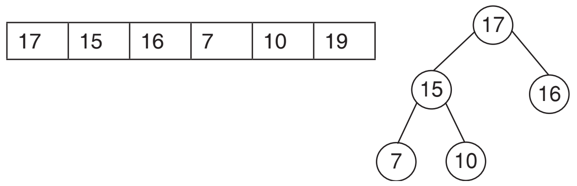
Percolate down the hole



Percolate once more (10 is less than 15, so it cannot be inserted in the previous hole)



Now 10 can be inserted in the hole



Repeat the step *B* till the array is sorted.

Heap sort analysis

Heap sort uses a data structure called (binary) heap binary, heap is viewed as a complete binary tree. An Array A that represents a heap is an object with 2 attributes: length [*A*], which is the number of elements in the array and heap size [*A*], the number of elements in the heap stored within array *A*.

No element past *A* [heap size [*A*]], where heap size [*A*] ≤ length [*A*], is an element of the heap.

There are 2 kinds of binary heaps:

1. Max-heaps
2. Min-heaps

In both kinds the values in the nodes satisfy a heap-property.

Max-heap property *A*[PARENT (i)] ≥*A*[*i*]

The value of a node is almost the value of its parent. Thus the largest element in a max-heap is stored at the root, and the sub tree rooted at a node contains values no larger than that contained at the node itself.

Min-heap property For every mode '*i*' other than the root [PARENT (i)] ≤ *A*[*i*]. The smallest element in a min-heap is at the root.

Max-heaps are used in heap sort algorithm.
Min-heaps are commonly used in priority queues.

Basic operations on heaps run in time almost proportional to the height of the tree and thus take $O(\log n)$ time

- MAX-HEAPIFY procedure, runs in $O(\log n)$ time.
- BUILD-MAX-HEAP procedure, runs in linear time.
- HEAP SORT procedure, runs in $O(n \log n)$ time, sorts an array in place.
- MAX-HEAP-INSERT
  HEAP- EXTRACT-MAX
  HEAP-INCREASE-KEY
  HEAP-MAXIMUM

  All these procedures, run in $O(\log n)$ time, allow the heap data structure to be used as a priority queue.
- Each call to MAX-HEAPIFY costs $O(\log n)$ time, and there are $O(n)$ such calls. Thus, the running time is $O(n \log n)$
- The HEAPSORT procedure takes time $O(n \log n)$, since the call to BUILD-MAX-HEAP takes time $O(n)$ and each of the $(n-1)$ calls to MAX-HEAPIFY takes time $O(\log n)$.

Priority Queues

The most popular application of a heap is its use as an efficient priority queue.

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. A max-priority queue supports the following operations:

INSERT: INSERT (s, x) inserts the element x into the set S. This operation can be written as $S \leftarrow S\ U\ \{x\}$.

MAXIMUM: MAXIMUM (S) returns the element of S with the largest key

EXTRACT-MAX: EXTRACT-MAX(S) removes and returns the element of S with the largest key.

INCREASE-KEY: INCREASE-KEY(s, x, k) increases the value of element x's key to the new value k, which is assumed to be atleast as large as x's current key value.

One application of max–priority queue is to schedule jobs on a shared computer.

Exercises

Practice Problems 1

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. Solve the recurrence relation $T(n) = 2T(n/2) + k.n$ where k is constant then $T(n)$ is

(A) $O(\log n)$ (B) $O(n \log n)$
(C) $O(n)$ (D) $O(n^2)$

2. What is the time complexity of the given code?

```
Void f(int n)
{
if (n > 0)
f (n/2);
}
```

(A) $\theta(\log n)$ (B) $\theta(n \log n)$
(C) $\theta(n^2)$ (D) $\theta(n)$

3. The running time of an algorithm is represented by the following recurrence relation;

$$T(n) = \begin{cases} n & n \le 3 \\ T\left[\dfrac{n}{3}\right] + cn & \text{otherwise} \end{cases}$$

What is the time complexity of the algorithm?

(A) $\theta(n)$ (B) $\theta(n \log n)$
(C) $\theta(n^2)$ (D) $\theta(n^2 \log n)$

Common data for questions 4 and 5:

4. The following pseudo code does which sorting?

```
xsort [A, n]
for j ← 2 to n
do key ← A [ i ]
i ← j – 1
While i > 0 and A [ i ] > key
do A [i + i] ← A [ i ]
i ← i – 1
A [i + 1] ← key
```

(A) Selection sort (B) Insertion sort
(C) Quick sort (D) Merge sort

5. What is the order of elements after 2 iterations of the above-mentioned sort on given elements?

8	2	4	9	3	6

(A)

2	4	9	8	3	6

(B)

2	4	8	9	3	6

(C)

2	4	6	3	8	9

(D)

2	4	6	3	8	9

Common data for questions 6 and 7:

6. The following pseudo code does which sort?

```
1. If n = 1 done
2. Recursively sort
     A [1…[n/2] ]and
     A [[n/2] + 1 … n]
3. Combine 2 ordered lists
```

(A) Insertion sort (B) Selection sort
(C) Merge sort (D) Quick sort

7. What is the complexity of the above pseudo code?
(A) $\theta(\log n)$ (B) $\theta(n^2)$
(C) $\theta(n \log n)$ (D) $\theta(2^n)$

8. Apply Quick sort on a given sequence 6 10 13 5 8 3 2 11. What is the sequence after first phase, pivot is first element?
(A) 5 3 2 6 10 8 13 11
(B) 5 2 3 6 8 13 10 11
(C) 6 5 13 10 8 3 2 11
(D) 6 5 3 2 8 13 10 11

9. Selection sort is applied on a given sequence:
89, 45, 68, 90, 29, 34, 17. What is the sequence after 2 iterations?
(A) 17, 29, 68, 90, 45, 34, 89
(B) 17, 45, 68, 90, 29, 34, 89
(C) 17, 68, 45, 90, 34, 29, 89
(D) 17, 29, 68, 90, 34, 45, 89

10. Suppose there are log n sorted lists of $\left\lfloor \frac{n}{\log n} \right\rfloor$ elements each. The time complexity of producing sorted lists of all these elements is: (hint: use a heap data structure)
(A) $\theta(n \log \log n)$ (B) $\theta(n \log n)$
(C) $\Omega(n \log n)$ (D) $\Omega(n^{3/2})$

11. If Divide and conquer methodology is applied on powering a Number X^n. Which one the following is correct?
(A) $X^n = X^{n/2} \cdot X^{n/2}$
(B) $X^n = X^{\frac{n-1}{2}} \cdot X^{\frac{n-1}{2}} . X$
(C) $X^n = X^{\frac{n+1}{2}} \cdot X^{\frac{n}{2}}$
(D) Both (A) and (B)

12. The usual $\theta(n^2)$ implementation of insertion sort to sort an array uses linear search to identify the position, where an element is to be inserted into the already sorted part of the array. If binary search is used instead of linear search to identify the position, the worst case running time would be.
(A) $\theta(n \log n)$
(B) $\theta(n^2)$
(C) $\theta(n(\log n)^2)$
(D) $\theta(n)$

13. Consider the process of inserting an element into a max heap where the max heap is represented by an array, suppose we perform a binary search on the path from the new leaf to the root to find the position for the newly inserted element, the number of comparisons performed is:
(A) $\theta(\log n)$ (B) $\theta(\log \log n)$
(C) $\theta(n)$ (D) $\theta(n \log n)$

14. Consider the following algorithm for searching a given number 'X' in an unsorted array $A[1 \cdots n]$ having 'n' distinct values:

```
(1) Choose an 'i' uniformly at random from 1 ··· n
(2) If A [i] = x
Then stop
else
goto(1);
```

Assuming that X is present in A, what is the expected number of comparisons made by the algorithm before it terminates.
(A) n (B) $n - 1$
(C) $2n$ (D) $n/2$

15. The recurrence equation for the number of additions $A(n)$ made by the divide and conquer algorithm on input size $n = 2^K$ is
(A) $A(n) = 2A(n/2)+ 1$ (B) $A(n) = 2A(n/2) + n^2$
(C) $A(n) = 2A(n/4) + n^2$ (D) $A(n) = 2A(n/8) + n^2$

Practice Problems 2

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1.

Input Array	Linear Search $W(n)$	Binary search $W(n)$
128 elements	128	8
1024 elements	1024	x

Find x value?
(A) 10 (B) 11
(C) 12 (D) 13

2. Choose the correct one
(i) $\log n$ (ii) n
(iii) $n \log n$ (iv) n^2

(a) A result of cutting a problem size by a constant factor on each iteration of the algorithm.
(b) Algorithm that scans a list of size 'n'.
(c) Many divide and conquer algorithms fall in this category.
(d) Typically characterizes efficiency of algorithm with two embedded loops.

(A) i – b, ii – c, iii – a, iv – d
(B) i – a, ii – b, iii – c, iv – d
(C) i – c, ii – d, iii – a, iv – b
(D) i – d, ii – a, iii – b, iv – c

3. Insertion sort analysis in worst case
(A) $\theta(n)$
(B) $\theta(n^2)$
(C) $\theta(n \log n)$
(D) $\theta(2^n)$

4. From the recurrence relation. Of merge sort $T(n) = 2T(n/2) + \theta(n)$.
Which option is correct?
I. $n/2$ II. 2T III. $\theta(n)$
(a) Extra work (divide and conquer)
(b) Sub-problem size
(c) Number of sub-problems
(A) III – b, II – a, I – c (B) I – b, II – c, III – a
(C) I – a, II – c, III – b (D) I – c, II – a, III – b

5. What is the number of swaps required to sort 'n' elements using selection sort, in the worst case?
(A) $\theta(n)$ (B) $\theta(n^2)$
(C) $\theta(n \log n)$ (D) $\theta(n^2 \log n)$

6. In a binary max heap containing 'n' numbers, the smallest element can be found in time
(A) $O(n)$ (B) $O(\log n)$
(C) $O(\log \log n)$ (D) $O(1)$

7. What is the worst case complexity of sorting 'n' numbers using quick sort?
(A) $\theta(n)$ (B) $\theta(n \log n)$
(C) $\theta(n^2)$ (D) $\theta(n\,!)$

8. The best case analysis of quick sort is, if partition splits the array of size n into
(A) $n/2 : n/m$ (B) $n/2 : n/2$
(C) $n/3 : n/2$ (D) $n/4 : n/2$

9. What is the time complexity of powering a number, by using divide and conquer methodology?
(A) $\theta(n^2)$ (B) $\theta(n)$
(C) $\theta(\log n)$ (D) $\theta(n \log n)$

10. Which one of the following in-place sorting algorithm needs the minimum number of swaps?
(A) Quick sort (B) Insertion sort
(C) Selection sort (D) Heap sort

11. As the size of the array grows what is the time complexity of finding an element using binary search (array of elements are ordered)?
(A) $\theta(n \log n)$ (B) $\theta(\log n)$
(C) $\theta(n^2)$ (D) $\theta(n)$

12. The time complexity of heap sort algorithm is
(A) $n \log n$ (B) $\log n$
(C) n^2 (D) None of these.

13. As part of maintenance work, you are entrusted with the work of rearranging the library books in a shelf in a proper order, at the end of each day. The ideal choices will be_____.
(A) Heap sort (B) Quick sort
(C) Selection sort (D) Insertion sort

14. The value for which you are searching is called
(A) Binary value
(B) Search argument
(C) Key
(D) Serial value

15. To sort many large objects and structures it would be most efficient to _____.
(A) Place them in an array and sort the array
(B) Place the pointers on them in an array and sort the array
(C) Place them in a linked list and sort the linked list
(D) None of the above

Previous Years' Questions

1. What is the number of swaps required to sort n elements using selection sort, in the worst case? **[2009]**
(A) $\theta(n)$
(B) $\theta(n \log n)$
(C) $\theta(n^2)$
(D) $\theta(n^2 \log n)$

2. Which one of the following is the tightest upper bound that represents the number of swaps required to sort n numbers using selection sort? **[2013]**
(A) $O(\log n)$ (B) $O(n)$
(C) $O(n \log n)$ (D) $O(n^2)$

3. Let P be a quick sort program to sort numbers in ascending order using the first element as the pivot. Let t_1 and t_2 be the number of comparisons made by P for the inputs [1 2 3 4 5] and [4 1 5 3 2] respectively. Which one of the following holds? **[2014]**
(A) $t_1 = 5$ (B) $t_1 < t_2$
(C) $t_1 > t_2$ (D) $t_1 = t_2$

4. The minimum number of comparisons required to find the minimum and the maximum of 100 numbers is ———. **[2014]**

5. Suppose P, Q, R, S, T are sorted sequences having lengths 20, 24, 30, 35, 50 respectively. They are to be merged into a single sequence by merging together two sequences at a time. The number of comparisons that will be needed in the worst case by the optimal algorithm for doing this is ———. **[2014]**

6. You have an array of n elements. Suppose you implement quick sort by always choosing the central element of the array as the pivot. Then the tightest upper bound for the worst case performance is **[2014]**
(A) $O(n^2)$ (B) $O(n \log n)$
(C) $\theta(n \log n)$ (D) $O(n^3)$

7. What are the worst-case complexities of insertion and deletion of a key in a binary search tree? **[2015]**

(A) θ(log *n*) for both insertion and deletion
(B) θ(*n*) for both insertion and deletion
(C) θ(*n*) for insertion and θ(log *n*) for deletion
(D) θ(log *n*) for insertion and θ(*n*) for deletion

8. The worst case running times of *Insertion sort, Merge sort* and *Quick sort*, respectively, are: **[2016]**
(A) $\Theta(n \log n)$, $\Theta(n \log n)$,and $\Theta(n^2)$
(B) $\Theta(n^2)$, $\Theta(n^2)$,and $\Theta(n \log n)$
(C) $\Theta(n^2)$, $\Theta(n \log n)$,and $\Theta(n \log n)$
(D) $\Theta(n^2)$, $\Theta(n \log n)$,and $\Theta(n^2)$

9. An operator delete(i) for a binary heap data structure is to be designed to delete the item in the i-th node. Assume that the heap is implemented in an array and i refers to the i-th index of the array. If the heap tree has depth d (number of edges on the path from the root to the farthest leaf), then what is the time complexity to re-fix the heap efficiently after the removal of the element? **[2016]**
(A) $O(1)$ (B) $O(d)$ but not $O(1)$
(C) $O(2^d)$ but not $O(d)$ (D) $O(d2^d)$ but not $O(2^d)$

10. Assume that the algorithms considered here sort the input sequences in ascending order. If the input is already in ascending order, which of the following are **TRUE**? **[2016]**
I. Quicksort runs in $\Theta(n^2)$ time
II. Bubblesort runs in $\Theta(n^2)$ time
III. Mergesort runs in $\Theta(n)$ time
IV. Insertion sort runs in Θ (n) time
(A) I and II only (B) I and III only
(C) II and IV only (D) I and IV only

11. A complete binary min - heap is made by including each integer in [1,1023] exactly once. The depth of a node in the heap is the length of the path from the root of the heap to that node. Thus, the root is depth 0. The maximum depth at which integer 9 can appear is _____ . **[2016]**

Answer Keys

Exercises

Practice Problems 1

1. B	**2.** A	**3.** A	**4.** B	**5.** B	**6.** C	**7.** C	**8.** B	**9.** A	**10.** B
11. D	**12.** A	**13.** A	**14.** B	**15.** A					

Practice Problems 2

1. B	**2.** B	**3.** B	**4.** B	**5.** A	**6.** A	**7.** C	**8.** B	**9.** C	**10.** C
11. B	**12.** A	**13.** D	**14.** C	**15.** B					

Previous Years' Questions

1. A	**2.** B	**3.** C	**4.** 148	**5.** 358	**6.** A	**7.** B	**8.** D	**9.** B	**10.** D
11. 8									