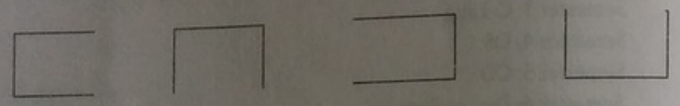
Hint: Run BFS two times. First run from v and second time from w. Find a DAG where the shortest ancestral path goes to a common ancestor x that is not an LCA.

Problem-33 Let us assume that we have two graphs G_1 and G_2, how do we check whether they are isomorphic or not?

Solution: If we take any graph, there can be many ways of representing the same graph. As an example, consider the following simple graph. It can be seen that all the below representations are having the same number of vertices and same number of edges.



Definition: Graphs $G_1 = \{V_1, E_1\}$ and $G_2 = \{V_2, E_2\}$ are isomorphic if

1) There is a one-to-one correspondence from V_1 to V_2 and
2) There is a one-to-one correspondence from E_1 to E_2 that map each edge of G_1 to G_2.

Now, for the given graphs how do we check whether they are isomorphic or not?

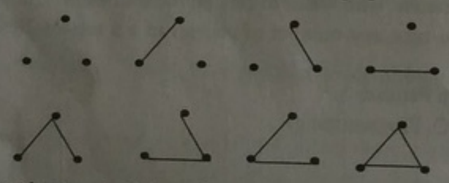
In general, it is not a simple task to prove that two graphs are isomorphic. For that reason what we do is, we consider some properties of isomorphic graphs. That means those properties must be satisfied if the graphs are isomorphic. If the given graph does not satisfy these properties then we say they are not isomorphic graphs.

Property: Two graphs are isomorphic if and only if for some ordering of their vertices their adjacency matrices are equal.

Based on the above property we decide whether the given graphs are isomorphic or not. For checking the property we need to do some matrix transformation operations.

Problem-34 How many simple undirected non-isomorphic graphs are there with n vertices?

Solution: We will try to answer this question in two steps. First, we count all labeled graphs. Assume all the below representations are labeled with $\{1, 2, 3\}$ as vertices. The set of all such graphs for $n = 3$ are:



There are only two choices for each edge, it either exists or it does not. Therefore, since the maximum number of edges is $\binom{n}{2}$ (since, the maximum number of edges in an undirected graph with n vertices are $\frac{n(n-1)}{2} = n_{C_2} = \binom{n}{2}$), the total number of undirected labeled graphs is $2^{\binom{n}{2}}$.

Problem-35 **Hamiltonian path in DAGs:** Given a DAG, design a linear time algorithm to determine whether there is a path that visits each vertex exactly once.

Solution: *Hamiltonian* path problem is a NP-Complete problem (for more details ref *Complexity Classes* chapter). To solve this problem, we will try to give the approximation algorithm (which solves the problem but it may not produce the optimal solution always).

Let us consider the topological sort algorithm for solving this problem. Topological sort has an interesting property that all pairs of consecutive vertices in the sorted order are connected by edges then these edges form a directed *Hamiltonian* path in the DAG. If a *Hamiltonian* path exists, the topological sort order is unique. Also, if a topological sort does not form a *Hamiltonian* path, the DAG will have two or more topological orderings.

Approximation Algorithm: Compute a topological sort and check if there is an edge between each consecutive pair of vertices in the topological order.

In an unweighted graph, finding a path from s to t that visits each vertex exactly once. The basic solution based on backtracking is, we start at s and try all of its neighbors recursively, making sure we never visit the same vertex twice. The algorithm based on this implementation can be given as:

```
bool seenTable[32];
void HamiltonianPath( struct Graph *G, int u ) {
        if( u == t )
                /* Check that we have seen all vertices. */
        else {
                for( int v = 0; v < n; v++ )
                        if( !seenTable[v] && G→Adj[u][v] ) {
                                seenTable[v] = true;
                                HamiltonianPath( v );
                                seenTable[v] = false;
                        }
        }
}
```

Note that if we have a partial path from s to u using vertices $s = v_1, v_2, \ldots, v_k = u$, then we don't care about the order in which we visited these vertices in order to figure out which vertex to visit next. All that we need to know is the set of vertices we have seen (the seenTable[] array) and which vertex we are at right now (u). There are 2^n possible sets of vertices and n choices for u. In other words, there are 2^n possible *seenTable*[ ] arrays and n different parameters to HamiltonianPath(). What HamiltonianPath() does during any particular recursive call is completely determined by the *seenTable*[ ] array and the parameter u.

Problem-36 For a given graph G with n vertices how many trees we can construct?

Solution: There is a simple formula for this problem and it was named after Arthur Cayley. For a given graph with n labeled vertices the formula for finding number of trees on is n^{n-2}. Below, the number of trees with different n values is shown.

n value	Formula value: n^{n-2}	Number of Trees
2	1	1 — 2
3	3	1 2 3 ; 2 1 3 ; 3 1 2

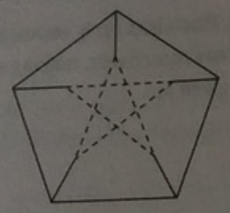
Problem-37 For a given graph G with n vertices how many spanning trees we can construct?

Solution: The solution to this problem is same as that of Problem-36. It is just other way of asking the same problem. Because, the number of edges in both regular tree and spanning tree are same.

Problem-38 The *Hamiltonian cycle* problem: Is it possible to traverse each of the vertices of a graph exactly once, starting and ending at the same vertex?

Solution: Since *Hamiltonian* path problem is a NP-Complete problem *Hamiltonian* cycle problem is a NP-Complete problem. A *Hamiltonian* cycle is a cycle that traverses every vertex of a graph exactly once. There are no known conditions which are both necessary and sufficient. There are a few sufficient conditions.

- For a graph to have a *Hamiltonian* cycle the degree of each vertex must be two or more.
- The Petersen graph does not have a *Hamiltonian* cycle and the graph is given below.



- In general, the more edges a graph has, the more likely it is to have a *Hamiltonian* cycle.
- Let G be a simple graph with n ≥ 3 vertices. If every vertex has degree at least $\frac{n}{2}$, then G has a *Hamiltonian* cycle.
- The best known algorithm for finding a *Hamiltonian* cycle has an exponential worst-case complexity.

As said above, for approximation algorithm of *Hamiltonian* path, please refer *Dynamic Programming* chapter.

Problem-39 What is the difference between *Dijkstra's* and *Prim's* algorithm?

Solution: *Dijkstra's* algorithm is almost identical to that of *Prim's*. The algorithm begins at a specific vertex and extends outward within the graph, until all vertices have been reached. The only distinction is that *Prim's* algorithm stores a minimum cost edge whereas *Dijkstra's* algorithm stores the total cost from a source vertex to the current vertex. More simply, *Dijkstra's* algorithm stores a summation of minimum cost edges whereas *Prim's* algorithm stores at most one minimum cost edge.

Problem-40 **Reversing Graph:** Give an algorithm that returns the reverse of the directed graph (each edge from v to w is replaced by an edge from w to v).

Solution: In graph theory, the reverse (also called as transpose) of a directed graph G is another directed graph on the same set of vertices with all of the edges reversed. That means, if G contains an edge (u, v) then the reverse of G contains an edge (v, u) and vice versa.

Algorithm:

```
Graph ReverseTheDirectedGraph(struct Graph *G) {
        Create new graph with name ReversedGraph and
                let us assume that this will contain the reversed graph.
        //The reversed graph also will contain same number of vertices and edges.
        for each vertex of given graph G {
                for each vertex w adjacent to v {
                        Add the w to v edge in ReversedGraph;
                        //That means we just need to reverse the bits in adjacency matrix.
                }
        }
        return ReversedGraph;
}
```

Problem-41 **Travelling Sales Person Problem:** Find the shortest path in a graph that visits each vertex at least once, starting and ending at the same vertex?

Solution: The Traveling Salesman Problem (*TSP*) is related to finding a Hamiltonian cycle. Given a weighted graph G, we want to find the shortest cycle (may be non-simple) that visits all the vertices.

Approximation algorithm: This algorithm does not solve the problem but gives a solution which is within a factor of 2 of optimal (in the worst-case).

1) Find a Minimal Spanning Tree (MST).

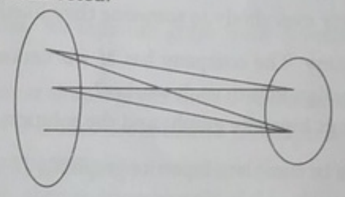
2) Do a DFS of the MST.

For more details, refer *Complexity Classes* chapter.

Problem-42 Discuss Bipartite matchings?

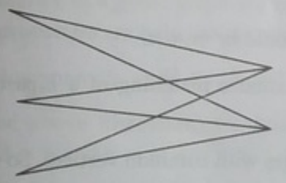
Solution: In Bipartite graphs, we divide the graphs in to two disjoint sets and each edge connects a vertex from one set to a vertex in another subset (as shown in figure).

Definition: A simple graph $G = (V, E)$ is called bipartite graph if its vertices can be divided into two disjoint sets $V = V_1 \cup V_2$, such that every edge has the form $e = (a, b)$ where $a \in V_1$ and $b \in V_2$. One important condition is that no vertices both in V_1 or both in V_2 are connected.

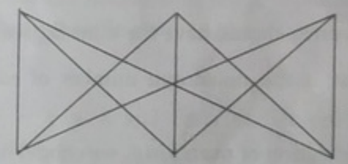


Properties of Bipartite Graphs

- A graph is called bipartite if and only if the given graph does not have an odd length cycle.
- A *complete bipartite graph* $K_{m,n}$ is a bipartite graph that has each vertex from one set adjacent to each vertex to another set.

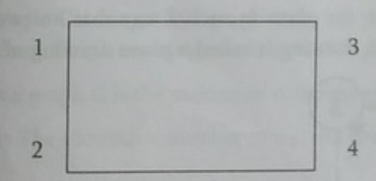


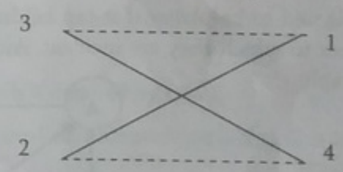
$K_{2,3}$



$K_{3,3}$

- A subset of edges $M \subset E$ is a *matching* if no two edges have a common vertex. As example, matching sets of edges are represented with dotted lines. A matching M is called *maximum* if it has a largest number of possible edges. In the graphs, the dotted edges represent the alternative matching for the given graph.





- A matching M is *perfect*, if it matches all vertices. We must have $V_1 = V_2$ in order to have perfect matching.
- An *alternating path* is a path whose edges alternate between matched and unmatched edges. If we find an alternating path then we can improve the matching. This is because an alternating path consists of matched and unmatched edges. The number of unmatched edges exceeds the number of matched edges by one. Therefore, an alternating path always increases the matching by one.

Next question is, how do we find a perfect matching? Based on the above theory and definition we can find the perfect matching with the following approximation algorithm.

Matching Algorithm (Hungarian algorithm)

1) Start at unmatched vertex.
2) Find an alternating path.

3) If it exists, change matching edges to no matching edges and conversely. If it does not exist, choose another unmatched vertex.
4) If the number of edges equals $V/2$ stop, otherwise proceed to step 1 and repeat as long all vertices have been examined without finding any alternating paths.

Time Complexity of the Matching Algorithm: The number of iterations is in $O(V)$. The complexity of finding an alternating path using BFS is $O(E)$. Therefore, the total time complexity is $O(V \times E)$.

Problem-43 Marriage and Personnel Problem?

Marriage Problem: There are X men and Y women who desire to get married. Participants indicate who among the opposite sex would be acceptable as a potential spouse. Every woman can be married to at most one man, and every man to at most one woman. How could we marry everybody to someone they liked?

Personnel Problem: You are the boss of a company. The company has M workers and N jobs. Each worker is qualified to do some jobs, but not others. How will you assign jobs to each worker?
Solution: This is just another way of asking about bipartite graphs and the solution is same as that of Problem-42.

Problem-44 How many edges will be there in complete bipartite graph $K_{m,n}$?

Solution: $m \times n$. This is because each vertex in first set can connect all vertices in second set.

Problem-45 A graph is called regular graph if it has no loops and multiple edges where each vertex has the same number of neighbors; i.e. every vertex has the same degree. Now, if $K_{m,n}$ is a regular graph what is the relation between m and n?

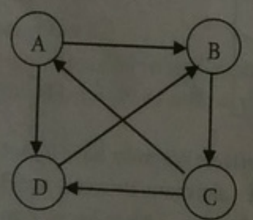
Solution: Since each vertex should have the same degree the relation should be $m = n$.

Problem-46 What is the maximum number of edges in the maximum matching of a bipartite graph with n vertices?

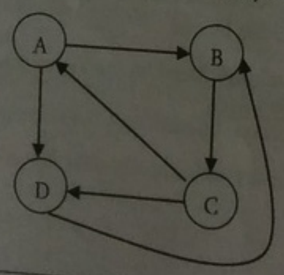
Solution: From the definition of *matching*, we should not have the edges with common vertices. So in bipartite graph, each vertex can connect to only one vertex. Since we divide the total vertices into two sets, we can get the maximum number of edges if we divide them into half. Finally the answer is $\frac{n}{2}$.

Problem-47 Discuss Planar Graphs. *Planar graph*: Is it possible to draw the edges of a graph in such a way that edges do not cross?

Solution: A graph G is said to be planar if it can be drawn in the plane in such a way that no two edges meet each other except at a vertex to which they are incident. Any such drawing is called a plane drawing of G. As an example consider the below graph:



This graph we can easily convert to planar graph as below (without any cross edges).



How do we decide whether a given graph is planar or not?

The solution to this problem is not simple. Instead some researchers found some interesting properties based on which we can decide whether the given graph is a planar graph or not.

Properties of Planar Graphs

- If a graph G is a connected planar simple graph with V vertices, where $V = 3$ and E edges then $E = 3V - 6$.
- K_5 is non-planar [K_5 stands for complete graph with 5 vertices].
- If a graph G is a connected planar simple graph with V vertices and E edges, and no triangles then $E = 2V - 4$.
- $K_{3,3}$ is non-planar [$K_{3,3}$ stands for bipartite graph with 3 vertices on one side and other 3 vertices on other side. $K_{3,3}$ contains 6 vertices].
- If a graph G is connected planar simple graph then G contains at least one vertex of degree 5 or less.
- A graph is planar if and only if it does not contain a subgraph which has K_5 and $K_{3,3}$ as a contraction.
- If a graph G contains a nonplanar graph as a subgraph, then G is non-planar.
- If a graph G is a planar graph, then every subgraph of G is planar;
- For any connected planar graph $G = (V, E)$, the following formula should holds $V + F - E = 2$, where F stands for the number of faces.
- For any planar graph $G = (V, E)$ with K components, the following formula holds $V + F - E = 1 + K$.

Inorder to test planarity of a given graph we use these properties and decide whether it is planar graph or not. Note that all the above properties are only the necessary conditions but not sufficient.

Problem-48 How many faces do $K_{2,3}$ have?

Solution: From the above discussion, we know that $V + F - E = 2$ and from earlier problem we know that $E = m \times n = 2 \times 3 = 6$ and $V = m + n = 5$. $\therefore 5 + F - 6 = 2 \Rightarrow F = 3$.

Problem-49 Discuss Graph Coloring

Solution: A k −coloring of a graph G is an assignment of one color to each vertex of G such that no more than k colors are used and no two adjacent vertices receive the same color. A graph is called k −colorable if and only if it has a k −coloring.

Applications of Graph Coloring: The graph coloring problem has many applications such as scheduling, register allocation in compilers, frequency assignment in mobile radios, etc.

Clique: A *clique* in a graph G is the maximum complete subgraph is denoted by $\omega(G)$.

Chromatic number: The chromatic number of a graph G is the smallest number k such that G is k −colorable, and it is denoted by $X(G)$.

Lower bound for $X(G)$ is $\omega(G)$, that means $\omega(G) \leq X(G)$.

Properties of Chromatic number: Let G be a graph with n vertices and G′ is its complement. Then,

- $X(G) \leq \Delta(G) + 1$, where Δ (G) is the maximum degree of G.
- $X(G)\, \omega(G') \geq n$
- $X(G) + \omega(G') \leq n + 1$
- $X(G) + (G') \leq n + 1$

K-colorability problem: Given a graph $G = (V, E)$ and a positive integer $k \leq V$. Check whether G is k −colorable?

This problem is NP-complete and will discuss more in *Complexity Classes* Chapter.

Graph coloring algorithm: As we discussed above, this problem is NP-Complete. So we do not have a polynomial time algorithm to determine $X(G)$. Let us consider the following approximation (no efficient) algorithm.

- Consider a graph G with two non-adjacent vertices a and b. The connection G_1 is obtained by joining the two non-adjacent vertices a and b with an edge. The contraction G_2 is obtained by shrinking $\{a, b\}$ into a single vertex $c(a, b)$ and by joining it to each neighbor in G of vertex a and of vertex b (and eliminating multiple edges).
- A coloring of G in which a and b have the same color yields a coloring of G_1. A coloring of G in which a and b have different colors yields a coloring of G_2.
- Repeat the operations of connection and contraction in each graph generated, until the resulting graphs are all cliques. If the smallest resulting clique is a K −clique, then $(G) = \mathrm{K}$.

Important notes on Graph Coloring

- Any simple planar graph G can be colored with 6 colors.
- Every simple planar graph can be colored with less than or equal to 5 colors.

Problem-50 What is the four coloring problem?

Solution: A graph can be constructed from any map. The regions of the map are represented by the vertices of the graph and two vertices are joined by an edge if the regions corresponding to the vertices are adjacent. The resulting graph is planar. That means it can be drawn in the plane without any edges crossing.

The *Four Color Problem* is whether if the vertices of a planar graph can be colored with at most 4 colors so that no two adjacent vertices use the same color.

History: The *Four − Color* problem was first given by *Francis Guthrie*. He was a student at *University College London* where he studied under *Augusts De Morgan*. After graduating from London he studied law but some years later his brother Frederick Guthrie had become a student of *De Morgan*. One day Francis asked his brother to discuss this problem with *De Morgan*.

Problem-51 When an adjacency-matrix representation is used, most graph algorithms require time $O(V^2)$. Show that determining whether a directed graph, represented in an adjacency-matrix contains a sink can be done in time $O(V)$. A sink is a vertex with in-degree $|V| - 1$ and out-degree 0 (Only one can exist in a graph).

Solution: A vertex i is a sink if and only if $M[i, j] = 0$ for all j and $M[j, i] = 1$ for all $j \neq i$. For any pair of vertices i and j:

$M[i, j] = 1 \rightarrow$ vertex i can't be a sink
$M[i, j] = 0 \rightarrow$ vertex j can't be a sink

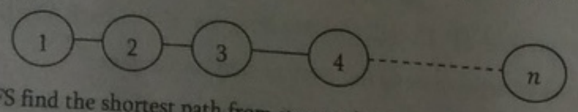
Algorithm:

- Start at $i = 1, j = 1$
- If $M[i, j] = 0 \rightarrow i$ wins, $j++$
- If $M[i, j] = 1 \rightarrow j$ wins, $i++$
- Proceed this process until $j = n$ or $i = n + 1$
- If $i == n + 1$, the graph does not contain a sink
- Otherwise, check row i - it should be all zeros, and column i – it should be all but $M[i, i]$ ones – if so, i is a sink.

Time Complexity: $O(V)$, because at most $2|V|$ cells in the matrix are examined.

Problem-52 What is the worst – case memory usage of DFS?

Solution: It happens when theO(|V|), which happens if the graph is actually a list. So the algorithm is memory efficient on graphs with small diameter.



Problem-53 Does DFS find the shortest path from start node to some node w ?

Solution: No. In DFS it is not compulsory to select smallest weight edge always.

SORTING

Chapter-10

10.1 What is Sorting?

Sorting is an algorithm that arranges the elements of a list in a certain order [either *ascending* or *descending*]. The output is a permutation or reordering of the input.

10.2 Why Sorting?

Sorting is one of the important categories of algorithms in computer science. Sometimes sorting significantly reduces the problem complexity. We can use sorting as a technique to reduce the search complexity. Great research went into this category of algorithms because of its importance. These algorithms are very much used in many computer algorithms [for example, searching elements], database algorithms and many more.

10.3 Classification

Sorting algorithms are generally classified into different categories based on the following parameters.

By Number of Comparisons

In this method, sorting algorithms are classified based on the number of comparisons. For comparison based sorting algorithms best case behavior is $O(nlogn)$ and worst case behavior is $O(n^2)$. Comparison-based sorting algorithms evaluate the elements of the list by key comparison operation and needs at least $O(nlogn)$ comparisons for most inputs.

Later in this chapter we will discuss few *non − comparison (linear)* sorting algorithms like Counting sort, Bucket sort, and Radix sort etc... Linear Sorting algorithms impose few restrictions on the inputs to improve the complexity.

By Number of Swaps

In this method, sorting algorithms are categorized by number of *swaps* (also called *inversions*).

By Memory Usage

Some sorting algorithms are "*in place*" and they need O(1) or $O(logn)$ memory to create auxiliary locations for sorting the data temporarily.

By Recursion

Sorting algorithms are either recursive [quick sort] or non-recursive [selection sort, and insertion sort]. There are some algorithms which uses both (merge sort).

By Stability

Sorting algorithm is *stable* if for all indices i and j such that the key $A[i]$ equals key $A[j]$, if record $R[i]$ precedes record $R[j]$ in the original file, record $R[i]$ precedes record $R[j]$ in the sorted list. Few sorting algorithms maintain the relative order of elements with equal keys (equivalent elements retain their relative positions even after sorting).

By Adaptability

Few sorting algorithms complexity changes based on presortedness [quick sort]: presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

10.4 Other Classifications

Other way of classifying the sorting algorithms are:

- Internal Sort
- External Sort

Internal Sort

Sort algorithms which use main memory exclusively during the sort are called *internal* sorting algorithms. This kind of algorithms assumes high-speed random access to all memory.

External Sort

Sorting algorithms which uses external memory, such as tape or disk, during the sort comes under this category.

10.5 Bubble sort

Bubble sort is the simplest sorting algorithm. It works by iterating the input array from the first element to last, comparing each pair of elements and swapping them if needed. Bubble sort continues its iterations until no swaps are needed. The algorithm got its name from the way smaller elements "*bubble*" to the top of the list. Generally, insertion sort has better performance than bubble sort. Some researchers suggest that we should not teach bubble sort because of its simplicity and bad complexity.

The only significant advantage that bubble sort has over other implementations is that it can detect whether the input list is already sorted or not.

Implementation

```
void BubbleSort(int A[], int n) {
        for (int pass = n - 1; pass >= 0; pass--)       {
                for (int i = 0; i < pass - 1 ; i++)     {
                        if(A[i] > A[i+1]) {
                                // swap elements
                                int temp = A[i];
                                A[i] = A[i+1];
                                A[i+1] = temp;
                        }
                }
        }
}
```

Algorithm takes $O(n^2)$ (even in best case). We can improve it by using one extra flag. When there are no more swaps, indicates the completion of sorting. If the list is already sorted, by using this flag we can skip the remaining passes.

```
void BubbleSortImproved(int A[], int n) {
        int pass, i, temp, swapped = 1;
        for (pass = n - 1; pass >= 0 && swapped; pass--) {
                swapped = 0;
```



```
                for (i = 0; i < pass - 1 ; i++) {
                        if(A[i] > A[i+1]) {
                                // swap elements
                                temp = A[i];
                                A[i] = A[i+1];
                                A[i+1] =  temp;
                                swapped = 1;
                        }
                }
        }
}
```

This modified version improves the best case of bubble sort to $O(n)$.

Performance

Worst case complexity : $O(n^2)$
Best case complexity (Improved version) : $O(n)$
Average case complexity (Basic version) : $O(n^2)$
Worst case space complexity : O(1) auxiliary

10.6 Selection Sort

Selection sort is an in-place sorting algorithm. Selection sort works well for small files. It is used for sorting the files with very large values and small keys. This is because of the fact that selection is made based on keys and swaps are made only when required.

Advantages:

- Easy to implement
- In-place sort (requires no additional storage space)

Disadvantages:

- Doesn't scale well: $O(n^2)$

Algorithm

1. Find the minimum value in the list
2. Swap it with the value in the current position
3. Repeat this process for all the elements until the entire array is sorted

This algorithm is called *selection sort* since it repeatedly *selects* the smallest element.

Implementation

```
void Selection(int A [], int n) {
        int i, j, min, temp;
        for (i = 0; i < n - 1; i++) {
                min = i;
                for (j = i+1; j < n; j++) {
                        if(A [j] < A [min])
                                min = j;
                }
                // swap elements
                temp = A[min];
                A[min] = A[i];
```

```
            A[i] = temp;
        }
}
```

Performance

Worst case complexity : $O(n^2)$
Best case complexity : $O(n)$
Average case complexity : $O(n^2)$
Worst case space complexity: O(1) auxiliary

10.7 Insertion sort

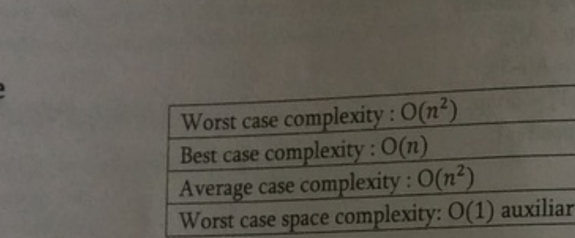
Insertion sort is a simple and efficient comparison sort. In this algorithm each iteration removes an element from the input data and inserts it into the correct position in the list being sorted. The choice of the element being removed from the input is random and this process is repeated until all input elements have been gone through.

Advantages

- Simple implementation
- Efficient for small data
- Adaptive: If the input list is presorted [may not be completely] then insertions sort takes $O(n + d)$, where d is the number of inversions
- Practically more efficient than selection and bubble sorts even though all of them have $O(n^2)$ worst case complexity
- Stable: Maintains relative order of input data if the keys are same
- In-place: It requires only a constant amount O(1) of additional memory space
- Online: Insertion sort can sort the list as it receives it

Algorithm

Every repetition of insertion sort removes an element from the input data, inserts it into the correct position in the already-sorted list until no input elements remain. Sorting is typically done in-place. The resulting array after k iterations has the property where the first $k + 1$ entries are sorted.



Each element greater than x copied to the right as it is compared against x.

Implementation

```
void InsertionSort(int A[], int n) {
    int i, j, v;
    for (i = 2; i <= n - 1; i++) {
        v = A[i];
        j = i;
        while (A[j-1] > v && j >= 1) {
            A[j] = A[j-1];
```



```
            j--;
        }
        A[j] = v;
    }
}
```

Example

Given an array: 6 8 1 4 5 3 7 2 and the goal is to put them in ascending order.

6 8 1 4 5 3 7 2 (Consider index 0)
6 8 1 4 5 3 7 2 (Consider indices 0 - 1)
1 6 8 4 5 3 7 2 (Consider indices 0 - 2: insertion places 1 in front of 6 and 8)
1 4 6 8 5 3 7 2 (Process same as above is repeated until array is sorted)
1 4 5 6 8 3 7 2
1 3 4 5 6 7 8 2
1 2 3 4 5 6 7 8 (The array is sorted!)

Analysis

Worst case analysis

Worst case occurs when for every i the inner loop has to move all elements $A[1], \ldots, A[i-1]$ (which happens when $A[i]$ = key is smaller than all of them), that takes $\Theta(i-1)$ time.

$$T(n) = \Theta(1) + \Theta(2) + \theta(2) + \ldots\ldots + \Theta(n-1)$$
$$= \Theta(1 + 2 + 3 + \ldots\ldots + n - 1) = \Theta(\frac{n(n-1)}{2}) \approx \Theta(n^2)$$

Average case analysis

For the average case, the inner loop will insert $A[i]$ in the middle of $A[1], \ldots, A[i-1]$. This takes $\Theta(i/2)$ time.

$$T(n) = \sum_{i=1}^{n} \Theta(i/2) \approx \Theta(n^2)$$

Performance

Worst case complexity : $O(n^2)$
Best case complexity : $O(n^2)$
Average case complexity : $O(n^2)$
Worst case space complexity: $O(n^2)$ total, O(1) auxiliary

Comparisons to Other Sorting Algorithms

Insertion sort is one of the elementary sorting algorithms with $O(n^2)$ worst-case time. Insertion sort is used when the data is nearly sorted (due to its adaptiveness) or when the input size is small (due to its low overhead). For these reasons and due to its stability, insertion sort is used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort.

Note:

- Bubble sort takes $\frac{n^2}{2}$ comparisons and $\frac{n^2}{2}$ swaps (inversions) in both average case and in worst case.
- Selection sort takes $\frac{n^2}{2}$ comparisons and n swaps.
- Insertion sort takes $\frac{n^2}{4}$ comparisons and $\frac{n^2}{8}$ swaps in average case and in the worst case they are double.
- Insertion sort is almost linear for partially sorted input.
- Selection sort is best suits for elements with bigger values and small keys.

10.8 Shell sort

Shell sort (also called *diminishing increment sort*) is invented by *Donald Shell*. This sorting algorithm is a generalization of insertion sort. Insertion sort works efficiently on input that is already almost sorted.

In insertion sort, comparison happens between the adjacent elements. At most 1 inversion is eliminated for each comparison done with insertion sort. The variation used in shell sort is to avoid comparing adjacent elements until the last step of the algorithm. So, the last step of shell sort is effectively the insertion sort algorithm. It improves insertion sort by allowing the comparison and exchange of elements that are far away. This is the first algorithm which got less than quadratic complexity among comparison sort algorithms.

Shellsort uses a sequence $h1, h2, \ldots, ht$ called the *increment sequence*. Any increment sequence is fine as long as $h1 = 1$ and some choices are better than others. Shellsort makes multiple passes through input list and sorts a number of equally sized sets using the insertion sort. Shellsort improves on the efficiency of insertion sort by *quickly* shifting values to their destination.

Implementation

```
void ShellSort(int A[], int array_size) {
        int i, j, h, v;
        for (h = 1; h = array_size/9; h = 3*h+1);
        for ( ; h > 0; h = h/3) {
                for (i = h+1; i = array_size; i += 1) {
                        v = A[i];
                        j = i;
                        while (j > h && A[j-h] > v) {
                                A[j] = A[j-h];
                                j -= h;
                        }
                        A[j] = v;
                }
        }
}
```

Note that when $h == 1$, the algorithm makes a pass over the entire list, comparing adjacent elements, but doing very few element exchanges. For $h == 1$, shell sort works just like insertion sort, except the number of inversions that have to be eliminated is greatly reduced by the previous steps of the algorithm with $h > 1$.

Analysis

Shell sort is efficient for medium size lists. For bigger lists, the algorithm is not the best choice. Fastest of all $O(n^2)$ sorting algorithms.

Disadvantage of Shell sort is that: it is a complex algorithm and not nearly as efficient as the merge, heap, and quick sorts. Shell sort is still significantly slower than the merge, heap, and quick sorts, but it is relatively simple algorithm which makes it a good choice for sorting lists of less than 5000 items unless speed important. It is also a good choice for repetitive sorting of smaller lists.

The best case in the Shell sort is when the array is already sorted in the right order. The number of comparisons is less. Running time of Shell sort depends on the choice of increment sequence.

Performance

Worst case complexity depends on gap sequence. Best known: $O(nlog^2n)$
Best case complexity: $O(n)$

Average case complexity depends on gap sequence
Worst case space complexity: $O(n)$

10.9 Merge sort

Merge sort is an example of the divide and conquer.

Important Notes

- *Merging* is the process of combining two sorted files to make one bigger sorted file.
- *Selection* is the process of dividing a file into two parts: k smallest elements and $n - k$ largest elements.
- Selection and merging are opposite operations
    - selection splits a list into two lists
    - merging joins two files to make one file
- Merge sort is Quick sorts complement
- Merge sort accesses the data in a sequential manner
- This algorithm is used for sorting a linked list
- Merge sort is insensitive to the initial order of its input
- In Quick sort most of the work is done before the recursive calls. Quick sort starts with the largest subfile and finishes up with the small ones and as a result it needs stack and also this algorithm is not stable. Whereas Merge sort divides the list into two parts, then each part is conquered individually. Merge sort starts with the small subfiles and finishes up with the largest one and as a result it doesn't need stack and this algorithm is stable.

Implementation

```
void Mergesort(int A[], int temp[], int left, int right) {
        int mid;
        if(right > left) {
                mid = (right + left) / 2;
                Mergesort(A, temp, left, mid);
                Mergesort(A, temp, mid+1, right);
                Merge(A, temp, left, mid+1, right);
        }
}
void Merge(int A[], int temp[], int left, int mid, int right) {
        int i, left_end, size, temp_pos;
        left_end = mid - 1;
        temp_pos = left;
        size = right - left + 1;
        while ((left <= left_end) && (mid <= right)) {
                if(A[left] <= A[mid]) {
                        temp[temp_pos] = A[left];
                        temp_pos = temp_pos + 1;
                        left = left +1;
                }
                else {  temp[temp_pos] = A[mid];
                        temp_pos = temp_pos + 1;
                        mid = mid + 1;
                }
        }
```

```
        while (left <= left_end) {
                temp[temp_pos] = A[left];
                left = left + 1;
                temp_pos = temp_pos + 1;
        }
        while (mid <= right) {
                temp[temp_pos] = A[mid];
                mid = mid + 1;
                temp_pos = temp_pos + 1;
        }
        for (i = 0; i <= size; i++) {
                A[right] = temp[right];
                right = right - 1;
        }
}
```

Analysis

In Merge sort the input list is divided into two parts and solve them recursively. After solving the sub problems merge them by scanning the resultant sub problems. Let us assume $T(n)$ is the complexity of Merge sort with n elements. The recurrence for the Merge Sort can be defined as:

Recurrence for Mergesort is $T(n) = 2T(\frac{n}{2}) + \Theta(n)$.

Using Master theorem, we get, $T(n) = \Theta(nlogn)$.

Note: For more details, refer *Divide and Conquer* chapter.

Performance

Worst case complexity : $\Theta(nlogn)$
Best case complexity : $\Theta(nlogn)$
Average case complexity : $\Theta(nlogn)$
Worst case space complexity: $\Theta(n)$ auxiliary

10.10 Heapsort

Heapsort is a comparison-based sorting algorithm and is part of the selection sort family. Although somewhat slower in practice on most machines than a good implementation of Quick sort, it has the advantage of a more favorable worst-case $\Theta(nlogn)$ runtime. Heapsort is an in-place algorithm but is not a stable sort.

Performance

Worst case performance: $\Theta(nlogn)$
Best case performance: $\Theta(nlogn)$
Average case performance: $\Theta(nlogn)$
Worst case space complexity: $\Theta(n)$ total, $\Theta(1)$ auxiliary

For other details on Heapsort refer *Priority Queues* chapter.

10.11 Quicksort

The quick sort is an example for divide-and-conquer algorithmic technique. It is also called *partition exchange sort*. It uses recursive calls for sorting the elements. It is one of famous algorithm among comparison based sorting algorithms.

Divide: The array $A[low \dots high]$ is partitioned into two non-empty sub arrays $A[low \dots q]$ and $A[q+1 \dots high]$, such that each element of $A[low \dots high]$ is less than or equal to each element of $A[q+1 \dots high]$. The index q is computed as part of this partitioning procedure.

Conquer: The two sub arrays $A[low \dots q]$ and $A[q+1 \dots high]$ are sorted by recursive calls to Quick sort.

Algorithm

The recursive algorithm consists of four steps:

1) If there is one or no elements in the array to be sorted, return.
2) Pick an element in the array to serve as "*pivot*" point. (Usually the left-most element in the array is used.)
3) Split the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot.
4) Recursively repeat the algorithm for both halves of the original array.

Implementation

```
Quicksort( int A[], int low, int high ) {
        int pivot;
        /* Termination condition! */
        if( high > low ) {
                pivot = Partition( A, low, high );
                Quicksort( A, low, pivot-1 );
                Quicksort( A, pivot+1, high );
        }
}
int Partition( int A, int low, int high ) {
        int left, right, pivot_item = A[low];
        left = low;
        right = high;
        while ( left < right ) {
                /* Move left while item < pivot */
                while( A[left] <= pivot_item )
                        left++;
                /* Move right while item > pivot */
                while( A[right] > pivot_item )
                        right--;
                if( left < right )
                        swap(A,left,right);
        }
        /* right is final position for the pivot */
        A[low] = A[right];
        A[right] = pivot_item;
        return right;
}
```

Analysis

Let us assume that T(n) be the complexity of Quick sort and also assume that all elements are distinct. Recurrence for $T(n)$ depends on two subproblem sizes which depend on partition element. If pivot is i^{th} smallest element then exactly $(i-1)$ items will be in left part and (n − i) in right part. Let us call it as i −split. Since each element has equal probability of selecting it as pivot the probability of selecting i^{th} element is $\frac{1}{n}$.

Best Case: Each partition splits array in halves and gives

$T(n) = 2T(n/2) + \Theta(n) = \Theta(n\log n)$, [using *Divide and Conquer* master theorem]

Worst Case: Each partition gives unbalanced splits and we get

$T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$[*using Subtraction and Conquer master theorem*]

The worst-case occurs when the list is already sorted and last element chosen as pivot.

Average Case: In the average case of Quick sort, we do not know where the split happens. For this reason, we take all possible values of split locations, add all of their complexities and divide with n to get the average case complexity.

$$T(n) = \sum_{i=1}^{n} \frac{1}{n}(\textit{runtime with } i-\textit{split}) + n + 1$$

$$= \frac{1}{n}\sum_{i=1}^{N}\big(T(i-1) + T(n-i)\big) + n + 1$$

//since we are dealing with best case we can assume $T(n-i)$ and $T(i-1)$ are equal

$$= \frac{2}{n}\sum_{i=1}^{n} T(i-1) + n + 1$$

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

Multiply both sides by n.

$$nT(n) = 2\sum_{i=0}^{n-1} T(i) + n^2 + n$$

Same formula for $n-1$.

$$(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + (n-1)^2 + (n-1)$$

Subtract the $n-1$ formula from n.

$$nT(n) - (n-1)T(n-1) = 2\sum_{i=0}^{n-1} T(i) + n^2 + n - (2\sum_{i=0}^{n-2} T(i) + (n-1)^2 + (n-1))$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n$$

$$nT(n) = (n+1)T(n-1) + 2n$$

Divide with $n(n+1)$.

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

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

.

.

$$= O(1) + 2\textstyle\sum_{i=3}^{n} \frac{1}{i}$$

$$= O(1) + O(2\log n)$$

$$\frac{T(n)}{n+1} = O(\log n)$$

$$T(n) = O\big((n+1)\log n\big) = O(n\log n)$$

Time Complexity, $T(n) = O(n\log n)$.

Performance

Worst case Complexity: $O(n^2)$
Best case Complexity: $O(n\log n)$
Average case Complexity: $O(n\log n)$
Worst case space Complexity: $O(1)$

Randomized Quick sort

In average-case behavior of Quicksort, we assumed that all permutation of the input numbers are equally likely. However, we cannot always expect it to hold. We can add randomization to an algorithm in order to reduce the probability of getting worst case in Quick sort.

There are two ways of adding randomization in Quick sort: either by randomly placing the input data in the array or by randomly choosing an element in the input data for pivot. The second choice is easier to analyze and implement. The change will only be done at the Partition algorithm.

In normal Quicksort, *pivot* element was always the leftmost element in the list to be sorted. Instead of always using $A[low]$ as the *pivot* we will use a randomly chosen element from the subarray $A[low..high]$ in the randomized version of Quicksort. It is done by exchanging element $A[low]$ with an element chosen at random from $A[low..high]$. This ensures that the *pivot* element is equally likely to be any of the $high - low + 1$ elements in the subarray. Since the pivot element is randomly chosen, we can expect the split of the input array to be reasonably well balanced on average. This can help in preventing the worst-case behavior of quick sort which in unbalanced partitioning occurs.

Even though, the randomized version improves the worst case complexity, its worst case complexity is still $O(n^2)$. One way to improve the $Randomized - QuickSort$ is to choose the pivot for partitioning more carefully than by picking a random element from the array. One common approach is to choose the pivot as the median of a set of 3 elements randomly selected from the array.

10.12 Tree Sort

Tree sort uses a binary search tree. It involves scanning each element of the input and placing it into its proper position in a binary search tree. This has two phases:

- First phase is creating a binary search tree using the given array elements.
- Second phase is traverse the given binary search tree in inorder, thus resulting in a sorted array.

Performance

The average number of comparisons for this method is $O(n\log n)$. But in worst case, number of comparisons is reduced by $O(n^2)$, a case which arises when the sort tree is skew tree.

10.13 Comparison of Sorting Algorithms

Name	Average Case	Worst Case	Auxiliary Memory	Is Stable?	Other Notes
Bubble	$O(n^2)$	$O(n^2)$	1	yes	Small code
Selection	$O(n^2)$	$O(n^2)$	1	no	Stability depends on the implementation.
Insertion	$O(n^2)$	$O(n^2)$	1	yes	Average case is also $O(n+d)$, where d is the number of inversions
Shell	-	$O(n\log^2 n)$	1	no	
Merge	$O(n\log n)$	$O(n\log n)$	depends	yes	
Heap	$O(n\log n)$	$O(n\log n)$	1	no	
Quick Sort	$O(n\log n)$	$O(n^2)$	$O(\log n)$	depends	Can be implemented as a stable sort depending on how the pivot is handled.

Tree sort	$O(nlogn)$	$O(n^2)$	$O(n)$	depends	Can be implemented as a stable sort.

Note: In the n denotes the number of elements in the input.

10.14 Linear Sorting Algorithms

In earlier sections, we have seen many examples on comparison based sorting algorithms. Among all of them, the best comparison-based sorting can has the complexity $O(nlogn)$. In this section, we will discuss other type of algorithms: Linear Sorting Algorithms. To improve the time complexity of sorting these algorithms make some assumptions about the input. Few examples of Linear Sorting Algorithms are:

- Counting Sort
- Bucket Sort
- Radix Sort

10.15 Counting Sort

Counting sort is not a comparison sort algorithm and gives $O(n)$ complexity for sorting. To achieve $O(n)$ complexity, Counting sort assumes that each of the elements is an integer in the range 1 to K, for some integer K. When $K = O(n)$, the Counting-sort runs in $O(n)$ time. The basic idea of Counting sort is to determine, for each input elements X, the number of elements less than X. This information can be used to place directly into its correct position. For example, if there 10 elements less than X, then X belongs to position 11 in output.

In the below code, $A[0..n-1]$ is the input array with length n. In counting sort we need two more arrays: let us assume array $B[0..n-1]$ contains the sorted output and the array $C[0..K-1]$ provides temporary storage.

```
void CountingSort (int A[], int n, int B[], int K) {
        int C[K], i, j;
        //Complexity: O(K)
        for (i =0 ; i<K; i++)
            C[i] = 0;
        //Complexity: O(n)
        for (j =0 ; j<n; j++)
            C[A[j]] = C[A[j]] + 1;
        //C[i] now contains the number of elements equal to i
        //Complexity: O(K)
        for (i =1 ; i<K; i++)
            C[i] = C[i] + C[i-1];
        // C[i] now contains the number of elements ≤ i
        //Complexity: O(n)
        for (j = n-1; j>=0; j--) {
            B[C[A[j]]] = A[j];
            C[A[j]] = C[A[j]] - 1;
        }
}
```

Total Complexity: $O(K) + O(n) + O(K) + O(n) = O(n)$ if $K = O(n)$. Space Complexity: $O(n)$ if $K = O(n)$.
Note: Counting works well if $K = O(n)$. Otherwise, the complexity will be more.

10.16 Bucket sort [or Bin Sort]

As similar to *Counting* sort, *Bucket* sort also imposes restrictions on the input to improve the performance. In other words, Bucket sort works well if the input is drawn from fixed set. *Bucket* sort is the generalization of *Counting* Sort. For example, suppose that all the input elements from $\{0, 1, \ldots, K-1\}$, i.e., the set of integers in the interval $[0, K-1]$. That means, K is the number of distant elements in the input. *Bucket* sort uses K counters. The i^{th} counter

keeps track of the number of occurrences of the i^{th} element. Bucket sort with two buckets is effectively a version of Quick sort with two buckets.

```
#define BUCKETS 10
void BucketSort(int A[], int array_size) {
        int i, j, k;
        int buckets[BUCKETS];
        for(j =0; j < BUCKETS; j++)
                buckets[j] = 0;
        for(i =0; i < array_size; i++)
                ++ buckets[A[i]];
        for(i =0, j=0; j < BUCKETS; j++)
                for(k = buckets[j];k > 0; --k)
                        A[i++] = j;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

10.17 Radix sort

Similar to *Counting* sort and *Bucket* sort, this sorting algorithm also assumes some kind of information about the input elements. Suppose that the input values to be sorted are from base d. That means all numbers are d-digit numbers.

In radix sort, first sort the elements based on last digit [least significant digit]. These results were again sorted by second digit [next to least significant digits]. Continue this process for all digits until we reach most significant digits. Use some stable sort to sort them by last digit. Then stable sort them by the second least significant digit, then by the third, etc. If we use counting sort as the stable sort, the total time is $O(nd) \approx O(n)$.

Algorithm:

1) Take the least significant digit of each element.
2) Sort the list of elements based on that digit, but keep the order of elements with the same digit (this is the definition of a stable sort).
3) Repeat the sort with each more significant digit.

The speed of Radix sort depends on the inner basic operations. If the operations are not efficient enough, Radix sort can be slower than other algorithms such as Quick sort and Merge sort. These operations include the insert and delete functions of the sub-lists and the process of isolating the digit we want. If the numbers were not of equal length then a test is needed to check for additional digits that need sorting. This can be one of the slowest parts of Radix sort and also one of the hardest to make efficient.

Since Radix sort depends on the digits or letters, it is less flexible than other sorts. For every different type of data, Radix sort needs to be rewritten and if the sorting orders changes, the sort needs to be rewritten again. In short, Radix sort takes more time to write, and it is very difficult to write a general purpose Radix sort that can handle all kinds of data.

For many programs that need a fast sort, Radix sort is a good choice. Still, there are faster sorts, which is one reason why Radix sort is not used as much as some other sorts.

Time Complexity: $O(nd) \approx O(n)$, if d is small.

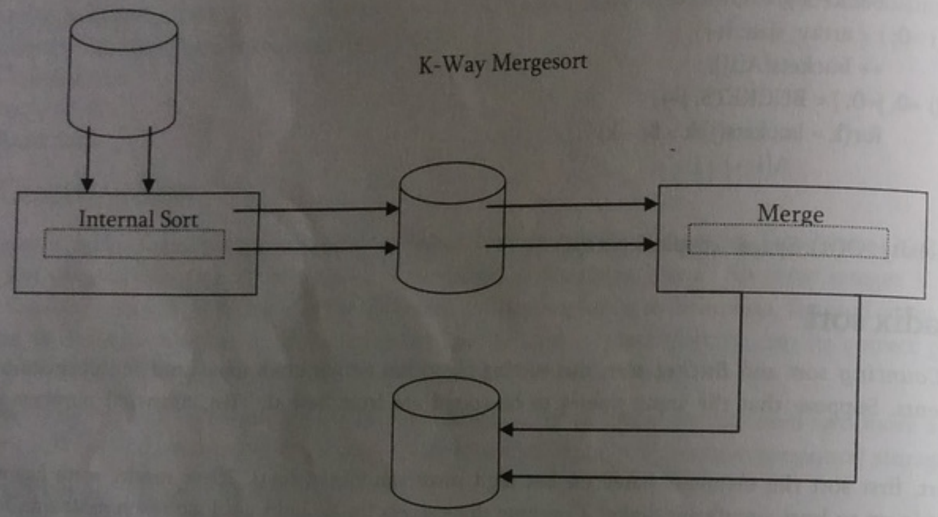
10.18 Topological Sort

Refer *Graph Algorithms* Chapter.

10.19 External Sorting

External sorting is a generic term for a class of sorting algorithms that can handle massive amounts of data. These External sorting algorithms are useful when the files are too big and cannot fit int main memory.

Like internal sorting algorithms, there are number for algorithms for external sorting also. One such algorithm is External Mergesort. In practice these external sorting algorithms are being supplemented by internal sorts.



Simple External Mergesort

A number of records from each tape would be read into main memory and sorted using an internal sort and then output to the tape. For the sake of clarity, let us assume that 900 megabytes of data needs to be sorted using only 100 megabytes of RAM.

1) Read 100MB of the data into main memory and sort by some conventional method (let us say Quick sort).
2) Write the sorted data to disk.
3) Repeat steps 1 and 2 until all of the data is sorted in chunks of 100MB. Now we need to merge them into one single sorted output file.
4) Read the first 10MB of each sorted chunk (call them input buffers) in main memory (90MB total) and allocate the remaining 10MB for output buffer.
5) Perform a 9-way Mergesort and store the result in the output buffer. If the output buffer is full, write it to the final sorted file. If any of the 9 input buffers gets empty, fill it with the next 10MB of its associated 100MB sorted chunk or otherwise mark it as exhausted if there is no more data in the sorted chunk and do not use it for merging.

The above algorithm can be generalized by assuming that the amount of data to be sorted exceeds the available memory by a factor of K. Then, K chunks of data need to be sorted and a K -way merge has to be completed.

If X is the amount of main memory available, there will be K input buffers and 1 output buffer of size $X/(K+1)$ each. Depending on various factors (how fast the hard drive is?) better performance can be achieved if the output buffer is made larger (for example twice as large as one input buffer).

Complexity of the 2-way External Merge sort: In each pass we read + write each page in file. Let us assume that there are n pages in file. That means we need $\lceil logn \rceil + 1$ number of passes. The total cost is $2n(\lceil logn \rceil + 1)$.

10.20 Problems on Sorting

Problem-1 Given an array $A[0 \ldots n-1]$ of n numbers containing repetition of some number. Give an algorithm for checking whether there are repeated elements or not. Assume that we are not allowed to use additional space (i.e., we can use a few temporary variables, O(1) storage).

Solution: Since we are not allowed to use any extra space, one simple way is to scan the elements one by one and for each element check whether that elements appears in the remaining elements. If we find a match we return true.

```
int CheckDuplicatesInArray(in A[], int n) {
        for (int i  =  0; i < n; i++)
                for (int j = i + 1; j < n; j++)
                        if(A[i]==A[j])
                                reutrn true;
        return false;
}
```

Each iteration of the inner, j-indexed loop uses O(1) space, and for a fixed value of i, the j loop executes $n-i$ times. The outer loop executes $n-1$ times, so the entire function uses time proportional to

$$\sum_{i=1}^{n-1} n-i = n(n-1) - \sum_{i=1}^{n-1} i = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2} = O(n^2)$$

Time Complexity: $O(n^2)$. Space Complexity: O(1).

Problem-2 Can we improve the time complexity of Problem-1?

Solution: Yes, using sorting technique.

```
int CheckDuplicatesInArray(in A[], int n) {
        //for heap sort algorithm refer Priority Queues chapter
        Heapsort( A, n );
        for (int i  =  0; i < n-1; i++)
                if(A[i]==A[i+1])
                        reutrn true;
        return false;
}
```

Heapsort function takes $O(nlogn)$ time, and requires O(1) space. The scan clearly takes for $n-1$ iterations, each iteration using O(1) time. The overall time is $O(nlogn + n) = O(nlogn)$.

Time Complexity: $O(nlogn)$. Space Complexity: O(1).

Note: For variations of this problem, refer *Searching* chapter.

Problem-3 Given an array $A[0 \ldots n-1]$, where each element of the array represents a vote in the election. Assume that each vote is given as an integer representing the ID of the chosen candidate. Give an algorithm for determining who wins the election.

Solution: This problem is nothing but finding the element which repeated maximum number of times. Solution is similar to Problem-1 solution: keep track of counter.

```
int CheckWhoWinsTheElection(in A[], int n) {
        int i, j, counter = 0, maxCounter = 0, candidate;
        candidate = A[0];
        for (i  =  0; i < n; i++) {
                candidate = A[i];
                counter = 0;
                for (j   = i + 1; j < n; j++) {
                        if(A[i]==A[j]) counter++;
```

```
            }
            if(counter > maxCounter) {
                maxCounter = counter;
                candidate = A[i];

            }
        }
        return candidate;
}
```

Time Complexity: $O(n^2)$. Space Complexity: $O(1)$.

Note: For variations of this problem, refer *Searching* chapter.

Problem-4 Can we improve the time complexity of Problem-3? Assume we don't have any extra space.

Solution: Yes. The approach is to sort the votes based on candidate ID, then scan the sorted array and count up which candidate so far has the most votes. We only have to remember the winner, so we don't need a clever data structure. We can use heapsort as it is an in-place sorting algorithm.

```
int CheckWhoWinsTheElection(in A[], int n) {
    int i, j, currentCounter = 1, maxCounter = 1;
    int currentCandidate, maxCandidate;
    currentCandidate = maxCandidate= A[0];
    //for heap sort algorithm refer Priority Queues Chapter
    Heapsort( A, n );
    for (int i = 1; i <= n; i++) {
        if( A[i] == currentCandidate)
            currentCounter ++;
        else {  currentCandidate = A[i];
            currentCounter = 1;
        }
        if(currentCounter > maxCounter)
            maxCounter = currentCounter;
        else {  maxCandidate = currentCandidate;
            maxCounter = currentCounter;
        }
    }
    return candidate;
}
```

Since Heapsort time complexity is $O(nlogn)$ and in-place, so it only uses an additional $O(1)$ of storage in addition to the input array. The scan of the sorted array does a constant-time conditional $n - 1$ times, thus using $O(n)$ time. The overall time bound is $O(nlogn)$.

Problem-5 Can we further improve the time complexity of Problem-3?

Solution: In the given problem, number of candidates is less but the number of votes is significantly large. For this problem we can use counting sort.

Time Complexity: $O(n)$, n is the number of votes (elements) in array.
Space Complexity: $O(k)$, k is the number of candidates participated in election.

Problem-6 Given an array A of n elements, each of which is an integer in the range $[1, n^2]$. How do we sort the array in $O(n)$ time?

Solution: If we subtract each number by 1 then we get the range $[0, n^2 - 1]$. If we consider all number as 2 –digit base n. Each digit ranges from 0 to n^2 - 1. Sort this using radix sort. This uses only two calls to counting sort. Finally, add 1 to all the numbers. Since there are 2 calls, the complexity is $O(2n) \approx O(n)$.

Problem-7 For the Problem-6, what if the range is $[1 \ldots n^3]$?

Solution: If we subtract each number by 1 then we get the range $[0, n^3 - 1]$. Consideromg all number as 3-digit base n: each digit ranges from 0 to n^3 - 1. Sort this using radix sort. This uses only three calls to counting sort. Finally, add 1 to all the numbers. Since there are 3 calls, the complexity is $O(3n) \approx O(n)$.

Problem-8 Given an array with n integers each of value less than n^{100}, can it be sorted in linear time?

Solution: Yes. Reasoning is same as that of Problem-6 and Problem-7.

Problem-9 Let A and B be two arrays of n elements, each. Given a number K, give an $O(nlogn)$ time algorithm for determining whether there exists a $\in$ A and b $\in$ B such that $a + b = K$.

Solution: Since we need $O(nlogn)$, it gives us a pointer that we need sorting. So, we will do that.

```
int Find( int A[], int B[], int n, K ) {
    int i, c;
    Heapsort( A, n );                   //O(nlogn)
    for (i =0; i< n; i++) {             //O(n)
        c = k-B[i];                     //O(1)
        if(BinarySearch(A, c))          //O(logn)
            return 1;
    }
    return 0;
}
```

Note: For variations of this problem, refer *Searching* chapter.

Problem-10 Let A, B and C are three arrays of n elements, each. Given a number K, give an $O(nlogn)$ time algorithm for determining whether there exists $a \in A$, $b \in B$ and $c \in C$ such that $a + b + c = K$.

Solution: Refer *Searching* chapter.

Problem-11 Given an array of n elements, can we output in sorted order the K elements following the median in sorted order in time $O(n + KlogK)$.

Solution: Yes. Find the median and partition about the median. With this we can find all the elements greater than it. Now find the K^{th} largest element in this set and partition about it and get all the elements less than it. Output the sorted list of final set of elements. Clearly, this operation takes $O(n + KlogK)$ time.

Problem-12 Consider the sorting algorithms: Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Heap Sort, and Quick Sort. Which of these are stable?

Solution: Let us assume that A is the array to be sorted. Also, let us say R and S are having the same key and R appears earlier in the array than S. That means, R is at $A[i]$ and S is at $A[j]$, with $i < j$. To show any algorithm stable, in the sorted output R must precede S.

Bubble sort: Yes. Elements change order only when a smaller record follows a larger. Since S is not smaller than R it cannot precede it.

Selection sort: No. It divides the array into sorted and unsorted portions and iteratively finds the minimum values in the unsorted portion. After finding a minimum x, if the algorithm moves x into the sorted portion of the array by means of a swap then the element swapped out could be R which then could be moved behind S. This would invert the positions of R and S, so in general it is not stable. If swapping is avoided, it could be made stable but the cost in time would probably be very significant.

Insertion sort: Yes. As presented, when S is to be inserted into sorted subarray $A[1..j-1]$, only records larger than S are shifted. Thus R would not be shifted during S's insertion and hence would always precede it.
Merge sort: Yes, In the case of records with equal keys, the record in the left subarray gets preference. Those are the records that came first in the unsorted array. As a result, they will precede later records with the same key.

Heap sort: No. Suppose $i = 1$ and R and S happen to be the two records with the largest keys in the input. Then R will remain in location 1 after the array is heapified, and will be placed in location n in the first iteration of Heapsort. Thus S will precede R in the output.

Quick sort: No. The partitioning step can swap the location of records many times, and thus two records with equal keys could swap position in the final output.

Problem-13 Consider the same sorting algorithms as that of Problem-12. Which of them are in-place?

Solution:
Bubble sort: Yes, because only two integers are required.

Insertion sort: Yes, since we need to store two integers and a record.

Selection sort: Yes. This algorithm would likely need space for two integers and one record.

Merge sort: No. Arrays need to perform the merge. (If the data is in the form of a linked list, the sorting can be done in-place, but this is a nontrivial modification.)

Heap sort: Yes, since the heap and partially-sorted array occupy opposite ends of the input array.

Quicksort: No, since it is recursive and stores $O(logn)$ activation records on the stack. Modifying it to be non-recursive is feasible but nontrivial.

Problem-14 Among, Quick sort, Insertion sort, Selection sort, Heap sort algorithms, which one needs the minimum number of swaps?

Solution: Selected sort, it needs n swaps only (refer theory section).

Problem-15 What is the minimum number of comparisons required to determine if an integer appears more than $n/2$ times in a sorted array of n integers?

Solution: Refer *Searching* chapter.

Problem-16 **Sort an array of 0's, 1's and 2's:** Given an array A[] consisting $0's$, $1's$ and $2's$, give an algorithm for sorting $A[]$. The algorithm should put all $0's$ first, then all $1's$ and all $2's$ in last.
Example: Input = {0,1,1,0,1,2,1,2,0,0,0,1}, Output = {0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2}

Solution: Use Counting Sort. Since only three elements are there and the maximum value is 2, we need a temporary array with 3 elements.

Time Complexity: $O(n)$. Space Complexity: $O(1)$.

Note: For variations of this problem, refer *Searching* chapter.

Problem-17 Is there any other way of solving the Problem-16?

Solution: Using Quick Sort. Since we know that only 3 elements 0, 1 and 2 are there in the array, we can select 1 as a pivot element for Quick Sort. Quick Sort finds the correct place for 1 by moving all 0's to the left of 1 and all 2's to the right of 1. For doing this it uses only one scan.

Time Complexity: $O(n)$. Space Complexity: $O(1)$.

Note: For efficient algorithm, refer *Searching* chapter.

Problem-18 How do we find the number which appeared maximum number of times in an array?

Solution: One simple approach is to sort the given array and scan the sorted array. While scanning keep track of the elements which occurred maximum number of times.

Algorithm:

```
QuickSort(A, n);
int i, j, count=1, Number=A[0], j=0;
for(i=0;i<n;i++) {
        if(A[j]==A) {
                count++;
                Number=A[j];
        }
        j=i;
}
printf("Number:%d, count:%d", Number, count);
```

Time Complexity = Time for Sorting + Time for Scan = $O(nlogn) + O(n) = O(nlogn)$. Space Complexity: $O(1)$.

Note: For variations of this problem, refer *Searching* chapter.

Problem-19 Is there any other way of solving the Problem-18?

Solution: Using Binary Tree. Create a binary tree with an extra field *count* which indicates the number of times an element appeared in the input. Let us say we have created a Binary Search Tree [BST]. Now, do the In-Order of the tree. In-Order traversal of BST produces sorted list. While doing In-Order traversal keep track of maximum element.

Time Complexity: $O(n) + O(n) \approx O(n)$. First parameter is for constructing the BST and the second parameter is for Inorder Traversal. Space Complexity: $O(2n) \approx O(n)$, since every node in BST needs two extra pointers.

Problem-20 Is there yet other way of solving the Problem-18?

Solution: Using Hash Table: For each element of the given array we use a counter and for each occurrence of the element we increment the corresponding counter. At the end we can just return the element which is having maximum counter.

Time Complexity: $O(n)$. Space Complexity: $O(n)$. For constructing hash table we need $O(n)$.

Note: For efficient algorithm, refer *Searching* chapter.

Problem-21 Given a 2 GB file with one string per line, which sorting algorithm would we use to sort the file and why?

Solution: When we have a size limit of 2GB, it means that we cannot bring all the data into main memory.
Algorithm: How much memory do we have available? Let's assume we have X MB of memory available. Divide the file into K chunks, where $X * K\ 2\ GB$.

- Bring each chunk into memory and sort the lines as usual (any $O(nlogn)$ algorithm).
- Save the lines back to the file.
- Now bring next chunk into memory and sort.
- Once we're done, merge them one by one in case of one set finish bring more data from concerned chunk.

The above algorithm is also known as external sort. Step $3 - 4$ is known as K-way merge. The idea behind going for external sort is the size of data. Since data is huge and we can't bring it to the memory, we need e to go for a disk based sorting algorithm.

Problem-22 **Nearly sorted:** Given an array of n elements, each which is at most K positions from its target position, devise an algorithm that sorts in $O(n\ logK)$ time.

Solution: Divide the elements into n/K groups of size K, and sort each piece in $O(KlogK)$ time, say using Mergesort. This preserves the property that no element is more than K elements out of position. Now, merge each block of K elements with the block to its left.

Problem-23 Is there any other way of solving the Problem-22?

Solution: Insert the first K elements into a binary heap. Insert the next element from the array into the heap, and delete the minimum element from the heap. Repeat.

Problem-24 **Merging K sorted lists:** Given K sorted lists with a total of n elements, give an $O(nlogK)$ algorithm to produce a sorted list of all n elements.

Solution: Simple Algorithm for merging K sorted lists: Consider groups each having $\frac{n}{K}$ elements. Take the first list and merge it with the second list using a linear-time algorithm for merging two sorted lists, such as the merging algorithm used in merge sort. Then, merge the resulting list of $\frac{2n}{K}$ elements with the third list, merge the list of $\frac{3n}{K}$ elements that results with the fourth list. Repeat this until we end up with a single sorted list of all n elements.

Time Complexity: In each iteration we are merging K elements.

$$T(n) = \frac{2n}{K} + \frac{3n}{K} + \frac{4n}{K} + \cdots \frac{Kn}{K}(n) = \frac{n}{K}\sum_{i=2}^{K} i$$

$$T(n) = \frac{n}{K}\left[\frac{K(K+1)}{2}\right] \approx O(nK)$$

Problem-25 Can we improve the time complexity of Problem-24?

Solution: One method is to repeatedly pair up the lists, and merge each pair. This method can also be seen as a tail component of the execution merge sort, where the analysis is clear. This method is called as Tournament Method. Maximum depth of this Tournament Method is $logK$ and in each iteration we are scanning all the n elements.

Time Complexity: O(nlogK).

Problem-26 Is there any other way of solving the Problem-24?

Solution: Other method is to use a *min* priority queue for the minimum elements of each of the K lists. At each step, we output the extracted minimum of the priority queue, and determine from which of the K lists it came, and insert the next element from that list into the priority queue. Since we are using priority queue, that maximum depth of priority queue is $logK$.

Time Complexity: O(nlogK).

Problem-27 Which sorting method is better for Linked Lists?

Solution: Merge Sort is a better choice. At a first appearance, merge sort may not be a good selection since the middle node is required to subdivide the given list into two sub-lists of equal length. We can easily solve this problem by moving the nodes alternatively to two lists would also solve this problem (refer *Linked Lists* chapter). Then, sorting these two lists recursively and merging the results into a single list will sort the given one [32].

```
typedef struct ListNode {
        int data;
        struct ListNode *next;
};
struct ListNode * LinkedListMergeSort(struct ListNode * first) {
        struct ListNode * list1HEAD = NULL;
        struct ListNode * list1TAIL = NULL;
        struct ListNode * list2HEAD = NULL;
        struct ListNode * list2TAIL = NULL;
        if(first==NULL || first→next==NULL)
                return first;
        while (first != NULL) {
                Append(first, list1HEAD, list1TAIL);
                if(first != NULL)
```



```
                        Append(first, list2HEAD, list2TAIL);
        }
        list1HEAD = LinkedListMergeSort(list1HEAD);
        list2HEAD = LinkedListMergeSort(list2HEAD);
        return Merge(list1HEAD, list2HEAD);
}
```

Note: Append() appends the first argument to the tail of a singly linked list whose head and tail are defined by the second and third arguments.
All external sorting algorithms can be used for sorting linked lists since each involved file can be considered as a linked list that can only be accessed sequentially. We can sort a doubly linked list using its next fields as if it is a singly linked one and reconstruct the prev fields after sorting with an additional scan.

Problem-28 Can we implement Linked Lists Sorting with Quick Sort?

Solution: Original Quick Sort cannot be used for sorting the Singly Linked Lists. This is because we cannot move backward in Singly Linked Lists. We can modify the original Quick Sort and make it work for Singly Linked Lists.

Let us consider the following modified Quick Sort implementation. The first node of the input list is considered as a *pivot* and is moved to *equal*. The value of each node is compared with the *pivot* and moved *to less* (*respectively*, *equal* or *larger*) if the nodes value is smaller than (respectively, *equal* to or *larger* than) the *pivot*. Then, *less* and *larger* are sorted recursively. Finally, joining *less*, *equal* and *larger* into a single list yields a sorted one.

Append() appends the first argument to the tail of a singly linked list whose head and tail are defined by the second and third arguments. On return, the first argument will be modified so that it *equal* points to the next node of the list. *Join*() appends the list whose head and tail are defined by the third and fourth arguments to the list whose head and tail are defined by the first and second arguments. For simplicity, the first and fourth arguments become the head and tail of the resulting list [32].

```
typedef struct ListNode {
        int data;
        struct ListNode *next;
};
void Qsort(struct ListNode *first, struct ListNode * last) {
        struct ListNode *lesHEAD=NULL, lesTAIL=NULL;
        struct ListNode *equHEAD=NULL, equTAIL=NULL;
        struct ListNode *larHEAD=NULL, larTAIL=NULL;
        struct ListNode *current = *first;
        int pivot, info;
        if(current == NULL)
                return;
        pivot = current→data;
        Append(current, equHEAD, equTAIL);
        while (current != NULL) {
                info = current→data;
                if(info < pivot)
                        Append(current, lesHEAD, lesTAIL)
                else if(info > pivot)
                        Append(current, larHEAD, larTAIL)
                else
                        Append(current, equHEAD, equTAIL);
        }
        Quicksort(&lesHEAD, &lesTAIL);
```

```
    Quicksort(&larHEAD, &larTAIL);
    Join(lesHEAD, lesTAIL,equHEAD, equTAIL);
    Join(lesHEAD, equTAIL,larHEAD, larTAIL);
    *first = lesHEAD;
    *last = larTAIL;
}
```

Problem-29 Given an array of $1,00,000$ pixel color values and each of which is an integer in the range $[0,255]$. For sorting them which sorting algorithm is preferable?

Solution: Counting Sort. There are only 256 key values, so the auxiliary array would be only of size 256, and there would be only two passes through the data which would be very efficient in both time and space.

Problem-30 Similar to Problem-29, if we have a telephone directory with $1,00,000$ entries, which sorting algorithm is best?

Solution: Bucket sort. In Bucket sort the buckets are defined by the last 7 digits. This requires an auxiliary array of size 10 million, and has the advantage of requiring only one pass through the data on disk. Each bucket contains all telephone numbers with the same last 7 digits but different area codes. The buckets can then be sorted on area code by selection or insertion sort, there are only a handful of area codes.

Problem-31 Give an algorithm for merging K-sorted lists.

Solution: Refer *Priority Queues* chapter.

Problem-32 Given a big file containing billions of numbers. Find maximum 10 numbers from those file.

Solution: Refer *Priority Queues* chapter.

Problem-33 There are two sorted arrays A and B. First one is of size $m+n$ containing only m elements. Another one is of size n and contains n elements. Merge these two arrays into the first array of size $m+n$ such that the output is sorted.

Solution: The trick for this problem is to start filling the destination array from the back with the largest elements. We will end up with a merged and sorted destination array.

```
void Merge(int[] A[], int m, int B[], int n) {
        int count = m;
        int i = n – 1, j = count – 1, k = m - 1;
        for(;k>=0;k--) {
                if(B[i] > A[j] || j < 0) {
                        A[k] =B[i];
                        i--;
                        if(i<0)
                                break;
                }
                else {
                        A[k] = A[j];
                        j--;
                }
        }
}
```

Time Complexity: $O(m+n)$. Space Complexity: $O(1)$.

Problem-34 **Nuts and Bolts Problem:** Given a set of n nuts of different sizes and n bolts such that there is a one-to-one correspondence between the nuts and the bolts, find for each nut its corresponding bolt. Assume that we can only compare nuts to bolts: we cannot compare nuts to nuts and bolts to bolts.

Otherway of asking: We are given a box which contains bolts and nuts. Assume there are n nuts and n bolts and that each nut matches exactly one bolt (and vice versa). By trying to match a bolt and a nut we can see which one is bigger, but we cannot compare two bolts or two nuts directly. Design an efficient algorithm for matching the nuts and bolts.

Solution: Brute Force Approach: start with the first bolt and compare it with each nut until we find a match. In the worst case, we require n comparisons. Repeating this for successive bolt on all remaining gives $O(n^2)$ complexity.

Problem-35 For Problem-34, can we improve the complexity?

Solution: In Problem-34, we got $O(n^2)$ complexity in the worst case (if bolts are in ascending order and nuts are in descending order). Its analysis is same as that of quick sort. The improvement is also on same line.

To reduce the worst case complexity, instead of selecting the first bold every time, we can select some random bolt and match it with nuts. This randomized selection reduces the probability of getting the worst case but still the worst case is $O(n^2)$ only.

Problem-36 For Problem-34, can we further improve the complexity?

Solution: We can use a divide-and-conquer technique for solving this problem and the solution is very similar to randomized Quicksort. For simplicity let us assume that bolts and nuts are represented in two arrays B and N.

The algorithm first performs a partition operation as follows: pick a random bolt $B[i]$. Using this bolt, rearrange the array of nuts into three groups of elements:

- First the nuts smaller than $B[i]$
- Nut that matches $B[i]$, and
- Finally, the nuts larger than $B[i]$.

Next, using the nut that matches $B[i]$ perform a similar partition of the array of bolts. This pair of partitioning operations can easily implemented in $O(n)$ time, and it leaves the bolts and nuts nicely partitioned so that the "*pivot*" bolt and nut are aligned with each-other and all other bolts and nuts are on the correct side of these pivots -- smaller nuts and bolts precede the pivots, and larger nuts and bolts follow the pivots. Our algorithm then completes by recursively applying itself to the subarray to the left and right of the pivot position to match these remaining bolts and nuts. We can assume by induction on n that these recursive calls will properly match the remaining bolts.

To analyze the running time of our algorithm, we can use the same analysis as that of randomized Quicksort. Therefore, applying the analysis from Quicksort, the time complexity of our algorithm is $O(nlogn)$.

Otherway of Analysis: We can solve this problem by making little change to quick sort. Let us assume that we pick the last element as pivot, say it is a nut. Compare the nut with only bolts as we walk down the array. This will partition the array for the bolts. Every bolt less than the partition nut will be on the left. And every bolt greater than the partition nut will be on the right.

While traversing down the list, the matching bolt for the partition nut will have been found. Now we do the partition again using the matching bolt. As a result, all the nuts less than the matching bolt will be on the left side and all the nuts greater than the matching bolt will be on the right side. Recursively call on the left and right arrays.

The time complexity is $O(2nlogn) \approx O(nlogn)$.

Problem-37 Given a binary tree, can we print its elements in sorted order in $O(n)$ time by performing an In-order tree traversal?

Solution: Yes if the tree is a Binary Search Tree [BST]. For more details refer *Trees* chapter.

Chapter-11

SEARCHING

11.1 What is Searching?

In computer science, searching is the process of finding an item with specified properties among a collection of items. The items may be stored as records in a database, simple data elements in arrays, text in files, nodes in trees, vertices and edges in graphs or may be elements of other search space.

11.2 Why Searching?

Searching is one of core computer science algorithms. We know that todays computers store lot of information. To retrieve this information efficiently we need very efficient searching algorithms.

There are certain ways of organizing the data which improves the searching process. That means, if we keep the data in some proper order , it is easy to search the required element. Sorting is one of the techniques for making the elements ordered. In this chapter we will see different searching algorithms.

11.3 Types of Searching

The following are the types of searches which we will be discussing in this book.

- Unordered Linear Search
- Sorted/Ordered Linear Search
- Binary Search
- Symbol Tables and Hashing
- String Searching Algorithms: Tries, Ternary Search and Suffix Trees

11.4 Unordered Linear Search

Let us assume that given an array whose elements order is not known. That means the elements of the array are not sorted. In this case if we want to search for an element then we have to scan the complete array and see if the element is there in the given list or not.

```
int UnsorteddLinearSearch (int A[], int n, int data) {
        for (int i  = 0; i < n; i++) {
                if(A[i] == data)
                        return i;
        }
        return -1;
}
```

Time complexity: $O(n)$, in the worst case we need to scan the complete array. Space complexity: $O(1)$.

11.5 Sorted/Ordered Linear Search

If the elements of the array are already sorted then in many cases we don't have to scan the complete array to see if the element is there in the given array or not. In the below algorithm, it can be seen that, at any point if the value at $A[i]$ is greater than the *data* to be searched then we just return -1 without searching the remaining array.


```
int SortedLinearSearch(int A[], int n, int data) {
        for (int i  = 0; i < n; i++) {
                if(A[i] == data)
                        return i;
                else if(A[i] > data)
                        return -1;
        }
        return -1;
}
```

Time complexity of this algorithm is $O(n)$. This is because in the worst case we need to scan the complete array. But in the average case it reduces the complexity even though the growth rate is same. Space complexity: $O(1)$.

Note: For the above algorithm we can make further improvement by incrementing the index at faster rate (say, 2). This will reduce the number of comparisons for searching in the sorted list.

11.6 Binary Search

Let us consider the problem of searching a word in a dictionary, in general we directly go some approximate page [say, middle page] start searching from that point. If the *name* that we are searching is same then we are done with the search. If the page is before the selected pages then apply the same process for the first half otherwise apply the same process to the second half. Binary search also works in the same way. The algorithm applying such a strategy is referred to as *binary search* algorithm.

```
//Iterative Binary Search Algorithm
int BinarySearchIterative[int A[], int n, int data) {
        int low =  0;
        int high = n-1;
        while (low <= high) {
                mid  = low + (high-low)/2; //To avoid overflow
                if(A[mid] == data)
                        return mid;
                else if(A[mid] < data)
                        low =  mid + 1;
                else    high =  mid - 1;
        }
        return -1;
}
//Recursive Binary Search Algorithm
int BinarySearchRecursive[int A[], int low, int high, int data) {
        int mid  = low + (high-low)/2; //To avoid overflow
        if(A[mid] == data)
                return mid;
        else if(A[mid] < data)
                return BinarySearchRecursive (A, mid + 1, high, data);
        else    return BinarySearchRecursive (A, low, mid - 1 , data);
        return -1;
}
```

Recurrence for binary search is $T(n) = T(\frac{n}{2}) + \Theta(1)$. This is because we are always considering only half of the input list and throwing out the other half. Using *Divide and Conquer* master theorem, we get, $T(n = O(logn)$.

Time Complexity: O($logn$). Space Complexity: O(1) [for iterative algorithm].

11.7 Comparing Basic Searching Algorithms

Implementation	Search-Worst Case	Search-Avg. Case
Unordered Array	n	$n/2$
Ordered Array (Binary Search)	$logn$	$logn$
Unordered List	n	$n/2$
Ordered List	n	$n/2$
Binary Search Trees (for skew trees)	n	$logn$

Note: For discussion on binary search trees refer *Trees* chapter.

11.8 Symbol Tables and Hashing

Refer *Symbol Tables* and *Hashing* chapters.

11.9 String Searching Algorithms

Refer *String Algorithms* chapter.

11.10 Problems on Searching

Problem-1 Given an array of n numbers, give an algorithm for checking whether there are any duplicated elements in the array or not?

Solution: This is one of the simplest problems. One obvious answer to this is, exhaustively searching for duplicated in the array. That means, for each input element check whether there is any element with same value. This we can solve just by using two simple *for* loops. The code for this solution can be given as:

```
void CheckDuplicatesBruteForce(int A[], int n) {
        for(int i = 0; i < n; i++) {
                for(int j = i+1; j < n; j++) {
                        if(A[i] == A[j])    {
                                printf("Duplicates exist: %d", A[i]);
                                return;
                        }
                }
        }
        printf("No duplicates in given array.");
}
```

Time Complexity: O(n^2), for two nested *for* loops. Space Complexity: O(1).

Problem-2 Can we improve the complexity of Problem-1's solution?

Solution: Yes. Sort the given array. After sorting all the elements with equal values come adjacent. Now, just do another scan on this sorted array and see if there are elements with same value and adjacent.

```
void CheckDuplicatesSorting(int A[], int n) {
        Sort(A, n);                          //sort the array
        for(int i = 0; i < n-1; i++) {
                if(A[i] == A[i+1]) {
                        printf("Duplicates exist: %d", A[i]);
                        return;
                }
```

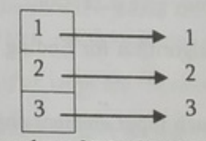


```
        }
        printf("No duplicates in given array.");
}
```

Time Complexity: O($nlogn$), for sorting. Space Complexity: O(1).

Problem-3 Is there any other way of solving the Problem-1?

Solution: Yes, using hash table. Hash tables are a simple and effective method to implement dictionaries. *Average* time to search for an element is O(1), while worst-case time is O(n). Refer *Hashing* chapter for more details on hashing algorithms. As an example, consider the array, $A = \{3, 2, 1, 2, 2, 3\}$. Scan the input array and insert the elements into the hash. For inserted element, keep the *counter* as 1 (assume initially all entires are filled with zeros). This indicates that the corresponding element has occurred already. For the given array, the hash table will look like (after inserting first three elements 3, 2 and 1):



Now if we try inserting 2, since counter value of 2 is already 1, we can say the element is appearing twice.

Time Complexity: O(n). Space Complexity: O(n).

Problem-4 Can we further improve the complexity of Problem-1 solution?

Solution: Let us assume that the array elements are positive numbers and also all the elements are in the range 0 to $n-1$. For each element $A[i]$, go to the array element whose index is $A[i]$. That means we select $A[A[i]]$ and mark - $A[A[i]]$ (that means we negate the value at $A[A[i]]$). Continue this process until we encounter the element whose value is already negated. If one such element exists then we say duplicate elements exist in the given array. As an example, consider the array, $A = \{3, 2, 1, 2, 2, 3\}$.

Initially,

3	2	1	2	2	3
0	1	2	3	4	5

At step-1, negate A[abs(A[0])],

3	2	1	-2	2	3
0	1	2	3	4	5

At step-2, negate A[abs(A[1])],

3	2	-1	-2	2	3
0	1	2	3	4	5

At step-3, negate A[abs(A[2])],

3	-2	- 1	-2	2	3
0	1	2	3	4	5

At step-4, negate A[abs(A[3])],

3	-2	- 1	-2	2	3
0	1	2	3	4	5

At step-4, observe that $A[abs(A[3])]$ is already negative. That means we have encountered the same value twice.

```
void CheckDuplicates(int A[], int n) {
        for(int i = 0; i < n; i++) {
                if(A[abs(A[i])] < 0) {
```