Laboratory Manual

For

Design and Analysis of Algorithm

(IT XXX)

B.Tech (IT)

SEM IV

July 2023

Faculty of Technology
Dharmsinh Desai University
Nadiad.
www.ddu.ac.in

Table of Contents

Sample Experiment

1) **Aim:** implement selection sort algorithm .
2) **Tools/Apparatus:** C /C++ , linux OS.
3) **Standard Procedure:**

3.1) Design and write the algorithm.

```
ALGORITHM: SELECTION SORT (A)
//A contains "n" data to be sorted in ascending order.
1. for j ←1 to n-1
2.        smallest ← j
3.        for I ← j + 1 to n
4.                if A [i] < A [ smallest]
5.                        then smallest ← i
6.        exchange (A [j], A [smallest])
```

3.2) Do apriori analysis of algorithm designed.

Pass	Number of Comparison
1st	(n-1)
2nd	(n-2)
3rd	(n-3)
...	...
last	1

Number of comparisons:(n-1) + (n-2) + (n-3) +.....+ 1 = n(n-1)/2 nearly equals to n^2

Time Complexity = O(n^2) OR

Another Method for analysis-

Select a barometer instruction and find the time complexity in best,worst and average cases. And based on it , conclude that it is O(n^2) as there are two nested loops.

Time Complexities:

- **Worst Case Complexity:** O(n^2)
  If we want to sort in ascending order and the array is in descending order then, the worst case occurs.

- **Best Case Complexity:** O(n^2)
  It occurs when the the array is already sorted

- **Average Case Complexity:** O(n^2)
  It occurs when the elements of the array are in jumbled order (neither ascending nor

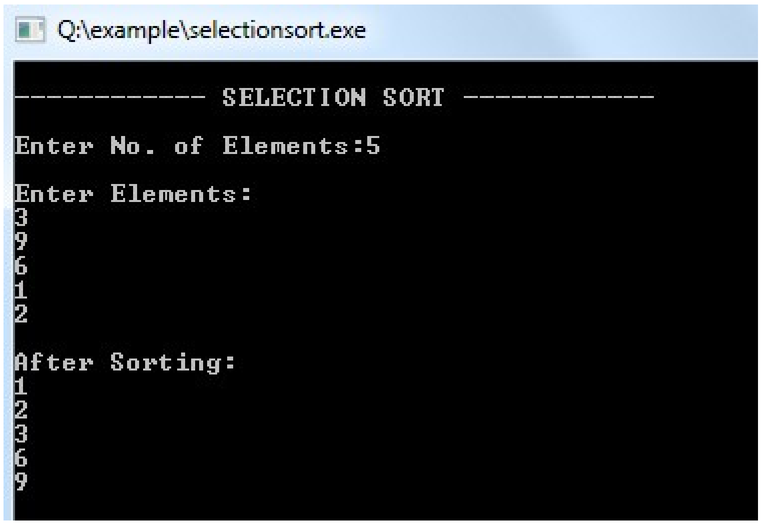
descending).

In Place : Yes, it does not require extra space.
Space complexity :O(1) because an extra variable temp is used.

3.3.) Solve the algorithm in appropriate programming language.

```
// Sample implementation in C++ language
#include <iostream>
using namespace std;

int main()
{
   int a[100], i, n, p, k, min, loc, tmp;

   cout << "\n------------ SELECTION SORT ------------ \n\n";
   cout << "Enter No. of Elements:"; cin >> n;

   cout << "\nEnter Elements:\n";
   for (i = 1; i <= n; i++) { cin >> a[i];
   }

   for (p = 1; p <= n - 1; p++) // Loop for Pass
   {
      min = a[p]; // Element Selection
      loc = p;

      for (k = p + 1; k <= n; k++) // Finding Min Value { if (min > a[k]) {
            min = a[k];
            loc = k;
         }
      }

      tmp = a[p];
      a[p] = a[loc];
      a[loc] = tmp;
   }
   cout << "\nAfter Sorting: \n";
   for (i = 1; i <= n; i++) {
      cout << a[i] << endl;
   }
}
```

Output:



```
Q:\example\selectionsort.exe
------------ SELECTION SORT ------------

Enter No. of Elements:5

Enter Elements:
3
9
6
1
2

After Sorting:
1
2
3
6
9
```

3.4) Execute the various possible Testing cases and note observations.

Type of Data	Size of Data	Actual time taken by algorithm to solve problem (in microseconds)
Unsorted	10,000	0.245
	1,00,000	0.505
	10,00,000	1.09
sorted	10,000	0.249
	1,00,000	0.575
	10,00,000	1.15

4) Give your conclusion of the algorithm features ,based on your observations of theoretical and practical performances .

Conclusion- Selection sort method is having equal time complexity in all cases of data. Hence the selection sort can be useful when:

- Small list is to be sorted
- Cost of swapping does not matter
- Checking of all the elements is compulsory
- Extra memory is not available to do sorting.

Note:- Methods for Timing the procedures→

- **Using tools like gprof (described in experiment1)**
- **Using commands provided by**

  (1) O.S. or (2) Programming language→

Method1) How to find time taken by a command/Problem on Linux Shell?

We can use **time** command for this purpose. The time taken is shown in three forms.
real: Total end to end time taken by Problem/command
user: Time taken in user mode.
sys: Time taken in kernel mode

- **A Command Example (Time taken by ls-l):**

```
$ time ls -l
```

The above command runs "ls -l" and shows
contents of current directory followed by
the time taken by command "ls -l".

- **A Problem example (Time taken by fib(30)):**

  let us consider below Problem.

```
#include<stdio.h>
int fib(int n)
{
  if (n <= 1)
     return n;
  return fib(n-1) + fib(n-2);
}

int main ()
{
  printf("Fibonacci Number is %d", fib(30));
  return 0;
}
```

Let we save above Problem as fib.c.

```
// Compiling above Problem on shell
~$ gcc fib.c

// Running the generated executable with time
~$ time ./a.out
Fibonacci Number is 832040
                         real    0m0.017s
                         user    0m0.017s
```

```
sys    0m0.000s
```

Note: 0.017 seconds (shown with real) is total time taken by Problem.

Method2) Using Programming language timing functions:

- **How to measure time taken by a function in C?**

To calculate time taken by a process, we can use clock() function which is available *time.h*. We can call the clock function at the beginning and end of the code for which we measure time, subtract the values, and then divide by CLOCKS_PER_SEC (the number of clock ticks per second) to get processor time, like following.

```
#include <time.h>

clock_t start, end;
double cpu_time_used;

start = clock();
... /* Do the work. */
end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

Example:- Following is a sample C Problem where we measure time taken by fun(). The function fun() waits for enter key press to terminate.

```
/* Problem to demonstrate time taken by function fun() */
#include <stdio.h>
#include <time.h>

// A function that terminates when enter key is pressed
void fun()
{
   printf("fun() starts \n");
   printf("Press enter to stop fun \n");
   while(1)
   {
      if (getchar())
         break;
   }
   printf("fun() ends \n");
}

// The main Problem calls fun() and measures time taken by fun()
int main()
{
   // Calculate the time taken by fun()
   clock_t t;
   t = clock();
   fun();
   t = clock() - t;
   double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds

   printf("fun() took %f seconds to execute \n", time_taken);
   return 0;
}
```

Output: The following output is obtained after waiting for around 4 seconds and then hitting enter key.

```
fun() starts
Press enter to stop fun

fun() ends
fun() took 4.017000 seconds to execute
```

<u>Another method:-</u>

```
#include <sys/time.h>

struct timeval  tv1, tv2;
gettimeofday(&tv1, NULL);
/* stuff to do! */
gettimeofday(&tv2, NULL);

printf ("Total time = %f seconds\n",
        (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 +
        (double) (tv2.tv_sec - tv1.tv_sec));
```

Note that this measures in microseconds, not just seconds.

<u>Another method:-</u>

- On linux use: clock_gettime(CLOCK_MONOTONIC_RAW, &time_variable);

  It's not affected if the system-admin changes the time.

```
#include <time.h>
#include <unistd.h> /* for sleep() */
int main() {
   struct timespec begin, end;
   clock_gettime(CLOCK_MONOTONIC_RAW, &begin);
```

```
  sleep(1);      // waste some time
  clock_gettime(CLOCK_MONOTONIC_RAW, &end);
  printf ("Total time = %f seconds\n",
         (end.tv_nsec - begin.tv_nsec) / 1000000000.0 +
         (end.tv_sec  - begin.tv_sec));
}
```

EXPERIMENT - 1

Aim:Study usage of GNU profiler tool and other time related functions . And use it to analyze recursive and iterative solutions of the problems: Fibonacci and Tower of Hanoi.

Procedure(for using gprof to profile the file):

1. Write a program in .c file. Example- myprog.c
2. Compile a program using gcc filename-pg option.
   ```
   $gcc -pg myprog.c
   ```
3. Execute output file.
   ```
   $ ./a.out
   ```
4. Do profiling
   ```
   $ gprof > filename.txt
   ```
5. Analyze the profile results obtained
   ```
   $ vi filename.txt
   ```

- Output Sample of gprof:-

```
                 Flat profile:

                 Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
...
```

The functions are sorted first by decreasing run-time spent in them, then by decreasing number of calls, then alphabetically by name.

Here is what the fields in each line mean:

- % time

This is the percentage of the total execution time your program spent in this function. These should all add up to 100%.

- cumulative seconds

This is the cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table.

- self seconds

This is the number of seconds accounted for by this function alone. The flat profile listing is sorted first by this number.

- calls

This is the total number of times the function was called. If the function was never called, or the number of times it was called cannot be determined (probably because the function was not compiled with profiling enabled), the calls field is blank.

- self ms/call

This represents the average number of milliseconds spent in this function per call, if this function is profiled. Otherwise, this field is blank for this function.

- total ms/call

This represents the average number of milliseconds spent in this function and its descendants per call, if this function is profiled. Otherwise, this field is blank for this function. This is the only field in the flat profile that uses call graph analysis.

- name

This is the name of the function. The flat profile is sorted by this field alphabetically after the self seconds and calls fields are sorted.

<u>The Call Graph</u>

The call graph shows how much time was spent in each function and its children. From this information, you can find functions that, while they themselves may not have used much time, called other functions that did use unusual amounts of time.

```
granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds

index % time    self  children    called     name

                                         <spontaneous>

[1]    100.0    0.00    0.05               start [1]

               0.00    0.05       1/1          main [2]

               0.00    0.00       1/2          on_exit [28]

               0.00    0.00       1/1          exit [59]

-----------------------------------------------

               0.00    0.05       1/1          start [1]

[2]    100.0    0.00    0.05       1         main [2]

               0.00    0.05       1/1          report [3]
```

```
-----------------------------------------------

            0.00    0.05       1/1          main [2]

-----------------------------------------------
```

The lines full of dashes divide this table into entries, one for each function. Each entry has one or more lines.

In each entry, the primary line is the one that starts with an index number in square brackets. The end of this line says which function the entry is for. The preceding lines in the entry describe the callers of this function and the following lines describe its subroutines (also called children when we speak of the call graph).

The entries are sorted by time spent in the function and its subroutines.

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function.

The lines above it list the functions that called this function, and the lines below it list the functions this one called.

This line lists:

- index A unique number given to each element of the table.
- Index numbers are sorted numerically.

The index number is printed next to every function name so it is easier to look up where the function in the table.

- % time- This is the percentage of the `total' time that was spent in this function and its children. Note that due to

different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.

- self -This is the total amount of time spent in this function.
- children -This is the total amount of time propagated into this function by its children.
- called -This is the number of times the function was called.If the function called itself recursively, the number only includes non-recursive calls, and is followed by a `+' and the number of recursive calls.
- name- The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

- self -This is the amount of time that was propagated directly from the function into this parent.
- children -This is the amount of time that was propagated fromthe function's children into this parent.
- called- This is the number of times this parent called the function `/' the total number of times the function was called. Recursive calls to the function are not included in the number after the `/'.

- name- This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number. If the parents of the function cannot be determined, the word `' is printed in the `name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

- self - This is the amount of time that was propagated directly from the child into the function.
- children This is the amount of time that was propagated from the child's children to the function.
- called This is the number of times the function called this child `/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the `/'.
- name This is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number. If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.) The `+' recursive calls entry shows the number of function calls that were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

- Observation Table:-

Tower of Hanoii / Fibonacci series

Number of Discs/ Total elements of series	Time reported by gprof (recursive method)	Time reported by gprof (Iterative method)	Average Time reported by timing method (recursive solution)	Average Time reported by timing method (Iterative solution)
n=10	0.XXX micro seconds	0.XXX micro seconds	0.XXX micro seconds	0.XXX micro seconds
...	...	..		
...	..	..		
n=20	0.XXX micro seconds	0.XXX micro seconds	0.XXX micro seconds	0.XXX micro seconds
….				
….				

EXPERIMENT - 2

Aim: Do empirical comparison of following algorithms for solving searching problems. -

2.1) Sequential Search

2.2) Binary Search

Procedure:-

2.1) The sequential search (sometimes called a linear search) is the simplest type of search, it is used when a list of integers is not in any order. It examines the first element in the list and then examines each "sequential" element in the list until a match is found.

2.2) Binary Search is a searching algorithm for finding an element's position in a sorted array. In this approach, the element is always searched in the middle of a portion of an array. Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

EXPERIMENT –3

Aim: Do empirical comparison of following
algorithms for solving sorting problems. -

3.1 Selection Sort

3.2 Insertion Sort

Procedure 3.1: Selection Sort-

This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

Procedure 3.2 : Insertion Sort-

The idea behind the insertion sort is that first take one element, iterate it through the sorted array.

EXPERIMENT – 4

Aim: Do empirical comparison of following algorithms for solving sorting problems. -

4.1 Quick Sort

4.2 Merge Sort

Procedure:

4.1) Quick Sort-

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value. Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.

4.2) Procedure Merge Sort-

Merge sort is a sorting technique based on divide and conquer technique.
Merge sort first divides the array into equal halves and then combines them in a sorted manner.

EXPERIMENT – 5

Aim: Solve and compare the Prim's and Kruskal's algorithm, for solving the Graph problem to find a minimum spanning tree.

Procedure:-

Prim: - Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. It finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized. It starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

Kruskal:- In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached.

EXPERIMENT –6

Aim: Solve the Graph problem to find shortest path, using greedy technique- Dijkstra’s algorithm.

Procedure:

- Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
- The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
- Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.
- The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

EXPERIMENT – 7

Aim: Solve the knapsack using dynamic programming technique.

Procedure:
First, we create a 2-dimensional array (i.e. a table) of n + 1 rows and w + 1 columns. A row number i represents the set of all the items from rows 1— i. For instance, the values in row 3 assumes that we only have items 1, 2, and 3.
A column number j represents the weight capacity of our knapsack. Therefore, the values in column 5, for example, assumes that our knapsack can hold 5 weight units. An entry in row i, column j represents the maximum value that can be obtained with items 1, 2, 3 … i, in a knapsack that can hold j weight units.
At row 0, when we have no items to pick from, the maximum value that can be stored in any knapsack must be 0. Similarly, at column 0, for a knapsack which can hold 0 weight units, the maximum value that can be stored in it is 0.
The relationship between the value at row i, column j and the values to the previous sub-problems is as follows:

Now, at row i and column j, we are tackling a sub-problem consisting of items 1, 2, 3 … i with a knapsack of j capacity. There are 2 options at this point: we can either include item i or not. Therefore, we need to compare the maximum value that we can obtain with and without item i.

The maximum value that we can obtain without item i can be found at row i-1, column j. The reasoning is straightforward: whatever maximum value we can obtain with items 1, 2, 3 … i must obviously be the same maximum value we can obtain with items 1, 2, 3 … i - 1, if we choose not to include item i.

To calculate the maximum value that we can obtain with item i, we first need to compare the weight of item i with the knapsack's weight capacity. Obviously, if item i weighs more than what the knapsack can hold, we can't include it, so it does not make sense to perform the calculation. In that case, the solution to this problem is simply the maximum value that we can obtain without item i (i.e. the value in the row above, at the same column).However, suppose that item i weighs less than the knapsack's capacity. We thus have the option to include it, if it potentially increases the maximum obtainable value. The maximum obtainable value by including item i is thus = the value of item i itself + the maximum value that can be obtained with the remaining capacity of the knapsack. We obviously want to make full use of the capacity of our knapsack, and not let any remaining capacity go to waste.
Therefore, at row i and column j (which represents the maximum value we can obtain there), we would pick either the maximum value that we can obtain without item i, or the maximum value that we can obtain with item i, whichever is larger.
Once the table has been populated, the final solution can be found at the last row in the last column, which represents the maximum value obtainable with all the items and the full capacity of the knapsack.

EXPERIMENT – 8

Aim: Solve the string editing problem using dynamic programming technique.

Procedure:

Let d be the edit distance between two sub sequences, which can have at most n or m edits to transform the string into the other.

Use a n x m matrix where the first row and column are given to us (as we know that we can have at most n or m edits, this is the same as the index).In dynamic programming, we solve the sub problems to solve the more complex problem. In this case the complex problem is the entire sequence, and the simple problem is the smallest subsequence at each iteration (character level). We start at d(1,1), and evaluate the 3 options

$$d(i, j) = \min \begin{cases} d(i-1, j) + 1 & \textit{(deleting } x_i\textit{)} \\ d(i, j-1) + 1 & \textit{(inserting } y_j\textit{)} \\ d(i-1, j-1) + t(i, j) & \textit{(substituting } x_i \textit{ with } y_j\textit{, or not)} \end{cases}$$

After filling out that position, we move to (x+1,y) and complete the row.

Our target is d(n,m), where we would have solved every sub problem along the way, and can trace back our edit steps through linking the edits. Tracking the least number of changes needed at each row may not be useful, for example, knowing that a subsequence of X and Y has a longer edit distance than a different subsequence. If we trace the diagonal, it will show to be more useful in real-world use cases. Simply go to the target at distance_matrix(n,m) to get the least number of edits to make the sequences the same.

EXPERIMENT – 9

Aim: Solve the n-queens problem using backtracking technique.

Procedure:

In N queen problem, we have to arrange n number of queens on n*n chessboard such that no two queens can be on same row, same column or same diagonal. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false.

EXPERIMENT – 10

Aim: Write a program that implements Knapsack using backtracking.

Procedure:

1. Define the solution space of the problem

For each item, there are only two results, load and no load, but whether the item is loaded or not is unknown. Therefore, for convenience, the variable Xi can be defined to indicate whether the ith item is loaded. If it is loaded, the value of Xi is 1, otherwise it is 0. The solution of the problem is an n-tuple, and the value of each component is 0 or 1, The solution space of the problem is {X1, X2, X3, ⋯⋯, Xn}, where Xi is 0 or 1 and i is 1, 2, 3, 4, ⋯⋯, n.

2. Determine the organizational structure of the solution space

The solution of the problem is the number of all subsets of a set composed of n elements.

For example, if the value of n is 3 (i.e. there are 3 items in total), the solution space is {0, 0, 0}, {0, 0, 1}, {0, 1, 1}, {1, 1, 1}, {0, 1, 0}, {1, 0, 1}, {1, 1, 0}.

It can be seen that the depth of the solution space tree of the problem is the scale n of the problem (that is, the total number of items).

3. Search solution space

Constraints:

Σ wi Xi < = w (the lower bound is i=1 and the upper bound is n)

search process:

Starting from the root node, the depth first search. The root node first becomes the live node and the current expansion node. The left branch is defined as 1, that is, the left subtree indicates that the item is placed in the shopping cart, and the right branch is 0, indicating that it is not placed in the shopping cart. Start to expand along the left branch, and then you need to judge whether the limiting conditions are met. If so, continue to expand along the left branch, No Then expand along the right branch. If not, go back to the nearest parent node and continue to expand other cases until all cases are considered.

EXPERIMENT – 11

Aim**:** Write a program that implements Make a change using dynamic. Procedure:

Make change for n units using the least possible number of coins. C is candidate sets which contains different amount of coins.

S is the solution set which contains total number of coins.

Procedure:

We need to use a 2D array (i.e memo table) to store the subproblem's solution.
Size of dpTable is (number of coins +1)*(Total Sum +1)
First column value is 1 because if total amount is 0, then is one way to make the change (we do not include any coin).
Row: Number of coins. The 1st-row index is 0 means no coin is available. 2nd row is 1, it means only 1st coin is available, similarly, the 3rd-row value index is 2 means the first two coins are available to make to the total amount and so on.Row index represent index of coin in coins array not the coin value.
Coulmn: Total Amount (sum). The 1st-column index is 0, it means sum value is 0. 2nd column index is 1 therefore combination of coins should make the sum of 1, similarly, 3rd column value is 2, means change of 2 is required and so on.
Thus each table field is storing the solution of subproblems. For Example, dpTable[2][3]=2 means there are 2 ways to get the sum of 3 using the first two coins {1,2}.

The last column and last row value will give the final result.

Here We need to loop through all the indexes (except 1st row and 1st column) in the memo table and make use of previously stored solutions of the subproblems.

If coin value is greater than the dpSum, then do not consider the coin i.e dpTable[i][j]=dpTable[i-1][j].
If coin value is smaller than the dpSum we can consider the coin i.e dpTable[i][j]=dpTable[i-1][dpSum]+dpTable[i][j-coins[i-1]].

EXPERIMENT – 12

Aim: Write a program that implements All pair shortest path problem.

Procedure:

All pair shortest path algorithm finds length of the shortest path between each pair of nodes.

C is a candidate set and S is the solution set.

References

Reference books:

- Fundamentals of Computer Algorithms by Horowitz, Sahni, Galgotia Pub. 2001 ed.
- Fundamentals of Algorithms by Brassard & Brately, PHI. Introduction to Algorithms by Coreman, Tata McGraw Hill.
- Design & Analysis of Computer Algorithms, Aho, Ullman, Addision Wesley

The art of Computer Programming Vol.I & III, Kunth, Addision Wesley.