**Batch: A3 Roll No.: 16010121045**

**Experiment No. 1**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title: Implementation of selection sort/ Insertion sort** |

**Objective:** To analyse performance of sorting methods

**CO to be achieved:**

|  |  |
| --- | --- |
| CO 1 | Analyze the asymptotic running time and space complexity of algorithms. |

**Books/ Journals/ Websites referred:**

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran,” Fundamentals of computer algorithm”, University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein,” Introduction to algortihtms”,2nd Edition ,MIT press/McGraw Hill,2001**
3. [**http://en.wikipedia.org/wiki/Insertion\_sort**](http://en.wikipedia.org/wiki/Insertion_sort)
4. [**http://www.sorting-algorithms.com/insertion-sort**](http://www.sorting-algorithms.com/insertion-sort)
5. [**http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Insertion\_sort.html**](http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Insertion_sort.html)
6. [**http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/insertionSort.htm**](http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/insertionSort.htm)
7. [**http://en.wikipedia.org/wiki/Selection\_sort**](http://en.wikipedia.org/wiki/Selection_sort)
8. [**http://www.sorting-algorithms.com/selection-sort**](http://www.sorting-algorithms.com/selection-sort)
9. [**http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/selectionSort.htm**](http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/selectionSort.htm)
10. **http://courses.cs.vt.edu/~csonline/Algorithms/Lessons/SelectionCardSort/selectioncardsort.html**

**Pre Lab/ Prior Concepts:**

Data structures, sorting techniques.

**Historical Profile:**

There are various methods to sort the given list. As the size of input changes, the performance of these strategies tends to differ from each other. In such case, the priori analysis can helps the engineer to choose the best algorithm.

**New Concepts to be learned:**

Space complexity, time complexity, size of input, order of growth.

**Topic: Sorting Algorithms**

**Theory:** Given a function to compute on n inputs the divide-and-conquer strategy suggests splitting the inputs into k distinct subsets, 1< k ≤n, yielding k sub problems. These sub problems must be solved and then a method must be found to combine sub solutions into a solution of the whole. If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied. Often the sub problems resulting from a divide-and-conquer design are the same type as the original problem. For those cases the reapplication of the divide-and- conquer principle is naturally expressed by a recursive algorithm. Now smaller and smaller sub problems of the same kind are generated until eventually sub problems that are small enough to be solved without splitting are produced.

**Algorithm Insertion Sort**

INSERTION\_SORT (*A,n*)

//The algorithm takes as parameters an array *A*[1.. *n*] and the length *n* of the array.

//The array *A* is sorted in place: the numbers are rearranged within the array

// A[1..n] of eletype, n: integer

**FOR** j ← 2 **TO** length[*A*]   
             **DO**  key ← *A*[*j*]      
                   {Put *A*[*j*] into the sorted sequence *A*[1 . . *j* − 1]}     
                    *i* ← *j* − 1      
                    **WHILE** *i* > 0 and *A*[*i*] > key  
                                 **DO** *A*[*i* +1] ← *A*[*i*]              
                                         *i* ← *i* − 1       
                     *A*[*i* + 1] ← key

**Algorithm Selection Sort**

SELECTION\_SORT (A,n)

//The algorithm takes as parameters an array *A*[1.. *n*] and the length *n* of the array.

//The array *A* is sorted in place: the numbers are rearranged within the array

// A[1..n] of eletype, n: integer

**FOR** *i* ← 1 **TO** *n*-1 **DO**    
    min *j* ← *i*;  
    min *x* ← A[*i*]  
   **FOR** *j* ← *i* + 1 to n do  
        **IF** A[*j*] < min x then  
            min *j* ← *j*  
            min *x* ← A[j]  
    A[min *j*] ← A [*i*]  
    A[*i*] ← min *x*

**Code:**

*#include* <bits/stdc++.h>

using namespace std;

void insertion(long \**arr*, int *n*)

{

*for* (long i = 0; i < *n* - 1; i++)

{

long key = *arr*[i + 1];

*for* (long j = i + 1; j > 0; j--)

*if* (*arr*[j] < *arr*[j - 1])

{

long temp = *arr*[j];

*arr*[j] = *arr*[j - 1];

*arr*[j - 1] = temp;

}

}

}

void selection(long *arr*[], int *n*)

{

*for* (long i = 0; i < *n* - 1; i++)

{

long min = i;

*for* (long j = i + 1; j < *n*; j++)

*if* (*arr*[j] < *arr*[min])

min = j;

long temp = *arr*[min];

*arr*[min] = *arr*[i];

*arr*[i] = temp;

}

}

int main()

{

long n = 10000;

double tim1[10], tim2[10];

*for* (int j = 0; j < 10; j++)

{

long int arr1[n], arr2[n];

*for* (int i = 0; i < n; i++)

{

arr1[i] = n - i;

arr2[i] = n - i;

}

clock\_t start, end;

start = clock();

ios\_base::sync\_with\_stdio(false);

insertion(arr1, n);

end = clock();

tim1[j] = ((double) (end - start)) / CLOCKS\_PER\_SEC;

start = clock();

selection(arr2, n);

end = clock();

tim2[j] = ((double) (end - start)) / CLOCKS\_PER\_SEC;

cout << "n= " << n << " Insertion = "<< tim1[j] << setprecision(5)<< " Selection = " << tim2[j] << endl;

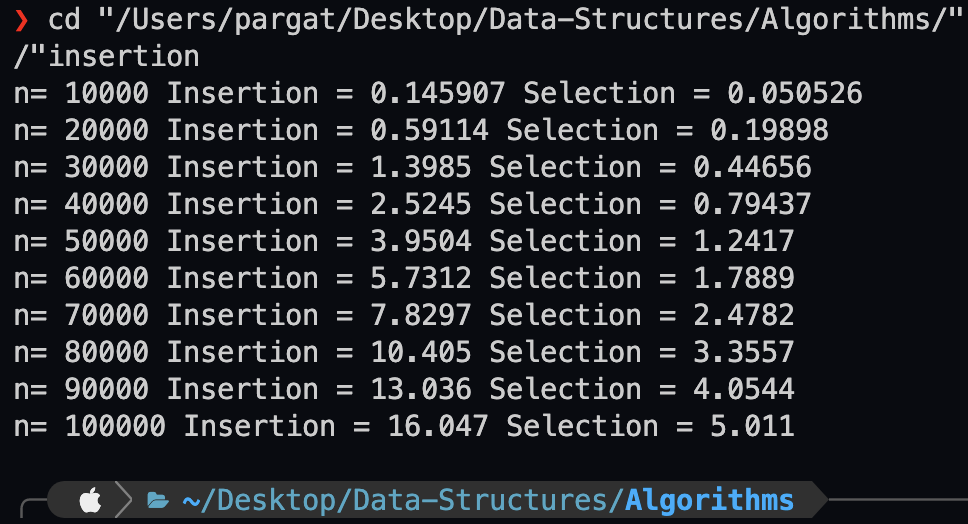
n += 10000;

}

*return* 0;

}

**Output:**

****

**The space complexity of Insertion sort: O(n)**

It takes in a total of n + 5 elements of space

Hence O(n)

**The space complexity of Selection sort: O(n)**

It takes in a total of n + 5 elements of space

Hence O(n)

**Time complexity for Insertion sort: O(n2)**

(n-1) + (n(n-1))/2 = (n2+n+2)/2

Hence O(n2)

**Time complexity for selection sort: O(n2)**

(n-1) + (n(n-1))/2 = (n2+n+2)/2

Hence O(n2)

**Graphs for varying input sizes: (Insertion Sort & Selection sort)**

Both of them have the same time complexity O(n2 ), but selection sort has been proven to be worse than insertion sort for large arrays.

**CONCLUSION:**

Understood the logic behind insertion sort and selection sort and the analysis of their space and time complexities.

**Batch: A3 Roll No.: 16010121045**

**Experiment No. 2**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title: Implementation of Binary search/Max-Min algorithm** |

**Objective:** To learn the divide and conquer strategy of solving the problems of different types

**CO to be achieved:**

|  |  |
| --- | --- |
| CO 2 | Describe various algorithm design strategies to solve different problems and analyse Complexity. |

**Books/ Journals/ Websites referred:**

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran,” Fundamentals of computer algorithm”, University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein,” Introduction to algortihtms”,2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://en.wikipedia.org/wiki/Binary\_search\_algorithm**
4. **https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Binary\_search\_algorithm.html**
5. **http://video.franklin.edu/Franklin/Math/170/common/mod01/binarySearchAlg.html**
6. **http://xlinux.nist.gov/dads/HTML/binarySearch.html**
7. **https://www.cs.auckland.ac.nz/software/AlgAnim/searching.html**

**Pre Lab/ Prior Concepts:**

Data structures

**Historical Profile:**

Finding maximum and minimum or Binary search are few problems those are solved with the divide-and-conquer technique. This is one the simplest strategies which basically works on dividing the problem to the smallest possible level.

Binary Search is an extremely well-known instance of divide-and-conquer paradigm. Given an ordered array of n elements, the basic idea of binary search is that for a given element , "probe" the middle element of the array. Then continue in either the lower or upper segment of the array, depending on the outcome of the probe until the required (given) element is reached.

**New Concepts to be learned:**

Number of comparisons, Application of algorithmic design strategy to any problem, Classical problem solving Vs Divide-and-Conquer problem solving.

**Algorithm IterativeBinarySearch**

int binary\_search(int A[ ], int key, int imin, int imax)

//The algorithm takes as parameters an array *A*[1.. *n*] , the search key and lower-higher index pair of the array.

// Output- The algorithm returns index of the search key in the given array, if it’s present.

{

// continue searching while [imin, imax] is not empty

**WHILE** (imax >= imin)

{

// calculate the midpoint for roughly equal partition

int imid = midpoint(imin, imax);

**IF**(A[imid] == key)

// key found at index imid

return imid;

// determine which subarray to search

**ELSE** **If** (A[imid] < key)

// change min index to search upper subarray

imin = imid + 1;

**ELSE**

// change max index to search lower subarray

imax = imid - 1;

}

// key was not found

**RETURN** KEY\_NOT\_FOUND;

}

**The space complexity of Iterative Binary Search:**

The space complexity of iterative binary search is O(1) .

It means that it only requires a constant amount of extra space, regardless of the size of the input array. It only needs two variables to keep track of the range of elements that are to be checked.

**Algorithm Recursive Binary Search**

int binary\_search(int A[], int key, int imin, int imax)

//The algorithm takes as parameters an array *A*[1.. *n*] , the search key and lower-higher index pair of the array.

// Output- The algorithm returns index of the search key in the given array, if it’s present.

{

// test if array is empty

**IF** (imax < imin)

// set is empty, so return value showing not found

**RETURN** KEY\_NOT\_FOUND;

**ELSE**{

// calculate midpoint to cut set in half

int imid = midpoint(imin, imax);

// three-way comparison

**IF** (A[imid] > key)

// key is in 🡨 lower subset

**RETURN** binary\_search(A, key, imin, imid-1);

**ELSE IF** (A[imid] < key)

// key is in 🡪 higher subset

**RETURN** binary\_search(A, key, imid+1, imax);

**ELSE**

// key has been found

**RETURN** imid;

}

}

**The space complexity of Recursive Binary Search:**

The space complexity of recursive binary search is O(logN).

It means that it requires a logarithmic amount of extra space, proportional to the size of the input array. This is because in the worst case, there will be logN recursive calls and all these recursive calls will be stacked in memory.

**The Time complexity of Binary Search:**

The time complexity of recursive binary search is O(log n) where n is the number of elements in the sorted array. This means that in each iteration or recursive call, the search gets reduced to half of the array size.

**Binary search code:**

*#include* <bits/stdc++.h>

using namespace std;

void binary(int \*arr, int ele, int n)

{

int l = 0, r = n - 1;

*while* (l <= r)

{

int mid = (l + r) / 2;

*if* (arr[mid] == ele)

{

cout << mid << endl;

*break*;

}

*else* *if* (arr[mid] > ele)

r = mid - 1;

*else*

l = mid + 1;

}

}

void search(int \*arr, int l, int r, int ele)

{

*if* (arr[(l + r) / 2] == ele)

{

cout << (l + r) / 2 << endl;

*return*;

}

*if* (l >= r)

{

cout << "Not Found" << endl;

*return*;

}

*if* (arr[(l + r) / 2] > ele)

search(arr, l, (l + r) / 2 - 1, ele);

*else*

search(arr, (l + r) / 2 + 1, r, ele);

}

int main()

{

int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

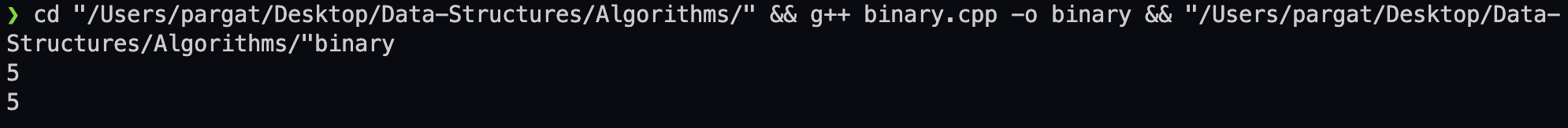
binary(arr, 6, 10);

search(arr, 0, 10, 6);

*return* 0;

}

**Output:**

****

**Algorithm StraightMaxMin:**

**VOID** StraightMaxMin (Type a[], int n, Type& max, Type& min)

// Set max to the maximum and min to the minimum of a[1:n].

{ max = min = a[1];

**FOR** (int i=2; i<=n; i++)

{

**IF** (a[i]>max) then max = a[i];

**IF** (a[i]<min) min = a[i];

}

}

**Algorithm: Recursive Max-Min**

**VOID** MaxMin(int i, int j, Type& max, Type& min)

// A[1:n] is a global array. Parameters i and j are integers, 1 <= i <= j <= n.

//The effect is to set max and min to the largest and smallest values in a[i:j], respectively.

{

**IF** (i == j) max = min = a[i]; // Small(P)

**ELSE IF** (i == j-1) { // Another case of Small(P)

**IF** (a[i] < a[j])

max = a[j]; min = a[i];

**ELSE** { max = a[i]; min = a[j];

}

**ELSE** { Type max1, min1;

// If P is not small divide P into sub problems. Find where to split the set.

int mid=(i+j)/2;

// solve the sub problems.

MaxMin(i, mid, max, min);

MaxMin(mid+1, j, max1, min1);

// Combine the solutions.

**IF** (max < max1) max = max1;

**IF** (min > min1) min = min1;

}

}

*#include* <bits/stdc++.h>

using namespace std;

int recMax(int arr[], int len)

{

*if* (len == 1)

*return* arr[0];

*return* max(arr[len - 1], recMax(arr, len - 1));

}

int recMin(int arr[], int len)

{

*if* (len == 1)

*return* arr[0];

*return* min(arr[len - 1], recMin(arr, len - 1));

}

void minMax(int\* arr,int n){

int max=arr[0],min=arr[0];

*for*(int i=1;i<n;i++){

*if*(arr[i]>max)

max=arr[i];

*if*(arr[i]<min)

min=arr[i];

}

cout<<max<<" "<<min<<endl;

}

int main()

{

int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

cout << recMax(arr, 10) << endl;

cout << recMin(arr, 10) << endl;

minMax(arr,10);

*return* 0;

}

**The Time complexity of Max-Min:**

Time complexity is O(n)

**Space complexity for Max-Min:**

Space complexity is O(1).

**Code:**

*#include* <bits/stdc++.h>

using namespace std;

int recMax(int arr[], int len)

{

*if* (len == 1)

*return* arr[0];

*return* max(arr[len - 1], recMax(arr, len - 1));

}

int recMin(int arr[], int len)

{

*if* (len == 1)

*return* arr[0];

*return* min(arr[len - 1], recMin(arr, len - 1));

}

void minMax(int\* arr,int n){

int max=arr[0],min=arr[0];

*for*(int i=1;i<n;i++){

*if*(arr[i]>max)

max=arr[i];

*if*(arr[i]<min)

min=arr[i];

}

cout<<max<<" "<<min<<endl;

}

int main()

{

int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

cout << recMax(arr, 10) << endl;

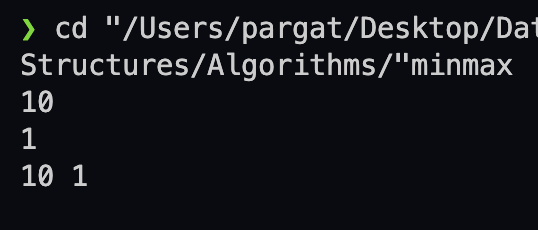
cout << recMin(arr, 10) << endl;

minMax(arr,10);

*return* 0;

}

**Output:**

****

**CONCLUSION:**

The divide and conquer strategy solves problems by dividing them into smaller subproblems and combining their solutions. Binary search and min-max are two examples of this strategy that can find an element or a pair of elements in an array efficiently.

**Batch: A3 Roll No.: 16010121045**

**Experiment No. 3**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title: Implementation of Quick sort/Merge sort algorithm** |

**Objective:** To learn the divide and conquer strategy of solving the problems of different types

**CO to be achieved:**

|  |  |
| --- | --- |
| CO 2 | Describe various algorithm design strategies to solve different problems and analyze Complexity. |

**Books/ Journals/ Websites referred:**

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran,” Fundamentals of computer algorithm”, University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein,” Introduction to algortihtms”,2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://en.wikipedia.org/wiki/Quicksort**
4. **https://www.cs.auckland.ac.nz/~jmor159/PLDS210/qsort.html**
5. **http://www.cs.rochester.edu/~gildea/csc282/slides/C07-quicksort.pdf**
6. **http://www.sorting-algorithms.com/quick-sort**
7. **http://www.cse.ust.hk/~dekai/271/notes/L01a/quickSort.pdf**
8. **http://en.wikipedia.org/wiki/Merge\_sort**
9. **http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/mergeSort.htm**
10. **http://www.sorting-algorithms.com/merge-sort**
11. **http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Merge\_sort.html**

**Pre Lab/ Prior Concepts:**

Data structures, various sorting techniques

**Historical Profile:**

**Quicksort and merge sort are** divide**-**and-conquer sorting algorithm in which division is dynamically carried out. They are one the most efficient sorting algorithms.

**New Concepts to be learned:**

Number of comparisons, Application of algorithmic design strategy to any problem, Classical problem solving vs Divide-and-Conquer problem solving.

**Algorithm** **Recursive Quick Sort:**

**void** quicksort( Integer A[ ], Integer left, Integer right)

**//**sorts A[left.. right] by using partition() to partition A[left.. right], and then //calling itself // twice to sort the two subarrays.

{ **IF** ( left < right ) then

{ q = partition( A, left, right);

quicksort( A, left, q–1);

quicksort( A, q+1, right);

}

}

**Integer *partition(integer A*T[], Integer *left*, Integer *right*)**

*//This function*rarranges *A*[*left***..***right*] and finds and returns an integer *q*, such that *A*[*left*], ..., //*A*[*q*–1] **<**∼*pivot*, *A*[*q*] = *pivot*, *A*[*q*+1], ..., *A*[*right*] > *pivot*, where *pivot* is the first element of //a[left…right], before partitioning**.**

{

pivot = A[left]; lo = left+1; hi = right;

**WHILE** ( lo ≤ hi)

{ **WHILE** (A[hi] > pivot) hi = hi – 1;

**WHILE** ( lo ≤ hi and A[lo] <∼pivot) lo = lo + 1;

**IF** ( lo ≤ hi) then swap( A[lo], A[hi]);

}

swap(pivot, A[hi]);

**RETURN** hi;

}

**The space complexity of Quick Sort: O(1)**

**Derivation of best case and worst-case time complexity (Quick Sort)**

**Best case: pivot element is middle element or near to middle element**

**= O(nlogn)**

**Worst case: pivot element is either greatest element or smallest element**

**= O(n2)**

**Algorithm Merge Sort**

MERGE-SORT (*A*, *p*, *r*)

// To sort the entire sequence A[1 .. n], make the initial call  to the procedure MERGE-SORT (*A*, //1, *n*). Array *A* and indices *p*, *q*, *r* such that *p* ≤ *q* ≤ r and sub array *A*[*p* .. *q*] is sorted and sub array //*A*[*q* + 1 .. *r*] is sorted. By restrictions on *p*, *q*, *r*, neither sub array is empty.

**//OUTPUT**: The two sub arrays are merged into a single sorted sub array in *A*[*p* .. *r*].

**IF** *p* < *r*                                                    // Check for base case  
         **THEN** *q* = FLOOR [(*p* + *r*)/2]                 // Divide step  
                 **MERGE** (A, *p*, *q*)                          // Conquer step.  
                 MERGE (A, *q* + 1, *r*)                     // Conquer step.  
                 MERGE (A, *p*, *q*, *r*)                       // Conquer step.

MERGE (*A*, *p*, *q*, *r*)

{

*n*1 ← *q* − *p* + 1  
      *n*2 ← *r* − *q*  
      Create arrays L[1 . . *n*1 + 1] and R[1 . . *n*2 + 1]  
      **FOR** *i* ← 1 **TO** *n*1  
            **DO** L[*i*] ← A[*p* + *i* − 1]  
      **FOR** *j* ← 1 **TO** *n*2  
            **DO** R[*j*] ← A[*q* + *j* ]  
      L[*n*1 + 1] ← ∞  
      R[*n*2 + 1] ← ∞  
    *i* ← 1  
    *j* ← 1  
    **FOR** *k* ← *p* **TO** *r*  
         **DO IF** L[*i* ] ≤ R[ *j*]  
                **THEN** A[*k*] ← L[*i*]  
                        *i* ← *i* + 1  
                **ELSE** A[k] ← R[j]  
                        *j* ← *j* + 1

}

**The space complexity of Merge sort: O(n)**

**Derivation of best case and worst-case time complexity (Merge Sort)**

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

**a = b = 2**

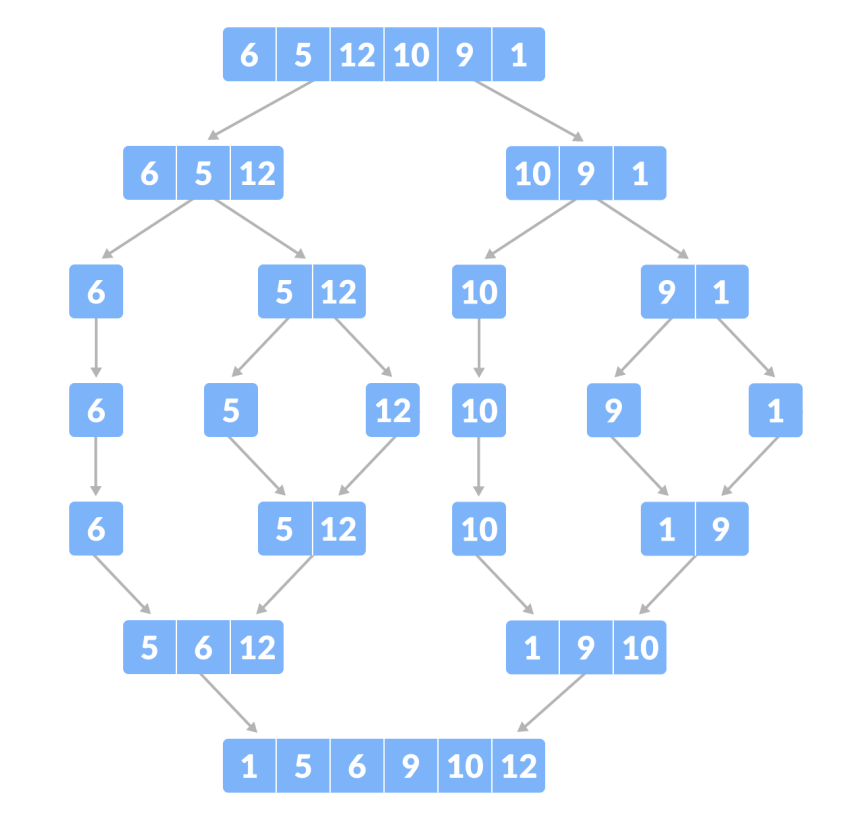
**logb a = 1 , c=1**

**Hence, θ( nclog n) = θ(n log n)**

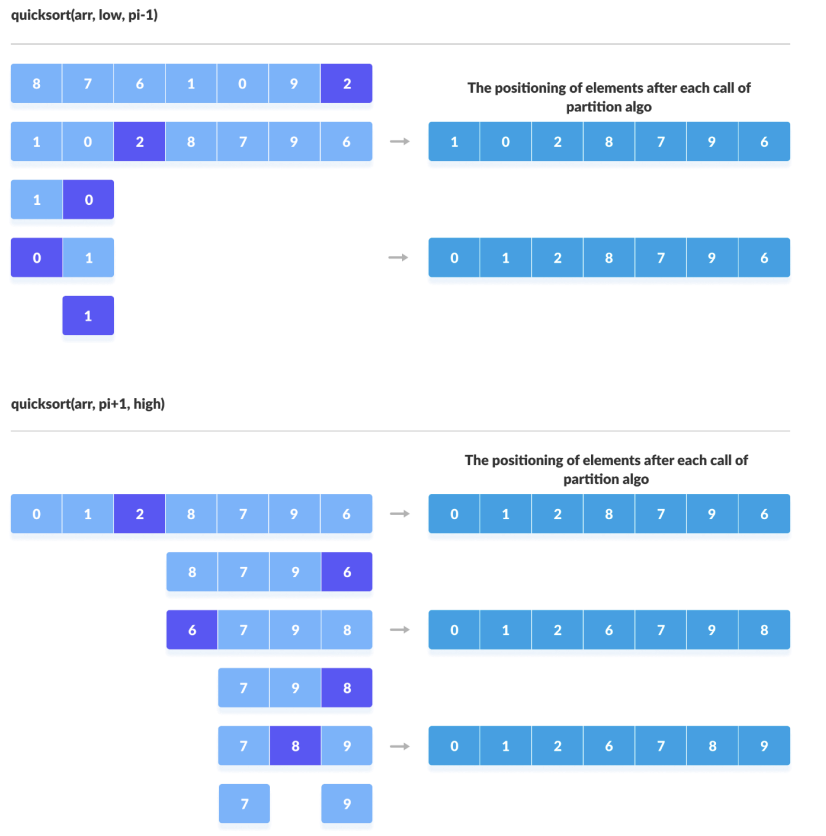
**Best Case = Worst Case = θ(n log n)**

**Example for quicksort/Merge tree for merge sort:**

**Merge Sort**

****

**Quick Sort:**

****

**Code:**

*#include* <bits/stdc++.h>

using namespace std;

void merge(int \*arr, int l, int mid, int r)

{

int arr1[r + 1];

int i = l, j = mid + 1, k = l;

*while* (i <= mid && j <= r)

{

*if* (arr[i] < arr[j])

{

arr[j] = arr[i];

i++;

}

*else*

{

arr1[k] = arr[j];

j++;

}

k++;

}

*if* (i > mid)

*while* (j <= r)

{

arr1[k] = arr[j];

k++;

j++;

}

*else* *if* (j > r)

*while* (i <= mid)

{

arr1[k] = arr[i];

k++;

i++;

}

*for* (k = l; k <= r; k++)

arr[k] = arr1[k];

}

void sort(int \*arr, int l, int r)

{

*if* (l < r)

{

int mid = (l + r) / 2;

sort(arr, l, mid);

sort(arr, mid + 1, r);

merge(arr, l, mid, r);

}

}

int main()

{

int n;

cin >> n;

int arr[n];

*for* (int i = 0; i < n; i++)

cin >> arr[i];

sort(arr, 0, n - 1);

*for* (int i = 0; i < n; i++)

cout << arr[i] << " ";

cout << endl;

*return* 0;

}

**CONCLUSION:**

Understood and implemented Merge sort and quick sort their time and space complexities

**Batch: A3 Roll No.: 16010121045**

**Experiment No: 4**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title:** Implementation ofSingle source shortest path by Greedy strategy |

**Objective:** To learn the Greedy strategy of solving the problems for different types of problems

**CO to be achieved:**

|  |  |
| --- | --- |
| CO 2 | Describe various algorithm design strategies to solve different problems and analyse Complexity. |

**Books/ Journals/ Websites referred:**

1. **1. Ellis horowitz, Sarataj Sahni, S.Rajsekaran,” Fundamentals of computer algorithm”, University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein,” Introduction to algortihtms”,2nd Edition ,MIT press/McGraw Hill,2001**
3. **https://www.mpi-inf.mpg.de/~mehlhorn/ftp/ShortestPathSeparator.pdf**
4. **en.wikipedia.org/wiki/Shortest\_path\_problem**
5. **www.cs.princeton.edu/~rs/AlgsDS07/15ShortestPaths.pdf**

**Pre Lab/ Prior Concepts:**

Data structures, Concepts of algorithm analysis

**Historical Profile:**

Sometimes the problems have more than one solution. With the size of the problem, every time it’s not feasible to solve all the alternative solutions and choose a better one. The greedy algorithms aim at choosing a greedy strategy as solutioning method and proves how the greedy solution is better one.

Though greedy algorithms do not guarantee optimal solution, they generally give a better and feasible solution.

The path finding algorithms work on graphs as input and represent various problems in the real world.

**New Concepts to be learned:** Application of algorithmic design strategy to any problem, Greedy method of problem solving Vs other methods of problem solving, optimality of the solution

**Topic: GREEDY METHOD**

**Theory:** The greedy method suggests that one can devise an algorithm that work in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution. Otherwise, it is added. The selection procedure itself is based on some optimization measures may be plausible for a given problem. Most of these, however, will result in algorithms that generate suboptimal solutions. This version of the greedy technique is called the **subset paradigm**.

**Control Abstraction**:

SolType Greedy (Type s [], int n)

// a[1:n] contains the n inputs.

{SolType solution = EMPTY;

// Initialize the solution.

For (int i=1; I<=n; i++) {

Type x = Select (a);

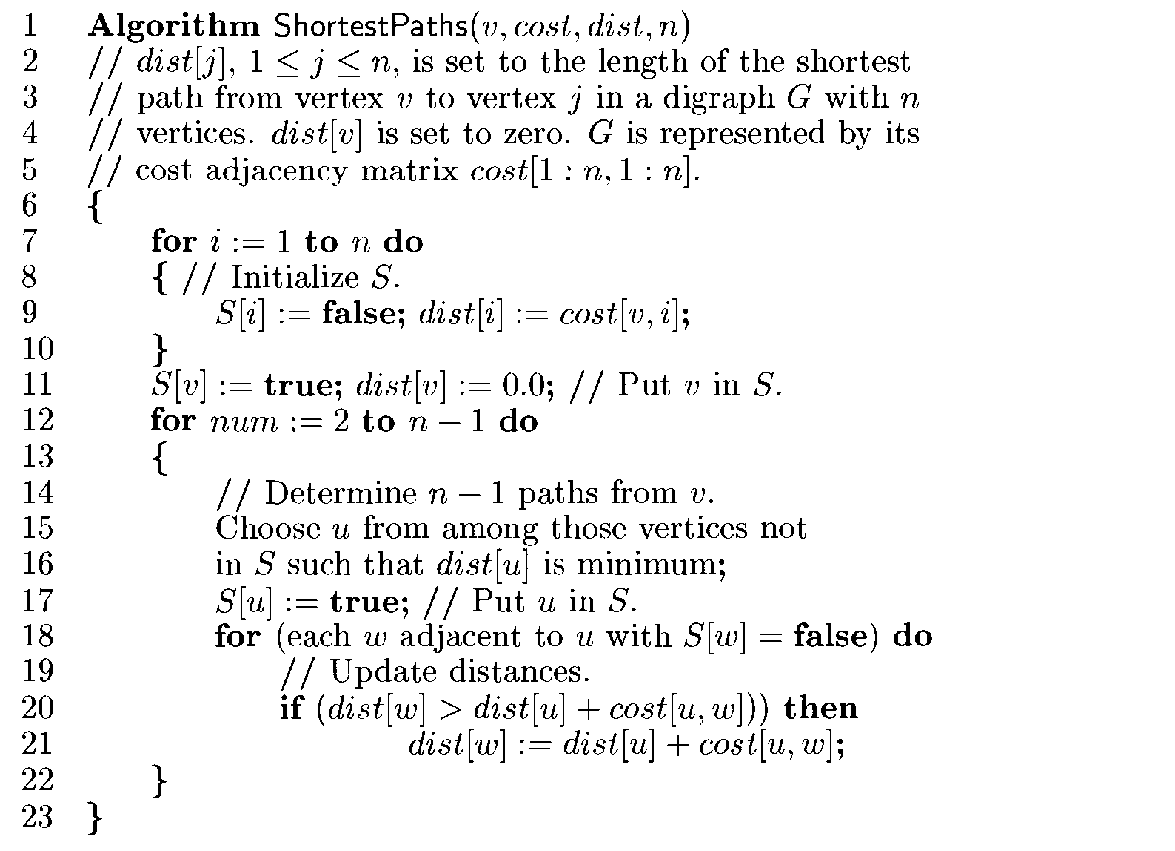
If Feasible (solution, x)

Solution = Union (solution, x) ;

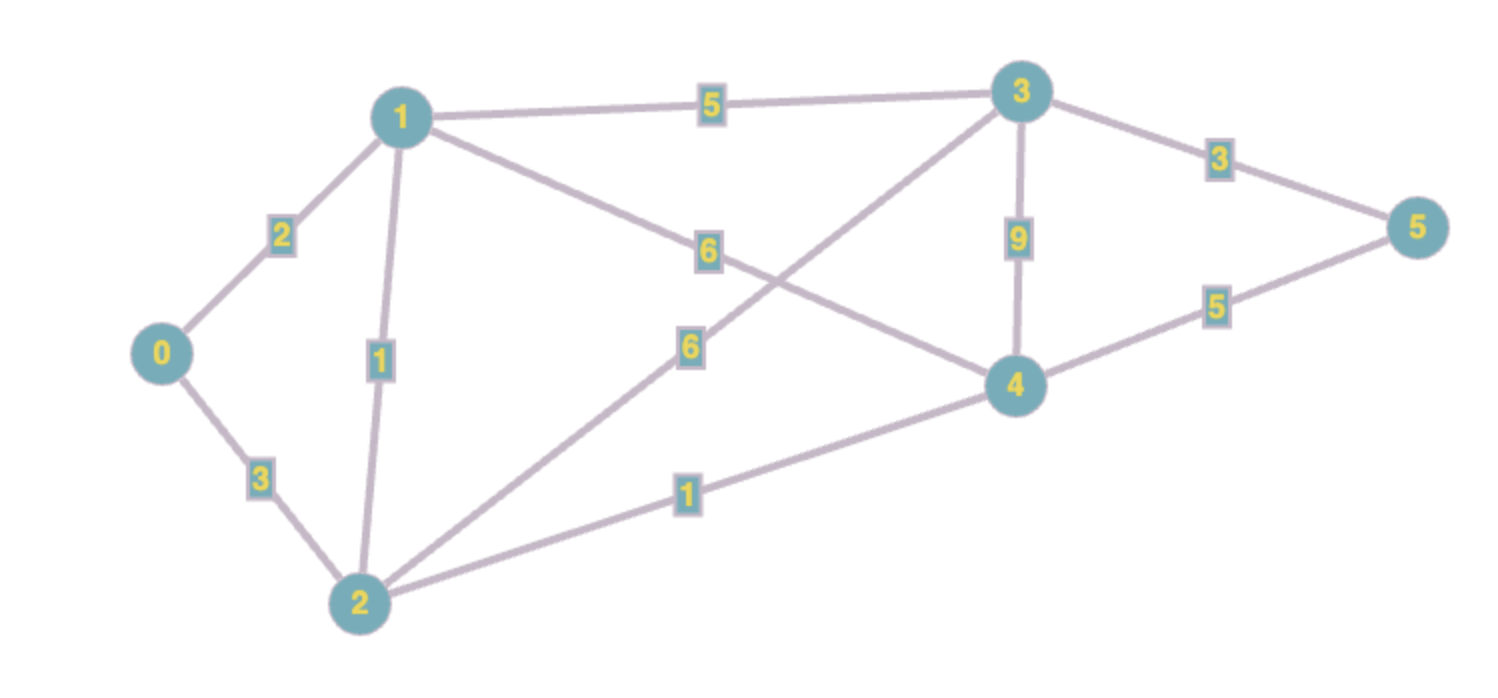
}

return solution;

}

**Algorithm**: 

**Example Graph:**



**Solution:**

|  |  |
| --- | --- |
| Vertex | Shortest Distance |
| 0 | 0 |
| 1 | 2 |
| 2 | 3 |
| 3 | 7 |
| 4 | 4 |
| 5 | 9 |

**Code:**

*#include* <bits/stdc++.h>

using namespace std;

*#define* INF 0x3f3f3f3f

typedef pair<int, int> p;

void addEdge(vector<p> adj[], int v, int u, int wt)

{

adj[v].push\_back(make\_pair(u, wt));

adj[u].push\_back(make\_pair(v, wt));

}

void dijkstras(vector<p> adj[], int src, int n)

{

priority\_queue<p, vector<p>, greater<p>> pq;

vector<int> dist(n, INF);

vector<int> parent(n, -1); *// Keep track of parent nodes*

pq.push(make\_pair(0, src));

dist[src] = 0;

*while* (!pq.empty())

{

int u = pq.top().second;

pq.pop();

*for* (auto x : adj[u])

{

int v = x.first;

int weight = x.second;

*if* (dist[v] > dist[u] + weight)

{

dist[v] = dist[u] + weight;

pq.push(make\_pair(dist[v], v));

parent[v] = u; *// Update parent of v*

}

}

}

cout << "Vertex Distance from Source\n";

*for* (int i = 0; i < n; i++)

cout << i << "\t\t" << dist[i] << "\n";

*// Print the shortest path from source to each vertex*

cout << "\nShortest Paths:\n";

*for* (int i = 0; i < n; i++)

{

cout << "Path to vertex " << i << ": ";

*if* (dist[i] == INF)

{

cout << "No path";

}

*else*

{

vector<int> path;

int curr = i;

*while* (curr != src)

{

path.push\_back(curr);

curr = parent[curr];

}

path.push\_back(src);

reverse(path.begin(), path.end());

*for* (int j = 0; j < path.size(); j++)

{

cout << path[j];

*if* (j != path.size() - 1)

cout << " -> ";

}

}

cout << "\n";

}

}

int main()

{

int vertices, edges;

cout << "Enter Vertices: ";

cin >> vertices;

cout << "Enter Edges: ";

cin >> edges;

vector<p> adj[vertices];

*for* (int i = 0; i < edges; i++)

{

int u, v, wt;

cin >> u >> v >> wt;

addEdge(adj, u, v, wt);

}

dijkstras(adj, 0, vertices);

*return* 0;

}

**Output:**

**Text

Description automatically generated**

**Time Complexity for single source shortest path**

Min-priority queue : **O( V + E log V )**

**Conclusion:**

Successfully completed the given experiment on Dijkstra’s algorithm to findout the shortest path in a weighted graph.

**Batch: A3 Roll No.: 16010121045**

**Experiment No. 5**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title:** Implementation of Knapsack Problem using Greedy strategy |

**Objective:** To learn the Greedy strategy of solving the problems for different types of problems

**CO to be achieved:**

|  |  |
| --- | --- |
| CO 2 | Describe various algorithm design strategies to solve different problems and analyse Complexity. |

**Books/ Journals/ Websites referred:**

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran,” Fundamentals of computer algorithm”, University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein,” Introduction to algortihtms”,2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://lcm.csa.iisc.ernet.in/dsa/node184.htm**
4. **http://students.ceid.upatras.gr/~papagel/project/kruskal.htm**
5. [**http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/kruskalAlgor.html**](http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/kruskalAlgor.html)
6. **http://lcm.csa.iisc.ernet.in/dsa/node183.html**
7. **http://students.ceid.upatras.gr/~papagel/project/prim.htm**
8. **http://www.cse.ust.hk/~dekai/271/notes/L07/L07.pdf**

**Pre Lab/ Prior Concepts:**

Data structures, Concepts of algorithm analysis

**Historical Profile:**

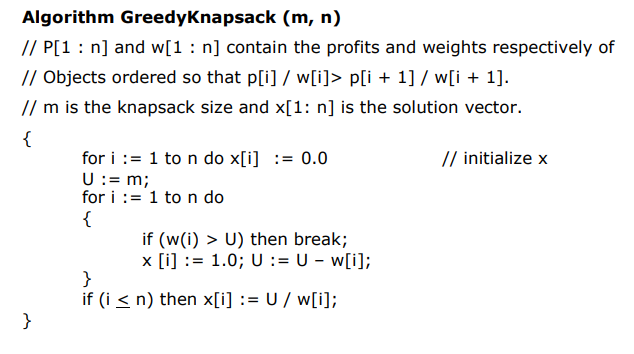
The knapsack problem represents constraint satisfaction optimization problems’ family. Based on nature of constraints, the knapsack problem can be solved with various problem saolving strategies. Typically, these problems represent resource optimization solution.

Given a set of n inputs. · Find a subset, called feasible solution, of the n inputs subject to some constraints, and satisfying a given objective function. · If the objective function is maximized or minimized, the feasible solution is optimal. · It is a locally optimal method.

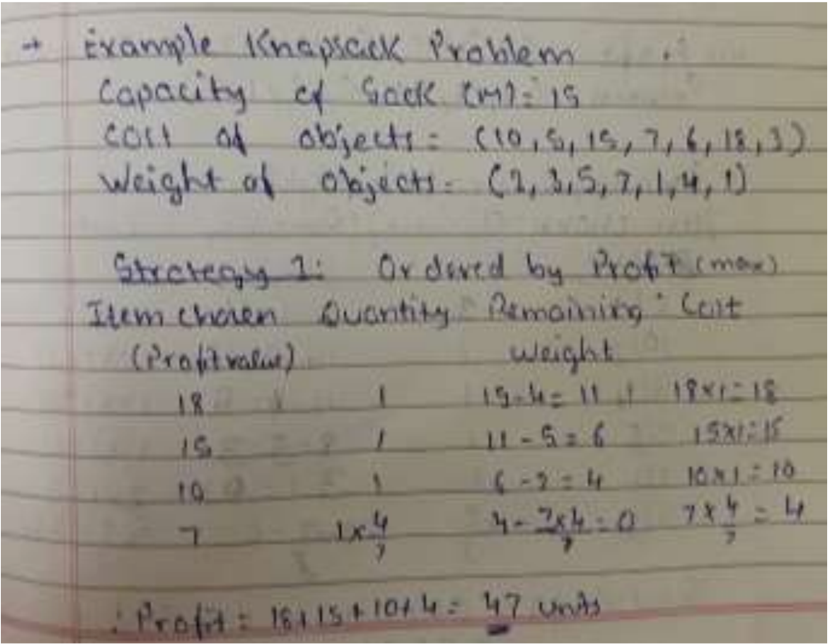
**New Concepts to be learned:**

Application of algorithmic design strategy to any problem, Greedy method of problem solving Vs other methods of problem solving, optimality of the solution, knapsack problem and their applications

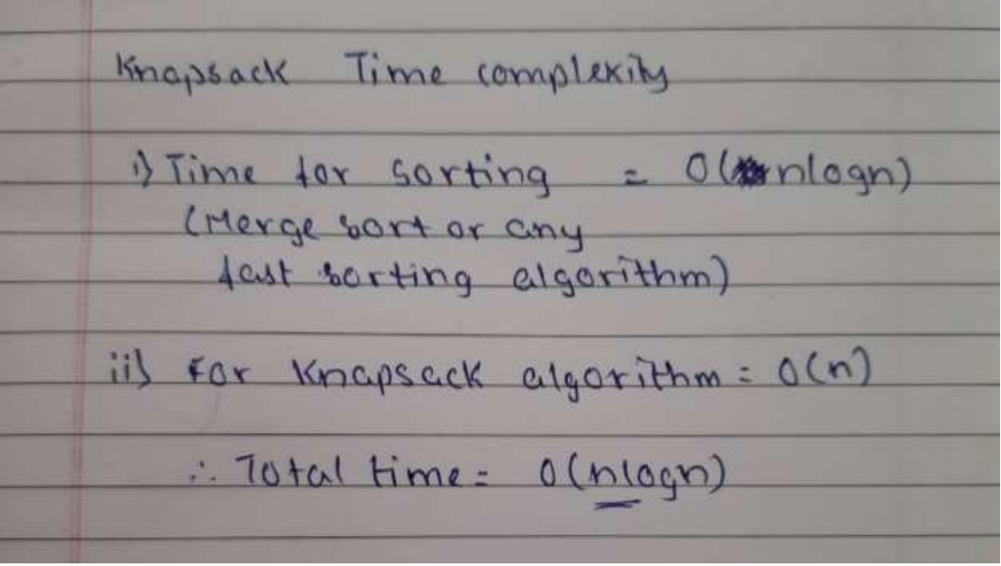
**Knapsack Problem Algorithm**

****

**Example: Knapsack Problem**



**Analysis of Knapsack Problem algorithm:**

****

*#include* <bits/stdc++.h>

using namespace std;

struct item {

int profit, weight, X;

item(int profit, int weight): profit(profit), weight(weight), X(0) {} };

bool compare(struct item a, struct item b)

{

double pwRatio1 = (double)a.profit / a.weight;

double pwRatio2 = (double)b.profit / b.weight;

*return* pwRatio1 > pwRatio2;

}

double knapsack(struct item items[], int capacity, int size)

{

sort(items, items + size, compare);

int currentWeight = 0;

double maxProfit = 0.0;

*for* (int i = 0; i < size; i++)

{

*if* (currentWeight + items[i].weight <= capacity)

{

currentWeight += items[i].weight;

maxProfit += items[i].profit;

items[i].X = 1;

}

*else*

{

int currentCapacity = capacity - currentWeight;

items[i].X = (double) currentCapacity / items[i].weight;

maxProfit += items[i].profit \* ((double) currentCapacity / items[i].weight); *break*;

}

}

*return* maxProfit;

}

int main() {

int capacity = 50;

item items[] = {{ 110, 40 },

{ 160, 30 },

{ 200, 50 }};

int size = sizeof(items) / sizeof(items[0]);

cout << "Maximum Profit: " << knapsack(items, capacity, size) << endl;

cout << "Fraction Collected: ";

*for* (auto &item: items)

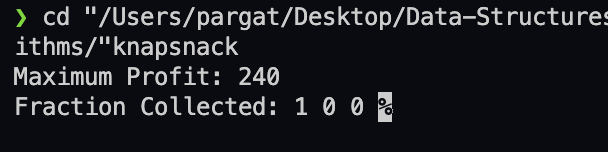
{

cout << item.X << " ";

}

*return* 0;

}

****

**Conclusion:**

Learnt the greedy approach for solving the knapsack problem, implemented all the strategies for solving the problem and analysed the time complexity of the algorithm.

**Batch: A3 Roll No.: 16010121045**

**Experiment No: 7**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title: Implementation of All Pair Shortest Path using Dynamic Programming** |

**Objective** To learn the All-Pair Shortest Path using Floyd-Warshall’salgorithm

**CO to be achieved:**

|  |  |
| --- | --- |
| CO 2 | Describe various algorithm design strategies to solve different problems and analyse Complexity. |

**Books/ Journals/ Websites referred:**

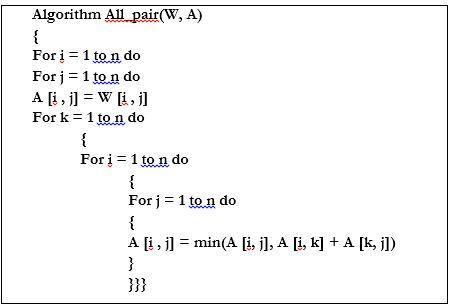
1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran,” Fundamentals of computer algorithm”, University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein,” Introduction to algortihtms”,2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://users.cecs.anu.edu.au/~Alistair.Rendell/Teaching/apac\_comp3600/module4/all\_pairs\_shortest\_paths.xhtml**
4. **https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/**
5. **http://www.cs.bilkent.edu.tr/~atat/502/AllPairsSP.ppt**

**Theory:**

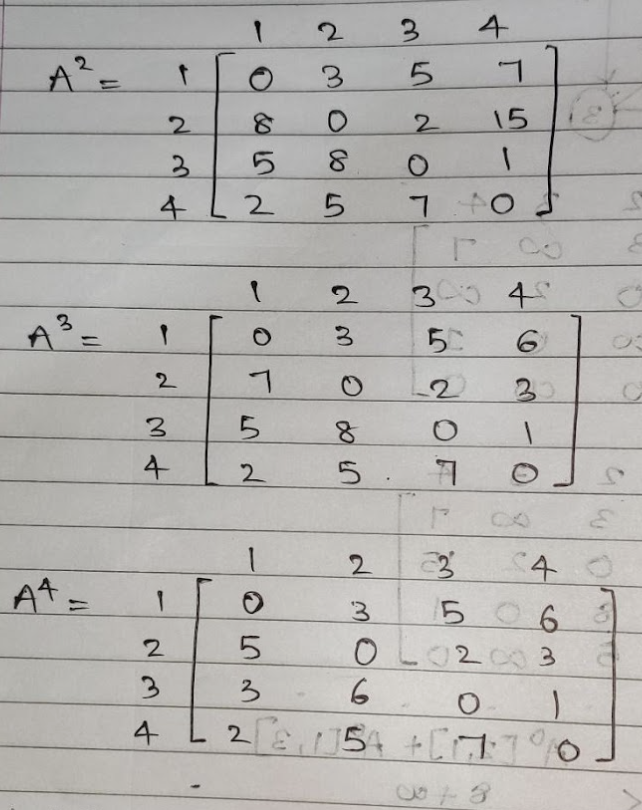
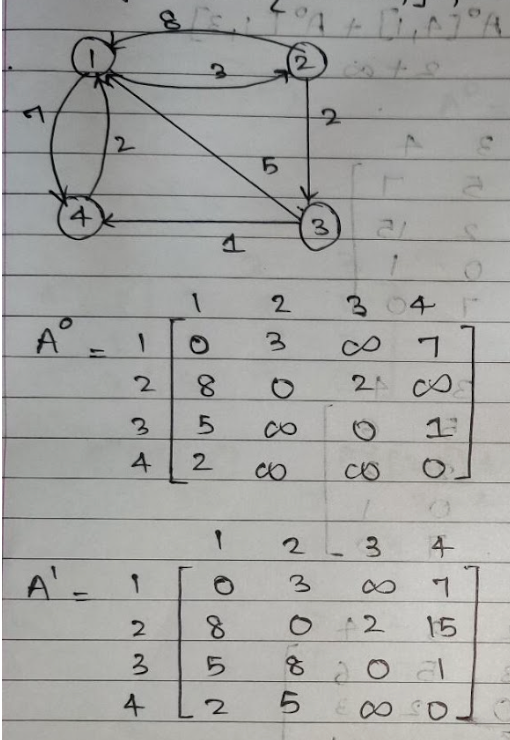
It aims to figure out the shortest path from each vertex v to every other u.

1. In all pair shortest path, when a weighted graph is represented by its weight matrix W then objective is to find the distance between every pair of nodes.
2. Apply dynamic programming to solve the all pairs shortest path.
3. In all pair shortest path algorithm, we first decomposed the given problem into sub problems.
4. In this principle of optimally is used for solving the problem.
5. It means any sub path of shortest path is a shortest path between the end nodes.

**Algorithm:**



**Example:**

****

**Code:**

*#include* <iostream>

*#include* <climits>

using namespace std;

*#define* V 4

void allshortestpath(int graph[][V])

{

int dist[V][V];

*for* (int i = 0; i < V; i++)

{

*for* (int j = 0; j < V; j++)

{

dist[i][j] = graph[i][j];

}

}

*for* (int k = 0; k < V; k++)

{

*for* (int i = 0; i < V; i++)

{

*for* (int j = 0; j < V; j++)

{

*if* (dist[i][k] != INT\_MAX && dist[k][j] != INT\_MAX && dist[i][k] + dist[k][j] < dist[i][j])

{

dist[i][j] = dist[i][k] + dist[k][j];

}

*if* (dist[i][j] == INT\_MAX)

cout << "INF\t";

*else*

cout << dist[i][j] << "\t";

}

cout << endl;

}

cout<<endl;

}

}

int main()

{

int graph[V][V] = {{0, 15, INT\_MAX, 10},

{INT\_MAX, 0, 4, INT\_MAX},

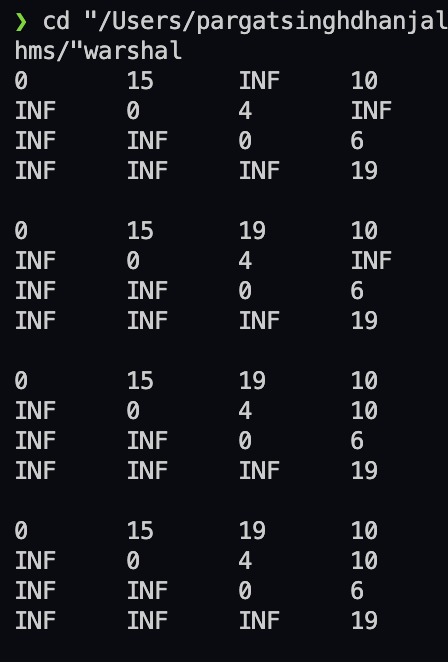
{INT\_MAX, INT\_MAX, 0, 6},

{INT\_MAX, INT\_MAX, INT\_MAX, 19}};

allshortestpath(graph);

*return* 0;

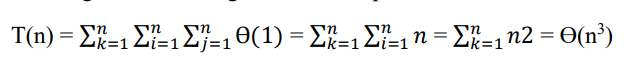
}



**Analysis of algorithm:**

It uses three nested loops. Innermost loop has only one statement. The complexity of that statement is Ɵ(1).

Running time of the algorithm is computed as

****

Thus, floyd's algorithm runs in cubic time.

**CONCLUSION:**

In this experiment, we have learnt Implementation of all Pair Shortest Path using Floyd-Warshall algorithm. We have understood the dynamic programming approach to solve all pairs shortest path problems.

**Batch: A3 Roll No.: 16010121045**

**Experiment No : 8**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title: Implementation Matrix Chain Multiplication of Dynamic Programming** |

**Objective:** To learn Matrix chain multiplication using Dynamic Programming Approach

**CO to be achieved:**

|  |  |
| --- | --- |
| CO 2 | Describe various algorithm design strategies to solve different problems and analyse Complexity. |

**Books/ Journals/ Websites referred:**

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran,” Fundamentals of computer algorithm”, University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein,” Introduction to algortihtms”,2nd Edition ,MIT press/McGraw Hill,2001**
3. [**http://www.lsi.upc.edu/~mjserna/docencia/algofib/P07/dynprog.pdf**](http://www.lsi.upc.edu/~mjserna/docencia/algofib/P07/dynprog.pdf)
4. [**http://www.geeksforgeeks.org/travelling-salesman-problem-set-1/**](http://www.geeksforgeeks.org/travelling-salesman-problem-set-1/)
5. [**http://www.mafy.lut.fi/study/DiscreteOpt/tspdp.pdf**](http://www.mafy.lut.fi/study/DiscreteOpt/tspdp.pdf)
6. [**https://class.coursera.org/algo2-2012-001/lecture/181**](https://class.coursera.org/algo2-2012-001/lecture/181)
7. [**http://www.quora.com/Algorithms/How-do-I-solve-the-travelling-salesman-problem-using-Dynamic-programming**](http://www.quora.com/Algorithms/How-do-I-solve-the-travelling-salesman-problem-using-Dynamic-programming)
8. [**www.cse.hcmut.edu.vn/~dtanh/download/Appendix\_B\_2.ppt**](http://www.cse.hcmut.edu.vn/~dtanh/download/Appendix_B_2.ppt)
9. **www.ms.unimelb.edu.au/~s620261/powerpoint/chapter9\_4.ppt‎**

**Pre Lab/ Prior Concepts:**

Data structures, Concepts of algorithm analysis

**Historical Profile:**

Dynamic Programming (DP) is used heavily in optimization problems (finding the maximum and the minimum of something). Applications range from financial models and operation research to biology and basic algorithm research. So the good news is that understanding DP is profitable. However, the bad news is that DP is not an algorithm or a data structure that you can memorize. It is a powerful algorithmic design technique.

**New Concepts to be learned:**

Application of algorithmic design strategy to any problem, dynamic Programming method of problem solving Vs other methods of problem solving, optimality of the solution, Optimal Binary Search Tree Problems and their applications

**Theory:**

**Problem definition:**

Given a sequence of N matrices, the matrix chain multiplication problem is to find the most efficient way to [multiply these matrices](https://en.wikipedia.org/wiki/Matrix_multiplication) by minimizing the number of computations involved during multiplications.

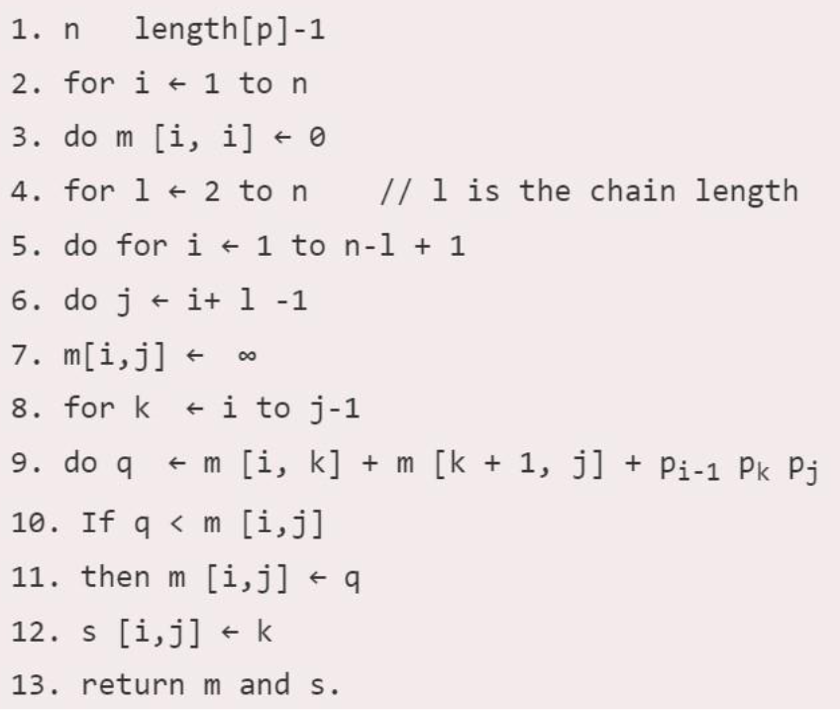
**Optimal Substructure:** parameterization/ select the subgroup of matrices that will result in least number of computations.

For multiplication of matrix series Ai to Aj, choose Ak such that multiplication of matrices through Ai..k and Ak+1…j will incur least number of computations for any k such that i<=k<j.

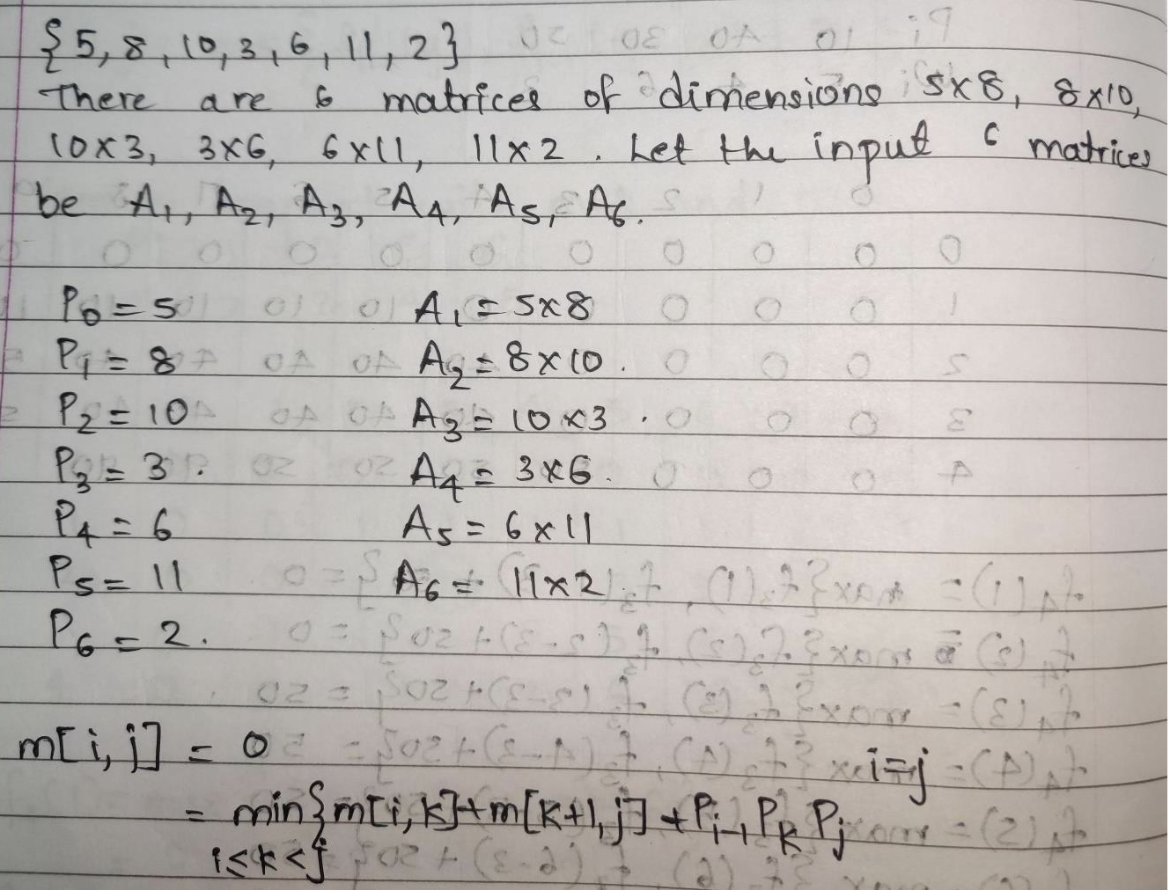
**Recursive Formula:**



**Algorithm:**

****

**Example:**



**Solution for the example:**

*#include* <bits/stdc++.h>

using namespace std;

void printParenthesis(int i, int j, int n, int \*bracket,

char &name)

{

*if* (i == j)

{

cout << name++;

*return*;

}

cout << "(";

printParenthesis(i, \*((bracket + i \* n) + j), n,

bracket, name);

printParenthesis(\*((bracket + i \* n) + j) + 1, j, n,

bracket, name);

cout << ")";

}

void matrixChainOrder(int p[], int n)

{

int m[n][n];

int bracket[n][n];

*for* (int i = 1; i < n; i++)

m[i][i] = 0;

*for* (int L = 2; L < n; L++)

{

*for* (int i = 1; i < n - L + 1; i++)

{

int j = i + L - 1;

m[i][j] = INT\_MAX;

*for* (int k = i; k <= j - 1; k++)

{

int q = m[i][k] + m[k + 1][j] + p[i - 1] \* p[k] \* p[j];

*if* (q < m[i][j])

{

m[i][j] = q;

bracket[i][j] = k;

}

}

}

}

char name = 'A';

cout << "Optimal Parenthesization is : ";

printParenthesis(1, n - 1, n, (int \*)bracket, name);

cout << "\nOptimal Cost is : " << m[1][n - 1] << endl;

}

int main()

{

int arr[] = {40, 20, 30, 10, 30};

int n = sizeof(arr) / sizeof(arr[0]);

matrixChainOrder(arr, n);

*return* 0;

}

Ans:

Text

Description automatically generated

**Analysis of algorithm:**

Time Complexity is: O (n3)

There are three nested loops. Each loop executes a maximum n times.

Space Complexity is O(n\*n) where n is the number present in the chain of the matrices. We create a DP matrix that stores the results after each operation.

**CONCLUSION:**

In this experiment, we have learnt Implementation of Matrix Chain Multiplication by dynamic programming approach.

**Topic: Backtracking**

**Theory:** In many applications of the backtrack method, the desired solution is expressible as an n-tuple *(x1,...,Xn),* where the x*i* are chosen from some finite set Si. Often the problem to be solved calls for finding one vector that maximizes (or minimizes or satisfies) a *criterion function P(x1,…..* . , *xn). Sometime*s it seeks all vectors that satisfy *P.* For example, sorting the array of integers in. *a[1* : n] is a problem whose solution is expressible by an *n- tuple, w*here x*i* is the index in *a* of the ith smallest element. The criterion function P is the inequality *a[xi]* ≤ *a[xi+1]* for 1 ≤ i < *n.* The set *Si* is finite and includes the integers 1 through *n.* Though sorting is not usually one of the problems solved by backtracking, it is one example of a familiar problem whose solution can be formulated as an n-tuple.

**Control abstraction**:

void Backtrack( int k )

// This is a schema that describes the backtracking process //using recursion. On entering, the first k-1 values x[1], x[2], //…., x[k-1] of the solution vector x[1:n] have been //assigned. x[] and n are global.

{

for (each x[k] such that x[k] Є T(x[1], …, x[k-1])

{

if (Bk (x[1], x[2], …, x[k]))

{

if (x[1], x[2], …, x[k] is a path to an answer node)

output x[1:k];

if (k < n) Backtrack(k+1);

}

}

}

**Batch: A3 Roll No.: 16010121045**

**Experiment No: 8**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title: Implementation of Graph Colouring Backtracking Algorithm** |

**Objective:** To learn the Backtracking strategy of problem solving for Graph Colouring problem

**CO to be achieved:**

|  |  |
| --- | --- |
| CO 2 | Analyze and solve problems for divide and conquer strategy, greedy method, dynamic programming approach and backtracking and branch & bound policies. |

**Books/ Journals/ Websites referred:**

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran,” Fundamentals of computer algorithm”, University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein,” Introduction to algortihtms”,2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://www.math.utah.edu/~alfeld/queens/queens.html**
4. [**http://www-isl.ece.arizona.edu/ece175/assignments275/assignment4a/Solving%208%20queen%20problem.pdf**](http://www-isl.ece.arizona.edu/ece175/assignments275/assignment4a/Solving%208%20queen%20problem.pdf)
5. [**http://www.slideshare.net/Tech\_MX/8-queens-problem-using-back-tracking**](http://www.slideshare.net/Tech_MX/8-queens-problem-using-back-tracking)
6. [**http://www.mathcs.emory.edu/~cheung/Courses/170.2010/Syllabus/Backtracking/8queens.html**](http://www.mathcs.emory.edu/~cheung/Courses/170.2010/Syllabus/Backtracking/8queens.html)
7. [**http://www.geeksforgeeks.org/backtracking-set-3-n-queen-problem/**](http://www.geeksforgeeks.org/backtracking-set-3-n-queen-problem/)
8. **http://www.hbmeyer.de/backtrack/achtdamen/eight.htm**

**Pre Lab/ Prior Concepts:**

Data structures, Concepts of algorithm analysis

**Historical Profile:**

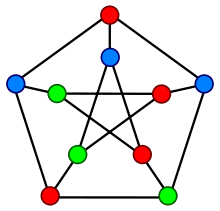
Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices.

***nput:***

1) A 2D array graph [V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph.

***Output:***

An array color [V] that should have numbers from 1 to m. color[i] should represent the color assigned to the ith vertex. The code should also return false if the graph cannot be colored with m colors.

Following is an example graph can be colored with 3 colors.  


**New Concepts to be learned:**

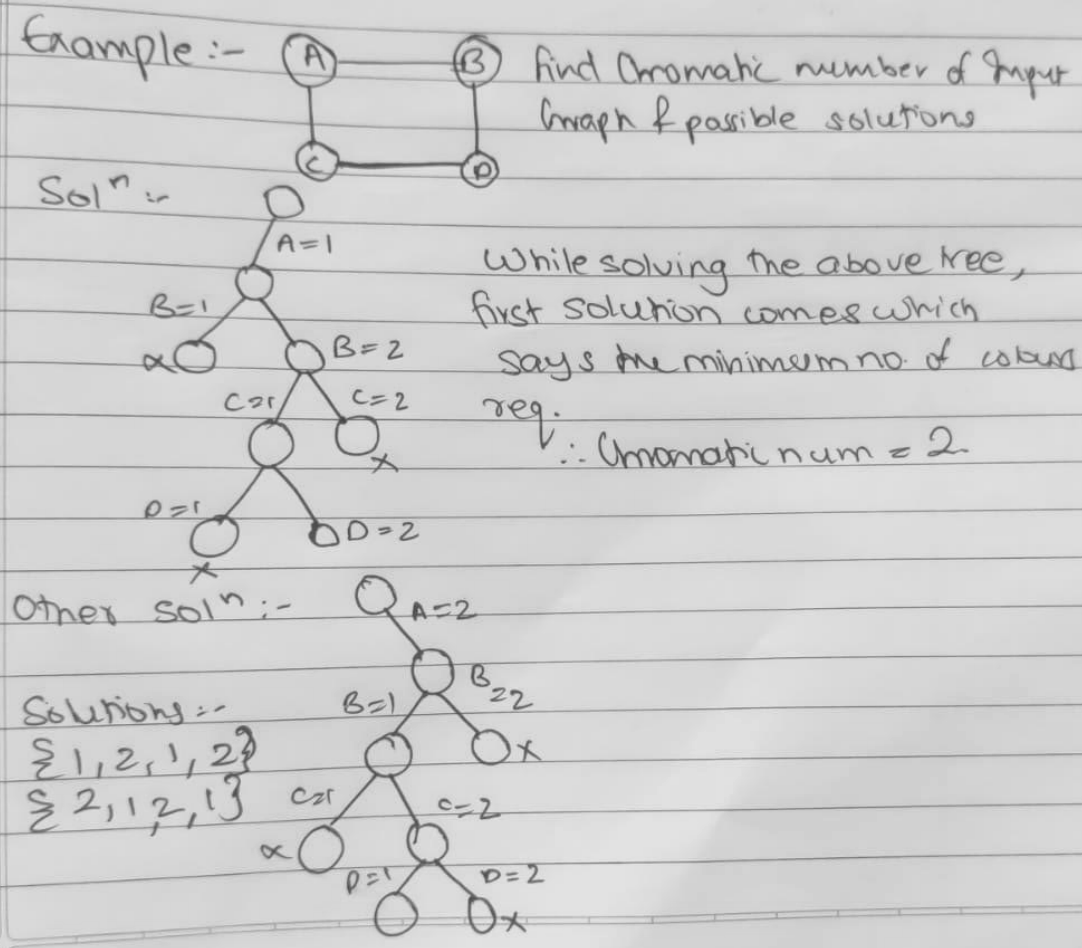
Application of algorithmic design strategy to any problem, Backtracking method of problem solving Vs other methods of problem solving problem graph colouring and its applications.

**Algorithm Graph colouring Problem:-**

**Text, letter

Description automatically generated**

**Example Graph Colouring Problem:**

****

**Code:**

*#include* <iostream>

using namespace std;

const int MAXN = 100; *// maximum number of vertices in the graph*

int graph[MAXN][MAXN]; *// adjacency matrix of the graph*

int colors[MAXN]; *// colors of the vertices*

int n, m; *// number of vertices and edges in the graph*

int num\_colors; *// number of colors available*

bool isSafe(int v, int c)

{

*// Check if any adjacent vertex has the same color*

*for* (int i = 0; i < n; i++)

{

*if* (graph[v][i] && colors[i] == c)

{

*return* false;

}

}

*return* true;

}

bool graphColoring(int v)

{

*if* (v == n)

{

*// All vertices have been colored*

*return* true;

}

*for* (int c = 1; c <= num\_colors; c++)

{

*if* (isSafe(v, c))

{

*// Color the vertex with the current color*

colors[v] = c;

*// Recur for the next vertex*

*if* (graphColoring(v + 1))

{

*return* true;

}

*// Backtrack and remove the color from the current vertex*

colors[v] = 0;

}

}

*// Vertex cannot be colored with any available color*

*return* false;

}

void printColors()

{

*for* (int i = 0; i < n; i++)

{

cout << "Vertex " << i + 1 << " is colored with " << colors[i] << endl;

}

}

int main()

{

cout << "Enter the number of vertices and edges in the graph: ";

cin >> n >> m;

*// Initialize the adjacency matrix to 0*

*for* (int i = 0; i < n; i++)

{

*for* (int j = 0; j < n; j++)

{

graph[i][j] = 0;

}

}

*// Read the edges of the graph*

cout << "Enter the edges of the graph:" << endl;

*for* (int i = 0; i < m; i++)

{

int u, v;

cin >> u >> v;

graph[u - 1][v - 1] = graph[v - 1][u - 1] = 1;

}

cout << "Enter the number of colors available: ";

cin >> num\_colors;

*if* (graphColoring(0))

{

cout << "Solution exists" << endl;

printColors();

}

*else*

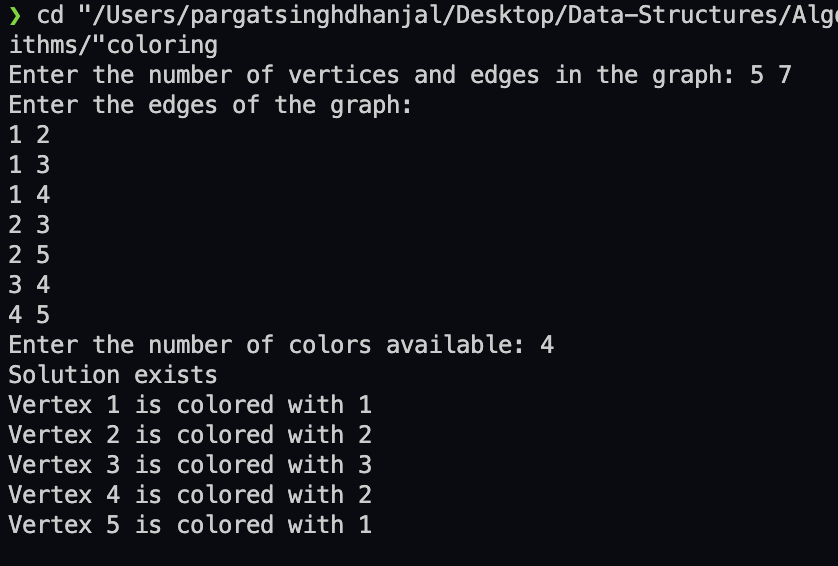
{

cout << "Solution does not exist" << endl;

}

*return* 0;

}

****

**Analysis of Backtracking solution for Graph Colouring Problem:**

In the backtracking approach of the graph colouring problem, we are generating total mV combinations of the colour. So, it also requires exponential time.

**Time Complexity**: In the backtracking approach of the graph colouring problem, the time complexity is O(mV). In the backtracking approach, the average time complexity is less than O(mV).

**Space Complexity**: In the backtracking approach to the graph colouring problem, we are not using any extra space but we are using the recursive stack for the recursive function call. So, the space complexity is **O(V)**.

**Conclusion:**

We learnt the Backtracking strategy of problem solving for Graph Colouring problem.

**Batch: A3 Roll No.: 16010121045**

**Experiment No. 8**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title: Implementation of N-Queen Problem using Backtracking Algorithm** |

**Objective:** To learn the Backtracking strategy of problem solving for 8-Queens problem

**CO to be achieved:**

|  |  |
| --- | --- |
| Sr. No | Objective |
| CO 1 | Compare and demonstrate the efficiency of algorithms using asymptotic complexity notations. |
| CO 2 | Analyze and solve problems for divide and conquer strategy, greedy method, dynamic programming approach and backtracking and branch & bound policies. |
| CO 3 | Analyze and solve problems for different string matching algorithms. |

**Books/ Journals/ Websites referred:**

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran,” Fundamentals of computer algorithm”, University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein,” Introduction to algortihtms”,2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://www.math.utah.edu/~alfeld/queens/queens.html**
4. [**http://www-isl.ece.arizona.edu/ece175/assignments275/assignment4a/Solving%208%20queen%20problem.pdf**](http://www-isl.ece.arizona.edu/ece175/assignments275/assignment4a/Solving%208%20queen%20problem.pdf)
5. [**http://www.slideshare.net/Tech\_MX/8-queens-problem-using-back-tracking**](http://www.slideshare.net/Tech_MX/8-queens-problem-using-back-tracking)
6. [**http://www.mathcs.emory.edu/~cheung/Courses/170.2010/Syllabus/Backtracking/8queens.html**](http://www.mathcs.emory.edu/~cheung/Courses/170.2010/Syllabus/Backtracking/8queens.html)
7. [**http://www.geeksforgeeks.org/backtracking-set-3-n-queen-problem/**](http://www.geeksforgeeks.org/backtracking-set-3-n-queen-problem/)
8. [**http://www.hbmeyer.de/backtrack/achtdamen/eight.htm**](http://www.hbmeyer.de/backtrack/achtdamen/eight.htm)

**Pre Lab/ Prior Concepts:**

Data structures, Concepts of algorithm analysis

**Historical Profile:**

The **N-Queens puzzle** is the problem of placing N queens on an N×N chessboard so that no two queens attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

**New Concepts to be learned:**

Application of algorithmic design strategy to any problem, Backtracking method of problem-solving Vs other methods of problem solving, 8- Queens problem and its applications.

**Algorithm N Queens Problem: -**

void NQueens(int k, int n)

// Using backtracking, this procedure prints all possible placements of n queens on an n X n chessboard so that they are nonattacking.

{ for (int i=1; i<=n; i++)

{

if (Place(k, i))

{

x[k] = i;

if (k==n)

for (int j=1;j<=n;j++) Print x[j] ;

else NQueens(k+1, n);

}

}

}

Boolean Place(int k, int i)

// Returns true if a queen can be placed in kth row and ith column. Otherwise it returns false.

// x[] is a global array whose first (k-1) values have been set. abs(r) returns absolute value of r.

{

for (int j=1; j < k; j++)

if ((x[j] == i) // Two in the same column

|| (abs(x[j]-i) == abs(j-k))) // or in the same diagonal

return(false);

return(true);

}

**Example 8-Queens Problem:**

The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other i.e. no two queens share the same row, column, or diagonal.

**Solution Using Backtracking Approach:**

The idea is to place queens one by one in different columns, starting from the leftmost column. 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.

**State Space tree for N-Queens (Solution):**

**A picture containing diagram

Description automatically generated**

**Implementation (Code):**

*#include* <bits/stdc++.h>

using namespace std;

*// `N × N` chessboard*

*#define* N 4

int ans = 0;

int isSafe(char mat[][N], int r, int c)

{

*for* (int i = 0; i < r; i++)

{

*if* (mat[i][c] == 'Q')

{

*return* 0;

}

}

*for* (int i = r, j = c; i >= 0 && j >= 0; i--, j--)

{

*if* (mat[i][j] == 'Q')

{

*return* 0;

}

}

*for* (int i = r, j = c; i >= 0 && j < N; i--, j++)

{

*if* (mat[i][j] == 'Q')

{

*return* 0;

}

}

*return* 1;

}

void printSolution(char mat[][N])

{

*for* (int i = 0; i < N; i++)

{

*for* (int j = 0; j < N; j++)

{

printf("%c ", mat[i][j]);

}

printf("\n");

}

printf("\n");

}

void nQueen(char mat[][N], int r)

{

*if* (r == N)

{

printSolution(mat);

ans++;

*return*;

}

*for* (int i = 0; i < N; i++)

{

*if* (isSafe(mat, r, i))

{

mat[r][i] = 'Q';

nQueen(mat, r + 1);

mat[r][i] = '-';

}

}

}

int main()

{

char mat[N][N];

memset(mat, '-', sizeof mat);

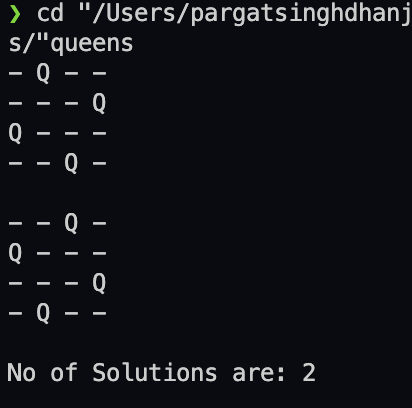
nQueen(mat, 0);

cout << "No of Solutions are: " << ans << endl;

*return* 0;

}

**OUTPUT:**



**Algorithm:**

0) Make a board, make a space to collect all solution states.

1) Start in the topmost row.

2) Make a recursive function which takes state of board and the current row number

as its parameter.

3) Fill a queen in a safe place and use this state of board to advance to next recursive

call, add 1 to the current row. Revert the state of board after making the call.

a) Safe function checks the current column, left top diagonal and right top diagonal.

b) If no queen is present then fill else return false and stop exploring that state

and track back to the next possible solution state

4) Keep calling the function till the current row is out of bound.

5) If current row reaches the number of rows in the board then the board is filled.

6) Store the state and return.

**Analysis of Backtracking solution:**

Time Complexity: O(N!)

Auxiliary Space: O(N2)

**CONCLUSION:**

**Successfully implemented the given problem using backtracking algorithm.**

**Batch: A3 Roll No.: 16010121045**

**Experiment No. 10**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title: Implementation of Longest Common Subsequence String Matching Algorithm** |

**Objective:** To compute longest common subsequence for the given two strings.

**CO to be achieved:**

|  |  |
| --- | --- |
| CO 2 | Analyze and solve problems for divide and conquer strategy, greedy method, dynamic programming approach and backtracking and branch & bound policies. |
| CO 3 | Analyze and solve problems for   different string matching algorithms. |

**Books/ Journals/ Websites referred:**

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran,” Fundamentals of computer algorithm”, University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein,” Introduction to algortihtms”,2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://www.math.utah.edu/~alfeld/queens/queens.**

**Pre Lab/ Prior Concepts:**

Data structures, Concepts of algorithm analysis

**Historical Profile:**

Given 2 sequences, *X* = *x*1 *, ..., xm*  and *Y* = *y*1 *, ... , yn* , find a subsequence common to both whose length is longest. A subsequence doesn’t have to be consecutive, but it has to be in order.

**New Concepts to be learned:**

String matching algorithm, Dynamic programming approach for LCS, Applications of LCS.

Recursive **Formulation:**

Define *c*[*i, j* ] = length of LCS of *Xi* and *Y j* .

Final answer will be computed with *c*[*m, n*].

c[i, j]= 0

if i=0 or j=0.

c[i, j]= c[i − 1, j − 1] + 1

if i,j>0 and xi=yj

c[i, j]= max(c[i − 1, j ], c[i, j − 1])

if i, j > 0 and xi <> yj

**Algorithm: Longest Common Subsequence**

**Compute length of optimal solution-**

**LCS-LENGTH** *( X , Y, m, n)*

**for** *i* ← 1 **to** *m*

**do** *c*[*i,* 0] ← 0

**for** *j* ← 0 **to** *n*

**do** *c*[0*, j* ] ← 0

**for** *i* ← 1 **to** *m*

**do for** *j* ← 1 **to** *n*

**do if** *xi* = *y j*

**then** *c*[*i, j* ] ← *c*[*i* − 1*, j* − 1] + 1

*b*[*i, j* ] ← “≈”

**else if** *c*[*i* − 1*, j* ] ≥ *c*[*i, j* − 1]

**then** *c*[*i, j* ] ← *c*[*i* − 1*, j* ]

*b*[*i, j* ] ← “↑”

**else** *c*[*i, j* ] ← *c*[*i, j* − 1]

*b*[*i, j* ] ← “←”

**return** *c* and *b*

**Print the solution-**

**PRINT-LCS*(b, X , i, j )***

**if** *i* = 0 or *j* = 0

**then return**

**if** *b*[*i, j* ] = “≈”

**then** PRINT-LCS*(b, X , i* − 1*, j* − 1*)*

print *xi*

**elseif** *b*[*i, j* ] = “↑”

**then** PRINT-LCS*(b, X , i* − 1*, j )*

**else** PRINT-LCS*(b, X , i, j* − 1*)*

Initial call is PRINT-LCS*(b, X , m, n)*.

*b*[*i, j* ] points to table entry whose subproblem we used in solving LCS of *Xi*

and *Y j.*

When *b*[*i, j* ] = ≈, we have extended LCS by one character. So longest com- mon subsequence = entries with ≈ in them.

**Example: LCS computation**

**Text, letter

Description automatically generated**

**Analysis of LCS computation**

Time Complexity: O(2 ^ N), i.e., exponential as we generate and compare all the subsequences of both the strings

Note: The total number of subsequences of a string is O(2 ^ N).

Space Complexity: O(1) as no extra space is being used.

Where ‘N’ is the length of the shortest of the two strings.

**Code:**

*#include* <bits/stdc++.h>

using namespace std;

string lcs(string X, string Y, int m, int n)

{

int L[m + 1][n + 1];

*for* (int i = 0; i <= m; i++)

{

*for* (int j = 0; j <= n; j++)

{

*if* (i == 0 || j == 0)

L[i][j] = 0;

*else* *if* (X[i - 1] == Y[j - 1])

L[i][j] = L[i - 1][j - 1] + 1;

*else*

L[i][j] = max(L[i - 1][j], L[i][j - 1]);

}

}

int len = L[m][n];

string lcs(len, ' ');

int i = m, j = n;

*while* (i > 0 && j > 0)

{

*if* (X[i - 1] == Y[j - 1])

{

lcs[--len] = X[i - 1];

i--;

j--;

}

*else* *if* (L[i - 1][j] > L[i][j - 1])

i--;

*else*

j--;

}

*return* lcs;

}

int main()

{

string X = "AGGTAB";

string Y = "GXTXAYB";

int m = X.length();

int n = Y.length();

string ans = lcs(X, Y, m, n);

cout << "LCS is " << ans << endl;

cout << "Length is " << ans.length() << endl;

*return* 0;

}

**Output:**

Text

Description automatically generated with medium confidence

**Algorithm:**

X and Y be two given sequences

Initialize a table LCS of dimension X.length \* Y.length

X.label = X

Y.label = Y

LCS[0][] = 0

LCS[][0] = 0

Start from LCS[1][1]

Compare X[i] and Y[j]

If X[i] = Y[j]

LCS[i][j] = 1 + LCS[i-1, j-1]

Point an arrow to LCS[i][j]

Else

LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])

Point an arrow to max(LCS[i-1][j], LCS[i][j-1])

**CONCLUSION:**

Hence, we can conclude that through this experiment we learnt and implemented the Longest

Common Subsequence String Matching Algorithm.