**K. J. Somaiya College of Engineering, Mumbai-77**

(Autonomous College Affiliated to University of Mumbai)

https://ssl.gstatic.com/ui/v1/icons/mail/images/cleardot.gif

Department of Computer Engineering

LIST OF EXPERIMENTS

Subject: Analysis of algorithms

Class : S. E. Computer Engineering (Semester – IV)

Year : 2018– 2019 (Second Term)



**(Programming Language – C/JAVA)**

|  |  |
| --- | --- |
| Introduction  to analysis of algorithm: | -Selection sort  -Insertion sort |
| Divide and Conquer | -Binary search/ Finding Minimum And Maximum  -Merge Sort Analysis /Quick Sort Analysis |
| Greedy Method | - Single Source Shortest Path  -Job sequencing with deadline |
| Dynamic Programming | -0/1 Knapsack  - Matrix Chain Multiplication |
| Backtracking | -8 Queen Problem( N-Queen Problem)  -Graph Coloring |
| String Matching  Algorithms | -Longest Common Subsequence Algorithm |

Subject In-charges:

Ms. Smita Sankhe Ms. Rajani Pamnani



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



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



**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://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.



**Algorithm InsertionSort**

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 SelectionSort**

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*

**The space complexity of Insertion sort:**

The space complexity is actually the *additional* space complexity used by your algorithm, i.e. the extra space that you need, apart from the initial space occupied by the data. Selection and insertion sort use only a constant additional space, apart from the original data, so they are O(1) in space complexity.

**The space complexity of Selection sort:**

The space complexity is actually the *additional* space complexity used by your algorithm, i.e. the extra space that you need, apart from the initial space occupied by the data. Selection and insertion sort use only a constant additional space, apart from the original data, so they are O(1) in space complexity.u

**Time complexity for Insertion sort:**

The time complexity of an algorithm is the amount of computer time it needs to run to completion.

Explaining with an example: 6 3 4 1 2 5

Pass 1: 3 6 4 1 2 5​ | 1 comparison

Pass 2: ​3 4 6 1 2 5​ | 2 comparisons

Pass 3:​ 1 3 4 6 2 5 |​ 3 comparisons

Pass 4:​ 1 2 3 4 6 5 |​ 4 comparisons

Pass 5:​ 1 2 3 4 5 6 |​ 5 comparisons

For 6 inputs, there are 1, 2,... 5 comparisons in each step. Similarly, for ‘n’ inputs, the number of comparisons go from 1, 2,... till (n-​2), (n​-1) comparisons.

I.e. a total of 1 + 2 + 3 + … + (n ​- 2) + (n -​ 1) = [n \* (n -​ 1)] / 2 = (n​2​​ -n) / 2

= O( n​2​)

Hence proved, the time complexity of Insertion sort is: O( n​2​) (asymptotically).

**Time complexity for selection sort:**

The time complexity of an algorithm is the amount of computer time it needs to run to completion.

Explaining with an example:​ 6 5 4 3 2 1

Pass 1: ​ ​ 1 6 5 4 3 2 | 5 comparisons

Pass 2: ​ ​ 1 2 6 5 4 3 | 4 comparisons

Pass 3: ​ ​ 1 2 3 6 5 4 | 3 comparisons

Pass 4: ​ ​ 1 2 3 4 6 5 | 2 comparisons

Pass 5: ​ ​ 1 2 3 4 5 6 | 1 comparisons

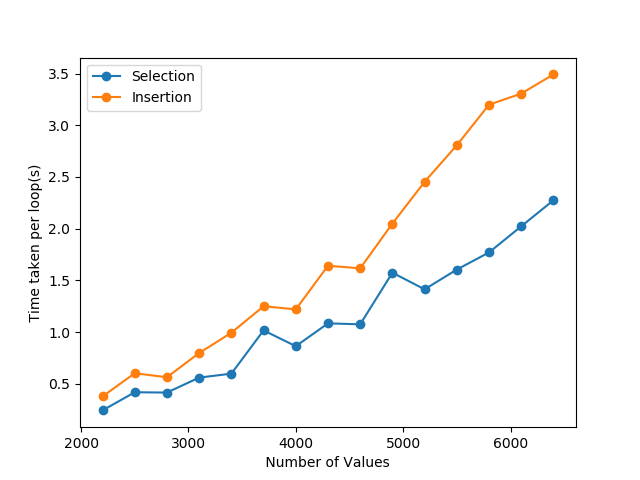
For 6 inputs, there are 5, 4,... 1 comparisons in each step.

Similarly, for ‘n’ inputs, the number of comparisons go from (n​-1), (n-​2),...till 2,1 comparisons (Here we are considering worst case scenario)

I.e. a total of (n -​ 1) + (n ​-2) + (n -​ 3) + … + 2 + 1 = [n \* (n -​ 1)] / 2

Hence proved, the time complexity of Selection sort is: O( n2​​) .

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



**Conclusion:** In this way, we have analyzed time and space complexities for both insertion and selection sorts.

**Topic: Divide and Conquer**

**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.

**Control Abstraction**:

Type DAndC(Problem P)

{

if small (P) return S(P);

else{

divide P into smaller instances P1, P2, …. ,Pk, k ≥1;

Apply DAndC to each of these sub problems;

Return combine(DAndC(P1), DAndC(P2),…., DAndC(Pk));

}

}



|  |
| --- |
| **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:**

|  |  |
| --- | --- |
| 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. **uhttp://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;

}

# Binary Search  
  
def bsr(arr, l, r, x): # Recursive  
 mid = l + (r - l) // 2   
 if l <= r:  
 if arr[mid] == x:  
 return mid  
 elif arr[mid] < x:  
 return bsr(arr, mid+1, r, x)  
 else:  
 return bsr(arr, l, mid-1, x)  
 else:  
 return -1   
  
def bsi(arr, l, r, x): # Iterative  
   
 while l <= r:  
 mid = l + (r - l) // 2  
   
 if arr[mid] == x:  
 return mid  
 elif arr[mid] < x:  
 l = mid + 1  
 else:  
 r = mid - 1  
 return -1  
  
def main():  
 l = [i+1 for i in range(10)]  
   
 posi = bsi(sorted(l), 0, len(l)-1, 2)  
 posr = bsr(sorted(l), 0, len(l)-1, 2)  
   
 if posi == -1 and posr == -1:  
 print('Element not found.')  
 else:  
 print('Element found at {}.'.format(posi))  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

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

It is O(1) i.e constant space complexity. This is because irrespective of the length of the array passed we can eventually find (or maybe not) the required key, by using certain variables.

**Algorithm RecursiveBinarySearch**

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 upper subset

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

**ELSE**

// key has been found

**RETURN** imid;

}

}

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

If one uses a recursive approach, then at each stage, we have to make a recursive call. That means leaving the current invocation on the stack, and calling a new one. When we are *k* levels deep, we have got *k* lots of stack frame, so the space complexity ends up being O(k).

**The Time complexity of Binary Search:**

It simply comes down to dividing an array of size ‘n’ into halves at each iteration until we reach size of one. Hence, mathematically,

1 = N / 2x

multiply by 2x:

2x = N

now do the log2:

log2(2x) = log2 N

x \* log2(2) = log2 N

x \* 1 = log2 N

Thus, we finally end up with logarithmic time complexity.

**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 subproblems. Find where to split the set.

int mid=(i+j)/2;

// Solve the subproblems.

MaxMin(i, mid, max, min);

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

// Combine the solutions.

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

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

}

}

import time, random  
import matplotlib.pyplot as plt  
  
def mmi(arr, mi, ma):  
 for i in range(1, len(arr)):  
 if arr[i] > ma:  
 ma = arr[i]  
 if arr[i] < mi:  
 mi = arr[i]  
 return ma, mi   
   
def mmr(arr, mi, ma):  
 if len(arr) == 1:  
 return arr[0], arr[0]  
 else:  
 # Passing two arrays and then just comparing the final return values  
   
 mi, ma = mmr(arr[: len(arr)//2], mi, ma)  
 mii, maa = mmr(arr[len(arr)//2: ], mi, ma)  
   
 if mi > mii:  
 mi = mii  
 if maa > ma:  
 ma = maa  
 return mi, ma   
   
def main():  
 inc = 250  
 xi = []  
 xr = []  
 yi = []  
 yr = []  
 for i in range(15):  
 l = random.sample(range(1, 10001), 2500 + i\*inc)  
   
 tic = time.time()  
 ma, mi = mmi(l, l[0], l[0])  
 tac = time.time()  
   
 xi.append(i\*inc)  
 yi.append(tac - tic)  
   
 tac = time.time()  
 ma, mi = mmr(l, l[0], l[0])  
 toe = time.time()  
   
 xr.append(i\*inc)  
 yr.append(toe - tac)  
 plt.plot(xi, yi, label='Iterative', marker='o')  
 plt.plot(xr, yr, label='Recursive', marker='x')  
 plt.legend()  
 plt.xlabel('Number of values (n)')  
 plt.ylabel('Time taken...')  
 plt.show()  
   
 # print('List:', l)  
 # print('Max {} Min {}'.format(ma, mi))  
   
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

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

Iterative simply requires O(1) as we obtain our min and max element by simply traversing through the array.

Recursive ends up requiring O(k), where k is the depth of stack which we get after the recursive calls.

**Time complexity for Max-Min:**

Iterative algorithm simply requires O(n) as we have to either way to traverse the entire list, be it best or worst case.

Recursive algorithm eventually requires O(n) but more specifically it requires:

Each call to partition performs a constant amount of work, plus 2 additional recursive calls, each with *half* of the input index range. We can thus construct a *recurrence relation* for the time complexity function:

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

This expands to a geometric series C \* (1 + 2 + 4 + ... ), which continues for log n terms (because at each level of recursion the input size halves, so it decreases geometrically to the stopping condition n = 2).

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

…

T(n) = 2^k \* T(n/ 2^k) + 2^k + 2^k-1 + … + 2

k = logn - 1

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

T(n) = 3n/2 - 2

**CONCLUSION:**

In this way, we successfully implemented divide and conquer approach by implementing min-max algorithm.



|  |
| --- |
| **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:**

|  |  |
| --- | --- |
| 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://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 s a** 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 QuickSort:**

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

**Algorithm MergeSort**

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 subarray *A*[*p* .. *q*] is sorted and subarray //*A*[*q* + 1 .. *r*] is sorted. By restrictions on *p*, *q*, *r*, neither subarray is empty.

**//OUTPUT**: The two subarrays are merged into a single sorted subarray 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:**

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

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

**CONCLUSION:**

**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 ;

}



|  |
| --- |
| **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:**

|  |  |
| --- | --- |
| 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. **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.

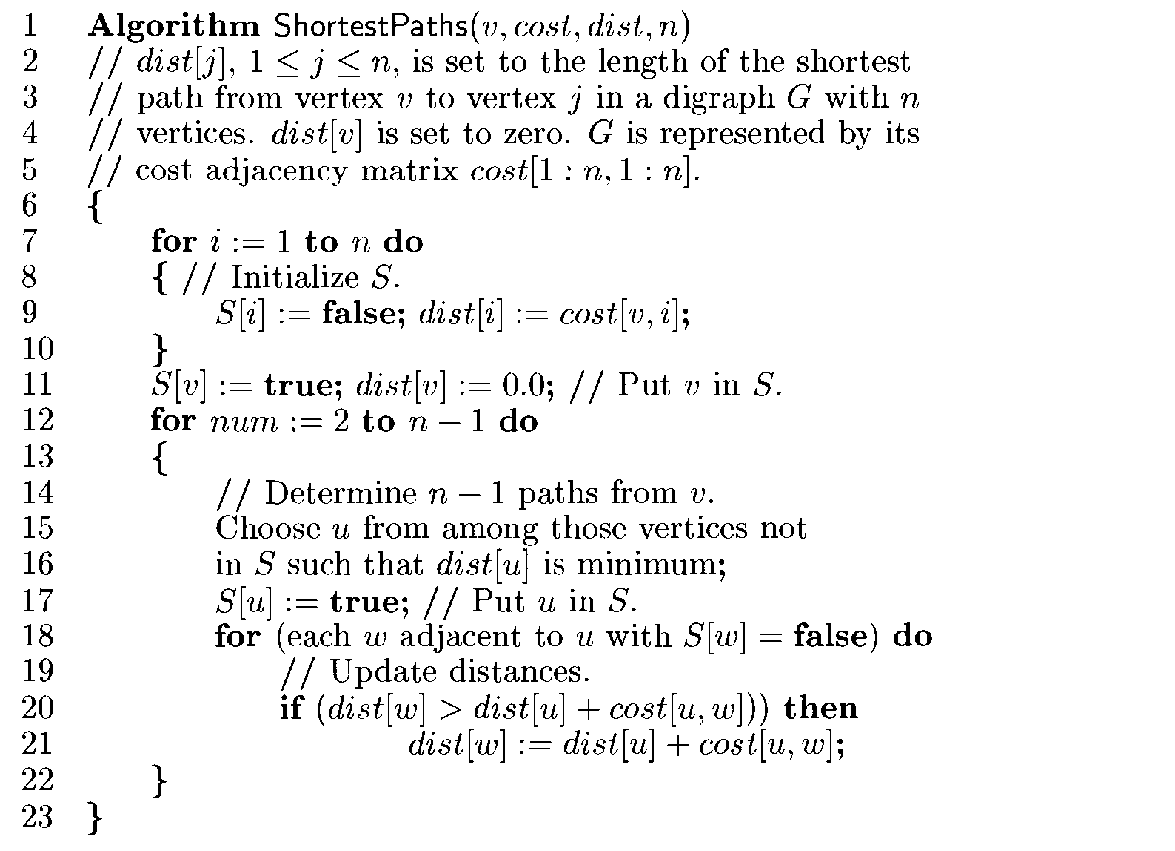


**New Concepts to be learned:**

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



**Algorithm**:



**Example Graph:**

**Solution:**

**Time Complexity for single source shortest path**

**CONCLUSION:**



|  |
| --- |
| **Title:** Implementation of Job sequencing with deadline algorithm using Greedy strategy |



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

**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://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

**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 ;

}

**Problem Definition:**

There are n jobs to be processed on a processor, Each job i has a deadline di ≥0, and profit pi ≥0.Profit pi is earned if and only if job is completed within the deadline.Only one processor is available and it can process only one job at a time.Job is completed if it is processed for a unit time,  so the minimum possible deadline for any job is 1.So a schedule S should be find consisting of a sequence of job “slots”, that maximizes profit.

A set of jobs is feasible if there exists at least one schedule that allows all the jobs to be executed within their deadlines. An optimal solution is a feasible solution with maximum profit value.



**Algorithm**:

algorithm JS (d, J, n) {

  d[0] = J[0] = 0 // Initialize

k = J[1] = 1 // Job 1 is already chosen

  for i = 2 to n do // Decreasing order of P jobs are scheduled

{ r = k //checking feasibility of insertion for i at position

while( d[ J[r]] > d[i]) and (d[J[r]]≠r)) do

r = r-1

if ( d[ J[r]] <= d[i] and (d[i]>r)

{

for q = k to r+1 do //insert I into J[]

J[q+1] = J[q]

J[r+1] = i

k = k+1 }

}

return k

}

**Example Problem :**

**Analysis of Job sequencing with deadline algorithm:**

**CONCLUSION:**

**Topic: DYNAMIC PROGRAMMING**

**Theory:** Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions. For some of the problems that may be viewed in this way ,an optimal sequence of decisions can be found by making the decisions one at a time and never making an erroneous decision .This is true for all problem solvable by the greedy method. For many other problems, it is not possible to make stepwise decisions in such a manner that the sequence of decisions made is optimal.

**Principle of Optimality:** It states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

**Example 1[knapsack]:**

The solution to the knapsack problem can be viewed as the result of a sequence of decisions. We have to decide the values of xi, 1<= i<+n. First we make a decision on xi , then on x2 . then on x3 ,etc. An optimal sequence of decisions maximizes the objective function ∑pixi.

**Example 2 [Optimal merge patterns]:**

An optimal merge pattern tells us which pair of files should be merged at each step. As a decision sequence, the problem calls for us to decide which pair of files should be merged first, which pair second, which pair third, etc. An optimal sequence of decisions is a least-cost sequence.



|  |
| --- |
| **Title: Implementation of Knapsack Problem using Dynamic Programming** |



**Objective** To learn the Dynamic Programming using Knapsack Problemalgorithm

**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. **en.wikipedia.org/wiki/Knapsack\_problem**
4. **www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf**
5. **cse.unl.edu/~ylu/raik283/notes/0-1-knapsack.ppt**
6. **www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf**
7. **cse.unl.edu/~ylu/raik283/notes/0-1-knapsack.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.

**Principle of Optimality:** It states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

The solution to the knapsack problem can be viewed as the result of a sequence of decisions. We have to decide the values of xi, 1<= i<+n. First we make a decision on xi , then on x2 . then on x3 ,etc. An optimal sequence of decisions maximizes the objective function ∑pixi.



**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,



**Algorithm**

// Input:

// Values (stored in array v)

// Weights (stored in array w)

// Number of distinct items (n)

// Knapsack capacity (W)

for j from 0 to W do

m[0, j] := 0

end for

for i from 1 to n do

for j from 0 to W do

if w[i] <= j then

m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])

else

m[i, j] := m[i-1, j]

end if

end for

end for

**Example:**

**Analysis of 0/1 Knapsack algorithm:**

**CONCLUSION:**



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



**Objective:**

**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.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:**

**Algorithm:**

**Example :**

**Solution for the example :**

**Analysis of algorithm:**

**CONCLUSION:**

**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);

}

}

}



|  |
| --- |
| **Title: Implementation of 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**



**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:**

**Solution Using Backtracking Approach:**

**Analysis of Backtracking solution for 8-Queens Problem:**

**CONCLUSION:**



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



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



**Pre Lab/ Prior Concepts:**

Data structures, Concepts of algorithm analysis



**Historical Profile:**

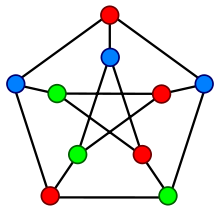
Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices.

***Input:***

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.  


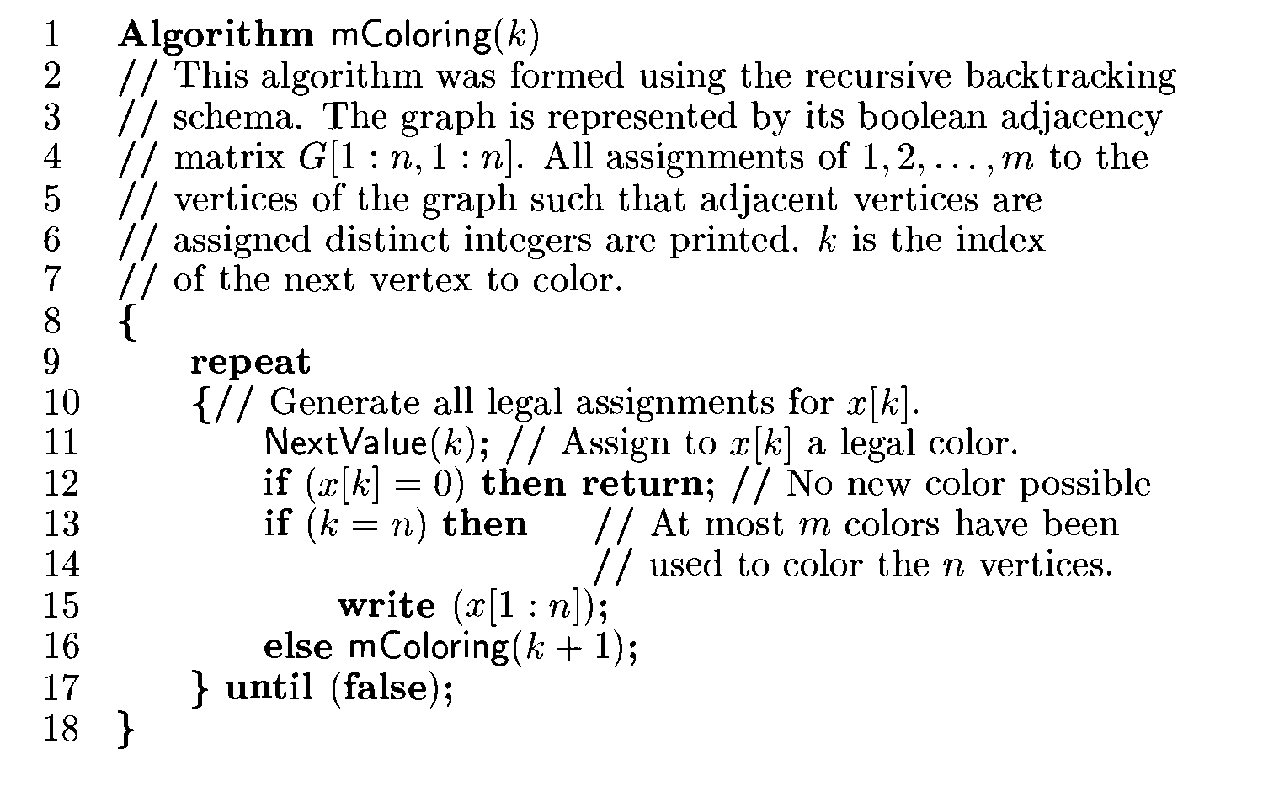


**New Concepts to be learned:**

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:-**

****

**Example Graph Colouring Problem:**

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

**CONCLUSION:**



|  |
| --- |
| **Title: Implementation Of String Matching Algorithm** |



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



**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://en.wikipedia.org/wiki/Longest\_common\_subsequence\_problem**](http://en.wikipedia.org/wiki/Longest_common_subsequence_problem)
4. [**http://www.columbia.edu/~cs2035/courses/csor4231.F11/lcs.pdf**](http://www.columbia.edu/~cs2035/courses/csor4231.F11/lcs.pdf)
5. **http://www-igm.univ-mlv.fr/~lecroq/seqcomp/node4.html**



**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 <> y j

**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**

**LCS computation**

**CONCLUSION:**