

**.CREDEL.**

**(PROGRESS REPORT)**

**MADE BY: CHAITANY BHARDWAJ (15103010 –B1)**

**SIDDHANT SAXENA (15103026 – B1)**

PROJECT TITLE:

**SMART BUSINESS MANAGER (TRANSPORTAION PORTAL)**

**WORK CONTRIBUTION:**

|  |  |  |  |
| --- | --- | --- | --- |
| S. No. | ENROLLMENT NUMBER | MEMBER NAME | CONTRIBUTION (%) |
| 1. | 15103010 | CHAITANY BHARDWAJ  (AREAS WORKED ON :  COINCHANGE , KNAPSACK,LOGIN PAGE,VIDEO EDITING,VIDEO SHOOTING,SCRIPTING) | 50 |
| 2. | 15103026 | SIDDHANT  SAXENA  (AREAS WORKED ON :  TRAVELLING SALESMAN PROBLEM , BIN PACKING , DELAYING ( LOADING DISPLAY ),SCREEN RECORDING , VIDEO SHOOTING , SCRIPTING ) | 50 |

* **ABOUT THE PROJECT :**

**INTRODUCTION:**

In this case, We will be having one shop (transportation portal) and various pickup Destinations (godowns). There will be many stations between these shop and destination. So basically various items will be loaded on the trucks/vans or we can say, items will be weighted in different containers by the use of **“BINPACKING”** algorithm to get the maximum of items/material to be loaded with minimum number of vans to be used thereby making it economically smoother for the owner.

Also in this project we will be mainly focusing on finding the shortest distances between the shop and destinations by the use of **“Travelling Salesman Problem (TSP**)” . By the use of this algorithm we have calculated the amount of cost, including the travel cost, fuel cost etc. Used to transfer our items to a particular destination in every successful way possible.

Cashing or money transactions for this transportation portal will be carried out by the help of **“Coin change algorithm “** , which is basically a kind of greedy algorithm only .

## **USERS OF THE SYSTEM:**

* BUSINESSMEN
* RETAILERS
* TRANSPORTATION DEPARTMENTS

## **FUNCTIONAL REQUIREMENTS:**

* Web accessible information base
* Provide templates for information entry – e.g. education, travel directions, public transportation feedback, etc.
* Allow for easy update of information by city employees
* Allow for easy retrieval of feedback collected to facilitate acting on feedback received.
* Allow to get the info for number of vans to be used.
* Allow to get the info for number of box that can be loaded on the van
* Allow report of fuel consumption and rates .
* Count of number of currency notes to be returned for balance , if there, after the transaction.(coin change algorithm).

## **user interface:**

Professional look and feel.

## **ALGORITHMS AND TOOLS TO BE USED:**

* KNAPSACK
* BINPACKING
* TRAVELLING SALESMAN
* BOYER-MOORE
* CASHING
* CODE BLOCKS
* SCREEN CORDER
* POWER DIRECTOR

## **USECASE DIAGRAM:**

* **Level Of Problems / Functions** (In this case , implementation of algorithm used in this project in order is rated on a scale of 10) :
* Delay function : 5/10
* Loading ( time delaying for the next screen ) : 4/10
* Frames and border ( co-ordinate system) : 7/10
* Welcome page :

- user authentication (4/10)

- user interface (5/10)

* Delivery:

- Travelling sales man problem algorithm (7/10)

- Mileage calculation (4/10)

-cost of petrol calculation(3/10)

* Loading – Unloading :

Binpacking Algorithm ( 7/10)

Knapsack Algorithm (7/10)

Successful delivery notification ( 4/10)

* Cashing :

- Coin change algorithm (6/10)

**OBJECTIVE:**

We will be preparing a “ video” lecture thereby taking up whole of a project ( C++) based on the functioning of legitimate algorithms to be used in it and side by side explaining the respective algorithms related to computer science and technology.

* **PRELIMS OF FEW OF THE ALGORITHMS TO BE USED :**

1. COIN CHANGE ALGORITHM :

FUNCTION USED IN OUR CODE :

void coin\_change(int amount)

{

int sum=0;

while(sum!=amount)

{

for(int i=0;i<9;i++)

{

if(coins[i]+sum<=amount)

{

sum+=coins[i];

cout<<coins[i]<<" ";

break;

}

}

}

}

void cashing(){

frame();

coord.X=55;

coord.Y=12;

SetConsoleCursorPosition(GetStdHandle(STD\_OUTPUT\_HANDLE),coord);

cout<<"Enter the amount paid by customer : ";

int paid,cost,amt;

cin>>paid;

cout<<"\n\n\t\t\t\t\t\t Enter cost of items purchased : ";

cin>>cost;

amt=paid-cost;

cout<<"\n\n\t\t\t\t\t\t Amount to be given back to customer : "<<amt;

cout<<"\n\n\n\t\t\t\t\t\t Give bills of : ";

coin\_change(amt);

cout<<"\t";

\_getch();

start();

}

Given a value N, if we want to make change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn’t matter.

For example, for N = 4 and S = {1,2,3}, there are four solutions: {1,1,1,1},{1,1,2},{2,2},{1,3}. So output should be 4. For N = 10 and S = {2, 5, 3, 6}, there are five solutions: {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}. So the output should be 5.

Optimal Substructure :  
To count total number solutions, we can divide all set solutions in two sets.  
1) Solutions that do not contain mth coin (or Sm).  
 Solutions that contain at least one Sm.  
Let count(S[], m, n) be the function to count the number of solutions, then it can be written as sum of count(S[], m-1, n) and count(S[], m, n-Sm).

Therefore, the problem has optimal substructure property as the problem can be solved using solutions to subproblems.

Following is a simple recursive implementation of the Coin Change problem. The implementation simply follows the recursive structure mentioned above.

|  |
| --- |
| #include<stdio.h>    // Returns the count of ways we can sum  S[0...m-1] coins to get sum n  int count( int S[], int m, int n )  {      // If n is 0 then there is 1 solution (do not include any coin)      if (n == 0)          return 1;        // If n is less than 0 then no solution exists      if (n < 0)          return 0;        // If there are no coins and n is greater than 0, then no solution exist      if (m <=0 && n >= 1)          return 0;        // count is sum of solutions (i) including S[m-1] (ii) excluding S[m-1]      return count( S, m - 1, n ) + count( S, m, n-S[m-1] );  }    // Driver program to test above function  int main()  {      int i, j;      int arr[] = {1, 2, 3};      int m = sizeof(arr)/sizeof(arr[0]);      printf("%d ", count(arr, m, 4));      getchar();      return 0;  } |

Since same suproblems are called again, this problem has Overlapping Subprolems property. So the Coin Change problem has both properties of a dynamic programming problem. Like other typical Dynamic Programming (DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array table[][] in bottom up manner.

**Dynamic Programming Solution**

|  |
| --- |
| #include<stdio.h>    int count( int S[], int m, int n )  {      int i, j, x, y;        // We need n+1 rows as the table is consturcted in bottom up manner using      // the base case 0 value case (n = 0)      int table[n+1][m];        // Fill the enteries for 0 value case (n = 0)      for (i=0; i<m; i++)          table[0][i] = 1;        // Fill rest of the table enteries in bottom up manner      for (i = 1; i < n+1; i++)      {          for (j = 0; j < m; j++)          {              // Count of solutions including S[j]              x = (i-S[j] >= 0)? table[i - S[j]][j]: 0;                // Count of solutions excluding S[j]              y = (j >= 1)? table[i][j-1]: 0;                // total count              table[i][j] = x + y;          }      }      return table[n][m-1];  }    // Driver program to test above function  int main()  {      int arr[] = {1, 2, 3};      int m = sizeof(arr)/sizeof(arr[0]);      int n = 4;      printf(" %d ", count(arr, m, n));      return 0;  } |

Output:

4

Time Complexity: O(mn)

1. **TRAVELLING SALESMAN PROBLEM:**

FUNCTION USED IN OUR CODE :

void delivery()

{

frame();

int arr[10][10],n,i,j,x[10][10];

coord.X=20;

coord.Y=12;

SetConsoleCursorPosition(GetStdHandle(STD\_OUTPUT\_HANDLE),coord);

cout<<"Enter the number of godowns the driver need to pick items from ( less than 10 ) : ";

cin>>n;

cout<<"\n\n\n";

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

{

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

{

if(i==j)

arr[i][j]=0;

if(i<j)

{

cout<<"\t\t\t\t\t Enter distance between Godown "<<(char)(i+65)<<" & "<<(char)(j+65)<<" : ";

cin>>arr[i][j];

arr[j][i]=arr[i][j];

}

}

}

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

{

cout<<"\n";

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

{

x[i][j]=arr[i][j];

}

}

for(i=0;i<n;i++) // min cost path using Dynamic Programming

{

for(j=1;j<n;j++)

{

if(i==0)

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

else if(j==0)

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

else

{ int min;

if(x[i][j-1]<x[i-1][j])

min=x[i][j-1];

else

min=x[i-1][j];

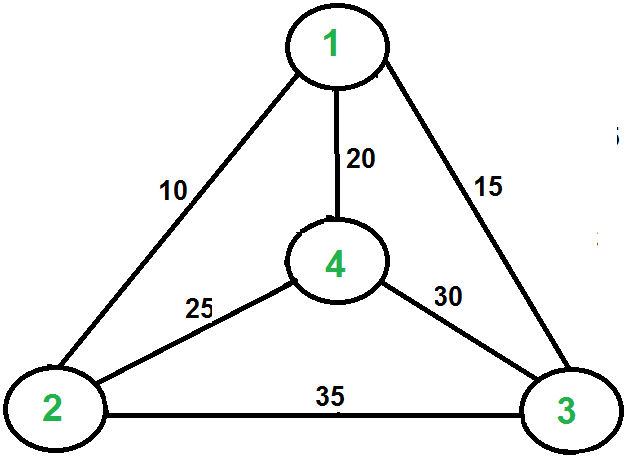
x[i][j]+=min;

}

}

}

**Travelling Salesman Problem (TSP):** Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.  
Note the difference between [Hamiltonian Cycle](http://www.geeksforgeeks.org/backtracking-set-7-hamiltonian-cycle/) and TSP. The Hamiltoninan cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

[](http://www.geeksforgeeks.org/wp-content/uploads/Euler12.png)

For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.

The problem is a famous [NP hard](http://www.geeksforgeeks.org/np-completeness-set-1/)problem. There is no polynomial time know solution for this problem.

Following are different solutions for the traveling salesman problem.

**Naive Solution:**  
1) Consider city 1 as the starting and ending point.  
2) Generate all (n-1)! [Permutations](http://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/)of cities.  
3) Calculate cost of every permutation and keep track of minimum cost permutation.  
4) Return the permutation with minimum cost.

Time Complexity: ?(n!)

**Dynamic Programming:**  
Let the given set of vertices be {1, 2, 3, 4,….n}. Let us consider 1 as starting and ending point of output. For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once. Let the cost of this path be cost(i), the cost of corresponding Cycle would be cost(i) + dist(i, 1) where dist(i, 1) is the distance from i to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far. Now the question is how to get cost(i)?  
To calculate cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems. Let us define a term C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i.  
We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

If size of S is 2, then S must be {1, i},

C(S, i) = dist(1, i)

Else if size of S is greater than 2.

C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.

For a set of size n, we consider n-2 subsets each of size n-1 such that all subsets don’t have nth in them.  
Using the above recurrence relation, we can write dynamic programming based solution. There are at most O(n\*2n) subproblems, and each one takes linear time to solve. The total running time is therefore O(n2\*2n). The time complexity is much less than O(n!), but still exponential. Space required is also exponential. So this approach is also infeasible even for slightly higher number of vertices.

1. **FRACTIONAL KNAPSACK ALGORITHM:**

**FUCTION USED IN OUR CODE :**

void delivery()

{

frame();

int arr[10][10],n,i,j,x[10][10];

coord.X=20;

coord.Y=12;

SetConsoleCursorPosition(GetStdHandle(STD\_OUTPUT\_HANDLE),coord);

cout<<"Enter the number of godowns the driver need to pick items from ( less than 10 ) : ";

cin>>n;

cout<<"\n\n\n";

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

{

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

{

if(i==j)

arr[i][j]=0;

if(i<j)

{

cout<<"\t\t\t\t\t Enter distance between Godown "<<(char)(i+65)<<" & "<<(char)(j+65)<<" : ";

cin>>arr[i][j];

arr[j][i]=arr[i][j];

}

}

}

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

{

cout<<"\n";

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

{

x[i][j]=arr[i][j];

}

}

for(i=0;i<n;i++) // min cost path using Dynamic Programming

{

for(j=1;j<n;j++)

{

if(i==0)

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

else if(j==0)

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

else

{ int min;

if(x[i][j-1]<x[i-1][j])

min=x[i][j-1];

else

min=x[i-1][j];

x[i][j]+=min;

}

}

}

Given weights and values of n items, we need put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

In the [0-1 Knapsack problem](http://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-knapsack-problem/), we are not allowed to break items. We either take the whole item or don’t take it.

Input:

Items as (value, weight) pairs

arr[] = {{60, 10}, {100, 20}, {120, 30}}

Knapsack Capacity, W = 50;

Output:

Maximum possible value = 220

by taking items of weight 20 and 30 kg

In **Fractional Knapsack**, we can break items for maximizing the total value of knapsack. This problem in which we can break item also called fractional knapsack problem.

Input :

Same as above

Output :

Maximum possible value = 240

By taking full items of 10 kg, 20 kg and

2/3rd of last item of 30 kg

A **brute-force solution** would be to try all possible subset with all different fraction but that will be too much time taking.

An **efficient solution** is to use Greedy approach. The basic idea of greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with highest ratio and add them until we can’t add the next item as whole and at the end add the next item as much as we can. Which will always be optimal solution of this problem.

A simple code with our own comparison function can be written as follows, please see sort function more closely, the third argument to sort function is our comparison function which sorts the item according to value/weight ratio in non-decreasing order.  
After sorting we need to loop over these items and add them in our knapsack satisfying above mentioned criteria.

|  |
| --- |
| // C/C++ program to solve fractional Knapsack Problem  #include <bits/stdc++.h>  using namespace std;    // Stucture for Item which store weight and corresponding  // value of Item  struct Item  {      int value, weight;        // Constructor      Item(int value, int weight) : value(value), weight(weight)      {}  };    // Comparison function to sort Item according to val/weight ratio  bool cmp(struct Item a, struct Item b)  {      double r1 = (double)a.value / a.weight;      double r2 = (double)b.value / b.weight;      return r1 > r2;  }    // Main greedy function to solve problem  double fractionalKnapsack(int W, struct Item arr[], int n)  {      //    sorting Item on basis of ration      sort(arr, arr + n, cmp);        //    Uncomment to see new order of Items with their ratio      /\*      for (int i = 0; i < n; i++)      {          cout << arr[i].value << "  " << arr[i].weight << " : "               << ((double)arr[i].value / arr[i].weight) << endl;      }      \*/        int curWeight = 0;  // Current weight in knapsack      double finalvalue = 0.0; // Result (value in Knapsack)        // Looping through all Items      for (int i = 0; i < n; i++)      {          // If adding Item won't overflow, add it completely          if (curWeight + arr[i].weight <= W)          {              curWeight += arr[i].weight;              finalvalue += arr[i].value;          }            // If we can't add current Item, add fractional part of it          else          {              int remain = W - curWeight;              finalvalue += arr[i].value \* ((double) remain / arr[i].weight);              break;          }      }        // Returning final value      return finalvalue;  }    // driver program to test above function  int main()  {      int W = 50;   //    Weight of knapsack      Item arr[] = {{60, 10}, {100, 20}, {120, 30}};        int n = sizeof(arr) / sizeof(arr[0]);        cout << "Maximum value we can obtain = "           << fractionalKnapsack(W, arr, n);      return 0;  } |

Output :

Maximum value in Knapsack = 240

1. **BIN-PACKING ALGORITHM :**

**FUCNTION USED IN OUR CODE :**

int n\_container;

n\_container=bin\_pack(items,n\_items,10);

cout<<"\n\n\t\t\t No of vans required : "<<n\_container;

if(n\_container+bins>20) // max no. of vans is 20 in the company

{

cout<<"\n\n\t\t\t QUANTITY EXCEEDING. HENCE WONT TAKE ITEMS FROM HERE. ";

\_getch();

drive(++godown,n,bins);

}

else

{

bins+=n\_container;

cout<<"\n\n\t\t\t ITEMS PICKED UP. TOTAL VANS : "<<bins;

cout<<endl;

\_getch();

drive(++godown,n,bins);

}

}

}

Given n items of different weights and bins each of capacity c, assign each item to a bin such that number of total used bins is minimized. It may be assumed that all items have weights smaller than bin capacity.

Example:

Input: wieght[] = {4, 8, 1, 4, 2, 1}

Bin Capacity c = 10

Output: 2

We need minimum 2 bins to accommodate all items

First bin contains {4, 4, 2} and second bin {8, 2}

Input: wieght[] = {9, 8, 2, 2, 5, 4}

Bin Capacity c = 10

Output: 4

We need minimum 4 bins to accommodate all items.

Input: wieght[] = {2, 5, 4, 7, 1, 3, 8};

Bin Capacity c = 10

Output: 3

**Lower Bound**

We can always find a lower bound on minimum number of bins required. The lower bound can be given as :

Min no. of bins >= Ceil ((Total Weight) / (Bin Capacity))

In the above examples, lower bound for first example is “ceil(4 + 8 + 1 + 4 + 2 + 1)/10” = 2 and lower bound in second example is “ceil(9 + 8 + 2 + 2 + 5 + 4)/10” = 3.  
This problem is a NP Hard problem and finding an exact minimum number of bins takes exponential time. Following are approximate algorithms for this problem.

**Applications**

1. Loading of containers like trucks.
2. Placing data on multiple disks.
3. Job scheduling.
4. Packing advertisements in fixed length radio/TV station breaks.
5. Storing a large collection of music onto tapes/CD’s, etc.

**Online Algorithms**

These algorithms are for Bin Packing problems where items arrive one at a time (in unknown order), each must be put in a bin, before considering the next item.

**1. Next Fit:**  
When processing next item, check if it fits in the same bin as the last item. Use a new bin only if it does not.

Below is C++ implementation for this algorithm.

|  |
| --- |
| // C++ program to find number of bins required using  // next fit algorithm.  #include <bits/stdc++.h>  using namespace std;    // Returns number of bins required using next fit  // online algorithm  int nextFit(int weight[], int n, int c)  {     // Initialize result (Count of bins) and remaining     // capacity in current bin.     int res = 0, bin\_rem = c;       // Place items one by one     for (int i=0; i<n; i++)     {         // If this item can't fit in current bin         if (weight[i] > bin\_rem)         {            res++;  // Use a new bin            bin\_rem = c - weight[i];         }         else           bin\_rem -= weight[i];     }     return res;  }    // Driver program  int main()  {      int weight[] = {2, 5, 4, 7, 1, 3, 8};      int c = 10;      int n = sizeof(weight) / sizeof(weight[0]);      cout << "Number of bins required in Next Fit : "           << nextFit(weight, n, c);      return 0;  } |

Output:

Number of bins required in Next Fit : 4

Next Fit is a simple algorithm. It requires only O(n) time and O(1) extra space to process n items.

Next Fit is 2 approximate, i.e., the number of bins used by this algorithm is bounded by twice of optimal. Consider any two adjacent bins. The sum of items in these two bins must be > c; otherwise, NextFit would have put all the items of second bin into the first. The same holds for all other bins. Thus, at most half the space is wasted, and so Next Fit uses at most 2M bins if M is optimal.

1. **BOYER-MOORE ALGORITHM :**

**A**

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results. A typical problem statement would be-  
Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

Examples:

Input: txt[] = "THIS IS A TEST TEXT"

pat[] = "TEST"

Output: Pattern found at index 10

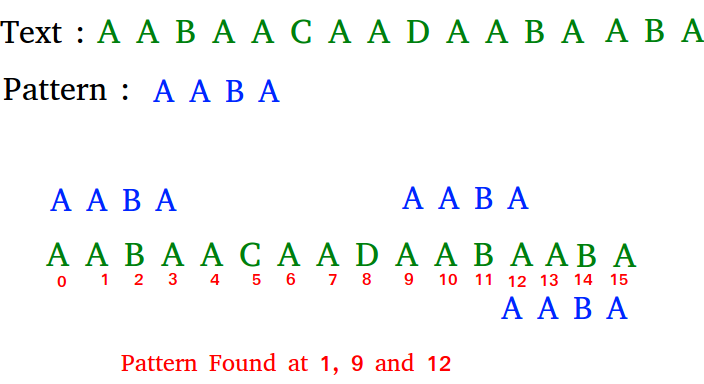
Input: txt[] = "AABAACAADAABAABA"

pat[] = "AABA"

Output: Pattern found at index 0

Pattern found at index 9

Pattern found at index 12



In this post, we will discuss Boyer Moore pattern searching algorithm. Like [KMP](http://www.geeksforgeeks.org/archives/11902)and [Finite Automata](http://www.geeksforgeeks.org/archives/18919)algorithms, Boyer Moore algorithm also preprocesses the pattern.  
Boyer Moore is a combination of following two approaches.  
1) Bad Character Heuristic  
2) Good Suffix Heuristic

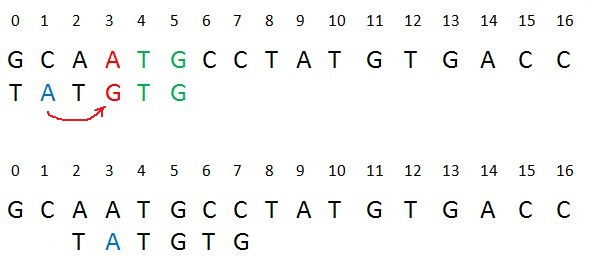
Both of the above heuristics can also be used independently to search a pattern in a text. Let us first understand how two independent approaches work together in the Boyer Moore algorithm. If we take a look at the [Naive algorithm](http://www.geeksforgeeks.org/archives/11871), it slides the pattern over the text one by one. [KMP algorithm](http://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/) does preprocessing over the pattern so that the pattern can be shifted by more than one. The Boyer Moore algorithm does preprocessing for the same reason. It preporcesses the pattern and creates different arrays for both heuristics. At every step, it slides the pattern by max of the slides suggested by the two heuristics. **So it uses best of the two heuristics at every step**.  
Unlike the previous pattern searching algorithms, **Boyer Moore algorithm starts matching from the last character of the pattern.**

In this post, we will discuss bad character heuristic, and discuss Good Suffix heuristic in the next post.

**Bad Character Heuristic**

The idea of bad character heuristic is simple. The character of the text which doesn’t match with the current character of pattern is called the **Bad Character**. Upon mismatch we shift the pattern until –  
1) The mismatch become a match  
2) Pattern P move past the mismatch character.

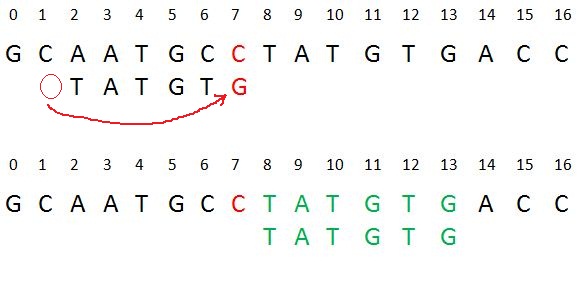
**Case 1 – Mismatch become match**  
We will lookup the position of last occurence of mismatching character in pattern and if mismatching character exist in pattern then we’ll shift the pattern such that it get aligned to the mismatching character in text T.



*case 1*

**Explanation:** In the above example, we got a mismatch at position 3. Here our mismatching character is “A”. Now we will search for last occurence of “A” in pattern. We got “A” at position 1 in pattern (displayed in Blue) and this is the last occurence of it. Now we will shift pattern 2 times so that “A” in pattern get aligned with “A” in text.

**Case 2 – Pattern move past the mismatch character**  
We’ll lookup the position of last occurence of mismatching character in pattern and if character does not exist we will shift pattern past the mismatching character.



*case2*

**Explanation:** Here we have a mismatch at position 7. The mismatching character “C” does not exist in pattern before position 7 so we’ll shift pattern past to the position 7 and eventually in above example we have got a perfect match of pattern (displayed in Green). We are doing this because, “C” do not exist in pattern so at every shift before position 7 we will get mismatch and our search will be fruitless.

In following implementation, we preprocess the pattern and store the last occurrence of every possible character in an array of size equal to alphabet size. If the character is not present at all, then it may result in a shift by m (length of pattern). Therefore, the bad character heuristic takes  time in the best case.

|  |
| --- |
| /\* Program for Bad Character Heuristic of Boyer     Moore String Matching Algorithm \*/  # include <limits.h>  # include <string.h>  # include <stdio.h>    # define NO\_OF\_CHARS 256    // A utility function to get maximum of two integers  int max (int a, int b) { return (a > b)? a: b; }    // The preprocessing function for Boyer Moore's  // bad character heuristic  void badCharHeuristic( char \*str, int size,                          int badchar[NO\_OF\_CHARS])  {      int i;        // Initialize all occurrences as -1      for (i = 0; i < NO\_OF\_CHARS; i++)           badchar[i] = -1;        // Fill the actual value of last occurrence      // of a character      for (i = 0; i < size; i++)           badchar[(int) str[i]] = i;  }    /\* A pattern searching function that uses Bad     Character Heuristic of Boyer Moore Algorithm \*/  void search( char \*txt,  char \*pat)  {      int m = strlen(pat);      int n = strlen(txt);        int badchar[NO\_OF\_CHARS];        /\* Fill the bad character array by calling         the preprocessing function badCharHeuristic()         for given pattern \*/      badCharHeuristic(pat, m, badchar);        int s = 0;  // s is shift of the pattern with                  // respect to text      while(s <= (n - m))      {          int j = m-1;            /\* Keep reducing index j of pattern while             characters of pattern and text are             matching at this shift s \*/          while(j >= 0 && pat[j] == txt[s+j])              j--;            /\* If the pattern is present at current             shift, then index j will become -1 after             the above loop \*/          if (j < 0)          {              printf("\n pattern occurs at shift = %d", s);                /\* Shift the pattern so that the next                 character in text aligns with the last                 occurrence of it in pattern.                 The condition s+m < n is necessary for                 the case when pattern occurs at the end                 of text \*/              s += (s+m < n)? m-badchar[txt[s+m]] : 1;            }            else              /\* Shift the pattern so that the bad character                 in text aligns with the last occurrence of                 it in pattern. The max function is used to                 make sure that we get a positive shift.                 We may get a negative shift if the last                 occurrence  of bad character in pattern                 is on the right side of the current                 character. \*/              s += max(1, j - badchar[txt[s+j]]);      }  }    /\* Driver program to test above funtion \*/  int main()  {      char txt[] = "ABAAABCD";      char pat[] = "ABC";      search(txt, pat);      return 0;  } |

Output:

pattern occurs at shift = 4

The Bad Character Heuristic may take  time in worst case. The worst case occurs when all characters of the text and pattern are same. For example, txt[] = “AAAAAAAAAAAAAAAAAA” and pat[] = “AAAAA”.

**THANK YOU !**