Contents

Chapter 1

Given a matrix of ‘O’ and ‘X’, find the largest subsquare surrounded by ‘X’

Given a matrix where every element is either ‘O’ or ‘X’, find the largest subsquare surrounded by ‘X’.

In the below article, it is assumed that the given matrix is also square matrix. The code given below can be easily extended for rectangular matrices.

Examples:

```
Input: mat[N][N] = { {'X', 'O', 'X', 'X', 'X'},
                     {'X', 'X', 'X', 'X', 'X'},
                     {'X', 'X', 'O', 'X', 'O'},
                     {'X', 'X', 'X', 'X', 'X'},
                     {'X', 'X', 'X', 'O', 'O'},
                    };
Output: 3
The square submatrix starting at (1, 1) is the largest
submatrix surrounded by 'X'

Input: mat[M][N] = { {'X', 'O', 'X', 'X', 'X', 'X'},
                     {'X', 'O', 'X', 'X', 'O', 'X'},
                     {'X', 'X', 'X', 'O', 'O', 'X'},
                     {'X', 'X', 'X', 'X', 'X', 'X'},
                     {'X', 'X', 'X', 'O', 'X', 'O'},
                    };
Output: 4
The square submatrix starting at (0, 2) is the largest
submatrix surrounded by 'X'
```

A **Simple Solution** is to consider every square submatrix and check whether is has all corner edges filled with ‘X’. The time complexity of this solution is O(N^4).

We can solve this problem **in O(N^3) time** using extra space. The idea is to create two auxiliary arrays hor[N][N] and ver[N][N]. The value stored in hor[i][j] is the number of horizontal continuous ‘X’ characters till mat[i][j] in mat[][]. Similarly, the value stored in ver[i][j] is the number of vertical continuous ‘X’ characters till mat[i][j] in mat[][]. Following is an example.

```
mat[6][6] =  X  O  X  X  X  X
             X  O  X  X  O  X
             X  X  X  O  O  X
             O  X  X  X  X  X
             X  X  X  O  X  O
             O  O  X  O  O  O

hor[6][6] = 1  0  1  2  3  4
            1  0  1  2  0  1
            1  2  3  0  0  1
            0  1  2  3  4  5
            1  2  3  0  1  0
            0  0  1  0  0  0

ver[6][6] = 1  0  1  1  1  1
            2  0  2  2  0  2
            3  1  3  0  0  3
            0  2  4  1  1  4
            1  3  5  0  2  0
            0  0  6  0  0  0
```

Once we have filled values in hor[][] and ver[][], we start from the bottommost-rightmost corner of matrix and move toward the leftmost-topmost in row by row manner. For every visited entry mat[i][j], we compare the values of hor[i][j] and ver[i][j], and pick the smaller of two as we need a square. Let the smaller of two be 'small'. After picking smaller of two, we check if both ver[][] and hor[][] for left and up edges respectively. If they have entries for the same, then we found a subsquare. Otherwise we try for small-1.

Below is C++ implementation of the above idea.

```
// A C++ program to find  the largest subsquare
// surrounded by 'X' in a given matrix of 'O' and 'X'
#include<iostream>
using namespace std;

// Size of given matrix is N X N
#define N 6

// A utility function to find minimum of two numbers
int getMin(int x, int y) { return (x<y)? x: y; }

// Returns size of maximum size subsquare matrix
// surrounded by 'X'
int findSubSquare(int mat[][N])
{
    int max = 1; // Initialize result

    // Initialize the left-top value in hor[][] and ver[][]
    int hor[N][N], ver[N][N];
    hor[0][0] = ver[0][0] = (mat[0][0] == 'X');

    // Fill values in hor[][] and ver[][]
    for (int i=0; i<N; i++)
    {
        for (int j=0; j<N; j++)
        {
            if (mat[i][j] == 'O')
                ver[i][j] = hor[i][j] = 0;
            else
            {
                hor[i][j] = (j==0)? 1: hor[i][j-1] + 1;
                ver[i][j] = (i==0)? 1: ver[i-1][j] + 1;
            }
        }
    }

    // Start from the rightmost-bottommost corner element and find
    // the largest ssubsquare with the help of hor[][] and ver[][]
    for (int i = N-1; i>=1; i--)
    {
        for (int j = N-1; j>=1; j--)
        {
            // Find smaller of values in hor[][] and ver[][]
            // A Square can only be made by taking smaller
            // value
            int small = getMin(hor[i][j], ver[i][j]);

            // At this point, we are sure that there is a right
            // vertical line and bottom horizontal line of length
            // at least 'small'.

            // We found a bigger square if following conditions
            // are met:
            // 1)If side of square is greater than max.
            // 2)There is a left vertical line of length >= 'small'
            // 3)There is a top horizontal line of length >= 'small'
            while (small > max)
            {
                if (ver[i][j-small+1] >= small &&
                    hor[i-small+1][j] >= small)
                {
                    max = small;
                }
                small--;
            }
        }
    }
    return max;
}

// Driver program to test above function
int main()
{
    int mat[][N] =  {{'X', 'O', 'X', 'X', 'X', 'X'},
                     {'X', 'O', 'X', 'X', 'O', 'X'},
                     {'X', 'X', 'X', 'O', 'O', 'X'},
                     {'O', 'X', 'X', 'X', 'X', 'X'},
                     {'X', 'X', 'X', 'O', 'X', 'O'},
                     {'O', 'O', 'X', 'O', 'O', 'O'},
                    };
    cout << findSubSquare(mat);
    return 0;
}
```

Output:

```
4
```

This article is contributed by **Anuj**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/given-matrix-o-x-find-largest-subsquare-surrounded-x/

Category: Arrays

Post navigation

← Print all possible strings that can be made by placing spaces D E Shaw Interview | Set 7 (Off Campus) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 2

Nuts & Bolts Problem (Lock & Key problem)

Given a set of n nuts of different sizes and n bolts of different sizes. There is a one-one mapping between nuts and bolts. Match nuts and bolts efficiently.
Constraint: Comparison of a nut to another nut or a bolt to another bolt is not allowed. It means nut can only be compared with bolt and bolt can only be compared with nut to see which one is bigger/smaller.

Other way of asking this problem is, given a box with locks and keys where one lock can be opened by one key in the box. We need to match the pair.

Brute force Way: Start with the first bolt and compare it with each nut until we find a match. In the worst case we require n comparisons. Doing this for all bolts gives us O(n^2) complexity.

Quick Sort Way: We can use quick sort technique to solve this. We represent nuts and bolts in character array for understanding the logic.

Nuts represented as array of character
char nuts[] = {'@', '#', '$', '%', '^', '&'}

Bolts represented as array of character
char bolts[] = {'$', '%', '&', '^', '@', '#'}

This algorithm first performs a partition by picking last element of bolts array as pivot, rearrange the array of nuts and returns the partition index 'i' such that all nuts smaller than nuts[i] are on the left side and all nuts greater than nuts[i] are on the right side. Next using the nuts[i] we can partition the array of bolts. Partitioning operations can easily be implemented in O(n). This operation also makes nuts and bolts array nicely partitioned. Now we apply this partitioning recursively on the left and right sub-array of nuts and bolts.

As we apply partitioning on nuts and bolts both so the total time complexity will be $\Theta(2*n\log n) = \Theta(n\log n)$ on average.

Here for the sake of simplicity we have chosen last element always as pivot. We can do randomized quick sort too.

A Java based implementation of idea is below:

```
// Java program to solve nut and bolt problem using Quick Sort
public class NutsAndBoltsMatch
{
```

```
//Driver method
public static void main(String[] args)
{
    // Nuts and bolts are represented as array of characters
    char nuts[] = {'@', '#', '$', '%', '^', '&'};
    char bolts[] = {'$', '%', '&', '^', '@', '#'};

    // Method based on quick sort which matches nuts and bolts
    matchPairs(nuts, bolts, 0, 5);

    System.out.println("Matched nuts and bolts are : ");
    printArray(nuts);
    printArray(bolts);
}

// Method to print the array
private static void printArray(char[] arr) {
    for (char ch : arr){
        System.out.print(ch + " ");
    }
    System.out.print("\n");
}

// Method which works just like quick sort
private static void matchPairs(char[] nuts, char[] bolts, int low,
                                                          int high)
{
    if (low < high)
    {
        // Choose last character of bolts array for nuts partition.
        int pivot = partition(nuts, low, high, bolts[high]);

        // Now using the partition of nuts choose that for bolts
        // partition.
        partition(bolts, low, high, nuts[pivot]);

        // Recur for [low...pivot-1] & [pivot+1...high] for nuts and
        // bolts array.
        matchPairs(nuts, bolts, low, pivot-1);
        matchPairs(nuts, bolts, pivot+1, high);
    }
}

// Similar to standard partition method. Here we pass the pivot element
// too instead of choosing it inside the method.
private static int partition(char[] arr, int low, int high, char pivot)
{
    int i = low;
    char temp1, temp2;
    for (int j = low; j < high; j++)
    {
        if (arr[j] < pivot){
            temp1 = arr[i];
            arr[i] = arr[j];
```

```
                arr[j] = temp1;
                i++;
            } else if(arr[j] == pivot){
                temp1 = arr[j];
                arr[j] = arr[high];
                arr[high] = temp1;
                j--;
            }
        }
        temp2 = arr[i];
        arr[i] = arr[high];
        arr[high] = temp2;

        // Return the partition index of an array based on the pivot
        // element of other array.
        return i;
    }
}
```

Output:

```
Matched nuts and bolts are :
# $ % & @ ^
# $ % & @ ^
```

This article is contributed by **Kumar Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/nuts-bolts-problem-lock-key-problem/

Chapter 3

Given n appointments, find all conflicting appointments

Given n appointments, find all conflicting appointments.

Examples:

```
Input: appointments[] = { {1, 5} {3, 7}, {2, 6}, {10, 15}, {5, 6}, {4, 100}}
Output: Following are conflicting intervals
[3,7] Conflicts with [1,5]
[2,6] Conflicts with [1,5]
[5,6] Conflicts with [3,7]
[4,100] Conflicts with [1,5]
```

An appointment is conflicting, if it conflicts with any of the previous appointments in array.

We strongly recommend to minimize the browser and try this yourself first.

A **Simple Solution** is to one by one process all appointments from second appointment to last. For every appointment i, check if it conflicts with i-1, i-2, … 0. The time complexity of this method is O(n^2).

We can use **Interval Tree** to solve this problem in O(nLogn) time. Following is detailed algorithm.

```
1) Create an Interval Tree, initially with the first appointment.
2) Do following for all other appointments starting from the second one.
   a) Check if the current appointment conflicts with any of the existing
     appointments in Interval Tree.  If conflicts, then print the current
     appointment.  This step can be done O(Logn) time.
   b) Insert the current appointment in Interval Tree. This step also can
      be done O(Logn) time.
```

Following is C++ implementation of above idea.

```
// C++ program to print all conflicting appointments in a
// given set of appointments
#include <iostream>
```

```
using namespace std;

// Structure to represent an interval
struct Interval
{
    int low, high;
};

// Structure to represent a node in Interval Search Tree
struct ITNode
{
    Interval *i;  // 'i' could also be a normal variable
    int max;
    ITNode *left, *right;
};

// A utility function to create a new Interval Search Tree Node
ITNode * newNode(Interval i)
{
    ITNode *temp = new ITNode;
    temp->i = new Interval(i);
    temp->max = i.high;
    temp->left = temp->right = NULL;
};

// A utility function to insert a new Interval Search Tree
// Node. This is similar to BST Insert.  Here the low value
//  of interval is used tomaintain BST property
ITNode *insert(ITNode *root, Interval i)
{
    // Base case: Tree is empty, new node becomes root
    if (root == NULL)
        return newNode(i);

    // Get low value of interval at root
    int l = root->i->low;

    // If root's low value is smaller, then new interval
    //  goes to left subtree
    if (i.low < l)
        root->left = insert(root->left, i);

    // Else, new node goes to right subtree.
    else
        root->right = insert(root->right, i);

    // Update the max value of this ancestor if needed
    if (root->max < i.high)
        root->max = i.high;

    return root;
}

// A utility function to check if given two intervals overlap
```

```
bool doOVerlap(Interval i1, Interval i2)
{
    if (i1.low < i2.high && i2.low < i1.high)
        return true;
    return false;
}

// The main function that searches a given interval i
// in a given Interval Tree.
Interval *overlapSearch(ITNode *root, Interval i)
{
    // Base Case, tree is empty
    if (root == NULL) return NULL;

    // If given interval overlaps with root
    if (doOVerlap(*(root->i), i))
        return root->i;

    // If left child of root is present and max of left child
    // is greater than or equal to given interval, then i may
    // overlap with an interval is left subtree
    if (root->left != NULL && root->left->max >= i.low)
        return overlapSearch(root->left, i);

    // Else interval can only overlap with right subtree
    return overlapSearch(root->right, i);
}

// This function prints all conflicting appointments in a given
// array of apointments.
void printConflicting(Interval appt[], int n)
{
     // Create an empty Interval Search Tree, add first
     // appointment
     ITNode *root = NULL;
     root = insert(root, appt[0]);

     // Process rest of the intervals
     for (int i=1; i<n; i++)
     {
         // If current appointment conflicts with any of the
         // existing intervals, print it
         Interval *res = overlapSearch(root, appt[i]);
         if (res != NULL)
            cout << "[" << appt[i].low << "," << appt[i].high
                 << "] Conflicts with [" << res->low << ","
                 << res->high << "]\n";

         // Insert this appointment
         root = insert(root, appt[i]);
     }
}
```

```
// Driver program to test above functions
int main()
{
    // Let us create interval tree shown in above figure
    Interval appt[] = { {1, 5}, {3, 7}, {2, 6}, {10, 15},
                        {5, 6}, {4, 100}};
    int n = sizeof(appt)/sizeof(appt[0]);
    cout << "Following are conflicting intervals\n";
    printConflicting(appt, n);
    return 0;
}
```

Output:

```
 Following are conflicting intervals
[3,7] Conflicts with [1,5]
[2,6] Conflicts with [1,5]
[5,6] Conflicts with [3,7]
[4,100] Conflicts with [1,5]
```

Note that the above implementation uses simple Binary Search Tree insert operations. Therefore, time complexity of the above implementation is more than O(nLogn). We can use Red-Black Tree or AVL Tree balancing techniques to make the above implementation O(nLogn).

This article is contributed by **Anmol**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/given-n-appointments-find-conflicting-appointments/

Category: Trees

Post navigation

← Amazon Interview Experience | Set 158 (Off-Campus) Given a linked list of line segments, remove middle points →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 4

Check a given sentence for a given set of simple grammer rules

A simple sentence if syntactically correct if it fulfills given rules. The following are given rules.

1. Sentence must start with a Uppercase character (e.g. Noun/ I/ We/ He etc.)
2. Then lowercase character follows.
3. There must be spaces between words.
4. Then the sentence must end with a full stop(.) after a word.
5. Two continuous spaces are not allowed.
6. Two continuous upper case characters are not allowed.
7. However the sentence can end after an upper case character.

Examples:

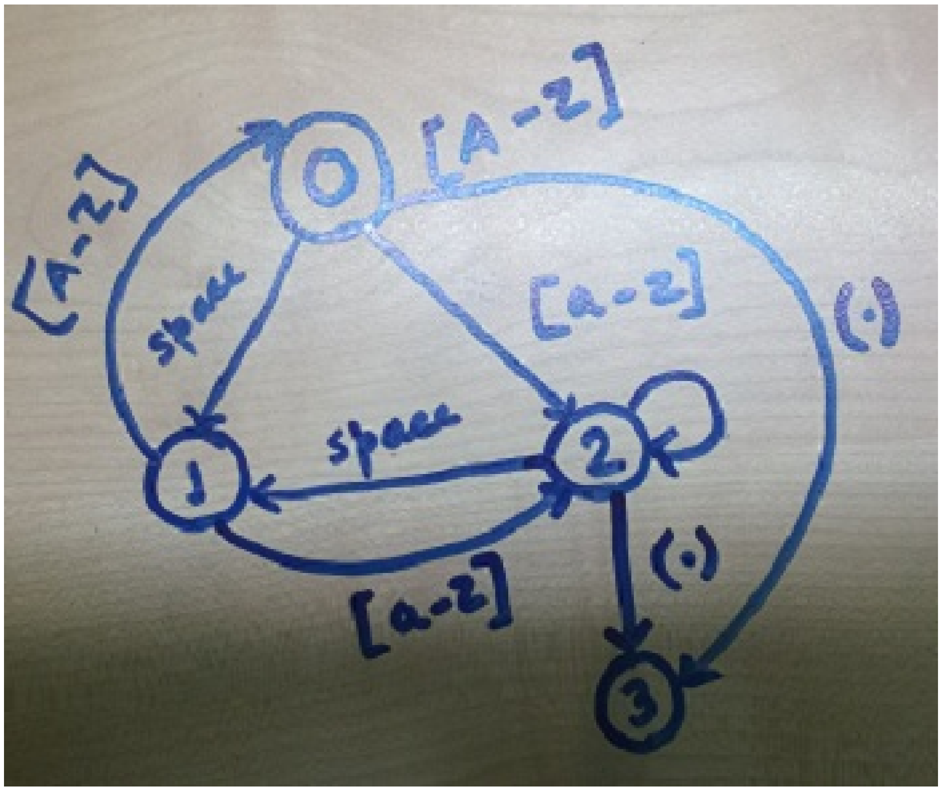
```
Correct sentences -
   "My name is Ram."
   "The vertex is S."
   "I am single."
   "I love Geeksquiz and Geeksforgeeks."

Incorrect sentence -
   "My name is KG."
   "I lovE cinema."
   "GeeksQuiz. is a quiz site."
   "  You are my friend."
   "I love cinema"
```

Question: Given a sentence, validate the given sentence for above given rules.

We strongly recommend to minimize the browser and try this yourself first.
The idea is to use an automata for the given set of rules.

Algorithm :
1. Check for the corner cases
…..1.a) Check if the first character is uppercase or not in the sentence.
…..1.b) Check if the last character is a full stop or not.

2. For rest of the string, this problem could be solved by following a state diagram. Please refer to the below state diagram for that.



3. We need to maintain previous and current state of different characters in the string. Based on that we can always validate the sentence of every character traversed.

A C based implementation is below. (By the way this sentence is also correct according to the rule and code)

```
// C program to validate a given sentence for a set of rules
#include<stdio.h>
#include<string.h>
#include<stdbool.h>

// Method to check a given sentence for given rules
bool checkSentence(char str[])
{
    // Calculate the length of the string.
    int len = strlen(str);

    // Check that the first character lies in [A-Z].
    // Otherwise return false.
    if (str[0] < 'A' || str[0] > 'Z')
        return false;

    //If the last character is not a full stop(.) no
    //need to check further.
    if (str[len - 1] != '.')
        return false;
    // Maintain 2 states. Previous and current state based
    // on which vertex state you are. Initialise both with
    // 0 = start state.
    int prev_state = 0, curr_state = 0;

    //Keep the index to the next character in the string.
    int index = 1;

    //Loop to go over the string.
    while (str[index])
    {
        // Set states according to the input characters in the
        // string and the rule defined in the description.
        // If current character is [A-Z]. Set current state as 0.
        if (str[index] >= 'A' && str[index] <= 'Z')
            curr_state = 0;

        // If current character is a space. Set current state as 1.
        else if (str[index] == ' ')
            curr_state = 1;

        // If current character is [a-z]. Set current state as 2.
        else if (str[index] >= 'a' && str[index] <= 'z')
            curr_state = 2;

        // If current state is a dot(.). Set current state as 3.
        else if (str[index] == '.')
            curr_state = 3;

        // Validates all current state with previous state for the
        // rules in the description of the problem.
        if (prev_state == curr_state && curr_state != 2)
            return false;

        if (prev_state == 2 && curr_state == 0)
            return false;

        // If we have reached last state and previous state is not 1,
        // then check next character. If next character is '\0', then
        // return true, else false
        if (curr_state == 3 && prev_state != 1)
            return (str[index + 1] == '\0');

        index++;

        // Set previous state as current state before going over
        // to the next character.
        prev_state = curr_state;
    }
    return false;
}

// Driver program
```

```
int main()
{
    char *str[] = { "I love cinema.", "The vertex is S.",
                    "I am single.", "My name is KG.",
                    "I lovE cinema.", "GeeksQuiz. is a quiz site.",
                    "I love Geeksquiz and Geeksforgeeks.",
                    "  You are my friend.", "I love cinema" };
    int str_size = sizeof(str) / sizeof(str[0]);
    int i = 0;
    for (i = 0; i < str_size; i++)
     checkSentence(str[i])? printf("\"%s\" is correct \n", str[i]):
                            printf("\"%s\" is incorrect \n", str[i]);

    return 0;
}
```

Output:

```
 "I love cinema." is correct
"The vertex is S." is correct
"I am single." is correct
"My name is KG." is incorrect
"I lovE cinema." is incorrect
"GeeksQuiz. is a quiz site." is incorrect
"I love Geeksquiz and Geeksforgeeks." is correct
"  You are my friend." is incorrect
"I love cinema" is incorrect
```

Time complexity – O(n), worst case as we have to traverse the full sentence where n is the length of the sentence.
Auxiliary space – O(1)

This article is contributed by **Kumar Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/check-given-sentence-given-set-simple-grammer-rules/

Category: Strings

Chapter 5

Find Index of 0 to be replaced with 1 to get longest continuous sequence of 1s in a binary array

Given an array of 0s and 1s, find the position of 0 to be replaced with 1 to get longest continuous sequence of 1s. Expected time complexity is O(n) and auxiliary space is O(1).
Example:

```
 Input:
   arr[] =  {1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1}
Output:
  Index 9
Assuming array index starts from 0, replacing 0 with 1 at index 9 causes
the maximum continuous sequence of 1s.

Input:
   arr[] =  {1, 1, 1, 1, 0}
Output:
  Index 4
```

We strongly recommend to minimize the browser and try this yourself first.

A **Simple Solution** is to traverse the array, for every 0, count the number of 1s on both sides of it. Keep track of maximum count for any 0. Finally return index of the 0 with maximum number of 1s around it. The time complexity of this solution is O(n^2).

Using an **Efficient Solution**, the problem can solved in O(n) time. The idea is to keep track of three indexes, current index (*curr*), previous zero index (*prev_zero*) and previous to previous zero index (*prev_prev_zero*). Traverse the array, if current element is 0, calculate the difference between *curr* and *prev_prev_zero* (This difference minus one is the number of 1s around the prev_zero). If the difference between *curr* and *prev_prev_zero* is more than maximum so far, then update the maximum. Finally return index of the prev_zero with maximum difference.

Following is C++ implementation of the above algorithm.

```
// C++ program to find Index of 0 to be replaced with 1 to get
// longest continuous sequence of 1s in a binary array
```

```
#include<iostream>
using namespace std;

// Returns index of 0 to be replaced with 1 to get longest
// continuous sequence of 1s.  If there is no 0 in array, then
// it returns -1.
int maxOnesIndex(bool arr[], int n)
{
    int max_count = 0;  // for maximum number of 1 around a zero
    int max_index;  // for storing result
    int prev_zero = -1;  // index of previous zero
    int prev_prev_zero = -1; // index of previous to previous zero

    // Traverse the input array
    for (int curr=0; curr<n; ++curr)
    {
        // If current element is 0, then calculate the difference
        // between curr and prev_prev_zero
        if (arr[curr] == 0)
        {
            // Update result if count of 1s around prev_zero is more
            if (curr - prev_prev_zero > max_count)
            {
                max_count = curr - prev_prev_zero;
                max_index = prev_zero;
            }

            // Update for next iteration
            prev_prev_zero = prev_zero;
            prev_zero = curr;
        }
    }

    // Check for the last encountered zero
    if (n-prev_prev_zero > max_count)
       max_index = prev_zero;

    return max_index;
}

// Driver program
int main()
{
    bool arr[] = {1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Index of 0 to be replaced is "
         << maxOnesIndex(arr, n);
    return 0;
}
```

Output:

```
 Index of 0 to be replaced is 9
```

Time Complexity: O(n)
Auxiliary Space: O(1)

This article is contributed by **Ankur Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-index-0-replaced-1-get-longest-continuous-sequence-1s-binary-array/

Category: Arrays

Post navigation

← Construct a Maximum Sum Linked List out of two Sorted Linked Lists having some Common nodes
Check a given sentence for a given set of simple grammer rules →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 6

How to check if two given sets are disjoint?

Given two sets represented by two arrays, how to check if the given two sets are disjoint or not? It may be assumed that the given arrays have no duplicates.

Difficulty Level: Rookie

```
Input: set1[] = {12, 34, 11, 9, 3}
       set2[] = {2, 1, 3, 5}
Output: Not Disjoint
3 is common in two sets.

Input: set1[] = {12, 34, 11, 9, 3}
       set2[] = {7, 2, 1, 5}
Output: Yes, Disjoint
There is no common element in two sets.
```

We strongly recommend to minimize your browser and try this yourself first.
There are plenty of methods to solve this problem, it's a good test to check how many solutions you can guess.

Method 1 (Simple)
Iterate through every element of first set and search it in other set, if any element is found, return false. If no element is found, return tree. Time complexity of this method is O(mn).

Following is C++ implementation of above idea.

```
// A Simple C++ program to check if two sets are disjoint
#include<iostream>
using namespace std;

// Returns true if set1[] and set2[] are disjoint, else false
bool areDisjoint(int set1[], int set2[], int m, int n)
{
    // Take every element of set1[] and search it in set2
    for (int i=0; i<m; i++)
      for (int j=0; j<n; j++)
         if (set1[i] == set2[j])
            return false;

    // If no element of set1 is present in set2
    return true;
}

// Driver program to test above function
int main()
{
    int set1[] = {12, 34, 11, 9, 3};
    int set2[] = {7, 2, 1, 5};
    int m = sizeof(set1)/sizeof(set1[0]);
    int n = sizeof(set2)/sizeof(set2[0]);
    areDisjoint(set1, set2, m, n)? cout << "Yes" : cout << " No";
    return 0;
}
```

Output:

```
 Yes
```

Method 2 (Use Sorting and Merging)
1) Sort first and second sets.
2) Use merge like process to compare elements.

Following is C++ implementation of above idea.

```
// A Simple C++ program to check if two sets are disjoint
#include<iostream>
#include<algorithm>
using namespace std;

// Returns true if set1[] and set2[] are disjoint, else false
bool areDisjoint(int set1[], int set2[], int m, int n)
{
    // Sort the given two sets
    sort(set1, set1+m);
    sort(set2, set2+n);

    // Check for same elements using merge like process
    int i = 0, j = 0;
    while (i < m && j < n)
    {
        if (set1[i] < set2[j])
            i++;
        else if (set2[j] < set1[i])
            j++;
        else /* if set1[i] == set2[j] */
            return false;
    }
    return true;
}

// Driver program to test above function
int main()
{
    int set1[] = {12, 34, 11, 9, 3};
    int set2[] = {7, 2, 1, 5};
    int m = sizeof(set1)/sizeof(set1[0]);
    int n = sizeof(set2)/sizeof(set2[0]);
    areDisjoint(set1, set2, m, n)? cout << "Yes" : cout << " No";
    return 0;
}
```

Output:

```
 Yes
```

Time complexity of above solution is O(mLogm + nLogn).

The above solution first sorts both sets, then takes O(m+n) time to find intersection. If we are given that the input sets are sorted, then this method is best among all.

Method 3 (Use Sorting and Binary Search)
This is similar to method 1. Instead of linear search, we use Binary Search.
1) Sort first set.
2) Iterate through every element of second set, and use binary search to search every element in first set. If element is found return it.

Time complexity of this method is O(mLogm + nLogm)

Method 4 (Use Binary Search Tree)
1) Create a self balancing binary search tree (Red Black, AVL, Splay, etc) of all elements in first set.
2) Iterate through all elements of second set and search every element in the above constructed Binary Search Tree. If element is found, return false.
3) If all elements are absent, return true.

Time complexity of this method is O(mLogm + nLogm).

Method 5 (Use Hashing)
1) Create an empty hash table.
2) Iterate through the first set and store every element in hash table.
3) Iterate through second set and check if any element is present in hash table. If present, then return false, else ignore the element.
4) If all elements of second set are not present in hash table, return true.

Following is Java implementation of this method.

```
/* Java program to check if two sets are distinct or not */
import java.util.*;

class Main
{
    // This function prints all distinct elements
    static boolean areDisjoint(int set1[], int set2[])
    {
        // Creates an empty hashset
        HashSet<Integer> set = new HashSet<>();

        // Traverse the first set and store its elements in hash
        for (int i=0; i<set1.length; i++)
            set.add(set1[i]);

        // Traverse the second set and check if any element of it
        // is already in hash or not.
        for (int i=0; i<set2.length; i++)
            if (set.contains(set2[i]))
                return false;

        return true;
    }

    // Driver method to test above method
    public static void main (String[] args)
    {
        int set1[] = {10, 5, 3, 4, 6};
        int set2[] = {8, 7, 9, 3};
        if (areDisjoint(set1, set2)
            System.out.println("Yes");
        else
            System.out.println("No");
    }
}
```

Output:

```
Yes
```

Time complexity of the above implementation is O(m+n) under the assumption that hash set operations like add() and contains() work in O(1) time.

This article is contributed by **Rajeev**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/check-two-given-sets-disjoint/

Category: Arrays

Post navigation

← Amazon Interview Experience | Set 149 (On-Campus for Internship) Amazon Interview Experience | Set 150 (SDE1 for 1 Year Experienced) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 7

Minimum Number of Platforms Required for a Railway/Bus Station

Given arrival and departure times of all trains that reach a railway station, find the minimum number of platforms required for the railway station so that no train waits.
We are given two arrays which represent arrival and departure times of trains that stop

Examples:

```
Input:  arr[]  = {9:00,  9:40, 9:50,  11:00, 15:00, 18:00}
        dep[]  = {9:10, 12:00, 11:20, 11:30, 19:00, 20:00}
Output: 3
There are at-most three trains at a time (time between 11:00 to 11:20)
```

We strongly recommend to minimize your browser and try this yourself first.
We need to find the maximum number of trains that are there on the given railway station at a time. A **Simple Solution** is to take every interval one by one and find the number of intervals that overlap with it. Keep track of maximum number of intervals that overlap with an interval. Finally return the maximum value. Time Complexity of this solution is O(n^2).

We can solve the above problem **in O(nLogn) time**. The idea is to consider all evens in sorted order. Once we have all events in sorted order, we can trace the number of trains at any time keeping track of trains that have arrived, but not departed.

For example consider the above example.

```
    arr[]  = {9:00,  9:40, 9:50,  11:00, 15:00, 18:00}
    dep[]  = {9:10, 12:00, 11:20, 11:30, 19:00, 20:00}

All events sorted by time.
Total platforms at any time can be obtained by subtracting total
departures from total arrivals by that time.
 Time     Event Type     Total Platforms Needed at this Time
 9:00       Arrival                  1
 9:10       Departure                0
 9:40       Arrival                  1
 9:50       Arrival                 2
 11:00      Arrival                 3
 11:20      Departure               2
 11:30      Departure               1
 12:00      Departure               0
 15:00      Arrival                 1
 18:00      Arrival                 2
 19:00      Departure               1
 20:00      Departure               0

Minimum Platforms needed on railway station = Maximum platforms
                                              needed at any time
                                           = 3
```

Following is C++ implementation of above approach. Note that the implementation doesn't create a single sorted list of all events, rather it individually sorts arr[] and dep[] arrays, and then uses merge process of merge sort to process them together as a single sorted array.

```
// Program to find minimum number of platforms required on a railway station
#include<iostream>
#include<algorithm>
using namespace std;

// Returns minimum number of platforms reqquired
int findPlatform(int arr[], int dep[], int n)
{
   // Sort arrival and departure arrays
   sort(arr, arr+n);
   sort(dep, dep+n);

   // plat_needed indicates number of platforms needed at a time
   int plat_needed = 1, result = 1;
   int i = 1, j = 0;

   // Similar to merge in merge sort to process all events in sorted order
   while (i < n && j < n)
   {
      // If next event in sorted order is arrival, increment count of
      // platforms needed
      if (arr[i] < dep[j])
      {
          plat_needed++;
          i++;
          if (plat_needed > result)  // Update result if needed
              result = plat_needed;
      }
      else // Else decrement count of platforms needed
      {
          plat_needed--;
          j++;
      }
   }
   return result;
}

// Driver program to test methods of graph class
int main()
{
    int arr[] = {900, 940, 950, 1100, 1500, 1800};
    int dep[] = {910, 1200, 1120, 1130, 1900, 2000};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Minimum Number of Platforms Required = "
         << findPlatform(arr, dep, n);
    return 0;
}
```

Output:

```
 Minimum Number of Platforms Required = 3
```

Algorithmic Paradigm: Dynamic Programming

Time Complexity: O(nLogn), assuming that a O(nLogn) sorting algorithm for sorting arr[] and dep[].

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/minimum-number-platforms-required-railwaybus-station/

Category: Arrays Tags: Greedy Algorithm

Post navigation

← SapientNitro Interview Experience | Set 2 (On-Campus) InfoEdge Interview Experience →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 8

Length of the largest subarray with contiguous elements | Set 1

Given an array of distinct integers, find length of the longest subarray which contains numbers that can be arranged in a continuous sequence.

Examples:

```
Input:  arr[] = {10, 12, 11};
Output: Length of the longest contiguous subarray is 3

Input:  arr[] = {14, 12, 11, 20};
Output: Length of the longest contiguous subarray is 2

Input:  arr[] = {1, 56, 58, 57, 90, 92, 94, 93, 91, 45};
Output: Length of the longest contiguous subarray is 5
```

We strongly recommend to minimize the browser and try this yourself first.

The important thing to note in question is, it is given that all elements are distinct. If all elements are distinct, then a subarray has contiguous elements if and only if the difference between maximum and minimum elements in subarray is equal to the difference between last and first indexes of subarray. So the idea is to keep track of minimum and maximum element in every subarray.

The following is C++ implementation of above idea.

```
#include<iostream>
using namespace std;

// Utility functions to find minimum and maximum of
// two elements
int min(int x, int y) { return (x < y)? x : y; }
int max(int x, int y) { return (x > y)? x : y; }

// Returns length of the longest contiguous subarray
int findLength(int arr[], int n)
```

```
{
    int max_len = 1;  // Initialize result
    for (int i=0; i<n-1; i++)
    {
        // Initialize min and max for all subarrays starting with i
        int mn = arr[i], mx = arr[i];

        // Consider all subarrays starting with i and ending with j
        for (int j=i+1; j<n; j++)
        {
            // Update min and max in this subarray if needed
            mn = min(mn, arr[j]);
            mx = max(mx, arr[j]);

            // If current subarray has all contiguous elements
            if ((mx - mn) == j-i)
                max_len = max(max_len, mx-mn+1);
        }
    }
    return max_len;  // Return result
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 56, 58, 57, 90, 92, 94, 93, 91, 45};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Length of the longest contiguous subarray is "
         << findLength(arr, n);
    return 0;
}
```

Output:

```
 Length of the longest contiguous subarray is 5
```

Time Complexity of the above solution is O(n^2).

We will soon be covering solution for the problem where duplicate elements are allowed in subarray.

This article is contributed by **Arjun**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/length-largest-subarray-contiguous-elements-set-1/

Category: Arrays

Chapter 9

Length of the largest subarray with contiguous elements | Set 2

Given an array of integers, find length of the longest subarray which contains numbers that can be arranged in a continuous sequence.

In theprevious post, we have discussed a solution that assumes that elements in given array are distinct. Here we discuss a solution that works even if the input array has duplicates.

Examples:

```
Input:  arr[] = cc
Output: Length of the longest contiguous subarray is 4

Input:  arr[] = {10, 12, 12, 10, 10, 11, 10};
Output: Length of the longest contiguous subarray is 2
```

We strongly recommend to minimize the browser and try this yourself first.

The idea is similar to previous post. In the previous post, we checked whether maximum value minus minimum value is equal to ending index minus starting index or not. Since duplicate elements are allowed, we also need to check if the subarray contains duplicate elements or not. For example, the array {12, 14, 12} follows the first property, but numbers in it are not contiguous elements.
To check duplicate elements in a subarray, we create a hash set for every subarray and if we find an element already in hash, we don't consider the current subarray.

Following is Java implementation of the above idea.

```
/* Java program to find length of the largest subarray which has
   all contiguous elements */
import java.util.*;

class Main
{
    // This function prints all distinct elements
    static int findLength(int arr[])
    {
        int n = arr.length;
        int max_len = 1; // Inialize result

        // One by one fix the starting points
        for (int i=0; i<n-1; i++)
        {
            // Create an empty hash set and add i'th element
            // to it.
            HashSet<Integer> set = new HashSet<>();
            set.add(arr[i]);

            // Initialize max and min in current subarray
            int mn = arr[i], mx = arr[i];

            // One by one fix ending points
            for (int j=i+1; j<n; j++)
            {
                // If current element is already in hash set, then
                // this subarray cannot contain contiguous elements
                if (set.contains(arr[j]))
                    break;

                // Else add curremt element to hash set and update
                // min, max if required.
                set.add(arr[j]);
                mn = Math.min(mn, arr[j]);
                mx = Math.max(mx, arr[j]);

                // We have already cheched for duplicates, now check
                // for other property and update max_len if needed
                if (mx-mn == j-i)
                    max_len = Math.max(max_len, mx-mn+1);
            }
        }
        return max_len; // Return result
    }

    // Driver method to test above method
    public static void main (String[] args)
    {
       int arr[] =  {10, 12, 12, 10, 10, 11, 10};
       System.out.println("Length of the longest contiguous subarray is " +
                           findLength(arr));
    }
}
```

Output:

```
 Length of the longest contiguous subarray is 2
```

Time complexity of the above solution is O(n^2) under the assumption that hash set operations like add() and contains() work in O(1) time.

This article is contributed by **Arjun**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/length-largest-subarray-contiguous-elements-set-2/

Category: Interview Experiences Tags: Hashing

Chapter 10

Print all increasing sequences of length k from first n natural numbers

Given two positive integers n and k, print all increasing sequences of length k such that the elements in every sequence are from first n natural numbers.

Examples:

```
Input: k = 2, n = 3
Output: 1 2
        1 3
        2 3

Input: k = 5, n = 5
Output: 1 2 3 4 5

Input: k = 3, n = 5
Output: 1 2 3
        1 2 4
        1 2 5
        1 3 4
        1 3 5
        1 4 5
        2 3 4
        2 3 5
        2 4 5
        3 4 5
```

We strongly recommend to minimize the browser and try this yourself first.

It's a good recursion question. The idea is to create an array of length k. The array stores current sequence. For every position in array, we check the previous element and one by one put all elements greater than the previous element. If there is no previous element (first position), we put all numbers from 1 to n.

Following is C++ implementation of above idea.

```
// C++ program to  print all increasing sequences of
// length 'k' such that the elements in every sequence
// are from first 'n' natural numbers.
#include<iostream>
using namespace std;

// A utility function to print contents of arr[0..k-1]
void printArr(int arr[], int k)
{
    for (int i=0; i<k; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// A recursive function to print all increasing sequences
// of first n natural numbers.  Every sequence should be
// length k. The array arr[] is used to store current
// sequence.
void printSeqUtil(int n, int k, int &len, int arr[])
{
    // If length of current increasing sequence becomes k,
    // print it
    if (len == k)
    {
        printArr(arr, k);
        return;
    }

    // Decide the starting number to put at current position:
    // If length is 0, then there are no previous elements
    // in arr[].  So start putting new numbers with 1.
    // If length is not 0, then start from value of
    // previous element plus 1.
    int i = (len == 0)? 1 : arr[len-1] + 1;

    // Increase length
    len++;

    // Put all numbers (which are greater than the previous
    // element) at new position.
    while (i<=n)
    {
        arr[len-1] = i;
        printSeqUtil(n, k, len, arr);
        i++;
    }

    // This is important. The variable 'len' is shared among
    // all function calls in recursion tree. Its value must be
    // brought back before next iteration of while loop
    len--;
}

// This function prints all increasing sequences of
// first n natural numbers. The length of every sequence
// must be k.  This function mainly uses printSeqUtil()
void printSeq(int n, int k)
{
    int arr[k];  // An array to store individual sequences
    int len = 0; // Initial length of current sequence
    printSeqUtil(n, k, len, arr);
}

// Driver program to test above functions
int main()
{
    int k = 3, n = 7;
    printSeq(n, k);
    return 0;
}
```

Output:

```
 1 2 3
1 2 4
1 2 5
1 2 6
1 2 7
1 3 4
1 3 5
1 3 6
1 3 7
1 4 5
1 4 6
1 4 7
1 5 6
1 5 7
1 6 7
2 3 4
2 3 5
2 3 6
2 3 7
2 4 5
2 4 6
2 4 7
2 5 6
2 5 7
2 6 7
3 4 5
3 4 6
3 4 7
3 5 6
3 5 7
3 6 7
4 5 6
4 5 7
4 6 7
5 6 7
```

This article is contributed by **Arjun**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/print-increasing-sequences-length-k-first-n-natural-numbers/

Category: Misc Tags: Recursion

Chapter 11

Given two strings, find if first string is a subsequence of second

Given two strings str1 and str2, find if str1 is a subsequence of str2. A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements (source: wiki). Expected time complexity is linear.

Examples:

```
Input: str1 = "AXY", str2 = "ADXCPY"
Output: True (str1 is a subsequence of str2)

Input: str1 = "AXY", str2 = "YADXCP"
Output: False (str1 is not a subsequence of str2)

Input: str1 = "gksrek", str2 = "geeksforgeeks"
Output: True (str1 is a subsequence of str2)
```

We strongly recommend to minimize the browser and try this yourself first.

The idea is simple, we traverse both strings from one side to other side (say from rightmost character to leftmost). If we find a matching character, we move ahead in both strings. Otherwise we move ahead only in str2.

Following is **Recursive Implementation** in C++ of the above idea.

```
// Recursive C++ program to check if a string is subsequence of another string
#include<iostream>
#include<cstring>
using namespace std;

// Returns true if str1[] is a subsequence of str2[]. m is
// length of str1 and n is length of str2
bool isSubSequence(char str1[], char str2[], int m, int n)
{
    // Base Cases
    if (m == 0) return true;
    if (n == 0) return false;

    // If last characters of two strings are matching
    if (str1[m-1] == str2[n-1])
        return isSubSequence(str1, str2, m-1, n-1);

    // If last characters are not matching
    return isSubSequence(str1, str2, m, n-1);
}

// Driver program to test methods of graph class
int main()
{
    char str1[] = "gksrek";
    char str2[] = "geeksforgeeks";
    int m = strlen(str1);
    int n = strlen(str2);
    isSubSequence(str1, str2, m, n)? cout << "Yes ":
                                     cout << "No";
    return 0;
}
```

Output:

```
 Yes
```

Following is **Iterative Implementation** in C++ for the same.

```
// Iterative C++ program to check if a string is subsequence of another string
#include<iostream>
#include<cstring>
using namespace std;

// Returns true if str1[] is a subsequence of str2[]. m is
// length of str1 and n is length of str2
bool isSubSequence(char str1[], char str2[], int m, int n)
{
   int j = 0; // For index of str1 (or subsequence

   // Traverse str2 and str1, and compare current character
   // of str2 with first unmatched char of str1, if matched
   // then move ahead in str1
   for (int i=0; i<n&&j<m; i++)
       if (str1[j] == str2[i])
         j++;

   // If all characters of str1 were found in str2
   return (j==m);
}
```

```
// Driver program to test methods of graph class
int main()
{
    char str1[] = "gksrek";
    char str2[] = "geeksforgeeks";
    int m = strlen(str1);
    int n = strlen(str2);
    isSubSequence(str1, str2, m, n)? cout << "Yes ":
                                     cout << "No";
    return 0;
}
```

Output:

```
 Yes
```

Time Complexity of both implementations above is O(n) where n is the length of str2.

This article is contributed by **Sachin Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/given-two-strings-find-first-string-subsequence-second/

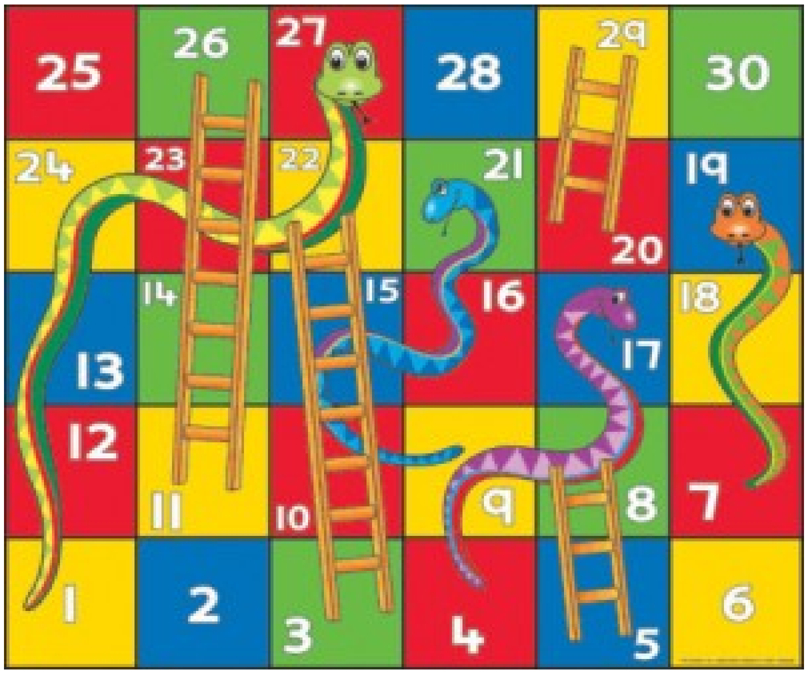
Category: Strings

Chapter 12

Snake and Ladder Problem

Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from source or 1st cell. Basically, the player has total control over outcome of dice throw and wants to find out minimum number of throws required to reach last cell.

If the player reaches a cell which is base of a ladder, the player has to climb up that ladder and if reaches a cell is mouth of the snake, has to go down to the tail of snake without a dice throw.



For example consider the board shown on right side (taken from here), the minimum number of dice throws required to reach cell 30 from cell 1 is 3. Following are steps.

a) First throw two on dice to reach cell number 3 and then ladder to reach 22
b) Then throw 6 to reach 28.
c) Finally through 2 to reach 30.

There can be other solutions as well like (2, 2, 6), (2, 4, 4), (2, 3, 5).. etc.

We strongly recommend to minimize the browser and try this yourself first.
The idea is to consider the given snake and ladder board as a directed graph with number of vertices equal to the number of cells in the board. The problem reduces to finding the shortest path in a graph. Every

vertex of the graph has an edge to next six vertices if next 6 vertices do not have a snake or ladder. If any of the next six vertices has a snake or ladder, then the edge from current vertex goes to the top of the ladder or tail of the snake. Since all edges are of equal weight, we can efficiently find shortest path using Breadth First Search of the graph.

Following is C++ implementation of the above idea. The input is represented by two things, first is 'N' which is number of cells in the given board, second is an array 'move[0…N-1]' of size N. An entry move[i] is -1 if there is no snake and no ladder from i, otherwise move[i] contains index of destination cell for the snake or the ladder at i.

```
// C++ program to find minimum number of dice throws required to
// reach last cell from first cell of a given snake and ladder
// board
#include<iostream>
#include <queue>
using namespace std;

// An entry in queue used in BFS
struct queueEntry
{
    int v;     // Vertex number
    int dist;  // Distance of this vertex from source
};

// This function returns minimum number of dice throws required to
// Reach last cell from 0'th cell in a snake and ladder game.
// move[] is an array of size N where N is no. of cells on board
// If there is no snake or ladder from cell i, then move[i] is -1
// Otherwise move[i] contains cell to which snake or ladder at i
// takes to.
int getMinDiceThrows(int move[], int N)
{
    // The graph has N vertices. Mark all the vertices as
    // not visited
    bool *visited = new bool[N];
    for (int i = 0; i < N; i++)
        visited[i] = false;

    // Create a queue for BFS
    queue<queueEntry> q;

    // Mark the node 0 as visited and enqueue it.
    visited[0] = true;
    queueEntry s = {0, 0};  // distance of 0't vertex is also 0
    q.push(s);  // Enqueue 0'th vertex

    // Do a BFS starting from vertex at index 0
    queueEntry qe;  // A queue entry (qe)
    while (!q.empty())
    {
        qe = q.front();
        int v = qe.v; // vertex no. of queue entry

        // If front vertex is the destination vertex,
        // we are done
        if (v == N-1)
            break;

        // Otherwise dequeue the front vertex and enqueue
        // its adjacent vertices (or cell numbers reachable
        // through a dice throw)
        q.pop();
        for (int j=v+1; j<=(v+6) && j<N; ++j)
        {
            // If this cell is already visited, then ignore
            if (!visited[j])
            {
                // Otherwise calculate its distance and mark it
                // as visited
                queueEntry a;
                a.dist = (qe.dist + 1);
                visited[j] = true;

                // Check if there a snake or ladder at 'j'
                // then tail of snake or top of ladder
                // become the adjacent of 'i'
                if (move[j] != -1)
                    a.v = move[j];
                else
                    a.v = j;
                q.push(a);
            }
        }
    }

    // We reach here when 'qe' has last vertex
    // return the distance of vertex in 'qe'
    return qe.dist;
}

// Driver program to test methods of graph class
int main()
{
    // Let us construct the board given in above diagram
    int N = 30;
    int moves[N];
    for (int i = 0; i<N; i++)
        moves[i] = -1;

    // Ladders
    moves[2] = 21;
    moves[4] = 7;
    moves[10] = 25;
    moves[19] = 28;

    // Snakes
    moves[26] = 0;
    moves[20] = 8;
    moves[16] = 3;
    moves[18] = 6;

    cout << "Min Dice throws required is " << getMinDiceThrows(moves, N);
    return 0;
}
```

Output:

```
 Min Dice throws required is 3
```

Time complexity of the above solution is O(N) as every cell is added and removed only once from queue. And a typical enqueue or dequeue operation takes O(1) time.

This article is contributed by **Siddharth**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/snake-ladder-problem-2/

Chapter 13

Connect n ropes with minimum cost

There are given n ropes of different lengths, we need to connect these ropes into one rope. The cost to connect two ropes is equal to sum of their lengths. We need to connect the ropes with minimum cost.

For example if we are given 4 ropes of lengths 4, 3, 2 and 6. We can connect the ropes in following ways.
1) First connect ropes of lengths 2 and 3. Now we have three ropes of lengths 4, 6 and 5.
2) Now connect ropes of lengths 4 and 5. Now we have two ropes of lengths 6 and 9.
3) Finally connect the two ropes and all ropes have connected.

Total cost for connecting all ropes is 5 + 9 + 15 = 29. This is the optimized cost for connecting ropes. Other ways of connecting ropes would always have same or more cost. For example, if we connect 4 and 6 first (we get three strings of 3, 2 and 10), then connect 10 and 3 (we get two strings of 13 and 2). Finally we connect 13 and 2. Total cost in this way is 10 + 13 + 15 = 38.

We strongly recommend to minimize the browser and try this yourself first.
If we observe the above problem closely, we can notice that the lengths of the ropes which are picked first are included more than once in total cost. Therefore, the idea is to connect smallest two ropes first and recur for remaining ropes. This approach is similar to Huffman Coding. We put smallest ropes down the tree so that they can be repeated multiple times rather than the longer ropes.

Following is complete algorithm for finding the minimum cost for connecting n ropes.
Let there be n ropes of lengths stored in an array len[0..n-1]
1) Create a min heap and insert all lengths into the min heap.
2) Do following while number of elements in min heap is not one.
……a) Extract the minimum and second minimum from min heap
……b) Add the above two extracted values and insert the added value to the min-heap.
3) Return the value of only left item in min heap.

Following is C++ implementation of above algorithm.

```
// C++ program for connecting n ropes with minimum cost
#include <iostream>
using namespace std;

// A Min Heap:  Collection of min heap nodes
struct MinHeap
{
    unsigned size;    // Current size of min heap
    unsigned capacity;   // capacity of min heap
    int *harr;  // Attay of minheap nodes
```

```
};

// A utility function to create a min heap of given capacity
struct MinHeap* createMinHeap(unsigned capacity)
{
    struct MinHeap* minHeap = new MinHeap;
    minHeap->size = 0;  // current size is 0
    minHeap->capacity = capacity;
    minHeap->harr = new int[capacity];
    return minHeap;
}

// A utility function to swap two min heap nodes
void swapMinHeapNode(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// The standard minHeapify function.
void minHeapify(struct MinHeap* minHeap, int idx)
{
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->size &&
        minHeap->harr[left] < minHeap->harr[smallest])
      smallest = left;

    if (right < minHeap->size &&
        minHeap->harr[right] < minHeap->harr[smallest])
      smallest = right;

    if (smallest != idx)
    {
        swapMinHeapNode(&minHeap->harr[smallest], &minHeap->harr[idx]);
        minHeapify(minHeap, smallest);
    }
}

// A utility function to check if size of heap is 1 or not
int isSizeOne(struct MinHeap* minHeap)
{
    return (minHeap->size == 1);
}

// A standard function to extract minimum value node from heap
int extractMin(struct MinHeap* minHeap)
{
    int temp = minHeap->harr[0];
    minHeap->harr[0] = minHeap->harr[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);
    return temp;
}

// A utility function to insert a new node to Min Heap
void insertMinHeap(struct MinHeap* minHeap, int val)
{
    ++minHeap->size;
    int i = minHeap->size - 1;
    while (i && (val < minHeap->harr[(i - 1)/2]))
    {
        minHeap->harr[i] = minHeap->harr[(i - 1)/2];
        i = (i - 1)/2;
    }
    minHeap->harr[i] = val;
}

// A standard funvtion to build min heap
void buildMinHeap(struct MinHeap* minHeap)
{
    int n = minHeap->size - 1;
    int i;
    for (i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}

// Creates a min heap of capacity equal to size and inserts all values
// from len[] in it. Initially size of min heap is equal to capacity
struct MinHeap* createAndBuildMinHeap(int len[], int size)
{
    struct MinHeap* minHeap = createMinHeap(size);
    for (int i = 0; i < size; ++i)
        minHeap->harr[i] = len[i];
    minHeap->size = size;
    buildMinHeap(minHeap);
    return minHeap;
}

// The main function that returns the minimum cost to connect n ropes of
// lengths stored in len[0..n-1]
int minCost(int len[], int n)
{
    int cost = 0;  // Initialize result

    // Create a min heap of capacity equal to n and put all ropes in it
    struct MinHeap* minHeap = createAndBuildMinHeap(len, n);

    // Iterate while size of heap doesn't become 1
    while (!isSizeOne(minHeap))
    {
        // Extract two minimum length ropes from min heap
        int min     = extractMin(minHeap);
        int sec_min = extractMin(minHeap);
        cost += (min + sec_min);  // Update total cost

        // Insert a new rope in min heap with length equal to sum
        // of two extracted minimum lengths
        insertMinHeap(minHeap, min+sec_min);
    }

    // Finally return total minimum cost for connecting all ropes
    return cost;
}

// Driver program to test above functions
int main()
{
    int len[] = {4, 3, 2, 6};
    int size = sizeof(len)/sizeof(len[0]);
    cout << "Total cost for connecting ropes is " << minCost(len, size);
    return 0;
}
```

Output:

```
 Total cost for connecting ropes is 29
```

Time Complexity: Time complexity of the algorithm is O(nLogn) assuming that we use a O(nLogn) sorting algorithm. Note that heap operations like insert and extract take O(Logn) time.

Algorithmic Paradigm: Greedy Algorithm

A simple implementation with STL in C++
Following is a simple implementation that uses priority_queue available in STL. Thanks to Pango89 for providing below code.

```
#include<iostream>
#include<queue>
using namespace std;

int minCost(int arr[], int n)
{
   // Create a priority queue ( http://www.cplusplus.com/reference/queue/priority_queue/ )
    // By default 'less' is used which is for decreasing order
    // and 'greater' is used for increasing order
    priority_queue< int, vector<int>, greater<int> > pq(arr, arr+n);

    // Initialize result
    int res = 0;

    // While size of priority queue is more than 1
    while (pq.size() > 1)
    {
        // Extract shortest two ropes from pq
        int first = pq.top();
        pq.pop();
        int second = pq.top();
        pq.pop();

        // Connect the ropes: update result and
        // insert the new rope to pq
        res += first + second;
        pq.push(first + second);
    }

    return res;
}

// Driver program to test above function
int main()
{
    int len[] = {4, 3, 2, 6};
    int size = sizeof(len)/sizeof(len[0]);
    cout << "Total cost for connecting ropes is " << minCost(len, size);
    return 0;
}
```

Output:

```
 Total cost for connecting ropes is 29
```

This article is compiled by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/connect-n-ropes-minimum-cost/

Chapter 14

Flood fill Algorithm - how to implement fill() in paint?

In MS-Paint, when we take the brush to a pixel and click, the color of the region of that pixel is replaced with a new selected color. Following is the problem statement to do this task.
Given a 2D screen, location of a pixel in the screen and a color, replace color of the given pixel and all adjacent same colored pixels with the given color.

Example:

```
Input:
       screen[M][N] = {{1, 1, 1, 1, 1, 1, 1, 1},
                      {1, 1, 1, 1, 1, 1, 0, 0},
                      {1, 0, 0, 1, 1, 0, 1, 1},
                      {1, 2, 2, 2, 2, 0, 1, 0},
                      {1, 1, 1, 2, 2, 0, 1, 0},
                      {1, 1, 1, 2, 2, 2, 2, 0},
                      {1, 1, 1, 1, 1, 2, 1, 1},
                      {1, 1, 1, 1, 1, 2, 2, 1},
                      };
    x = 4, y = 4, newColor = 3
The values in the given 2D screen indicate colors of the pixels.
x and y are coordinates of the brush, newColor is the color that
should replace the previous color on screen[x][y] and all surrounding
pixels with same color.

Output:
Screen should be changed to following.
       screen[M][N] = {{1, 1, 1, 1, 1, 1, 1, 1},
                      {1, 1, 1, 1, 1, 1, 0, 0},
                      {1, 0, 0, 1, 1, 0, 1, 1},
                      {1, 3, 3, 3, 3, 0, 1, 0},
                      {1, 1, 1, 3, 3, 0, 1, 0},
                      {1, 1, 1, 3, 3, 3, 3, 0},
                      {1, 1, 1, 1, 1, 3, 1, 1},
                      {1, 1, 1, 1, 1, 3, 3, 1},
                      };
```

Flood Fill Algorithm:
The idea is simple, we first replace the color of current pixel, then recur for 4 surrounding points. The following is detailed algorithm.

```
// A recursive function to replace previous color 'prevC' at  '(x, y)'
// and all surrounding pixels of (x, y) with new color 'newC' and
floodFil(screen[M][N], x, y, prevC, newC)
1) If x or y is outside the screen, then return.
2) If color of screen[x][y] is not same as prevC, then return
3) Recur for north, south, east and west.
    floodFillUtil(screen, x+1, y, prevC, newC);
    floodFillUtil(screen, x-1, y, prevC, newC);
    floodFillUtil(screen, x, y+1, prevC, newC);
    floodFillUtil(screen, x, y-1, prevC, newC);
```

The following is C++ implementation of above algorithm.

```
// A C++ program to implement flood fill algorithm
#include<iostream>
using namespace std;

// Dimentions of paint screen
#define M 8
#define N 8

// A recursive function to replace previous color 'prevC' at  '(x, y)'
// and all surrounding pixels of (x, y) with new color 'newC' and
void floodFillUtil(int screen[][N], int x, int y, int prevC, int newC)
{
    // Base cases
    if (x < 0 || x >= M || y < 0 || y >= N)
        return;
    if (screen[x][y] != prevC)
        return;

    // Replace the color at (x, y)
    screen[x][y] = newC;

    // Recur for north, east, south and west
    floodFillUtil(screen, x+1, y, prevC, newC);
    floodFillUtil(screen, x-1, y, prevC, newC);
    floodFillUtil(screen, x, y+1, prevC, newC);
    floodFillUtil(screen, x, y-1, prevC, newC);
}

// It mainly finds the previous color on (x, y) and
// calls floodFillUtil()
void floodFill(int screen[][N], int x, int y, int newC)
{
    int prevC = screen[x][y];
    floodFillUtil(screen, x, y, prevC, newC);
}

// Driver program to test above function
int main()
{
    int screen[M][N] = {{1, 1, 1, 1, 1, 1, 1, 1},
                      {1, 1, 1, 1, 1, 1, 0, 0},
                      {1, 0, 0, 1, 1, 0, 1, 1},
                      {1, 2, 2, 2, 2, 0, 1, 0},
                      {1, 1, 1, 2, 2, 0, 1, 0},
                      {1, 1, 1, 2, 2, 2, 2, 0},
                      {1, 1, 1, 1, 1, 2, 1, 1},
                      {1, 1, 1, 1, 1, 2, 2, 1},
                     };
    int x = 4, y = 4, newC = 3;
    floodFill(screen, x, y, newC);

    cout << "Updated screen after call to floodFill: \n";
    for (int i=0; i<M; i++)
    {
        for (int j=0; j<N; j++)
           cout << screen[i][j] << " ";
        cout << endl;
    }
}
```

Output:

```
 Updated screen after call to floodFill:
1 1 1 1 1 1 1 1
1 1 1 1 1 1 0 0
1 0 0 1 1 0 1 1
1 3 3 3 3 0 1 0
1 1 1 3 3 0 1 0
1 1 1 3 3 3 3 0
1 1 1 1 1 3 1 1
1 1 1 1 1 3 3 1
```

References:
http://en.wikipedia.org/wiki/Flood_fill

This article is contributed by **Anmol**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/flood-fill-algorithm-implement-fill-paint/

Category: Arrays

Post navigation

← Bottom View of a Binary Tree Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 1 →