# 1. Write a C++ program to calculate the product of two positive integers represented as strings. Return the result as a string.

Sample Input: sn1 = "12"

sn2 = "5"

Sample Output: 12 X 5 = 60

Sample Input: sn1 = "48"

sn2 = "85"

Sample Output: 48 X 85 = 4080

#include <iostream>

#include <algorithm>

#include <vector>

#include <functional>

using namespace std;

string multiply(string sn1, string sn2) {

const auto char\_to\_int = [](const char c) { return c - '0'; };

const auto int\_to\_char = [](const int i) { return i + '0'; };

vector<int> n1;

transform(sn1.rbegin(), sn1.rend(), back\_inserter(n1), char\_to\_int);

vector<int> n2;

transform(sn2.rbegin(), sn2.rend(), back\_inserter(n2), char\_to\_int);

vector<int> temp(n1.size() + n2.size());

for(int i = 0; i < n1.size(); ++i) {

for(int j = 0; j < n2.size(); ++j) {

temp[i + j] += n1[i] \* n2[j];

temp[i + j + 1] += temp[i + j] / 10;

temp[i + j] %= 10;

}

}

string result;

transform(find\_if(temp.rbegin(), prev(temp.rend()),

[](const int i) { return i != 0; }),

temp.rend(), back\_inserter(result), int\_to\_char);

return result;

}

int main()

{

string sn1 = "12";

string sn2 = "5";

cout << sn1 <<" X " << sn2 << " = " << multiply(sn1, sn2) << endl;

sn1 = "48";

sn2 = "85";

cout << sn1 <<" X " << sn2 << " = " << multiply(sn1, sn2) << endl;

return 0;

}

# 2. Write a program in C++ to produce a square matrix with 0's down the main diagonal, 1's in the entries just above and below the main diagonal, 2's above and below that, etc.

0 1 2 3 4

1 0 1 2 3

2 1 0 1 2

3 2 1 0 1

4 3 2 1 0

Sample Output:

Input number or rows: 8

0 1 2 3 4 5 6 7

1 0 1 2 3 4 5 6

2 1 0 1 2 3 4 5

3 2 1 0 1 2 3 4

4 3 2 1 0 1 2 3

5 4 3 2 1 0 1 2

6 5 4 3 2 1 0 1

7 6 5 4 3 2 1 0

#include <iostream>

using namespace std;

int main()

{

int n, i, j, k, m = 0;

cout << "\n\n Print patern........:\n";

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

cout << " Input number or rows: ";

cin >> n;

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

if (i == 1) {

for (j = 1; j <= i; j++) {

cout << m << " ";

}

for (k = 1; k <= n - i; k++) {

cout << k << " ";

}

}

else {

for (k = i - 1; k >= 1; k--) {

cout << k << " ";

}

cout << m << " ";

for (j = 1; j <= n - i; j++) {

cout << j << " ";

}

}

cout << endl;

}

cout << endl;

}

# 3. Write a program in C++ to create and display unique three-digit number using 1, 2, 3, 4. Also count how many three-digit numbers are there

Sample Output:

The three-digit numbers are:

123 124 132 134 142 143 213 214 231 234 241 243 312 314 321 324 341 342 412 413 421 423 431 432

Total number of the three-digit-number is: 24

#include <iostream>

using namespace std;

void revOfString(const string& a);

int main()

{

string str;

cout << "\n\n Create and display the unique three-digit number using 1, 2, 3, 4:\n";

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

cout<<" The three-digit numbers are: "<<endl;

int amount = 0;

cout<<" ";

for(int i = 1; i <= 4; i++)

{

for(int j = 1; j <= 4; j++)

{

for(int k = 1; k <= 4; k++)

{

if(k != i && k != j && i != j)

{

amount++;

cout<<i <<j<<k<<" ";

}

}

}

}

cout<<endl<<" Total number of the three-digit-number is: "<< amount<<endl<<endl;

}

# 4. Write a C++ Program to Swap Two Numbers without using third variable.

#include <iostream>

using namespace std;

int main()

{

int a,b ;

cout<<"Enter 1st number :: ";

cin>>a;

cout<<"\nEnter 2nd number :: ";

cin>>b;

cout << "\nBefore swapping, Numbers are :: " << endl;

cout << "\n\ta = " << a << ", b = " << b << endl;

a = a + b;

b = a - b;

a = a - b;

cout << "\nAfter swapping, Numbers are :: " << endl;

cout << "\n\ta = " << a << ", b = " << b << endl;

return 0;

}

# 5. Petya loves football very much. One day, as he was watching a football match, he was writing the players' current positions on a piece of paper.

To simplify the situation he depicted it as a string consisting of zeroes and ones. A zero corresponds to players of one team; a one corresponds to players of another team. If there are at least 7 players of some team standing one after another, then the situation is considered dangerous. For example, the situation 00100110111111101 is dangerous and 11110111011101 is not. You are given the current situation. Determine whether it is dangerous or not.

Input

The first input line contains a non-empty string consisting of characters "0" and "1", which represents players. The length of the string does not exceed 100 characters. There's at least one player from each team present on the field.

Output

Print "YES" if the situation is dangerous. Otherwise, print "NO".

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s;

cin >> s;

int contiguous = 1;

for (size\_t i = 1; i < s.length(); ++i)

{

if (s[i] == s[i - 1])

{

contiguous += 1;

if (contiguous == 7)

{

cout << "YES" << endl;

return 0;

}

}

else

{

contiguous = 1;

}

}

cout << "NO" << endl;

return 0;

}

# 6. Capitalization is writing a word with its first letter as a capital letter. Your task is to capitalize the given word.

Note, that during capitalization all the letters except the first one remains unchanged.

Input

A single line contains a non-empty word. This word consists of lowercase and uppercase English letters. The length of the word will not exceed 103.

Output

Output the given word after capitalization.

#include <stdio.h>

#include <stdlib.h>

int main()

{

int i,l;

char s[1000];

scanf("%s",s);

for(i=0;i<strlen(s);i++){

if(s[0]>=97&&s[0]<=122) s[0]=65+s[0]-97;

}

printf("%s\n",s);

return 0;

}

# **7. Bob is preparing to pass IQ test. The most frequent task in this test is to find out which one of the given *n* numbers differs from the others.** Bob observed that one number usually differs from the others in evenness. Help Bob — to check his answers, he needs a program that among the given *n* numbers finds one that is different in evenness.

Input

The first line contains integer *n* (3 ≤ *n* ≤ 100) — amount of numbers in the task. The second line contains *n* space-separated natural numbers, not exceeding 100. It is guaranteed, that exactly one of these numbers differs from the others in evenness.

Output

Output index of number that differs from the others in evenness. Numbers are numbered from 1 in the input order.

#include<stdio.h>

int main()

{

int n,a[101],b=0,c=0,i,x,y;

scanf("%d",&n);

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

scanf("%d",&a[i]);

}

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

if(a[i]%2==0){

c++;

x=i;

}

else{

b++;

y=i;

}

}

if(c==1) printf("%d\n",x);

else if(b==1) printf("%d\n",y);

return 0;

# 9. Write an algorithm to get 0 and 1 with equal probability using a function that generates random numbers from 1 to 5 with equal probability.

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Function to generate a random number from 1 to 5 with equal probability

int random() {

return (rand() % 5) + 1;

}

// Returns 0 or 1 with equal probability using `random()` function

int generate()

{

int r;

do {

// `r` could be any one of 1, 2, 3, 4, and 5

r = random();

} while (r == 5);

// `r` could any of 1, 2, 3, 4 now

// since there are 2 odd and 2 even numbers, return the last bit of `r`,

// which could be 0 or 1 with equal probability

return r & 1;

}

int main(void)

{

srand(time(NULL));

int x = 0, y = 0;

// make 10000 calls to `generate()`

for (int i = 1; i <= 10000; i++) {

generate()? x++: y++;

}

// print the results

printf("0 ~ %0.2f%\n", x/100.0);

printf("1 ~ %0.2f%\n", y/100.0);

return 0;

}

# 10. Write an algorithm to generate any one of the given n numbers according to given probabilities.

For example, consider the following integer array and their probabilities. The solution should return 1 with 30% probability, 2 with 10% probability, 3 with 20% probability, and so on for every array element.

nums[] = { 1, 2, 3, 4, 5 };

probability[] = { 30, 10, 20, 15, 25 }; // total probability should sum to 100%

Algorithm:

Construct a sum array S[] from the given probability array P[], where S[i] holds the sum of all P[j] for 0 <= j <= i.

Generate a random integer from 1 to 100 and check where it lies in S[].

Based on the comparison result, return the corresponding element from the input array.

#include <iostream>

#include <vector>

#include <unordered\_map>

#include <cstdlib>

#include <ctime>

using namespace std;

// Function to generate random nums from a vector according to the given probabilities

int random(vector<int> const &nums, vector<int> const &probability)

{

int n = nums.size();

if (n != probability.size()) {

return -1; // error

}

// construct a sum vector from the given probabilities

vector<int> prob\_sum(n, 0);

// `prob\_sum[i]` holds sum of all `probability[j]` for `0 <= j <=i`

prob\_sum[0] = probability[0];

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

prob\_sum[i] = prob\_sum.at(i-1) + probability[i];

}

// generate a random integer from 1 to 100 and check where it lies in `prob\_sum[]`

int r = (rand() % 100) + 1;

// based on the comparison result, return the corresponding

// element from the nums vector

if (r <= prob\_sum[0]) { // handle 0th index separately

return nums[0];

}

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

{

if (r > prob\_sum.at(i-1) && r <= prob\_sum[i]) {

return nums[i];

}

}

}

int main()

{

// Input: vector of integers and their probabilities

// Goal: generate `nums[i]` with probability equal to `probability[i]`

vector<int> nums = {1, 2, 3, 4, 5};

vector<int> probability = {30, 10, 20, 15, 25};

// initialize srand with a distinctive value

srand(time(NULL));

// maintain a frequency map to validate the results

unordered\_map<int, int> freq;

// make 1000000 calls to the `random()` function and

// store results in a map

for (int i = 0; i < 1000000; i++)

{

int val = random(nums, probability);

freq[val]++;

}

// print the results

for (int i = 0; i < nums.size(); i++) {

cout << nums[i] << " ~ " << freq[nums[i]]/10000.0 << "%" << endl;

}

return 0;

}

# 11. Find all n–digit strictly increasing numbers where n varies from 1 to 9. A number is strictly increasing if every digit is greater than its preceding digit.

8–digit strictly increasing numbers are:

12345678 12345679 12345689 12345789 12346789 12356789 12456789 13456789 23456789

7–digit strictly increasing numbers are:

1234567 1234568 1234569 1234578 1234579 1234589 1234678 1234679 1234689 1234789 1235678 1235679 1235689 1235789 1236789 1245678 1245679 1245689 1245789 1246789 1256789 1345678 1345679 1345689 1345789 1346789 1356789 1456789 2345678 2345679 2345689 2345789 2346789 2356789 2456789 3456789

#include <stdio.h>

// Function to find all n–digit strictly increasing

// numbers in a bottom-up manner

void printStrictlyInc(int prev, char result[], int index, int size)

{

// If the array becomes n–digit

if (index == size)

{

// null terminate the character array and print it

result[index] = '\0';

printf("%s ", result);

return;

}

// start from the next digit (the last digit is `prev` at `result[index-1]`)

for (int i = prev + 1; i <= 9; i++)

{

// put digit `i` in the current index

result[index] = '0' + i;

// recur for the next index

printStrictlyInc(i, result, index + 1, size);

}

}

int main(void)

{

int n = 7;

char result[n + 1]; // stores output

printStrictlyInc(0, result, 0, n);

return 0;

}

# 12. Write an efficient algorithm to construct the longest palindrome by shuffling or deleting characters from a given string.

Input: ABBDAB

Output: The longest palindrome is BABAB (or BADAB or ABBBA or ABDBA)

Input: ABCDD

Output: The longest palindrome is DAD (or DBD or DCD)

#include <iostream>

#include <unordered\_map>

using namespace std;

// Construct the longest palindrome by shuffling or deleting

// characters from a given string

string longestPalindrome(string str)

{

// create a frequency map for characters of a given string

unordered\_map<char, int> freq;

for (char ch: str) {

freq[ch]++;

}

string mid\_char; // stores odd character

string left; // stores left substring

// iterate through the frequency map

for (auto p: freq)

{

char ch = p.first; // get current character

int count = p.second; // get character frequency

// if the current character's frequency is odd,

// update mid to current char (and discard the old one)

if (count & 1) {

mid\_char = ch;

}

// append half of the characters to the left substring

// (the other half goes to the right substring in reverse order)

left.append(count/2, ch);

}

// the right substring will be the reverse of the left substring

string right(left.rbegin(), left.rend());

// return string formed by the left substring, mid-character (if any),

// and the right substring

return (left + mid\_char + right);

}

int main()

{

string str = "ABBDAB";

cout << "The longest palindrome is " << longestPalindrome(str);

return 0;

}

# 13. Given a positive number n, find all strings of length n containing balanced parentheses.

Input: n = 4

Output:

(())

()()

Input: n = 6

Output:

((()))

(()())

(())()

()(())

()()()

Input: n = 5

Output: Invalid input

| #include <iostream>  #include <string>  using namespace std;    // Function to find all strings of length `n` containing balanced parentheses  void balParenthesis(int n, string str, int open)  {  // if `n` is odd with no open parentheses, balanced parentheses  // cannot be formed  if ((n & 1) && !open) {  return;  }    // base case: length `n` is reached  if (n == 0)  {  // if the output string contains all balanced parenthesis, print it  if (open == 0) {  cout << str << endl;  }  return;  }    // Optimization: return if we cannot close all open parentheses with  // left characters  if (open > n) {  return;  }    // recur with open parentheses  balParenthesis(n - 1, str + "(", open + 1);    // recur with closed parentheses only if the output string has at least  // one unclosed parentheses  if (open > 0) {  balParenthesis(n - 1, str + ")", open - 1);  }  }    int main()  {  int n = 6;  balParenthesis(n, "", 0);  return 0;  } |
| --- |

# 14. Given two integer arrays, reorder elements of the first array by the order of elements defined by the second array.

The elements that are not present in the second array but present in the first array should be appended at the end sorted. The second array can contain some extra elements which are not part of the first array.

Input:first = [5, 8, 9, 3, 5, 7, 1, 3, 4, 9, 3, 5, 1, 8, 4]

second = [3, 5, 7, 2]

Output: [3, 3, 3, 5, 5, 5, 7, 1, 1, 4, 4, 8, 8, 9, 9]

#include <iostream>

#include <unordered\_map>

#include <algorithm>

using namespace std;

void customSort(int first[], int second[], int m, int n)

{

// map to store the frequency of each element of the first array

unordered\_map<int, int> freq;

// find the frequency of each element of the first array and

// store it in a map

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

freq[first[i]]++;

}

// Note that once we have the frequencies of all elements of

// the first array, we can overwrite elements of the first array

int index = 0;

// do for every element of the second array

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

{

// If the current element is present on the map, print it `n` times

// where `n` is the frequency of that element in the first array

while (freq[second[i]])

{

first[index++] = second[i];

freq[second[i]]--;

}

// erase the element from the map

freq.erase(second[i]);

}

// Now we are left with elements only present in the first array,

// but not in the second array.

// Sort the remaining elements present on the map (Note that the keys are

// already sorted if we use `std::map`)

int i = index;

for (auto it = freq.begin(); it != freq.end(); it++)

{

while (it->second--) {

first[index++] = (it->first);

}

}

// sort the remaining elements

sort(first + i, first + index);

}

// Utility function to print the first `n` elements of an array `arr`

void printArray(int arr[], int n)

{

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

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

}

cout << endl;

}

int main()

{

int first[] = { 5, 8, 9, 3, 5, 7, 1, 3, 4, 9, 3, 5, 1, 8, 4 };

int second[] = { 3, 5, 7, 2 };

int m = sizeof(first) / sizeof(first[0]);

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

customSort(first, second, m, n);

cout << "The array after sorting is ";

printArray(first, m);

return 0;

}

# 15. Given an M × N rectangular grid, print all routes in the grid that start at the first cell (0, 0) and ends at the last cell (M-1, N-1). We can move down or right or diagonally (down-right), but not up or left.

**Input:**

**{ 1, 2, 3 }**

**{ 4, 5, 6 }**

**{ 7, 8, 9 }**

**Output:**

**[ 1, 4, 7, 8, 9 ]**

**[ 1, 4, 5, 8, 9 ]**

**[ 1, 4, 5, 6, 9 ]**

**[ 1, 4, 5, 9 ]**

**[ 1, 4, 8, 9 ]**

**[ 1, 2, 5, 8, 9 ]**

**[ 1, 2, 5, 6, 9 ]**

**[ 1, 2, 5, 9 ]**

**[ 1, 2, 3, 6, 9 ]**

**[ 1, 2, 6, 9 ]**

**[ 1, 5, 8, 9 ]**

**[ 1, 5, 6, 9 ]**

**[ 1, 5, 9 ]**

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**// Recursive function to get all routes in a rectangular grid**

**// that start at cell (i, j) and ends at the last cell (M-1, N-1).**

**void printPaths(vector<vector<int>> const &mat, vector<int> &route, int i, int j)**

**{**

**// base case**

**if (mat.size() == 0) {**

**return;**

**}**

**// `M × N` matrix**

**int M = mat.size();**

**int N = mat[0].size();**

**// if the last cell is reached**

**if (i == M - 1 && j == N - 1)**

**{**

**// print the current route**

**for (int i: route) {**

**cout << i << ", ";**

**}**

**cout << mat[i][j] << endl;**

**return;**

**}**

**// include current cell in route**

**route.push\_back(mat[i][j]);**

**// move down**

**if (i + 1 < M) {**

**printPaths(mat, route, i + 1, j);**

**}**

**// move right**

**if (j + 1 < N) {**

**printPaths(mat, route, i, j + 1);**

**}**

**// move diagonally**

**if (i + 1 < M && j + 1 < N) {**

**printPaths(mat, route, i + 1, j + 1);**

**}**

**// backtrack**

**route.pop\_back();**

**}**

**// Print all routes in a rectangular grid**

**void printPaths(vector<vector<int>> const &mat)**

**{**

**// vector to store the current route**

**vector<int> route;**

**// start from the first cell (0, 0)**

**printPaths(mat, route, 0, 0);**

**}**

**int main()**

**{**

**vector<vector<int>> mat =**

**{**

**{ 1, 2, 3 },**

**{ 4, 5, 6 },**

**{ 7, 8, 9 }**

**};**

**printPaths(mat);**

**return 0;**

**}**

# 16. Write an efficient code to clone a linked list with each node containing an additional random pointer. The random pointer can point to any random node of the linked list or null.

To clone a linked list with random pointers, maintain a [hash table](https://www.techiedelight.com/hashing-in-data-structure/) for storing the mappings from a linked list node to its clone. We create a new node with the same data for each node in the original linked list and recursively set its next pointers. We also create a mapping from the original node to the duplicate node in the hash table. Finally, traverse the original linked list again and update the duplicate nodes’ random pointers using the hash table.

#include <iostream>

#include <unordered\_map>

using namespace std;

// A linked list node with a random pointer

struct Node

{

int data;

Node\* next;

Node\* random;

// Constructor

Node(int data)

{

this->data = data;

this->next = nullptr;

this->random = nullptr;

}

};

// Recursive function to print a linked list

void traverse(Node\* head)

{

if (head == nullptr)

{

cout << "null" << endl;

return;

}

// print current node data and random pointer data

if (head->random) {

cout << head->data << "(" << head->random->data << ") —> ";

}

else {

cout << head->data << "(" << "X" << ") —> ";

}

// recur for the next node

traverse(head->next);

}

// Recursive function to copy random pointers from the original linked list

// into the cloned linked list using the map

void updateRandomPointers(Node\* head, unordered\_map<Node\*, Node\*> &map)

{

// base case

if (map[head] == nullptr) {

return;

}

// update the random pointer of the cloned node

map[head]->random = map[head->random];

// recur for the next node

updateRandomPointers(head->next, map);

}

// Recursive function to clone the data and next pointer for each node

// of the linked list into a given map

Node\* cloneLinkedList(Node\* head, unordered\_map<Node\*, Node\*> &map)

{

// base case

if (head == nullptr) {

return nullptr;

}

// clone all fields of the head node except the random pointer

// create a new node with the same data as the head node

map[head] = new Node(head->data);

// clone the next node

map[head]->next = cloneLinkedList(head->next, map);

// return cloned head node

return map[head];

}

// Function to clone a linked list having random pointers

Node\* cloneLinkedList(Node\* head)

{

// Create a map to store mappings from a node to its clone

unordered\_map<Node\*, Node\*> map;

// clone data and next pointer for each node of the original

// linked list and put references into the map

cloneLinkedList(head, map);

// update random pointers from the original linked list in the map

updateRandomPointers(head, map);

// return the cloned head node

return map[head];

}

int main()

{

Node\* head = new Node(1);

head->next = new Node(2);

head->next->next = new Node(3);

head->next->next->next = new Node(4);

head->next->next->next->next = new Node(5);

head->random = head->next->next->next;

head->next->next->random = head->next;

cout << "Original Linked List:\n";

traverse(head);

Node\* clone = cloneLinkedList(head);

cout << "\nCloned Linked List:\n";

traverse(clone);

return 0;

}

# 17. Given a map containing employee to manager mappings, find all employees under each manager who directly or indirectly reports him.

**For example, consider the following employee-manager pairs:**

A —> A

B —> A

C —> B

D —> B

E —> D

F —> E

Here, A reports to himself, i.e., A is head of the company and is the manager of employee B. B is the manager of employees C and D, D is the manager of employee E, E is the manager of employee F, C, and F is not managers of any employee.

**Output:**

A —> [B, D, C, E, F]

B —> [D, C, E, F]

C —> []

D —> [E, F]

E —> [F]

F —> []

#include <iostream>

#include <unordered\_set>

#include <unordered\_map>

using namespace std;

// Utility function to print a unordered\_set

void printSet(char c, unordered\_set<char> const &v)

{

cout << c << " —> [";

int n = v.size();

for (auto i: v) {

cout << i;

if (--n) {

cout << ", ";

}

}

cout << "]\n";

}

// Recursive DP function to find all employees who directly or indirectly

// report to a given manager and store the result in `result`

unordered\_set<char> findAllReportingEmployees(char manager,

auto &managerToEmployeeMappings,

auto &result)

{

// if the subproblem is already seen before

if (result.find(manager) != result.end())

{

// return the already computed mapping

return result[manager];

}

// find all employees reporting directly to the current manager

unordered\_set<char> managerEmployees = managerToEmployeeMappings[manager];

// find all employees reporting indirectly to the current manager

for (char reportee: managerToEmployeeMappings[manager])

{

// find all employees reporting to the current employee

unordered\_set<char> employees = findAllReportingEmployees(reportee,

managerToEmployeeMappings, result);

// move those employees to the current manager

for (char c: employees) {

managerEmployees.insert(c);

}

}

// save the result to avoid recomputation and return it

result[manager] = managerEmployees;

return managerEmployees;

}

// Find all employees who directly or indirectly reports to a manager

unordered\_map<char, unordered\_set<char>> findEmployees(auto &employeeToManagerMappings)

{

// store manager to employee mappings in a new map.

// `unordered\_set<char>` is used since a manager can have several employees mapped

unordered\_map<char, unordered\_set<char>> managerToEmployeeMappings;

// fill the above map with the manager to employee mappings

for (auto it: employeeToManagerMappings)

{

char employee = it.first;

char manager = it.second;

// don't map an employee with itself

if (employee != manager) {

managerToEmployeeMappings[manager].insert(employee);

}

}

// construct an ordered map to store the result

unordered\_map<char, unordered\_set<char>> result;

// find all reporting employees (direct and indirect) for every manager

// and store the result in a map

for (auto p: employeeToManagerMappings) {

findAllReportingEmployees(p.first, managerToEmployeeMappings, result);

}

return result;

}

int main()

{

// construct a mapping from employee to manager

unordered\_map<char, char> employeeToManagerMappings = {

{'A', 'A'}, {'B', 'A'}, {'C', 'B'}, {'D', 'B'}, {'E', 'D'}, {'F', 'E'}

};

auto result = findEmployees(employeeToManagerMappings);

// print contents of the resulting map

for (auto p: result) {

printSet(p.first, p.second);

}

return 0;

}

# 18. Write an efficient algorithm to reverse the specified portion of a given linked list.

**Input:**

**Linked List: 1 —> 2 —> 3 —> 4 —> 5 —> 6 —> 7 —> None**

**start position = 2**

**end position = 5**

**Output: 1 —> 5 —> 4 —> 3 —> 2 —> 6 —> 7 —> None**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**// A Linked List Node**

**struct Node**

**{**

**int data;**

**Node\* next;**

**};**

**// Utility function to print a linked list**

**void printList(string msg, Node\* head)**

**{**

**cout << msg;**

**Node\* ptr = head;**

**while (ptr)**

**{**

**cout << ptr->data << " —> ";**

**ptr = ptr->next;**

**}**

**cout << "null" << endl;**

**}**

**// Helper function to insert a new node at the beginning of the linked list**

**void push(Node\*\* head, int data)**

**{**

**Node\* newNode = new Node();**

**newNode->data = data;**

**newNode->next = \*head;**

**\*head = newNode;**

**}**

**// Iteratively reverse a linked list from position `m` to `n`**

**// Note that the head is passed by reference**

**void reverse(Node\* &head, int m, int n)**

**{**

**// base case**

**if (m > n) {**

**return;**

**}**

**Node\* prev = NULL; // the previous pointer**

**Node\* curr = head; // the main pointer**

**// 1. Skip the first `m` nodes**

**for (int i = 1; curr != NULL && i < m; i++)**

**{**

**prev = curr;**

**curr = curr->next;**

**}**

**// `prev` now points to (m-1)'th node**

**// `curr` now points to m'th node**

**Node\* start = curr;**

**Node\* end = NULL;**

**// 2. Traverse and reverse the sublist from position `m` to `n`**

**for (int i = 1; curr != NULL && i <= n - m + 1; i++)**

**{**

**// Take note of the next node**

**Node\* next = curr->next;**

**// move the current node onto the `end`**

**curr->next = end;**

**end = curr;**

**// move to the next node**

**curr = next;**

**}**

**/\***

**`start` points to the m'th node**

**`end` now points to the n'th node**

**`curr` now points to the (n+1)'th node**

**\*/**

**// 3. Fix the pointers and return the head node**

**if (start)**

**{**

**start->next = curr;**

**if (prev != NULL) {**

**prev->next = end;**

**}**

**// when m = 1, `prev` is nullptr**

**else {**

**// fix the head pointer to point to the new front**

**head = end;**

**}**

**}**

**}**

**int main()**

**{**

**int m = 2, n = 5;**

**Node\* head = NULL;**

**for (int i = 7; i >= 1; i--) {**

**push(&head, i);**

**}**

**printList("Original linked list: ", head);**

**reverse(head, m, n);**

**printList("Reversed linked list: ", head);**

**return 0;**

**}**

# 19. Generate fair results from a biased coin that prefers one side of the coin over another and returns TAILS with p probability and HEADS with 1-p probability where p != (1-p).

We can use the given biased coin for fair results by making two calls from the biased coin instead of one call. If the results of both calls match (both are HEADS, or both are TAILS), discard the results and start over. If the results differ, consider the first result.

How this works?

Suppose we have a biased() function that returns TAILS with p probability and HEADS with 1-p probability. We make two independent subsequent calls to the biased() and store the results. Then there are four possible possibilities:

1. The probability that both calls returns TAILS = p × p
2. The probability that both calls returns HEADS = (1 - p) × (1 - p)
3. The probability that the first call returns TAILS, and the second call returns HEADS = p × (1 - p)
4. The probability that the first call returns HEADS, and the second call returns TAILS = (1 - p) × p

Clearly, the biased coin has the same probability of getting TAILS and then HEADS as the probability of getting HEADS and then TAILS.

So if we exclude the events of two HEADS and two TAILS by repeating the procedure, we are left with the only two remaining outcomes having equivalent probability. That’s the reason why we will get a fair result.

#include <stdio.h>

#include <stdlib.h>

#define HEADS 1

#define TAILS 0

// A biased function that returns TAILS with probability `p` and

// HEADS with `1-p` probability

int biased()

{

// generate a random number between 0–99, both inclusive

int r = rand() % 100;

// return TAILS if we got a number between [0–79]; otherwise, return HEADS

return (r <= 79)? TAILS: HEADS;

}

// Return HEADS and TAILS with equal probability using the specified function

int generate()

{

while (1)

{

int first = biased();

int second = biased();

if (first != second) {

return first; // or return second

}

}

}

int main(void)

{

int x = 0, y = 0;

for (int i = 0; i < 100000; i++)

{

int val = generate();

(val == 0) ? x++ : y++;

}

printf("0 ~ %f%\n1 ~ %f%%", x/1000.0, y/1000.0);

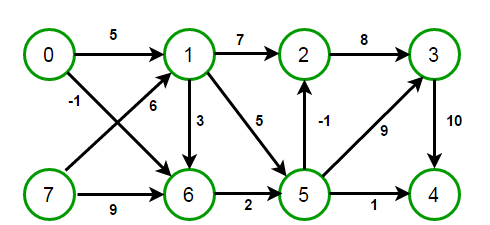
return 0;

}

# **20, Least cost path in a digraph from a given source to a destination having exactly `m` edges**

Given a weighted digraph (directed graph), find the least-cost path from a given source to a given destination with exactly m edges.

For example, consider the following graph,



Let source = 0, destination = 3, number of edges (m) = 4.

The graph has 3 routes from source 0 to destination 3 with 4 edges.

0—1—5—2—3 having cost 17

0—1—6—5—3 having cost 19

0—6—5—2—3 having cost 8

The solution should return the least-cost, i.e., 8.

[Practice this problem](https://techiedelight.com/practice/?problem=LeastCostPathII)

Whenever we see the term *shortest*, the first thing we should think about is doing a [BFS traversal](https://www.techiedelight.com/breadth-first-search/). So, here also, we start BFS traversal from the given source vertex. Usually, BFS doesn’t explore already discovered vertices again, but here we do the opposite. To cover all possible paths from source to destination, remove this check from BFS. But what if the graph contains a cycle? Removing this check will cause the program to go into an infinite loop. We can easily handle that if we don’t consider nodes having a BFS depth of more than m.

The solution below maintains the following information in a BFS queue node:

* The current vertex number.
* The current depth of BFS (i.e., how far the current node is from the source?).
* The cost of the current path chosen so far.

Whenever we encounter any node whose cost of a path is more and required BFS depth is reached, update the result. The BFS will terminate when we have explored every path in the given graph or BFS depth exceeds m.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108 | #include <iostream>  #include <queue>  #include <vector>  #include <climits>  using namespace std;    // Data structure to store a graph edge  struct Edge {  int src, dest, weight;  };    // A class to represent a graph object  class Graph  {  public:  // a vector of vectors to represent an adjacency list  vector<vector<Edge>> adjList;    // Graph Constructor  Graph(vector<Edge> const &edges, int n)  {  // resize the vector to hold `n` elements of type vector<Edge>  adjList.resize(n);    // add edges to the directed graph  for (auto &edge: edges) {  adjList[edge.src].push\_back(edge);  }  }  };    // A BFS Node  struct Node {  int vertex, depth, weight;  };    // Perform BFS on graph `g` starting from vertex `v`  int findLeastCost(Graph const &g, int src, int dest, int m)  {  // create a queue for doing BFS  queue<Node> q;    // enqueue source vertex  q.push({src, 0, 0});    // stores least-cost from source to destination  int minCost = INT\_MAX;    // loop till queue is empty  while (!q.empty())  {  // dequeue front node  Node node = q.front();  q.pop();    int v = node.vertex;  int depth = node.depth;  int cost = node.weight;    // if the destination is reached and BFS depth is equal to `m`,  // update the minimum cost calculated so far  if (v == dest && depth == m) {  minCost = min(minCost, cost);  }    // don't consider nodes having a BFS depth more than `m`.  // This check will result in optimized code and handle cycles  // in the graph (otherwise, the loop will never break)  if (depth > m) {  break;  }    // do for every adjacent edge of `v`  for (Edge edge: g.adjList[v])  {  // push every vertex (discovered or undiscovered) into  // the queue with depth as +1 of parent and cost equal  // to the cost of parent plus the current edge weight  q.push({edge.dest, depth + 1, cost + edge.weight});  }  }    // return least-cost  return minCost;  }    int main()  {  // vector of graph edges as per the above diagram  vector<Edge> edges =  {  {0, 6, -1}, {0, 1, 5}, {1, 6, 3}, {1, 5, 5}, {1, 2, 7}, {2, 3, 8}, {3, 4, 10},  {5, 2, -1}, {5, 3, 9}, {5, 4, 1}, {6, 5, 2}, {7, 6, 9}, {7, 1, 6}  };    // total number of nodes in the graph (labelled from 0 to 7)  int n = 8;    // build a graph from the given edges  Graph g(edges, n);    int src = 0, dest = 3, m = 4;    // Perform modified BFS traversal from source vertex `src`  cout << findLeastCost(g, src, dest, m);    return 0;  } |
| --- | --- |

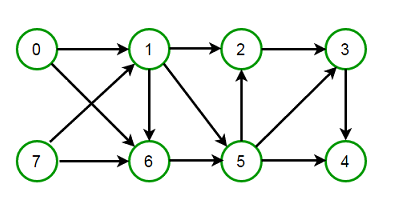
[Download](https://www.techiedelight.com/code/iokgoB) [Run Code](https://techiedelight.com/compiler/?run=iokgoB)

**Output:**

# **21, Total paths in a digraph from a given source to a destination having exactly `m` edges**

Given a digraph (directed graph), find the total number of routes to reach the destination from a given source with exactly m edges.

For example, consider the following graph:



Let source = 0, destination = 3, number of edges m = 4. The graph has 3 routes from source 0 to destination 3 with 4 edges. The solution should return the total number of routes 3.

0 —> 1 —> 5 —> 2 —> 3

0 —> 1 —> 6 —> 5 —> 3

0 —> 6 —> 5 —> 2 —> 3

[Practice this problem](https://techiedelight.com/practice/?problem=TotalPathsInDigraph)

The idea is to do a [BFS traversal](https://www.techiedelight.com/breadth-first-search/) from the given source vertex. BFS is generally used to find the shortest paths in graphs/matrices, but we can modify normal BFS to meet our requirements. Usually, BFS doesn’t explore already discovered vertices again, but here we do the opposite. To cover all possible paths from source to destination, remove this check from BFS. But if the graph contains a [cycle](https://www.techiedelight.com/check-undirected-graph-contains-cycle-not/), removing this check will cause the program to go into an infinite loop. We can easily handle that if we don’t consider nodes having a BFS depth of more than m. Basically, we maintain two things in the BFS queue node:

* The current vertex number.
* The current depth of BFS (i.e., how far away from the current node is from the source?).

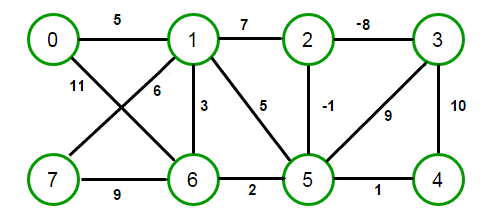
So, whenever the destination vertex is reached and BFS depth is equal to m, we update the result. The BFS will terminate when we have explored every path in the given graph or BFS depth exceeds m. Following is the implementation in C++, Java, and Python based on the above idea:

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106 | #include <iostream>  #include <vector>  #include <queue>  using namespace std;    // Data structure to store a graph edge  struct Edge {  int src, dest;  };    // A class to represent a graph object  class Graph  {  public:  // a vector of vectors to represent an adjacency list  vector<vector<int>> adjList;    // Graph Constructor  Graph(vector<Edge> const &edges, int n)  {  // resize the vector to hold `n` elements of type `vector<int>`  adjList.resize(n);    // add edges to the directed graph  for (auto &edge: edges) {  adjList[edge.src].push\_back(edge.dest);  }  }  };    // A BFS Node  struct Node  {  // stores current vertex number and the current depth of  // BFS (how far away from the source current node is)  int vertex, depth;  };    // Perform BFS on graph `graph` starting from vertex `v`  int findTotalPaths(Graph const &graph, int src, int dest, int m)  {  // create a queue for doing BFS  queue<Node> q;    // enqueue source vertex  q.push({src, 0});    // stores number of paths from source to destination having exactly `m` edges  int count = 0;    // loop till queue is empty  while (!q.empty())  {  // dequeue front node  Node node = q.front();  q.pop();    int v = node.vertex;  int depth = node.depth;    // if the destination is reached and BFS depth is equal to `m`, update count  if (v == dest && depth == m) {  count++;  }    // don't consider nodes having a BFS depth more than `m`.  // This check will result in optimized code and handle cycles  // in the graph (otherwise, the loop will never break)  if (depth > m) {  break;  }    // do for every adjacent vertex `u` of `v`  for (int u: graph.adjList[v])  {  // enqueue every vertex (discovered or undiscovered)  q.push({u, depth + 1});  }  }    // return number of paths from source to destination  return count;  }    int main()  {  // vector of graph edges as per the above diagram  vector<Edge> edges =  {  {0, 6}, {0, 1}, {1, 6}, {1, 5}, {1, 2}, {2, 3}, {3, 4},  {5, 2}, {5, 3}, {5, 4}, {6, 5}, {7, 6}, {7, 1}  };    // total number of nodes in the graph  int n = 8;    // build a graph from the given edges  Graph graph(edges, n);    int src = 0, dest = 3, m = 4;    // Do modified BFS traversal from the source vertex src  cout << findTotalPaths(graph, src, dest, m);    return 0;  } |
| --- | --- |

# **22, Find maximum cost path in a graph from a given source to a given destination**

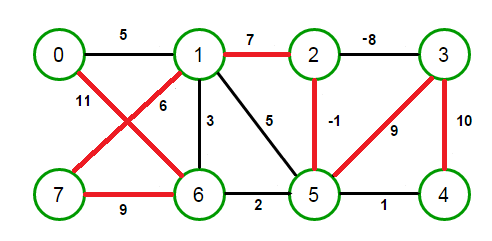
Given a weighted undirected graph, find the maximum cost path from a given source to any other vertex in the graph which is greater than a given cost. The path should not contain any cycles.

For example, consider the following graph,



Let source = 0 and cost = 50.

The maximum cost route from source vertex 0 is 0—6—7—1—2—5—3—4, having cost 51, which is more than cost 50. The solution should return 51.



[Practice this problem](https://techiedelight.com/practice/?problem=MaximumCostPath)

The idea is to do a [Breadth–first search (BFS)](https://www.techiedelight.com/breadth-first-search/) traversal. BFS is generally used to find the shortest paths in graphs/matrices, but we can modify normal BFS to meet our requirements. By modifying BFS, we don’t mean using a [priority queue](https://www.techiedelight.com/introduction-priority-queues-using-binary-heaps/) that picks up the maximum weighted edge at every step, as that approach will fail. A low-weight edge can also be involved in the maximum cost path as there might be higher weight edges connected through it.

So, how can we use BFS?

Usually, BFS doesn’t explore already discovered vertices again, but here we do the opposite. To cover all possible paths from a given source, remove this check from BFS. But if the graph contains a [cycle](https://www.techiedelight.com/check-undirected-graph-contains-cycle-not/), removing this check will cause the program to go into an infinite loop. We can easily handle that if we maintain a list of nodes visited so far in the current path for a node in a queue. Basically, we maintain three things in the BFS queue node:

* The current vertex number.
* The cost of the current path chosen so far.
* The set of nodes visited so far in the current path.

Whenever we encounter any node whose cost of a path is more, update the result. The BFS will terminate when we have explored every path that doesn’t result in a cycle.

#include <iostream>

#include <queue>

#include <vector>

#include <set>

#include <climits>

using namespace std;

// Data structure to store a graph edge

struct Edge {

int src, dest, weight;

};

// A class to represent a graph object

class Graph

{

public:

// a vector of vectors of `Edge` to represent an adjacency list

vector<vector<Edge>> adjList;

// Constructor

Graph(vector<Edge> const &edges, int n)

{

// resize the vector to hold `n` elements of type vector<Edge>

adjList.resize(n);

// add edges to the undirected graph

for (auto &edge: edges)

{

int src = edge.src;

int dest = edge.dest;

int weight = edge.weight;

adjList[src].push\_back({src, dest, weight});

adjList[dest].push\_back({dest, src, weight});

}

}

};

// A BFS Node

struct Node

{

// current vertex number and cost of the current path

int vertex, weight;

// set of nodes visited so far in the current path

set<int> s;

};

// Perform BFS on graph `graph` starting from vertex `v`

int findMaxCost(Graph const &graph, int src, int k)

{

// create a queue for doing BFS

queue<Node> q;

// add source vertex to set and enqueue it

set<int> vertices;

vertices.insert(src);

q.push({src, 0, vertices});

// stores maximum cost of a path from the source

int maxCost = INT\_MIN;

// loop till queue is empty

while (!q.empty())

{

// dequeue front node

Node node = q.front();

q.pop();

int v = node.vertex;

int cost = node.weight;

vertices = node.s;

// if the destination is reached and BFS depth is equal to `m`,

// update the minimum cost calculated so far

if (cost > k) {

maxCost = max(maxCost, cost);

}

// do for every adjacent edge of `v`

for (Edge edge: graph.adjList[v])

{

// check for a cycle

if (vertices.find(edge.dest) == vertices.end())

{

// add current node to the path

set<int> s = vertices;

s.insert(edge.dest);

// push every vertex (discovered or undiscovered) into

// the queue with a cost equal to the

// parent's cost plus the current edge's weight

q.push({edge.dest, cost + edge.weight, s});

}

}

}

// return max-cost

return maxCost;

}

int main()

{

// vector of graph edges as per the above diagram

vector<Edge> edges =

{

{0, 6, 11}, {0, 1, 5}, {1, 6, 3}, {1, 5, 5}, {1, 2, 7},

{2, 3, -8}, {3, 4, 10}, {5, 2, -1}, {5, 3, 9}, {5, 4, 1},

{6, 5, 2}, {7, 6, 9}, {7, 1, 6}

};

// total number of nodes in the graph (labelled from 0 to 7)

int n = 8;

// build a graph from the given edges

Graph graph(edges, n);

int src = 0;

int cost = 50;

// Start modified BFS traversal from source vertex `src`

int maxCost = findMaxCost(graph, src, cost);

if (maxCost != INT\_MIN) {

cout << maxCost;

}

else {

cout << "All paths from source have their costs < " << cost;

}

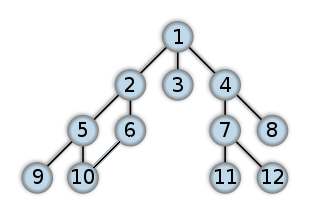
return 0;

}

# **23, Check if an undirected graph contains a cycle or not**

Given a connected undirected graph, check if it contains any cycle or not.

For example, the following graph contains a cycle 2–5–10–6–2:



When we do a [Breadth–first search (BFS)](https://www.techiedelight.com/breadth-first-search/) from any vertex v in an undirected graph, we may encounter **a cross-edge** that points to a previously discovered vertex that is neither an ancestor nor a descendant of the current vertex. Each “cross edge” defines a cycle in an undirected graph. If the cross edge is x —> y, then since y is already discovered, we have a path from v to y (or from y to v since the graph is undirected), where v is the starting vertex of BFS. So, we can say that we have a path v ~~ x ~ y ~~ v that forms a cycle. (Here, ~~ represents one more edge in the path, and ~ represents a direct edge).

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

// Data structure to store a graph edge

struct Edge {

int src, dest;

};

// A class to represent a graph object

class Graph

{

public:

// a vector of vectors to represent an adjacency list

vector<vector<int>> adjList;

// Graph Constructor

Graph(vector<Edge> const &edges, int n)

{

// resize the vector to hold `n` elements of type `vector<int>`

adjList.resize(n);

// add edges to the undirected graph

for (auto &edge: edges)

{

adjList[edge.src].push\_back(edge.dest);

adjList[edge.dest].push\_back(edge.src);

}

}

};

// Node to store vertex and its parent info in BFS

struct Node {

int v, parent;

};

// Perform BFS on the graph starting from vertex `src` and

// return true if a cycle is found in the graph

bool BFS(Graph const &graph, int src, int n)

{

// to keep track of whether a vertex is discovered or not

vector<bool> discovered(n);

// mark the source vertex as discovered

discovered[src] = true;

// create a queue for doing BFS and

// enqueue source vertex

queue<Node> q;

q.push({src, -1});

// loop till queue is empty

while (!q.empty())

{

// dequeue front node and print it

Node node = q.front();

q.pop();

// do for every edge (v, u)

for (int u: graph.adjList[node.v])

{

if (!discovered[u])

{

// mark it as discovered

discovered[u] = true;

// construct the queue node containing info

// about vertex and enqueue it

q.push({ u, node.v });

}

// `u` is discovered, and `u` is not a parent

else if (u != node.parent)

{

// we found a cross-edge, i.e., the cycle is found

return true;

}

}

}

// no cross-edges were found in the graph

return false;

}

int main()

{

// initialize edges

vector<Edge> edges =

{

{0, 1}, {0, 2}, {0, 3}, {1, 4}, {1, 5}, {4, 8},

{4, 9}, {3, 6}, {3, 7}, {6, 10}, {6, 11}, {5, 9}

// edge {5, 9} introduces a cycle in the graph

};

// total number of nodes in the graph (0 to 11)

int n = 12;

// build a graph from the given edges

Graph graph(edges, n);

// Perform BFS traversal in connected components of a graph

if (BFS(graph, 0, n)) {

cout << "The graph contains a cycle";

}

else {

cout << "The graph doesn't contain any cycle";

}

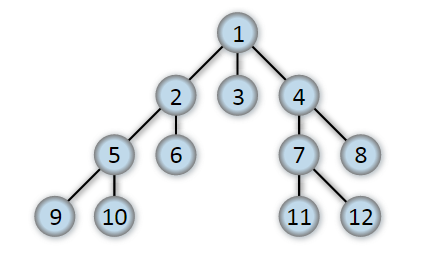
return 0;

}

# **24, Breadth-First Search (BFS) – Iterative and Recursive Implementation**

Breadth–first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a ‘search key’) and explores the neighbor nodes first before moving to the next-level neighbors.

The following graph shows the order in which the nodes are discovered in BFS:



#include <iostream>

#include <queue>

#include <vector>

using namespace std;

// Data structure to store a graph edge

struct Edge {

int src, dest;

};

// A class to represent a graph object

class Graph

{

public:

// a vector of vectors to represent an adjacency list

vector<vector<int>> adjList;

// Graph Constructor

Graph(vector<Edge> const &edges, int n)

{

// resize the vector to hold `n` elements of type `vector<int>`

adjList.resize(n);

// add edges to the undirected graph

for (auto &edge: edges)

{

adjList[edge.src].push\_back(edge.dest);

adjList[edge.dest].push\_back(edge.src);

}

}

};

// Perform BFS on the graph starting from vertex `v`

void BFS(Graph const &graph, int v, vector<bool> &discovered)

{

// create a queue for doing BFS

queue<int> q;

// mark the source vertex as discovered

discovered[v] = true;

// enqueue source vertex

q.push(v);

// loop till queue is empty

while (!q.empty())

{

// dequeue front node and print it

v = q.front();

q.pop();

cout << v << " ";

// do for every edge (v, u)

for (int u: graph.adjList[v])

{

if (!discovered[u])

{

// mark it as discovered and enqueue it

discovered[u] = true;

q.push(u);

}

}

}

}

int main()

{

// vector of graph edges as per the above diagram

vector<Edge> edges = {

{1, 2}, {1, 3}, {1, 4}, {2, 5}, {2, 6}, {5, 9},

{5, 10}, {4, 7}, {4, 8}, {7, 11}, {7, 12}

// vertex 0, 13, and 14 are single nodes

};

// total number of nodes in the graph (labelled from 0 to 14)

int n = 15;

// build a graph from the given edges

Graph graph(edges, n);

// to keep track of whether a vertex is discovered or not

vector<bool> discovered(n, false);

// Perform BFS traversal from all undiscovered nodes to

// cover all connected components of a graph

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

{

if (discovered[i] == false)

{

// start BFS traversal from vertex `i`

BFS(graph, i, discovered);

}

}

return 0;

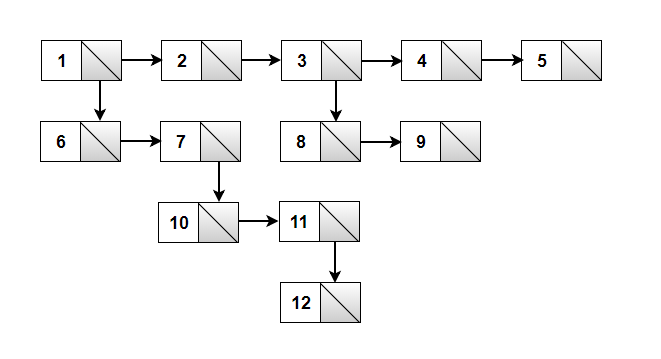
}

# **25, Convert a multilevel linked list to a singly linked list**

Given a multilevel linked list, convert it into a [singly linked list](https://www.techiedelight.com/introduction-linked-lists/) so that all nodes of the first level appear first, followed by all nodes of the second level, and so on.

The multilevel linked list is similar to the simple linked list, except that it has one extra field that points to that node’s child. The child may point to a separate list altogether, which may have children of its own.

For example, consider the following multilevel linked list:



We should convert it into list 1—>2—>3—>4—>5—>6—>7—>8—>9—>10—>11—>12—>null.

The idea is to use the [queue data structure](https://www.techiedelight.com/queue-implementation-cpp/) to solve this problem. We start by traversing the list horizontally from the head node using the next pointer, and whenever a node with a child is found, insert the child node into a queue. If the end of the list is reached, dequeue the front node, set it as the next node of the last encountered node, and repeat the entire process till the queue becomes empty.

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

// A Linked List Node

struct Node

{

int data;

Node \*next, \*child;

};

// Helper function to create a new node with the given data and

// pushes it onto the list's front

void push(Node\* &headRef, int data)

{

Node\* newNode = new Node;

newNode->data = data;

newNode->next = headRef;

newNode->child = nullptr;

headRef = newNode;

}

// Helper function to create a linked list with elements of a given vector

Node\* createHorizontalList(vector<int> const &input)

{

Node\* head = nullptr;

for (int i = input.size() - 1; i >= 0; i--) {

push(head, input[i]);

}

return head;

}

// Function to convert a multilevel linked list into a singly linked list

void convertList(Node\* const head)

{

Node\* curr = head;

queue<Node\*> q;

// process all nodes

while (curr)

{

// last node is reached

if (curr->next == nullptr)

{

// dequeue the front node and set it as the next node of the current node

curr->next = q.front();

q.pop();

}

// if the current node has a child

if (curr->child) {

q.push(curr->child);

}

// advance the current node

curr = curr->next;

}

}

// Helper function to print a given linked list

void printList(Node\* const head)

{

Node\* ptr = head;

while (ptr)

{

cout << ptr->data << " —> ";

ptr = ptr->next;

}

cout << "nullptr" << endl;

}

int main()

{

// create a multilevel linked list

Node\* head = createHorizontalList({1, 2, 3, 4, 5});

head->child = createHorizontalList({6, 7});

head->next->next->child = createHorizontalList({8, 9});

head->child->next->child = createHorizontalList({10, 11});

head->child->next->child->child = createHorizontalList({12});

convertList(head);

printList(head);

return 0;

}

# **26, Find the shortest distance of every cell from a landmine inside a maze**

Given a [maze](https://www.techiedelight.com/maze-problems-in-data-structures/) in the form of a rectangular matrix, filled with either O, X, or M, where O represents an open cell, X represents a blocked cell, and M represents landmines in the maze, find the shortest distance of every open cell in the maze from its nearest mine.

We are only allowed to travel either of the four directions, and diagonal moves are not allowed. We can assume cells with the mine have distance 0. Also, blocked and unreachable cells have distance -1.

For example,

**Input:** 6 × 5 matrix filled with O (Open cell), X (Blocked Cell), and M (Landmine).

O M O O X

O X X O M

O O O O O

O X X X O

O O M O O

O X X M O

**Output:**

1 0 1 2 -1

2 -1 -1 1 0

3 4 3 2 1

3 -1 -1 -1 2

2 1 0 1 2

3 -1 -1 0 1

[Practice this problem](https://techiedelight.com/practice/?problem=ShortestPathIII)

The idea is to perform a [BFS](https://www.techiedelight.com/breadth-first-search/) to solve this problem. Start by creating an empty [queue](https://www.techiedelight.com/circular-queue-implementation-c/) and enqueue all cells with the mines. Then loop through the queue and consider each of four adjacent cells of the front cell. Enqueue the adjacent cell (with updated distance) if it represents an open space, and its distance from the mine is yet to be calculated. Repeat the procedure till the queue is empty.

#include <iostream>

#include <vector>

#include <queue>

#include <iomanip>

using namespace std;

// A Queue Node

struct Node

{

int x; // stores x–coordinate of a matrix cell

int y; // stores y–coordinate of a matrix cell

int distance; // stores the distance of (x, y) from mine

};

// check if specified row and column are valid matrix index

bool isValid(int i, int j, int M, int N) {

return (i >= 0 && i < M) && (j >= 0 && j < N);

}

// check if the current cell is an open area, and its

// distance from the mine is not yet calculated

bool isSafe(int i, int j, vector<vector<char>> const &mat,

vector<vector<int>> const &result) {

return mat[i][j] == 'O' && result[i][j] == -1;

}

// Replace all O's in a matrix with their shortest distance

// from the nearest mine

vector<vector<int>> updateShortestDistance(vector<vector<char>> const &mat)

{

// base case

if (mat.size() == 0) {

return {};

}

// `M × N` matrix

int M = mat.size();

int N = mat[0].size();

vector<vector<int>> result(M, vector<int>(N));

// initialize an empty queue

queue<Node> q;

// find all mines location and add them to the queue

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

{

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

{

// if the current cell represents a mine

if (mat[i][j] == 'M')

{

q.push({i, j, 0});

// update mine distance as 0

result[i][j] = 0;

}

// otherwise, initialize the mine distance by -1

else {

result[i][j] = -1;

}

}

}

// arrays to get indices of four adjacent cells of a given cell

int row[] = { 0, -1, 0, 1 };

int col[] = { -1, 0, 1, 0 };

// do for each node in the queue

while (!q.empty())

{

// process front cell in the queue

int x = q.front().x;

int y = q.front().y;

int distance = q.front().distance;

// dequeue front cell

q.pop();

// update the four adjacent cells of the front node in the queue

for (int i = 0; i < 4; i++)

{

// enqueue adjacent cell if it is valid, unvisited,

// and has a path through it

if (isValid(x + row[i], y + col[i], M, N) &&

isSafe(x + row[i], y + col[i], mat, result))

{

result[x + row[i]][y + col[i]] = distance + 1;

q.push({x + row[i], y + col[i], distance + 1});

}

}

}

return result;

}

// Utility function to print a matrix

void printMatrix(vector<vector<int>> const &mat)

{

for (int i = 0; i < mat.size(); i++)

{

for (int j = 0; j < mat[0].size(); j++) {

cout << setw(3) << mat[i][j];

}

cout << endl;

}

}

int main()

{

vector<vector<char>> mat =

{

{'O', 'M', 'O', 'O', 'X'},

{'O', 'X', 'X', 'O', 'M'},

{'O', 'O', 'O', 'O', 'O'},

{'O', 'X', 'X', 'X', 'O'},

{'O', 'O', 'M', 'O', 'O'},

{'O', 'X', 'X', 'M', 'O'}

};

vector<vector<int>> output = updateShortestDistance(mat);

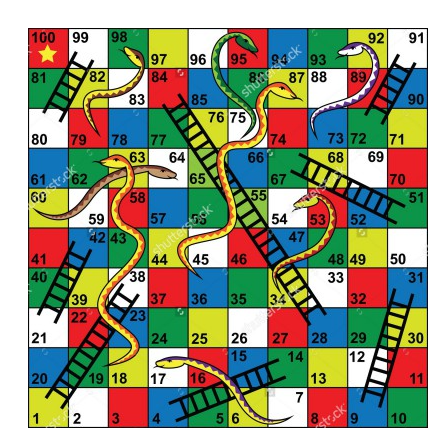
printMatrix(output);

return 0;

}

# **27, Snake and Ladder Problem**

Find the minimum number of throws required to win a given Snakes and Ladders board game. For example, the following game requires at least 7 dice throws to win:



The idea is to consider the snakes and ladders board as a directed graph and run [Breadth–first search (BFS)](https://www.techiedelight.com/breadth-first-search/) from the starting node, vertex 0, as per game rules. We construct a directed graph, keeping in mind the following conditions:

1. For any vertex in graph v, we have an edge from v to v+1, v+2, v+3, v+4, v+5, v+6 as we can reach any of these nodes in one throw of dice from node v.
2. If any of these neighbors of v has a ladder or snake, which takes us to position x, then x becomes the neighbor instead of the base of the ladder or head of the snake.

Now the problem is reduced to finding the shortest path between two nodes in a directed graph problem. We represent the snakes and ladders board using a map.

#include <iostream>

#include <queue>

#include <unordered\_map>

using namespace std;

// Total number of vertices in the graph

// 10 x 10 board

#define N 100

// Data structure to store a graph edge

struct Edge {

int src, dest;

};

// A class to represent a graph object

class Graph

{

public:

// a vector of vectors to represent an adjacency list

vector<int> adjList[N + 1];

// Constructor

Graph(vector<Edge> const &edges)

{

// add edges to the graph

for (Edge edge: edges)

{

// Please note that the graph is directed

adjList[edge.src].push\_back(edge.dest);

}

}

};

// A queue node

struct Node

{

// stores number associated with graph node

int ver;

// `min\_dist` stores the minimum distance of a node from the starting vertex

int min\_dist;

};

// Perform BFS on graph `g` starting from a given source vertex

int BFS(Graph const &g, int source)

{

// create a queue for doing BFS

queue<Node> q;

// to keep track of whether a vertex is discovered or not

vector<bool> discovered(N + 1);

// mark the source vertex as discovered

discovered[source] = true;

// assign the minimum distance of the source vertex as 0 and

// enqueue it

Node node = { source, 0 };

q.push(node);

// loop till queue is empty

while (!q.empty())

{

// dequeue front node

node = q.front();

q.pop();

// Stop BFS if the last node is reached

if (node.ver == N) {

return node.min\_dist;

}

// do for every adjacent node of the current node

for (int u: g.adjList[node.ver])

{

if (!discovered[u])

{

// mark it as discovered and enqueue it

discovered[u] = true;

// assign the minimum distance of the current node

// one more than the minimum distance of the parent node

Node n = {u, node.min\_dist + 1};

q.push(n);

}

}

}

}

int findMinimumMoves(unordered\_map<int, int> &ladder, unordered\_map<int, int> &snake)

{

// find all edges involved and store them in a vector

vector<Edge> edges;

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

{

for (int j = 1; j <= 6 && i + j <= N; j++)

{

int src = i;

// update destination if there is any ladder

// or snake from the current position.

int dest = (ladder[i + j] || snake[i + j]) ?

(ladder[i + j] + snake[i + j]) : (i + j);

// add an edge from src to dest

Edge e = { src, dest };

edges.push\_back(e);

}

}

// construct a directed graph

Graph g(edges);

// Find the shortest path between 1 and 100 using BFS

return BFS(g, 0);

}

int main()

{

// snakes and ladders are represented using a map

unordered\_map<int, int> ladder, snake;

// insert ladders into the map

ladder[1] = 38;

ladder[4] = 14;

ladder[9] = 31;

ladder[21] = 42;

ladder[28] = 84;

ladder[51] = 67;

ladder[72] = 91;

ladder[80] = 99;

// insert snakes into the map

snake[17] = 7;

snake[54] = 34;

snake[62] = 19;

snake[64] = 60;

snake[87] = 36;

snake[93] = 73;

snake[95] = 75;

snake[98] = 79;

cout << findMinimumMoves(ladder, snake);

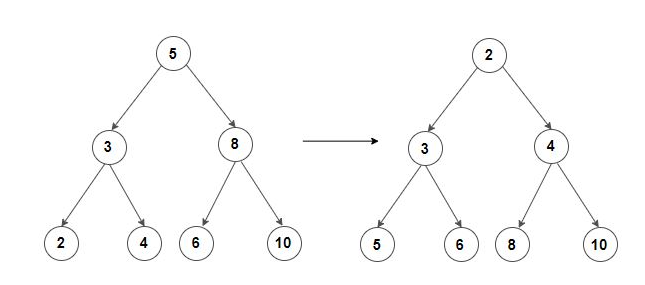
return 0;

}

# **28, Convert a Binary Search Tree into a Min Heap**

Given a binary search tree (BST), efficiently convert it into a min-heap. In order words, convert a binary search tree into a complete binary tree where each node has a higher value than its parent’s value.

For example, the solution should convert the BST on the left into the binary tree on the right, or any other binary tree with the same set of keys that satisfies the [structural and heap-ordering property](https://www.techiedelight.com/introduction-priority-queues-using-binary-heaps/#Heap) of min-heap data structure.



If the given BST is already a complete binary tree, the min-heap’s structural property is already satisfied, and we need to take care of the only heap-ordering property of the min-heap. Basically, we need to ensure that each node’s value is greater than its parent’s value, with the minimum element present at the root.

The idea is to traverse the binary search tree in an [inorder fashion](https://www.techiedelight.com/inorder-tree-traversal-iterative-recursive/) and enqueue all encountered keys. Then traverse the tree in a [preorder fashion](https://www.techiedelight.com/preorder-tree-traversal-iterative-recursive/) and for each encountered node, dequeue a key and assign it to the node.

Following is the implementation of the above algorithm in C++, Java, and Python. The logic works since the nodes get visited in the increasing order of their keys inorder traversal. The preorder traversal ensures that each node in the binary tree has a value greater than its parent’s.

#include <iostream>

#include <vector>

#include <queue>

#include <string>

#include <utility>

using namespace std;

// Data structure to store a binary tree node

struct Node

{

int data;

Node\* left = nullptr, \*right = nullptr;

Node() {}

Node(int data): data(data) {}

};

// Recursive function to insert a key into a BST

Node\* insert(Node\* root, int key)

{

// if the root is null, create a new node and return it

if (root == nullptr) {

return new Node(key);

}

// if the given key is less than the root node, recur for the left subtree

if (key < root->data) {

root->left = insert(root->left, key);

}

// if the given key is more than the root node, recur for the right subtree

else {

root->right = insert(root->right, key);

}

return root;

}

// Helper function to perform level order traversal on a binary tree

void printLevelOrderTraversal(Node\* root)

{

// base case: empty tree

if (root == nullptr) {

return;

}

queue<Node\*> q;

q.push(root);

while (!q.empty())

{

int n = q.size();

while (n--)

{

Node\* front = q.front();

q.pop();

cout << front->data << ' ';

if (front->left) {

q.push(front->left);

}

if (front->right) {

q.push(front->right);

}

}

cout << endl;

}

}

// Function to perform inorder traversal on a given binary tree and

// enqueue all nodes (in encountered order)

void inorder(Node\* root, queue<int> &keys)

{

if (root == nullptr) {

return;

}

inorder(root->left, keys);

keys.push(root->data);

inorder(root->right, keys);

}

// Function to perform preorder traversal on a given binary tree.

// Assign each encountered node with the next key from the queue

void preorder(Node\* root, queue<int> &keys)

{

// base case: empty tree

if (root == nullptr) {

return;

}

// replace the root's key value with the next key from the queue

root->data = keys.front();

keys.pop();

// process left subtree

preorder(root->left, keys);

// process right subtree

preorder(root->right, keys);

}

// Function to convert a BST into a min-heap

void convert(Node\* root)

{

// base case

if (root == nullptr) {

return;

}

// maintain a queue to store inorder traversal on the tree

queue<int> keys;

// fill the queue in an inorder fashion

inorder(root, keys);

// traverse tree in preorder fashion, and for each encountered node,

// dequeue a key and assign it to the node

preorder(root, keys);

}

int main()

{

vector<int> keys = { 5, 3, 2, 4, 8, 6, 10 };

/\* Construct the following BST

5

/ \

/ \

3 8

/ \ / \

/ \ / \

2 4 6 10

\*/

Node\* root = nullptr;

for (int key: keys) {

root = insert(root, key);

}

convert(root);

printLevelOrderTraversal(root);

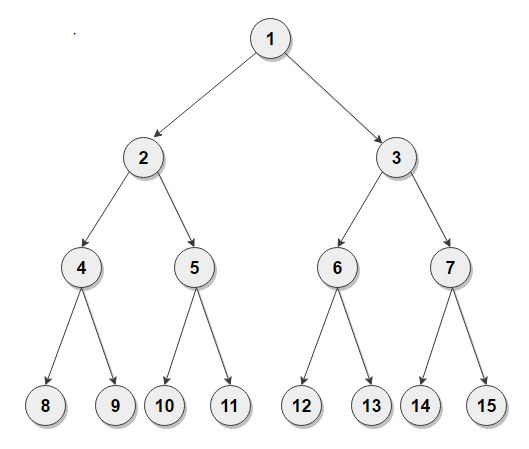
return 0;

}

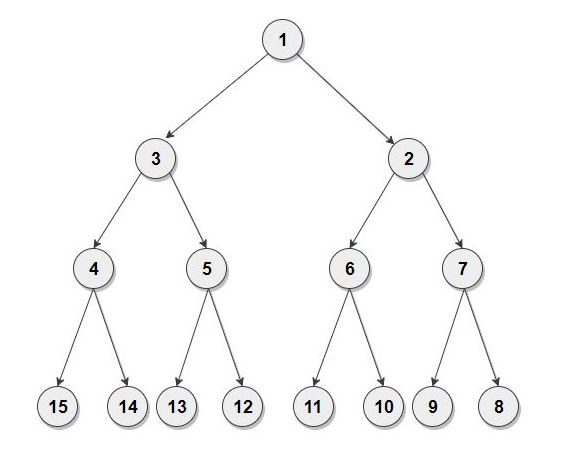
# **29, Invert alternate levels of a perfect binary tree**

Write an efficient algorithm to invert alternate levels of a perfect binary tree.

For example, consider the following tree:



We should convert it into the following tree:



The idea is to perform a [level order traversal](https://www.techiedelight.com/level-order-traversal-binary-tree/) of the perfect binary tree and traverse its nodes level-by-level. Then for each odd level, push all nodes present in that level into a [stack](https://www.techiedelight.com/stack-implementation/). Finally, at the end of each odd level, we put nodes present in the stack into their correct position. Following is the C++, Java, and Python program that demonstrates it:

#include <iostream>

#include <vector>

#include <algorithm>

#include <string>

#include <utility>

#include <queue>

#include <stack>

using namespace std;

// Data structure to store a binary tree node

struct Node

{

int data;

Node \*left, \*right;

Node(int data)

{

this->data = data;

this->left = this->right = nullptr;

}

};

// Function to print level order traversal of a perfect binary tree

void levelOrderTraversal(Node\* root)

{

if (root == nullptr) {

return;

}

// create an empty queue and enqueue the root node

queue<Node\*> queue;

queue.push(root);

// pointer to store the current node

Node\* curr = nullptr;

// loop till queue is empty

while (queue.size())

{

// process each node in the queue and enqueue their

// non-empty left and right child

curr = queue.front();

queue.pop();

cout << curr->data << " ";

if (curr->left) {

queue.push(curr->left);

}

if (curr->right) {

queue.push(curr->right);

}

}

}

// Iterative function to invert alternate levels of a perfect binary tree

// using level order traversal

void invertBinaryTree(Node\* root)

{

// base case: if the tree is empty

if (root == nullptr) {

return;

}

// maintain a queue and enqueue the root node

queue<Node\*> q;

q.push(root);

// to store current level information

bool level = false;

// maintain another queue to store nodes present at an odd level

queue<Node\*> level\_nodes;

// maintain a stack to store node's data on an odd level

stack<int> level\_data;

// loop till queue is empty

while (!q.empty())

{

// get the size of the current level

int n = q.size();

// process all nodes present at the current level

while (n--)

{

// dequeue front node

Node\* curr = q.front();

q.pop();

// if the level is odd

if (level)

{

// enqueue current node

level\_nodes.push(curr);

// push the current node data into the stack

level\_data.push(curr->data);

}

// if the current node is the last node of the level

if (n == 0)

{

// flip the level

level = !level;

// put elements present in the `level\_data` into their correct

// position using `level\_nodes`

while (!level\_nodes.empty())

{

Node\* front = level\_nodes.front();

front->data = level\_data.top();

level\_nodes.pop();

level\_data.pop();

}

}

// enqueue left child of the current node

if (curr->left) {

q.push(curr->left);

}

// enqueue right child of the current node

if (curr->right) {

q.push(curr->right);

}

}

}

}

int main()

{

/\* Construct the following tree

1

/ \

/ \

2 3

/ \ / \

4 5 6 7

/ \ / \ / \ / \

8 9 10 11 12 13 14 15

\*/

Node\* root = new Node(1);

root->left = new Node(2);

root->right = new Node(3);

root->left->left = new Node(4);

root->left->right = new Node(5);

root->right->left = new Node(6);

root->right->right = new Node(7);

root->left->left->left = new Node(8);

root->left->left->right = new Node(9);

root->left->right->left = new Node(10);

root->left->right->right = new Node(11);

root->right->left->left = new Node(12);

root->right->left->right = new Node(13);

root->right->right->left = new Node(14);

root->right->right->right = new Node(15);

invertBinaryTree(root);

levelOrderTraversal(root);

return 0;

}

# **30, Compute modulus division without division and modulo operator**

Compute modulus division of a positive number n by another positive number d, which is a power of 2, without using division or modulo operator.

In order words, compute n % d without using / and % operators, where d is one of 1, 2, 4, 8, 16, 32, …. Since d is a power of 2, d can be represented as 1 << s, where s is a positive number.

We can use the expression n & (d-1) to compute the value of the expression n % d when n and d are both positive and d is a power of 2.

How this works?

The expression d-1 has all the bits flipped (0 to 1) after the set bit of d (which is also flipped from 1 to 0), i.e.,

00001000 (d = 8)

00000111 (d-1 = 7)

So, the expression n & (d-1) converts all left bits of n starting from the i'th bit to 0 and leave bits from 0 to i-1 as it is. Here, i is the position of the only set bit in d. For example,

00010010 & (n = 18)

00000111 (d-1 = 7)

~~~~~~~~

00000010 (n % d)

This method is demonstrated below in C:

| #include <stdio.h>    // Function to compute `n % d` without using division and modulo operator  unsigned int compute(const unsigned int n, const unsigned int d) {  return n & (d - 1);  }    int main()  {  const unsigned int n = 18;    const unsigned int s = 3;  const unsigned int d = 1U << s; // So, `d` is one of 1, 2, 4, 8, 16, 32, …    unsigned int m; // `m` will be `n % d`  m = compute(n, d);    printf("%d %% %d = %d", n, d, m);    return 0;  }  **Output:**  18 % 8 = 2    Note that if d is not a power of 2, then to compute n % d, do repeated subtractions until we get the remainder, as demonstrated below in C:  #include <stdio.h>    // Function to compute `n % d` without using division and modulo operator  int compute(int n, int d)  {  while (n > 0) {  n = n - d;  }    return n + d;  }    int main()  {  const unsigned int n = 38;  const unsigned int d = 7;    unsigned int m = compute(n, d);  printf("%d %% %d = %d", n, d, m);    return 0;  } **31, Generate random input from an array according to given probabilities** Write an algorithm to generate any one of the given n numbers according to given probabilities.  For example, consider the following integer array and their probabilities. The solution should return 1 with 30% probability, 2 with 10% probability, 3 with 20% probability, and so on for every array element.  nums[] = { 1, 2, 3, 4, 5 };  probability[] = { 30, 10, 20, 15, 25 }; // total probability should sum to 100%  [Practice this problem](https://techiedelight.com/practice/?problem=ProbabilityVI)  Algorithm:   1. Construct a sum array S[] from the given probability array P[], where S[i] holds the sum of all P[j] for 0 <= j <= i. 2. Generate a random integer from 1 to 100 and check where it lies in S[]. 3. Based on the comparison result, return the corresponding element from the input array.  | #include <iostream>  #include <vector>  #include <unordered\_map>  #include <cstdlib>  #include <ctime>  using namespace std;    // Function to generate random nums from a vector according to the given probabilities  int random(vector<int> const &nums, vector<int> const &probability)  {  int n = nums.size();  if (n != probability.size()) {  return -1; // error  }    // construct a sum vector from the given probabilities  vector<int> prob\_sum(n, 0);    // `prob\_sum[i]` holds sum of all `probability[j]` for `0 <= j <=i`  prob\_sum[0] = probability[0];  for (int i = 1; i < n; i++) {  prob\_sum[i] = prob\_sum.at(i-1) + probability[i];  }    // generate a random integer from 1 to 100 and check where it lies in `prob\_sum[]`  int r = (rand() % 100) + 1;    // based on the comparison result, return the corresponding  // element from the nums vector    if (r <= prob\_sum[0]) { // handle 0th index separately  return nums[0];  }    for (int i = 1; i < n; i++)  {  if (r > prob\_sum.at(i-1) && r <= prob\_sum[i]) {  return nums[i];  }  }  }    int main()  {  // Input: vector of integers and their probabilities  // Goal: generate `nums[i]` with probability equal to `probability[i]`    vector<int> nums = {1, 2, 3, 4, 5};  vector<int> probability = {30, 10, 20, 15, 25};    // initialize srand with a distinctive value  srand(time(NULL));    // maintain a frequency map to validate the results  unordered\_map<int, int> freq;    // make 1000000 calls to the `random()` function and  // store results in a map  for (int i = 0; i < 1000000; i++)  {  int val = random(nums, probability);  freq[val]++;  }    // print the results  for (int i = 0; i < nums.size(); i++) {  cout << nums[i] << " ~ " << freq[nums[i]]/10000.0 << "%" << endl;  }    return 0;  } **32, Get 0 and 1 with equal probability using a specified function** Write an algorithm to get 0 and 1 with equal probability using a function that generates random numbers from 1 to 5 with equal probability.  The algorithm can be implemented as follows in C (self-explanatory):  #include <stdio.h>  #include <stdlib.h>  #include <time.h>    // Function to generate a random number from 1 to 5 with equal probability  int random() {  return (rand() % 5) + 1;  }    // Returns 0 or 1 with equal probability using `random()` function  int generate()  {  int r;    do {  // `r` could be any one of 1, 2, 3, 4, and 5  r = random();  } while (r == 5);    // `r` could any of 1, 2, 3, 4 now    // since there are 2 odd and 2 even numbers, return the last bit of `r`,  // which could be 0 or 1 with equal probability  return r & 1;  }    int main(void)  {  srand(time(NULL));    int x = 0, y = 0;    // make 10000 calls to `generate()`  for (int i = 1; i <= 10000; i++) {  generate()? x++: y++;  }    // print the results  printf("0 ~ %0.2f%\n", x/100.0);  printf("1 ~ %0.2f%\n", y/100.0);    return 0;  }  **Output (will vary):**  0 ~ 50.23%  1 ~ 49.77%    We can also do something like below, but this will increase the number of calls made to the random() function:   | 1  2  3  4  5  6  7  8  9  10  11  12  13 | int generate()  {  int r;    do {  // `r` could be any one of 1, 2, 3, 4, and 5  r = random();  } while (r > 2);    // `r` could be 1 or 2 now    return r - 1;  } | | --- | --- |  **33, Determine if two integers are equal without using comparison and arithmetic operators** This post will discuss how to determine whether two integers are equal without using comparison operators (==, !=, <, >, <=, >=) and arithmetic operators (+, -, \*, /, %).  The simplest solution is to use the bitwise XOR operator. We know that for equal numbers, the XOR operator returns 0.  #include <stdio.h>    // Determine if two integers are equal without using comparison  // and arithmetic operators  int checkForEquality(int x, int y) {  return !(x ^ y);  }    int main(void)  {  int x = 10, y = 10;    if (checkForEquality(x, y)) {  printf ("x=%d is equal to y=%d\n", x, y);  }  else {  printf ("x=%d is not equal to y=%d\n", x, y);  }    return 0;  } **34, Implement ternary operator without using conditional expressions** This post will implement a ternary-like operator in C without using conditional expressions like ternary operator, if–else expression, or switch-case statements.  The solution should implement the condition x ? a : b.  If x = 1, a is returned; if x = 0, b should be returned.  The idea is to use the expression x × a + !x × b or x × a + (1 - x) × b.  Let’s consider the first expression x × a + !x × b:   * For x = 1, the expression reduces to (1 × a) + (!1 × b) = a. * For x = 0, the expression reduces to (0 × a) + (!0 × b) = b.   The following C program demonstrates it:  #include <stdio.h>    // Function to return the result of the expression (x ? a : b)  int ternary(int x, int a, int b) {  return x \* a + !x \* b;  }    // Implement a ternary operator without using conditional expressions  int main(void)  {  int a = 10, b = 20;    printf ("%d\n", ternary(0, a, b));  printf ("%d\n", ternary(1, a, b));    return 0;  } **35, Generate numbers from 1 to 7 with equal probability using a specified function** Write an algorithm to generate numbers from 1 to 7 with equal probability using a specified function that produces random numbers between 1 and 5.  [Practice this problem](https://techiedelight.com/practice/?problem=ProbabilityIV)  Suppose the specified function is random(), which generates random numbers from 1 to 5 with equal probability. The idea is to use the expression 5 × (random() - 1) + random() which uniformly produces random numbers in the range [1–25]. So if we exclude the possibility of the random number being one among [8–25] by repeating the procedure, we are left with numbers between 1 and 7 having equivalent probability.  How this works?  Since random() returns random numbers from 1 to 5 with equal probability, R = 5 × (random() - 1) can be any of 0, 5, 10, 15 or 20. Now for the second random() call, let’s explore all possibilities:  If R = 0, R + random() can be any of 1, 2, 3, 4, 5  If R = 5, R + random() can be any of 6, 7, 8, 9, 10  If R = 10, R + random() can be any of 11, 12, 13, 14, 15  If R = 15, R + random() can be any of 15, 16, 17, 18, 19, 20  If R = 20, R + random() can be any of 21, 22, 23, 24, 25  So, the expression uniformly distributes the numbers in the range  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #include <string.h>    // Function to generate a random number from 1 to 5 with equal probability  int random() {  return (rand() % 5) + 1;  }    // Function to generate a random number from 1 to 7 with equal probability  // by using the specified function  int generate()  {  int r;  do {  r = 5 \* (random() - 1) + random();  } while (r > 7);    return r;  }    int main(void)  {  // initialize srand with a distinctive value  srand(time(NULL));    // count array to validate the results  int count[8];  memset(count, 0, sizeof count);    // make 1000000 calls to the `generate()` function and store the results  for (int i = 1; i <= 1000000; i++)  {  int val = generate();  count[val]++;  }    // print the results  for (int i = 1; i <= 7; i++) {  printf("%d ~ %0.2f%\n", i, count[i]/10000.0);  }    return 0;  } **36, Merge `M` sorted lists each containing `N` elements** Given m sorted lists, each containing n elements, print them efficiently in sorted order.  For example,  **Input:** 5 sorted lists of fixed size 4    [10, 20, 30, 40]  [15, 25, 35, 45]  [27, 29, 37, 48]  [32, 33, 39, 50]  [16, 18, 22, 28]    **Output:**    [10, 15, 16, 18, 20, 22, 25, 27, 28, 29, 30, 32, 33, 35, 37, 39, 40, 45, 48, 50]  A simple solution would be to create an auxiliary array containing all lists’ elements (order doesn’t matter). Then use an efficient sorting algorithm to [sort the array](https://www.techiedelight.com/sort-array-ascending-order-cpp/) in ascending order and print the elements. The worst-case time complexity of this approach will be O(m.n.log(m.n)). Also, this approach does not take advantage of the fact that each list is already sorted.    We can easily solve this problem in O(m.n.log(m)) time by using a [min-heap](https://www.techiedelight.com/introduction-priority-queues-using-binary-heaps/#Heap). The idea is to construct a min-heap of size m and insert the first element of each list in it. Then, pop the root element (minimum element) from the heap and insert the next element from the “same” list as the popped element. Repeat this process till the heap is exhausted. Depending upon the requirement, either print the popped element or store it in an auxiliary array.  #include <iostream>  #include <vector>  #include <queue>  using namespace std;    // Data structure to store a heap node  struct Node  {  // `val` stores the element,  // `i` stores the list number of the element  // `index` stores the column number of the i'th list from which element was taken  int val, i, index;  };    // Comparison object to be used to order the min-heap  struct comp  {  bool operator()(const Node &lhs, const Node &rhs) const {  return lhs.val > rhs.val;  }  };    // Function to merge `M` sorted lists each of size `N` and  // print them in ascending order  void printSorted(vector<vector<int>> lists)  {  // create an empty min-heap  priority\_queue<Node, vector<Node>, comp> pq;    // push the first element of each list into the min-heap  // along with the list number and their index in the list  for (int i = 0; i < lists.size(); i++) {  pq.push({lists[i][0], i, 0});  }    // run till min-heap is empty  while (!pq.empty())  {  // extract the minimum node from the min-heap  Node min = pq.top();  pq.pop();    // print the minimum element  cout << min.val << " ";    // take the next element from the "same" list and  // insert it into the min-heap  if (min.index + 1 < lists[min.i].size())  {  min.index += 1;  min.val = lists[min.i][min.index];  pq.push(min);  }  }  }    int main()  {  // `M` lists of size `N`, each in the form of a 2D-matrix  vector<vector<int>> lists =  {  { 10, 20, 30, 40 },  { 15, 25, 35, 45 },  { 27, 29, 37, 48 },  { 32, 33, 39, 50 },  { 16, 18, 22, 28 }  };    printSorted(lists);    return 0;  } **37, Count all paths in a matrix from the first cell to the last cell** Given an M × N rectangular grid, efficiently count all paths starting from the first cell (0, 0) to the last cell (M-1, N-1). We can either move down or move towards right from a cell.  For example,  **Input:**  3 × 3 matrix    **Output:** Total number of paths are 6    (0, 0) —> (0, 1) —> (0, 2) —> (1, 2) —> (2, 2)  (0, 0) —> (0, 1) —> (1, 1) —> (1, 2) —> (2, 2)  (0, 0) —> (0, 1) —> (1, 1) —> (2, 1) —> (2, 2)  (0, 0) —> (1, 0) —> (2, 0) —> (2, 1) —> (2, 2)  (0, 0) —> (1, 0) —> (1, 1) —> (1, 2) —> (2, 2)  (0, 0) —> (1, 0) —> (1, 1) —> (2, 1) —> (2, 2)  [Practice this problem](https://techiedelight.com/practice/?problem=MatrixPathVI)  The idea is to start from the top-left corner of the matrix and recur for the next cell, which can be either the immediate right cell or the immediate bottom cell.  **#include <stdio.h>**    **// Top-down recursive function to count all paths from cell (m, n)**  **// to the last cell (M-1, N-1) in a given `M × N` rectangular grid**  **int countPaths(int m, int n, int M, int N)**  **{**  **// there is only one way to reach the last cell**  **// when we are at the last row or the last column**  **if (m == M - 1 || n == N - 1) {**  **return 1;**  **}**    **return countPaths(m + 1, n, M, N) // move down**  **+ countPaths(m, n + 1, M, N); // move right**  **}**    **int main(void)**  **{**  **// `M × N` matrix**  **int M = 3;**  **int N = 3;**    **int k = countPaths(0, 0, M, N);**  **printf("The total number of paths is %d", k);**    **return 0;**  **}** **38, Calculate the minimum cost to reach the destination city from the source city** **Given an N × N matrix of non-negative integers, where each cell of the matrix (i, j) indicates the direct flight cost from the city i to city j. Find the minimum cost to reach the destination city N-1 from the source city 0.**  **For example,**  **Input: The cost matrix for 4 cities:**    **[ 0 20 30 100 ]**  **[ 20 0 15 75 ]**  **[ 30 15 0 50 ]**  **[ 100 75 50 0 ]**    **Output: The minimum cost is 80.**  **The minimum cost path is:**  **city 0 —> city 2 (cost = 30)**  **city 2 —> city 3 (cost = 50)**  **Input: The cost matrix for 5 cities:**  **[ 0 25 20 10 105 ]**  **[ 20 0 15 80 80 ]**  **[ 30 15 0 70 90 ]**  **[ 10 10 50 0 100 ]**  **[ 40 50 5 10 0 ]**  **Output: The minimum cost is 100.**  **The minimum cost path is:**  **city 0 —> city 3 (cost = 10)**  **city 3 —> city 1 (cost = 10)**  **city 1 —> city 4 (cost = 80)**  [**Practice this problem**](https://techiedelight.com/practice/?problem=MinimumCostPathII)  **The idea is to recur for all cities reachable from the source city and consider their minimum cost. The recurrence relation T(0) can be written as:**  **T(n) = minimum { T(i) + cost[i][n] }**  **for all cities i between source city 0 and destination city n**  **i.e., the minimum cost C(0, n) to reach city n from city 0 is**  **C(0, n) = minimum(cost[0][n],**  **C(0, 1) + cost[1][n],**  **C(0, 2) + cost[2][n],**  **…**  **C(0, n-1) + cost[n-1][n]**  **)**  **The time complexity of this solution would be exponential since we might end up computing the same subproblem repeatedly. We can use** [**dynamic programming**](https://www.techiedelight.com/dynamic-programming-interview-questions/) **to optimize the code since this problem exhibits both properties of dynamic programming, i.e.,** [**overlapping subproblems**](https://www.techiedelight.com/introduction-dynamic-programming/#overlapping-subproblems) **and** [**optimal substructure**](https://www.techiedelight.com/introduction-dynamic-programming/#optimal-substructure)**.**    **The idea is to construct an auxiliary array lookup[] for storing the subproblem solutions where each element lookup[i] of the lookup table stores the minimum cost to reach city i from city 0.**  **#include <iostream>**  **#include <vector>**  **#include <algorithm>**  **#include <climits>**  **using namespace std;**    **// DP function to calculate the minimum cost to reach the destination city `n`**  **// from the source city 0**  **int findMinCost(vector<vector<int>> const &cost)**  **{**  **// base case**  **if (cost.size() == 0) {**  **return 0;**  **}**    **// `N × N` matrix**  **int N = cost.size();**    **// `lookup[i]` stores the minimum cost to reach city `i` from city 0**  **int lookup[N];**    **// Initialize `lookup[]` with the direct ticket price from the source city**  **for (int i = 0; i < N; i++) {**  **lookup[i] = cost[0][i];**  **}**    **// repeat loop till `lookup[]` is filled with all minimum values**  **bool is\_filled = false;**  **while (!is\_filled)**  **{**  **is\_filled = true;**    **// fill `lookup[]` in a bottom-up manner**  **for (int i = 0; i < N; i++)**  **{**  **for (int j = 0; j < N; j++)**  **{**  **if (lookup[i] > lookup[j] + cost[j][i])**  **{**  **lookup[i] = lookup[j] + cost[j][i];**  **is\_filled = false; // mark lookup[] as NOT filled**  **}**  **}**  **}**  **}**    **// return the minimum cost to reach city `N-1` from city 0**  **return lookup[N-1];**    **}**    **int main()**  **{**  **vector<vector<int>> cost =**  **{**  **{ 0, 25, 20, 10, 105 },**  **{ 20, 0, 15, 80, 80 },**  **{ 30, 15, 0, 70, 90 },**  **{ 10, 10, 50, 0, 100 },**  **{ 40, 50, 5, 10, 0 }**  **};**    **cout << "The minimum cost is " << findMinCost(cost) << endl;**    **return 0;**  **}** **39, Find the shortest route in a device to construct a given string** **Given a device having left, right, top, and bottom buttons and an OK button to enter a text from a virtual keypad having alphabets from A–Y arranged in a 5 × 5 grid, as shown below. Find the shortest route in the device to construct a given string if we start from the top-left position in the keypad.**  **For example,**  **Keypad:**    **A B C D E**  **F G H I J**  **K L M N O**  **P Q R S T**  **U V W X Y**    **Device:**    **T**  **L M R**  **B**    **where,**    **T — Move up**  **B — Move down**  **L — Move left**  **R — Move right**  **M — Press OK**  **The shortest route to construct string TECHIE with the device’s help is BBBRRRRMTTTMLLMBMRMTRM.**    **The idea is to consider all characters of the specified string, and for each character, print out the shortest route to the next character from it. For finding the shortest route, compare the coordinates of the current character with the coordinates of the next character in the matrix. Based on the difference between the x–coordinate and y–coordinate of the current and next character, move left, right, top, or bottom.**   | **#include <iostream>**  **using namespace std;**    **// Find the shortest route in a device to construct the given string**  **void printPath(string str)**  **{**  **// start from the top-left corner with coordinates, i.e., (0, 0) cell**  **int x = 0, y = 0;**    **int n = str.length();**  **for (int i = 0; i < n; i++)**  **{**  **// find coordinates of the next character**  **int X = (str[i] - 'A') / 5;**  **int Y = (str[i] - 'A') % 5;**    **// if the next character is above the current character**  **while (x > X)**  **{**  **cout << "T";**  **x--; // Go up**  **}**    **// if the next character is below the current character**  **while (x < X)**  **{**  **cout << "B";**  **x++; // Go down**  **}**    **// if the next character is to the left of the current character**  **while (y > Y)**  **{**  **cout << "L";**  **y--; // Go left**  **}**    **// if the next character is to the right of the current character**  **while (y < Y)**  **{**  **cout << "R";**  **y++; // Go right**  **}**    **// next character is found**  **cout << "M";**  **}**  **}**    **int main()**  **{**  **string str = "TECHIE";**    **printPath(str);**    **return 0;**  **}** | | --- |  **40, Find the shortest distance of every cell from a landmine inside a maze** **Given a** [**maze**](https://www.techiedelight.com/maze-problems-in-data-structures/) **in the form of a rectangular matrix, filled with either O, X, or M, where O represents an open cell, X represents a blocked cell, and M represents landmines in the maze, find the shortest distance of every open cell in the maze from its nearest mine.**  **We are only allowed to travel either of the four directions, and diagonal moves are not allowed. We can assume cells with the mine have distance 0. Also, blocked and unreachable cells have distance -1.**    **For example,**  **Input: 6 × 5 matrix filled with O (Open cell), X (Blocked Cell), and M (Landmine).**    **O M O O X**  **O X X O M**  **O O O O O**  **O X X X O**  **O O M O O**  **O X X M O**    **Output:**    **1 0 1 2 -1**  **2 -1 -1 1 0**  **3 4 3 2 1**  **3 -1 -1 -1 2**  **2 1 0 1 2**  **3 -1 -1 0 1**  [**Practice this problem**](https://techiedelight.com/practice/?problem=ShortestPathIII)  **The idea is to perform a** [**BFS**](https://www.techiedelight.com/breadth-first-search/) **to solve this problem. Start by creating an empty** [**queue**](https://www.techiedelight.com/circular-queue-implementation-c/) **and enqueue all cells with the mines. Then loop through the queue and consider each of four adjacent cells of the front cell. Enqueue the adjacent cell (with updated distance) if it represents an open space, and its distance from the mine is yet to be calculated. Repeat the procedure till the queue is empty.**  **#include <iostream>**  **#include <vector>**  **#include <queue>**  **#include <iomanip>**  **using namespace std;**    **// A Queue Node**  **struct Node**  **{**  **int x; // stores x–coordinate of a matrix cell**  **int y; // stores y–coordinate of a matrix cell**  **int distance; // stores the distance of (x, y) from mine**  **};**    **// check if specified row and column are valid matrix index**  **bool isValid(int i, int j, int M, int N) {**  **return (i >= 0 && i < M) && (j >= 0 && j < N);**  **}**    **// check if the current cell is an open area, and its**  **// distance from the mine is not yet calculated**  **bool isSafe(int i, int j, vector<vector<char>> const &mat,**  **vector<vector<int>> const &result) {**  **return mat[i][j] == 'O' && result[i][j] == -1;**  **}**    **// Replace all O's in a matrix with their shortest distance**  **// from the nearest mine**  **vector<vector<int>> updateShortestDistance(vector<vector<char>> const &mat)**  **{**  **// base case**  **if (mat.size() == 0) {**  **return {};**  **}**    **// `M × N` matrix**  **int M = mat.size();**  **int N = mat[0].size();**    **vector<vector<int>> result(M, vector<int>(N));**    **// initialize an empty queue**  **queue<Node> q;**    **// find all mines location and add them to the queue**  **for (int i = 0; i < M; i++)**  **{**  **for (int j = 0; j < N; j++)**  **{**  **// if the current cell represents a mine**  **if (mat[i][j] == 'M')**  **{**  **q.push({i, j, 0});**    **// update mine distance as 0**  **result[i][j] = 0;**  **}**  **// otherwise, initialize the mine distance by -1**  **else {**  **result[i][j] = -1;**  **}**  **}**  **}**    **// arrays to get indices of four adjacent cells of a given cell**  **int row[] = { 0, -1, 0, 1 };**  **int col[] = { -1, 0, 1, 0 };**    **// do for each node in the queue**  **while (!q.empty())**  **{**  **// process front cell in the queue**  **int x = q.front().x;**  **int y = q.front().y;**  **int distance = q.front().distance;**    **// dequeue front cell**  **q.pop();**    **// update the four adjacent cells of the front node in the queue**  **for (int i = 0; i < 4; i++)**  **{**  **// enqueue adjacent cell if it is valid, unvisited,**  **// and has a path through it**  **if (isValid(x + row[i], y + col[i], M, N) &&**  **isSafe(x + row[i], y + col[i], mat, result))**  **{**  **result[x + row[i]][y + col[i]] = distance + 1;**  **q.push({x + row[i], y + col[i], distance + 1});**  **}**  **}**  **}**    **return result;**  **}**    **// Utility function to print a matrix**  **void printMatrix(vector<vector<int>> const &mat)**  **{**  **for (int i = 0; i < mat.size(); i++)**  **{**  **for (int j = 0; j < mat[0].size(); j++) {**  **cout << setw(3) << mat[i][j];**  **}**  **cout << endl;**  **}**  **}**    **int main()**  **{**  **vector<vector<char>> mat =**  **{**  **{'O', 'M', 'O', 'O', 'X'},**  **{'O', 'X', 'X', 'O', 'M'},**  **{'O', 'O', 'O', 'O', 'O'},**  **{'O', 'X', 'X', 'X', 'O'},**  **{'O', 'O', 'M', 'O', 'O'},**  **{'O', 'X', 'X', 'M', 'O'}**  **};**    **vector<vector<int>> output = updateShortestDistance(mat);**  **printMatrix(output);**    **return 0;** **41, Find maximum value of `M[c][d] – M[a][b]` over all choices of indexes** **Given a square matrix of integers, find the maximum value of M[c][d] - M[a][b] over every choice of indexes such that c > a and d > b in a single traversal of the matrix.**  **For example,**  **Input Matrix:**    **{ 1, 2, -1, -4, -20 }**  **{ -8, -3, 4, 2, 1 }**  **{ 3, 8, 6, 1, 3 }**  **{ -4, -1, 1, 7, -6 }**  **{ 0, -4, 10, -5, 1 }**    **Output: The maximum value is 18 as M[4][2] – M[1][0] has maximum difference.**  [**Practice this problem**](https://techiedelight.com/practice/?problem=MaximumDifferencePairII)  **A naive solution would be to find M[c][d] for all values M[a][b] in the matrix, having the maximum value and satisfies c > a and d > b. We keep track of the maximum value found so far in a variable and finally return the maximum value. The implementation can be seen** [**here**](https://techiedelight.com/compiler/?run=5yYWh1) **and runs in O(N4) time for an N × N matrix.**    **The efficient solution is to use an auxiliary matrix whose index (i, j) will store the value of the maximum element in the input matrix from coordinates (i, j) to (N-1, N-1). We keep track of the maximum value found so far in a variable and finally return the maximum value.**  **#include <iostream>**  **#include <algorithm>**  **#include <vector>**  **#include <climits>**  **using namespace std;**    **// Returns maximum value `M[c][d] - M[a][b]` over every choice of indexes**  **// such that `c > a` and `d > b`**  **int findMax(vector<vector<int>> const &M)**  **{**  **// base case**  **if (M.size() == 0) {**  **return 0;**  **}**    **// get size of the matrix**  **int n = M.size();**    **// `K[i][j]` stores a maximum of elements in the matrix from `(i, j)`**  **// to `(n-1, n-1)`**  **int K[n][n];**    **// last element of `K[][]` will be the same as that of the specified matrix**  **K[n-1][n-1] = M[n-1][n-1];**    **int max = M[n-1][n-1]; // Initialize max**    **// preprocess the last row**  **for (int j = n-2; j >= 0; j--)**  **{**  **if (M[n-1][j] > max) {**  **max = M[n-1][j];**  **}**  **K[n-1][j] = max;**  **}**    **max = M[n-1][n-1]; // Initialize max**    **// preprocess the last column**  **for (int i = n-2; i >= 0; i--)**  **{**  **if (M[i][n-1] > max) {**  **max = M[i][n-1];**  **}**  **K[i][n-1] = max;**  **}**    **max = INT\_MIN; // Initialize max**    **// preprocess the rest of the matrix from the bottom**  **for (int i = n-2; i >= 0; i--)**  **{**  **for (int j = n-2; j >= 0; j--)**  **{**  **// update the max value**  **if (K[i+1][j+1] - M[i][j] > max) {**  **max = K[i+1][j+1] - M[i][j];**  **}**    **// assign `K[i][j]`**  **K[i][j] = std::max(M[i][j], std::max(K[i][j+1], K[i+1][j]));**  **}**  **}**    **return max;**  **}**    **int main()**  **{**  **vector<vector<int>> M =**  **{**  **{ 1, 2, -1, -4, -20 },**  **{ -8, -3, 4, 2, 1 },**  **{ 3, 8, 6, 1, 3 },**  **{ -4, -1, 1, 7, -6 },**  **{ 0, -4, 10, -5, 1 }**  **};**    **cout << "The maximum value is " << findMax(M) << endl;**    **return 0;**  **}** **42, Calculate the size of the largest plus of 1’s in a binary matrix** **Given a square matrix of 0's and 1's, calculate the size of the largest plus formed by 1's.**  **For example, for the matrix below, we have highlighted the largest plus of ones having size 17.**    **We start by creating four auxiliary matrices left[][], right[][], top[][], bottom[][], where left[j][j], right[i][j], top[i][j], and bottom[i][j] store the maximum number of consecutive 1's present at the left, right, top and bottom of cell (i, j) including cell (i, j), respectively by using** [**dynamic programming**](https://www.techiedelight.com/introduction-dynamic-programming/)**:**  **if grid[i][j] == 1**  **left[i][j] = left[i][j – 1] + 1**    **if grid[i][j] == 1**  **top[i][j] = top[i – 1][j] + 1**    **if grid[i][j] == 1**  **bottom[i][j] = bottom[i + 1][j] + 1**    **if grid[i][j] == 1**  **right[i][j] = right[i][j + 1] + 1**  **After calculating the above matrices, find cell (i, j) that has maximum value in each direction (by considering the minimum of left[i][j], right[i][j], top[i][j], bottom[i][j]).**  **#include <stdio.h>**  **#include <string.h>**    **// size of the binary square matrix**  **#define N 10**    **int min(int a, int b) {**  **return (a < b) ? a : b;**  **}**    **int minimum(int a, int b, int c, int d) {**  **return min(min(a, b), min(c, d));**  **}**    **// Calculate the size of the largest `+` formed by 1's**  **int calculateSize(int grid[][N])**  **{**  **// left[j][j] stores the maximum number of consecutive 1's**  **// present at the left of grid[i][j] (including grid[i][j])**  **int left[N][N];**  **memset(left, 0, sizeof left);**    **// right[j][j] stores the maximum number of consecutive 1's**  **// present at the right of grid[i][j] (including grid[i][j])**  **int right[N][N];**  **memset(right, 0, sizeof right);**    **// top[j][j] stores the maximum number of consecutive 1's**  **// present at the top of grid[i][j] (including grid[i][j])**  **int top[N][N];**  **memset(top, 0, sizeof top);**    **// bottom[j][j] stores the maximum number of consecutive 1's**  **// present at the bottom of grid[i][j] (including grid[i][j])**  **int bottom[N][N];**  **memset(bottom, 0, sizeof bottom);**    **// initialize the above matrices**  **for (int i = 0; i < N; i++)**  **{**  **// initialize the first row of the top matrix**  **top[0][i] = grid[0][i];**    **// initialize the last row of the bottom matrix**  **bottom[N - 1][i] = grid[N - 1][i];**    **// initialize the first column of the left matrix**  **left[i][0] = grid[i][0];**    **// initialize the last column of the right matrix**  **right[i][N - 1] = grid[i][N - 1];**  **}**    **// fill all cells of the above matrices**  **for (int i = 0; i < N; i++)**  **{**  **for (int j = 1; j < N; j++)**  **{**  **// fill the left matrix**  **if (grid[i][j] == 1) {**  **left[i][j] = left[i][j - 1] + 1;**  **}**    **// fill the top matrix**  **if (grid[j][i] == 1) {**  **top[j][i] = top[j - 1][i] + 1;**  **}**    **// fill the bottom matrix**  **if (grid[N - 1 - j][i] == 1) {**  **bottom[N - 1 - j][i] = bottom[N - j][i] + 1;**  **}**    **// fill the right matrix**  **if (grid[i][N - 1 - j] == 1) {**  **right[i][N - 1 - j] = right[i][N - j] + 1;**  **}**  **}**  **}**    **// bar` stores the length of the longest `+` found so far**  **int bar = 0;**    **// compute the longest plus**  **for (int i = 0; i < N; i++)**  **{**  **for (int j = 0; j < N; j++)**  **{**  **// find minimum of left(i, j), right(i, j), top(i, j), bottom(i, j)**  **int len = minimum(top[i][j], bottom[i][j], left[i][j], right[i][j]);**    **// largest `+` would be formed by a cell that has a maximum value**  **if (len - 1 > bar) {**  **bar = len - 1;**  **}**  **}**  **}**    **return bar;**  **}**    **int main()**  **{**  **int grid[N][N] =**  **{**  **{ 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 },**  **{ 1, 0, 1, 0, 1, 1, 1, 0, 1, 1 },**  **{ 1, 1, 1, 0, 1, 1, 0, 1, 0, 1 },**  **{ 0, 0, 0, 0, 1, 0, 0, 1, 0, 0 },**  **{ 1, 1, 1, 0, 1, 1, 1, 1, 1, 1 },**  **{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 0 },**  **{ 1, 0, 0, 0, 1, 0, 0, 1, 0, 1 },**  **{ 1, 0, 1, 1, 1, 1, 0, 0, 1, 1 },**  **{ 1, 1, 0, 0, 1, 0, 1, 0, 0, 1 },**  **{ 1, 0, 1, 1, 1, 1, 0, 1, 0, 0 }**  **};**    **int bar = calculateSize(grid);**    **// 4 directions of length `4×bar+1` for a middle cell**  **if (bar) {**  **printf("The largest plus of 1's has a size of %d", 4\*bar + 1);**  **}**    **return 0;**  **}** **43, Find the shortest path in a maze** **Given a** [**maze**](https://www.techiedelight.com/maze-problems-in-data-structures/) **in the form of a binary rectangular matrix, find the shortest path’s length in the maze from a given source to a given destination. The path can only be constructed out of cells having value 1, and at any moment, we can only move one step in one of the four directions.**  **The valid moves are:**  **Go Top: (x, y) ——> (x – 1, y)**  **Go Left: (x, y) ——> (x, y – 1)**  **Go Down: (x, y) ——> (x + 1, y)**  **Go Right: (x, y) ——> (x, y + 1)**    **For example, consider the following binary matrix. If source = (0, 0) and destination = (7, 5), the shortest path from source to destination has length 12.**  **[ 1 1 1 1 1 0 0 1 1 1 ]**  **[ 0 1 1 1 1 1 0 1 0 1 ]**  **[ 0 0 1 0 1 1 1 0 0 1 ]**  **[ 1 0 1 1 1 0 1 1 0 1 ]**  **[ 0 0 0 1 0 0 0 1 0 1 ]**  **[ 1 0 1 1 1 0 0 1 1 0 ]**  **[ 0 0 0 0 1 0 0 1 0 1 ]**  **[ 0 1 1 1 1 1 1 1 0 0 ]**  **[ 1 1 1 1 1 0 0 1 1 1 ]**  **[ 0 0 1 0 0 1 1 0 0 1 ]**  [**Practice this problem**](https://techiedelight.com/practice/?problem=ShortestPath)    **To find the maze’s shortest path, search for all possible paths in the maze from the starting position to the goal position until all possibilities are exhausted. We can easily achieve this with the help of** [**backtracking**](https://www.techiedelight.com/backtracking-interview-questions/)**. The idea is to start from the given source cell in the matrix and explore all four paths possible and recursively check if they will lead to the destination or not. Then update the minimum path length whenever the destination cell is reached. If a path doesn’t reach the destination or explored all possible routes from the current cell, backtrack. To make sure that the path is simple and doesn’t contain any cycles, keep track of cells involved in the current path in a matrix, and before exploring any cell, ignore the cell if it is already covered in the current path.** | | --- | --- | --- | --- |   **44: Find the index of 0 to be replaced to get the maximum length sequence of continuous ones**  **Given a binary array, find the index of 0 to be replaced with 1 to get the maximum length sequence of continuous ones.**  **For example, consider the array { 0, 0, 1, 0, 1, 1, 1, 0, 1, 1 }. We need to replace index 7 to get the continuous sequence of length 6 containing all 1’s.**  **#include <stdio.h>**    **// Find the index of 0 to replace with 1 to get the maximum sequence**  **// of continuous 1's**  **int findIndexofZero(int arr[], int n)**  **{**  **int max\_count = 0; // stores maximum number of 1's (including zero)**  **int max\_index = -1; // stores index of 0 to be replaced**    **int prev\_zero\_index = -1; // stores index of previous zero**  **int count = 0; // stores current count of zeros**    **// consider each index `i` in the array**  **for (int i = 0; i < n; i++)**  **{**  **// if the current element is 1**  **if (arr[i] == 1) {**  **count++;**  **}**  **// if the current element is 0**  **else {**  **// reset count to 1 + number of ones to the left of current 0**  **count = i - prev\_zero\_index;**    **// update `prev\_zero\_index` to the current index**  **prev\_zero\_index = i;**  **}**    **// update maximum count and index of 0 to be replaced if required**  **if (count > max\_count)**  **{**  **max\_count = count;**  **max\_index = prev\_zero\_index;**  **}**  **}**    **// return index of 0 to be replaced or -1 if the array contains all 1's**  **return max\_index;**  **}**    **int main(void)**  **{**  **int arr[] = { 0, 0, 1, 0, 1, 1, 1, 0, 1, 1 };**  **int n = sizeof(arr) / sizeof(arr[0]);**    **int index = findIndexofZero(arr, n);**    **if (index != -1) {**  **printf("Index to be replaced is %d", index);**  **}**  **else {**  **printf("Invalid input");**  **}**    **return 0;**  **}**  **45: Maximum Sum Circular Subarray**  **Given a circular integer array, find a subarray with the largest sum in it.**  **Input: {2, 1, -5, 4, -3, 1, -3, 4, -1} Output: Subarray with the largest sum is {4, -1, 2, 1} with sum 6. Input: {-3, 1, -3, 4, -1, 2, 1, -5, 4} Output: Subarray with the largest sum is {4, -1, 2, 1} with sum 6.**   | **#include <iostream>**  **#include <numeric>**  **#include <algorithm>**  **using namespace std;**    **// Function to find contiguous subarray with the largest sum**  **// in a given set of integers**  **int kadane(int arr[], int n)**  **{**  **// stores the sum of maximum subarray found so far**  **int max\_so\_far = 0;**    **// stores the maximum sum of subarray ending at the current position**  **int max\_ending\_here = 0;**    **// traverse the given array**  **for (int i = 0; i < n; i++)**  **{**  **// update the maximum sum of subarray "ending" at index `i` (by adding the**  **// current element to maximum sum ending at previous index `i-1`)**  **max\_ending\_here = max\_ending\_here + arr[i];**    **// if the maximum sum is negative, set it to 0 (which represents**  **// an empty subarray)**  **max\_ending\_here = max(max\_ending\_here, 0);**    **// update result if the current subarray sum is found to be greater**  **max\_so\_far = max(max\_so\_far, max\_ending\_here);**  **}**    **return max\_so\_far;**  **}**    **// Function to find the maximum sum circular subarray in a given array**  **int runCircularKadane(int arr[], int n)**  **{**  **// empty array has sum of 0**  **if (n == 0) {**  **return 0;**  **}**    **// find the maximum element present in a given array**  **int max\_num = \*max\_element(arr, arr + n);**    **// if the array contains all negative values, return the maximum element**  **if (max\_num < 0) {**  **return max\_num;**  **}**    **// negate all the array elements**  **for (int i = 0; i < n; i++) {**  **arr[i] = -arr[i];**  **}**    **// run Kadane’s algorithm on the modified array**  **int neg\_max\_sum = kadane(arr, n);**    **// restore the array**  **for (int i = 0; i < n; i++) {**  **arr[i] = -arr[i];**  **}**    **/\* Return the maximum of the following:**  **1. Sum returned by Kadane’s algorithm on the original array.**  **2. Sum returned by Kadane’s algorithm on the modified array +**  **the sum of all the array elements.**  **\*/**    **return max(kadane(arr, n), accumulate(arr, arr + n, 0) + neg\_max\_sum);**  **}**    **int main()**  **{**  **int arr[] = { 2, 1, -5, 4, -3, 1, -3, 4, -1 };**  **int n = sizeof(arr)/sizeof(arr[0]);**    **cout << "The sum of the subarray with the largest sum is " <<**  **runCircularKadane(arr, n);**    **return 0;**  **}** | | --- |   **46: Maximum Product Subarray Problem**  **Given an integer array, find the subarray that has the maximum product of its elements. The solution should return the maximum product of elements among all possible subarrays.  Input: { -6, 4, -5, 8, -10, 0, 8 } Output: 1600 Explanation: The maximum product subarray is {4, -5, 8, -10} having product 1600**  **Input: { 40, 0, -20, -10 } Output: 200 Explanation: The maximum product subarray is {-20, -10} having product 200**  **#include <stdio.h>**    **// Utility function to find a minimum of two numbers**  **int min(int x, int y) {**  **return (x < y) ? x : y;**  **}**    **// Utility function to find a maximum of two numbers**  **int max(int x, int y) {**  **return (x > y) ? x : y;**  **}**    **// Function to return the maximum product of a subarray of a given array**  **int findMaxProduct(int arr[], int n)**  **{**  **// base case**  **if (n == 0) {**  **return 0;**  **}**    **// maintain two variables to store the maximum and minimum product**  **// ending at the current index**  **int max\_ending = arr[0], min\_ending = arr[0];**    **// to store the maximum product subarray found so far**  **int max\_so\_far = arr[0];**    **// traverse the given array**  **for (int i = 1; i < n; i++)**  **{**  **int temp = max\_ending;**    **// update the maximum product ending at the current index**  **max\_ending = max(arr[i], max(arr[i] \* max\_ending, arr[i] \* min\_ending));**    **// update the minimum product ending at the current index**  **min\_ending = min(arr[i], min(arr[i] \* temp, arr[i] \* min\_ending));**    **max\_so\_far = max(max\_so\_far, max\_ending);**  **}**    **// return maximum product**  **return max\_so\_far;**  **}**    **int main(void)**  **{**  **int arr[] = { -6, 4, -5, 8, -10, 0, 8 };**  **int n = sizeof(arr) / sizeof(arr[0]);**    **printf("The maximum product of a subarray is %d",**  **findMaxProduct(arr, n));**    **return 0;**  **}**  **47:Find the longest continuous sequence length with the same sum in given binary arrays**  **Given two binary arrays, X and Y, find the length of the longest continuous sequence that starts and ends at the same index in both arrays and have the same sum. In other words, find max(j-i+1) for every j >= i, where the sum of subarray X[i, j] is equal to the sum of subarray Y[i, j].**  **For example, consider the following binary arrays X and Y: X[]: {0, 0, 1, 1, 1, 1} Y[]: {0, 1, 1, 0, 1, 0}**  **The length of the longest continuous sequence with the same sum is 5 as X[0, 4]: {0, 0, 1, 1, 1} (sum = 3) Y[0, 4]: {0, 1, 1, 0, 1} (sum = 3)**  **#include <iostream>**  **#include <unordered\_map>**  **using namespace std;**    **// Given two binary arrays, `X` and `Y`, find the length of the longest**  **// continuous sequence that starts and ends at the same index in both**  **// arrays and have the same sum**  **int findMaxSubarrayLength(bool X[], bool Y[], int n)**  **{**  **// create an empty map**  **unordered\_map<int, int> map;**    **// to handle the case when the required sequence starts from index 0**  **map[0] = -1;**    **// stores length of the longest continuous sequence**  **int result = 0;**    **// `sum\_x` and `sum\_y` stores the sum of elements of `X[]` and `Y[]`,**  **// respectively, till the current index**  **int sum\_x = 0, sum\_y = 0;**    **// traverse both lists simultaneously**  **for (int i = 0; i < n; i++)**  **{**  **// update `sum\_x` and `sum\_y`**  **sum\_x += X[i];**  **sum\_y += Y[i];**    **// calculate the difference between the sum of elements in two lists**  **int diff = sum\_x - sum\_y;**    **// if the difference is seen for the first time, store the**  **// difference and current index in a map**  **if (map.find(diff) == map.end()) {**  **map[diff] = i;**  **}**    **// if the difference is seen before, then update the result**  **else {**  **result = max(result, i - map[diff]);**  **}**  **}**    **return result;**  **}**    **int main()**  **{**  **bool X[] = {0, 0, 1, 1, 1, 1};**  **bool Y[] = {0, 1, 1, 0, 1, 0};**    **int n = sizeof(X)/sizeof(X[0]);**    **cout << "The length of the longest continuous sequence with the same sum is " <<**  **findMaxSubarrayLength(X, Y, n);**    **return 0;**  **}**  **48: Rearrange array such that `A[A[i]]` is set to `i` for every element `A[i]`**  **Given an unsorted integer array A of size n, whose elements lie in the range 0 to n-1, rearrange the array such that A[A[i]] is set to i for every array element A[i]. Do this in linear time and without using any extra constant space.**  **Input: {1, 3, 4, 2, 0} Output: {4, 0, 3, 1, 2}  Explanation:**  **A[0] = 1, A[1] becomes 0**  **A[1] = 3, A[3] becomes 1**  **A[2] = 4, A[4] becomes 2**  **A[3] = 2, A[2] becomes 3**  **A[4] = 0, A[0] becomes 4**   | **#include <stdio.h>**    **// Function to rearrange an array such that `A[A[i]]` is set to `i`**  **// for every element `A[i]`**  **void rearrange(int A[], int n)**  **{**  **// create an auxiliary array of size `n`**  **int aux[n];**    **// for each element `A[i]` of the input array, set**  **// value `i` at index `A[i]` in the auxiliary array**  **for (int i = 0; i < n; i++) {**  **aux[A[i]] = i;**  **}**    **// copy auxiliary array elements back to the given array**  **for (int i = 0; i < n; i++) {**  **A[i] = aux[i];**  **}**  **}**    **int main()**  **{**  **int A[] = { 1, 3, 4, 2, 0 };**  **int n = sizeof(A) / sizeof(A[0]);**    **rearrange(A, n);**    **for (int i = 0; i < n; i++) {**  **printf("%d ", A[i]);**  **}**    **return 0;**  **}** | | --- |   **49: Find the maximum absolute difference between the sum of two non-overlapping subarrays**  **Given an array, find the maximum absolute difference between the sum of elements of two non-overlapping subarrays in linear time.**  **Input: A[] = { -3, -2, 6, -3, 5, -9, 3, 4, -1, -8, 2 } Output: The maximum absolute difference is 19.**  **The two subarrays are { 6, -3, 5 }, { -9, 3, 4, -1, -8 } whose sum of elements are 8 and -11, respectively. So, abs(8-(-11)) or abs(-11-8) = 19.**   | **#include <iostream>**  **#include <algorithm>**  **#include <climits>**  **using namespace std;**    **// `diff` ——> counter for loop from `i` to `j` in `A[]` (`diff` can be `+1` or `-1`)**  **// If the `diff` is 1, fill `aux[k]` with the maximum sum of subarray `A[0, k]`**  **// If the `diff` is -1, fill `aux[k]` with the maximum sum of subarray `A[k, n-1]`**  **// using Kadane’s algorithm**    **void findMaxSumSubarray(int A[], int aux[], int i, int j, int diff)**  **{**  **int max\_so\_far = A[i];**  **int max\_ending\_here = A[i];**  **aux[i] = A[i];**    **for (int k = i + diff; k != j; k += diff)**  **{**  **// update the maximum sum of subarray "ending" or "starting" at index `k`**  **max\_ending\_here = max(A[k], max\_ending\_here + A[k]);**    **// update the result if the current subarray sum is found to be greater**  **max\_so\_far = max(max\_so\_far, max\_ending\_here);**  **aux[k] = max\_so\_far;**  **}**  **}**    **void fillArrays(int A[], int left\_max[], int right\_max[],**  **int left\_min[], int right\_min[], int n)**  **{**  **findMaxSumSubarray(A, left\_max, 0, n - 1, 1);**  **findMaxSumSubarray(A, right\_max, n - 1, 0, -1);**    **// negate `A[]` for finding the minimum sum of subarrays using**  **// the same logic for finding the maximum sum of subarrays**  **transform(A, A + n, A, negate<int>());**    **// `transform()` is equivalent to**    **/\* for (int i = 0; i < n; i++) {**  **A[i] = -A[i];**  **} \*/**    **findMaxSumSubarray(A, left\_min, 0, n - 1, 1);**  **findMaxSumSubarray(A, right\_min, n - 1, 0, -1);**    **// negate `left\_min[]` and `right\_min[]` to get the minimum sum of subarrays**  **transform(left\_min, left\_min + n, left\_min, negate<int>());**  **transform(right\_min, right\_min + n, right\_min, negate<int>());**    **// restore the sign of `A[]`**  **transform(A, A + n, A, negate<int>());**  **}**    **// Find the maximum absolute difference between the sum of elements of**  **// two non-overlapping subarrays in a given array**  **int findMaxAbsDiff(int A[], int n)**  **{**  **// base case**  **if (n == 0) {**  **return 0;**  **}**    **// base case**  **if (n == 1) {**  **return A[0];**  **}**    **// `left\_max[i]` stores maximum sum of subarray in `A(0, i)`**  **// `right\_max[i]` stores maximum sum of subarray in `A(i, n-1)`**  **// `left\_min[i]` stores minimum sum of subarray in `A(0, i)`**  **// `right\_min[i]` stores minimum sum of subarray in `A(i, n-1)`**    **int left\_max[n], right\_max[n], left\_min[n], right\_min[n];**  **fillArrays(A, left\_max, right\_max, left\_min, right\_min, n);**    **// stores the maximum absolute difference**  **int max\_abs\_diff = INT\_MIN;**    **// do for each index `i` in the array**  **for (int i = 0; i < n - 1; i++)**  **{**  **max\_abs\_diff = max(max\_abs\_diff, max(abs(left\_max[i] - right\_min[i+1]),**  **abs(left\_min[i] - right\_max[i+1])));**  **}**    **return max\_abs\_diff;**  **}**    **int main()**  **{**  **int A[] = { -3, -2, 6, -3, 5, -9, 3, 4, -1, -8, 2 };**  **int n = sizeof(A) / sizeof(A[0]);**    **cout << "The maximum absolute difference is " << findMaxAbsDiff(A, n);**    **return 0;**  **}** | | --- |   **50: Print all combinations of positive integers in increasing order that sums to a given number**  **Write code to print all combinations of positive integers in increasing order that sum to a given positive number.**  **Input: N = 3 1 1 1 1 2 3**  **Input: N = 4 1 1 1 1 1 1 2 1 3 2 2 4**  **Input: N = 5 1 1 1 1 1 1 1 1 2 1 1 3 1 2 2 1 4 2 3 5**   | **#include <stdio.h>**    **// Utility function to print given array**  **void printArray(int nums[], int n)**  **{**  **for (int i = 0; i <= n; i++) {**  **printf("%d ", nums[i]);**  **}**    **printf("\n");**  **}**    **// Recursive function to print all combinations of positive integers**  **// in increasing order that sum to a given number**  **void printCombinations(int nums[], int i, int sum, int sum\_left)**  **{**  **// to maintain the increasing order, start the loop from the**  **// previous number stored in `nums[]`**  **int prev\_num = (i > 0) ? nums[i - 1] : 1;**  **for (int k = prev\_num; k <= sum; k++)**  **{**  **// set the next array element to `k`**  **nums[i] = k;**    **// recur with the sum left and the next location in the array**  **if (sum\_left > k) {**  **printCombinations(nums, i + 1, sum, sum\_left - k);**  **}**    **// if the sum is found**  **if (sum\_left == k) {**  **printArray(nums, i);**  **}**  **}**  **}**    **// Wrapper over `printCombinations()` function**  **void findCombinations(int sum)**  **{**  **// create a temporary array for storing the combinations**  **int nums[sum];**    **// recur for all combinations**  **int starting\_index = 0;**  **printCombinations(nums, starting\_index, sum, sum);**  **}**    **int main(void)**  **{**  **int sum = 5;**  **findCombinations(sum);**    **return 0;**  **}** | | --- |   **51: Find a minimum range with at least one element from each of the given arrays**  **Given three sorted arrays of variable length, efficiently compute the minimum range with at least one element from each array. Input: 3 sorted arrays of variable length [ 3, 6, 8, 10, 15 ] [ 1, 5, 12 ] [ 4, 8, 15, 16 ] Output: Minimum range is 3–5  Input: 3 sorted arrays of variable length [ 2, 3, 4, 8, 10, 15 ] [ 1, 5, 12 ] [ 7, 8, 15, 16 ] Output: Minimum range is 4–7**  **#include <iostream>**  **#include <vector>**  **#include <limits>**  **using namespace std;**    **// Function to find the minimum range with at least one element from**  **// each of the given arrays**  **pair<int, int> findMinRange(auto &a, auto &b, auto &c)**  **{**  **// create a pair to store the result**  **pair<int, int> pair;**    **// stores the low difference**  **int diff = numeric\_limits<int>::max();**    **// consider all triplets formed by `(a[i], b[j], c[k])`**  **for (int i = 0; i < a.size(); i++)**  **{**  **for (int j = 0; j < b.size(); j++)**  **{**  **for (int k = 0; k < c.size(); k++)**  **{**  **// find the minimum and maximum value in the current triplet**  **int low = min(min(a[i], b[j]), c[k]);**  **int high = max(max(a[i], b[j]), c[k]);**    **// update the low difference if the current difference is more**  **// and store the range in a pair**  **if (diff > high - low)**  **{**  **pair = make\_pair(low, high);**  **diff = high - low;**  **}**  **}**  **}**  **}**    **return pair;**  **}**    **int main()**  **{**  **vector<int> a = { 3, 6, 8, 10, 15 };**  **vector<int> b = { 1, 5, 12 };**  **vector<int> c = { 4, 8, 15, 16 };**    **auto pair = findMinRange(a, b, c);**  **cout << "The minimum range is [" << pair.first << ", " << pair.second << "]";**    **return 0;**  **}**  **52: 3–partition problem extended | Printing all partitions**  **Given an array of positive integers, which can be partitioned into three disjoint subsets having the same sum, print the partitions.  For example, consider the following set: S = { 7, 3, 2, 1, 5, 4, 8 } We can partition S into three partitions, each having a sum of 10. S1 = {7, 3} S2 = {5, 4, 1} S3 = {8, 2}**   | **#include <iostream>**  **#include <vector>**  **#include <numeric>**  **using namespace std;**    **// Helper function to 3–partition problem.**  **// It returns true if there exist three subsets with a given sum**  **bool isSubsetExist(vector<int> const &S, int n, int a, int b, int c, vector<int> &arr)**  **{**  **// return true if the subset is found**  **if (a == 0 && b == 0 && c == 0) {**  **return true;**  **}**    **// base case: no items left**  **if (n < 0) {**  **return false;**  **}**    **// Case 1. The current item becomes part of the first subset**  **bool A = false;**  **if (a - S[n] >= 0)**  **{**  **arr[n] = 1; // current element goes to the first subset**  **A = isSubsetExist(S, n - 1, a - S[n], b, c, arr);**  **}**    **// Case 2. The current item becomes part of the second subset**  **bool B = false;**  **if (!A && (b - S[n] >= 0))**  **{**  **arr[n] = 2; // current element goes to the second subset**  **B = isSubsetExist(S, n - 1, a, b - S[n], c, arr);**  **}**    **// Case 3. The current item becomes part of the third subset**  **bool C = false;**  **if ((!A && !B) && (c - S[n] >= 0))**  **{**  **arr[n] = 3; // current element goes to the third subset**  **C = isSubsetExist(S, n - 1, a, b, c - S[n], arr);**  **}**    **// return true if we get a solution**  **return A || B || C;**  **}**    **// Function for solving the 3–partition problem. It prints the subset if**  **// the given set `S[0…n-1]` can be divided into three subsets with an equal sum**  **void partition(vector<int> const &S)**  **{**  **// get the sum of all elements in the set**  **int sum = accumulate(S.begin(), S.end(), 0);**    **// total number of items in `S`**  **int n = S.size();**    **// construct an array to track the subsets**  **// `arr[i] = k` represents i'th item of `S` is part of k'th subset**  **vector<int> arr(n);**    **// set result to true if the sum is divisible by 3 and the set `S` can**  **// be divided into three subsets with an equal sum**  **bool result = (n >= 3) && !(sum % 3) &&**  **isSubsetExist(S, n - 1, sum/3, sum/3, sum/3, arr);**    **if (!result)**  **{**  **cout << "3-Partition of set is not possible";**  **return;**  **}**    **// print the partitions**  **for (int i = 0; i < 3; i++)**  **{**  **cout << "Partition " << i << " is ";**  **for (int j = 0; j < n; j++)**  **{**  **if (arr[j] == i + 1) {**  **cout << S[j] << " ";**  **}**  **}**  **cout << endl;**  **}**  **}**    **int main()**  **{**  **// Input: a set of integers**  **vector<int> S = { 7, 3, 2, 1, 5, 4, 8 };**    **partition(S);**    **return 0;**  **}** | | --- |   **53. Inversion count of an array**  **Given an array, find the total number of inversions of it. If (i < j) and (A[i] > A[j]), then pair (i, j) is called an inversion of an array A. We need to count all such pairs in the array. Input: A[] = [1, 9, 6, 4, 5] Output: The inversion count is 5 There are 5 inversions in the array: (9, 6), (9, 4), (9, 5), (6, 4), (6, 5)**   | **#include <stdio.h>**    **// Function to find inversion count of a given array**  **int findInversionCount(int arr[], int n)**  **{**  **int inversionCount = 0;**  **for (int i = 0; i < n - 1; i++)**  **{**  **for (int j = i + 1; j < n; j++)**  **{**  **if (arr[i] > arr[j]) {**  **inversionCount++;**  **}**  **}**  **}**  **return inversionCount;**  **}**    **int main()**  **{**  **int arr[] = { 1, 9, 6, 4, 5 };**  **int N = sizeof(arr)/sizeof(arr[0]);**    **printf("The inversion count is %d", findInversionCount(arr, N));**    **return 0;**  **}** | | --- |   **54: Find surpasser count for each array element**  **Given an integer array having distinct elements, find the surpasser count for each element in it. In other words, for each array element, find the total number of elements to its right, which are greater than it.**  **Input: { 4, 6, 3, 9, 7, 10 } Output: { 4, 3, 3, 1, 1, 0 }**  **#include <iostream>**  **#include <unordered\_map>**  **#include <cstring>**  **using namespace std;**    **// Function to merge two sorted subarrays `arr[low … mid]` and**  **// `arr[mid+1 … high]`**  **void merge(int arr[], int aux[], int low, int mid, int high,**  **auto &surpasser\_count)**  **{**  **int k = low, i = low, j = mid + 1;**  **int count = 0;**    **// run if there are elements in the left and right runs**  **while (i <= mid && j <= high)**  **{**  **if (arr[i] > arr[j])**  **{**  **// update surpasser count of `arr[i]`**  **surpasser\_count[arr[i]] += count;**    **aux[k++] = arr[i++];**  **}**  **else {**  **aux[k++] = arr[j++];**  **count++;**  **}**  **}**    **// copy remaining elements**  **while (i <= mid)**  **{**  **surpasser\_count[arr[i]] += count;**  **aux[k++] = arr[i++];**  **}**    **/\* no need to copy the second half (since the remaining items**  **are already in their correct position in the temporary array) \*/**    **// copy back to the original array to reflect sorted order**  **for (int i = low; i <= high; i++) {**  **arr[i] = aux[i];**  **}**  **}**    **// Function to sort array `arr[low…high]` in descending order**  **void merge\_sort(int arr[], int aux[], int low, int high,**  **auto &surpasser\_count)**  **{**  **// base case: run size is less than or equal to 1**  **if (high <= low) {**  **return;**  **}**    **// find midpoint**  **int mid = (low + ((high - low) >> 1));**    **// recursively split runs into two halves until run size == 1,**  **// merge them, and return up the call chain**    **merge\_sort(arr, aux, low, mid, surpasser\_count);**  **merge\_sort(arr, aux, mid + 1, high, surpasser\_count);**    **merge(arr, aux, low, mid, high, surpasser\_count);**  **}**    **// Function to find the surpasser count for each array element**  **auto surpasserCount(int const A[], int n)**  **{**  **unordered\_map<int, int> surpasser\_count;**    **// create two copies of the original array**  **int aux[n], arr[n];**  **memcpy(aux, A, n \* sizeof(A[0]));**  **memcpy(arr, A, n \* sizeof(A[0]));**    **// sort array `arr[]` in descending order using auxiliary array aux[]**  **merge\_sort(arr, aux, 0, n - 1, surpasser\_count);**    **return surpasser\_count;**  **}**    **int main()**  **{**  **int arr[] = { 4, 6, 3, 9, 7, 10 };**  **int n = sizeof(arr) / sizeof(arr[0]);**    **// find the surpasser count for array elements**  **unordered\_map<int, int> surpasser\_count = surpasserCount(arr, n);**  **for (int i = 0; i < n; i++) {**  **cout << surpasser\_count[arr[i]] << " ";**  **}**    **return 0;**  **}**  **55: Find the area of the largest rectangle of 1’s in a binary matrix**  **Given a rectangular binary matrix, calculate the area of the largest rectangle of 1's in it. Assume that a rectangle can be formed by swapping any number of columns with each other.**  **Input: [0, 1, 0, 1, 1] [1, 1, 0, 0, 1] [1, 1, 0, 1, 1] [1, 1, 1, 1, 1]**  **Output: The area of the largest rectangle of 1’s is 9 Explanation: The largest rectangle of 1’s can be formed by swapping column 3 with column 5. [0, 1, 1, 1, 0] [1, 1, 1, 0, 0] [1, 1, 1, 1, 0] [1, 1, 1, 1, 1]**  **Input: [0, 1, 1, 0] [1, 0, 0, 1] [1, 1, 0, 1] [1, 1, 1, 1]**  **Output: The area of the largest rectangle of 1’s is 6 Explanation: The largest rectangle of 1’s can be formed by swapping column 2 with column 4 or swapping column 3 with column 4. [0, 0, 1, 1] [1, 1, 0, 0] [1, 1, 0, 1] [1, 1, 1, 1]**  **OR**  **[0, 1, 0, 1] [1, 0, 1, 0] [1, 1, 1, 0] [1, 1, 1, 1]**   | **#include <stdio.h>**  **#include <limits.h>**    **// `M × N` matrix**  **#define M 4**  **#define N 5**    **// Utility function to replace all non-zero values in a matrix by 1**  **void resetMatrix(int mat[][N])**  **{**  **for (int i = 0; i < M; i++)**  **{**  **for (int j = 0; j < N; j++)**  **{**  **if (mat[i][j] != 0) {**  **mat[i][j] = 1;**  **}**  **}**  **}**  **}**    **// Utility function to find the maximum of two numbers**  **int max(int x, int y) {**  **return (x > y) ? x : y;**  **}**    **// Function to calculate the area of the largest rectangle of 1's where swapping of**  **// columns is allowed**  **int findMaxRectArea(int mat[][N])**  **{**  **// update the matrix to store the counts of consecutive 1's present in each column**  **for (int j = 0; j < N; j++)**  **{**  **// process each column from bottom to top**  **for (int i = M - 2; i >= 0; i--)**  **{**  **if (mat[i][j] == 1) {**  **mat[i][j] = mat[i+1][j] + 1;**  **}**  **}**  **}**    **// keep track of the largest rectangle of 1's found so far**  **int maxArea = 0;**    **// traverse each row in the modified matrix to find the maximum area**  **for (int i = 0; i < M; i++)**  **{**  **// create a count array for each row `i`**  **int count[M + 1] = { 0 };**    **// process row `i`**  **for (int j = 0; j < N; j++)**  **{**  **if (mat[i][j] > 0)**  **{**  **// increment value in the count array using the current element**  **// as an index**  **count[mat[i][j]] += 1;**    **// the area can be calculated by multiplying the current**  **// element `mat[i][j]` with the corresponding value**  **// in the count array `count[mat[i][j]]`**    **maxArea = max(maxArea, mat[i][j] \* count[mat[i][j]]);**  **}**  **}**  **}**    **// reset matrix before returning**  **resetMatrix(mat);**    **return maxArea;**  **}**    **int main(void)**  **{**  **int mat[M][N] =**  **{**  **{ 0, 1, 0, 1, 1 },**  **{ 1, 1, 0, 0, 1 },**  **{ 1, 1, 0, 1, 1 },**  **{ 1, 1, 1, 1, 1 }**  **};**    **printf("The area of the largest rectangle of 1's is %d", findMaxRectArea(mat));**    **return 0;**  **}** | | --- |   **56: Sort an array using Young tableau**  **In this post, we will see how to sort N2 numbers in increasing order using an N × N Young tableau in O(N3) time.**  **An N × N Young tableau is an N × N matrix such that entries of each row are sorted from left to right and the entries of each column are sorted from top to bottom. Some entries of a Young tableau may be infinity, which indicates an empty entry. Thus, a Young tableau can be used to hold n <= N2 finite numbers.**  **#include <iostream>**  **#include <vector>**  **#include <algorithm>**  **#include <sstream>**  **#include <cstring>**  **#include <climits>**  **#include <cmath>**  **using namespace std;**    **class YoungTableau**  **{**  **// Recursive function to fix the tableau property in an `N × N` Young tableau.**  **// An infinite value is initially placed at the first cell `(0, 0)` of the tableau.**  **// The function works by swapping the smallest of `[i+i, j]` and `[i, j+1]` with**  **// `[i, j]` and recur for the smaller value.**  **void fixTableau(vector<vector<int>> &tableau, int i, int j)**  **{**  **int N = tableau.size();**    **// get the values present at the bottom and right cell of the current cell**  **int bottom = (i + 1 < N) ? tableau[i + 1][j] : INT\_MAX;**  **int right = (j + 1 < N) ? tableau[i][j + 1] : INT\_MAX;**    **if (bottom == INT\_MAX && right == INT\_MAX) {**  **return;**  **}**    **if (bottom < right) // go down**  **{**  **swap(tableau[i][j], tableau[i + 1][j]);**  **fixTableau(tableau, i + 1, j);**  **}**  **else // go right**  **{**  **swap(tableau[i][j], tableau[i][j + 1]);**  **fixTableau(tableau, i, j + 1);**  **}**  **}**    **// Recursive function to insert a new element into a non-full `N × N` Young tableau.**  **// The new element is initially placed at the bottom-right corner of the tableau.**  **// The function works by swapping the smallest of `[i-i, j]` and `[i, j-1]` with**  **// `[i, j]` and recur for the smaller value.**  **void insert(vector<vector<int>> &tableau, int i, int j)**  **{**  **// base case**  **if (i == 0 && j == 0) {**  **return;**  **}**    **// handle separately for the first row**  **if (i == 0)**  **{**  **if (tableau[i][j] < tableau[i][j - 1])**  **{**  **swap(tableau[i][j], tableau[i][j - 1]);**  **insert(tableau, i, j - 1);**  **}**  **return;**  **}**    **// handle separately for the first column**  **if (j == 0)**  **{**  **if (tableau[i][j] < tableau[i - 1][j])**  **{**  **swap(tableau[i][j], tableau[i - 1][j]);**  **insert(tableau, i - 1, j);**  **}**  **return;**  **}**    **if (tableau[i][j] < tableau[i - 1][j]) // go up**  **{**  **swap(tableau[i][j], tableau[i - 1][j]);**  **insert(tableau, i - 1, j);**  **}**    **if (tableau[i][j] < tableau[i][j - 1]) // go left**  **{**  **swap(tableau[i][j], tableau[i][j - 1]);**  **insert(tableau, i, j - 1);**  **}**  **}**    **public:**    **// Function construct an `N × N` Young tableau from the given keys**  **vector<vector<int>> construct(vector<int> &keys)**  **{**  **// initialize the Young tableau by infinity**  **int N = (int) ceil(sqrt(keys.size()));**  **vector<vector<int>> tableau(N, vector<int>(N, INT\_MAX));**    **// do for each key**  **for (int key: keys)**  **{**  **// check for overflow**  **if (tableau[N - 1][N - 1] != INT\_MAX) {**  **break;**  **}**    **// place the key at the bottom-right corner of the tableau**  **tableau[N - 1][N - 1] = key;**    **// move the key to its correct position in the tableau**  **insert(tableau, N - 1, N - 1);**  **}**    **return tableau;**  **}**    **// Function to extract the next minimum element from the Young tableau**  **int extractMin(vector<vector<int>> &tableau)**  **{**  **// the first cell of the tableau stores the minimum element**  **int min = tableau[0][0];**    **// make the first element as infinity**  **tableau[0][0] = INT\_MAX;**    **// fix the Young tableau property**  **fixTableau(tableau, 0, 0);**    **return min;**  **}**  **};**    **void sort(vector<int> &keys)**  **{**  **if (keys.size() == 0) {**  **return;**  **}**    **// construct a Young tableau from the above keys**  **YoungTableau s;**  **vector<vector<int>> tableau = s.construct(keys);**    **// repeatedly call `extractMin()` and fill `keys[]` with the returned values**  **for (int i = 0; i < keys.size(); i++) {**  **keys[i] = s.extractMin(tableau);**  **}**  **}**    **int main()**  **{**  **// unsorted input**  **vector<int> keys { 6, 4, 8, 7, 2, 3, 1, 5 };**    **// sort the input keys**  **sort(keys);**    **// print the sorted input**  **for (int i: keys) {**  **cout << i << ' ';**  **}**    **return 0;**  **}**  **57:Print all possible solutions to N–Queens problem**  **The N–queens puzzle is the problem of placing N chess queens on an N × N chessboard so that no two queens threaten each other. Thus, the solution requires that no two queens share the same row, column, or diagonal.**  **For example, for a standard 8 × 8 chessboard, below is one such configuration:**  **Q – – – – – – –  – – – – Q – – –  – – – – – – – Q  – – – – – Q – –  – – Q – – – – –  – – – – – – Q –  – Q – – – – – –  – – – Q – – – –**  **#include <stdio.h>**  **#include <string.h>**    **// `N × N` chessboard**  **#define N 8**    **// Function to check if two queens threaten each other or not**  **int isSafe(char mat[][N], int r, int c)**  **{**  **// return 0 if two queens share the same column**  **for (int i = 0; i < r; i++)**  **{**  **if (mat[i][c] == 'Q') {**  **return 0;**  **}**  **}**    **// return 0 if two queens share the same `` diagonal**  **for (int i = r, j = c; i >= 0 && j >= 0; i--, j--)**  **{**  **if (mat[i][j] == 'Q') {**  **return 0;**  **}**  **}**    **// return 0 if two queens share the same `/` diagonal**  **for (int i = r, j = c; i >= 0 && j < N; i--, j++)**  **{**  **if (mat[i][j] == 'Q') {**  **return 0;**  **}**  **}**    **return 1;**  **}**    **void printSolution(char mat[][N])**  **{**  **for (int i = 0; i < N; i++)**  **{**  **for (int j = 0; j < N; j++) {**  **printf("%c ", mat[i][j]);**  **}**  **printf("\n");**  **}**  **printf("\n");**  **}**    **void nQueen(char mat[][N], int r)**  **{**  **// if `N` queens are placed successfully, print the solution**  **if (r == N)**  **{**  **printSolution(mat);**  **return;**  **}**    **// place queen at every square in the current row `r`**  **// and recur for each valid movement**  **for (int i = 0; i < N; i++)**  **{**  **// if no two queens threaten each other**  **if (isSafe(mat, r, i))**  **{**  **// place queen on the current square**  **mat[r][i] = 'Q';**    **// recur for the next row**  **nQueen(mat, r + 1);**    **// backtrack and remove the queen from the current square**  **mat[r][i] = '-';**  **}**  **}**  **}**    **int main()**  **{**  **// `mat[][]` keeps track of the position of queens in the current configuration**  **char mat[N][N];**    **// initialize `mat[][]` by `-`**  **memset(mat, '-', sizeof mat);**    **nQueen(mat, 0);**    **return 0;**  **}**  **58: Print all possible Knight’s tours on a chessboard**  **Given a chessboard, print all sequences of moves of a knight on a chessboard such that the knight visits every square only once. For example, for the standard 8 × 8 chessboards, below is one such tour. We have started the tour from the top-leftmost of the board (marked as 1), and the next number represents the knight’s consecutive moves.**  **1 50 45 62 31 18 9 64 46 61 32 49 10 63 30 17 51 2 47 44 33 28 19 8 60 35 42 27 48 11 16 29 41 52 3 34 43 24 7 20 36 59 38 55 26 21 12 15 53 40 57 4 23 14 25 6 58 37 54 39 56 5 22 13**   | **#include <iostream>**  **#include <cstring>**  **using namespace std;**    **// `N × N` chessboard**  **#define N 5**    **// Below arrays detail all eight possible movements for a knight.**  **// It is important to avoid changing the sequence of the below arrays**  **int row[] = { 2, 1, -1, -2, -2, -1, 1, 2, 2 };**  **int col[] = { 1, 2, 2, 1, -1, -2, -2, -1, 1 };**    **// Check if `(x, y)` is valid chessboard coordinates.**  **// Note that a knight cannot go out of the chessboard**  **bool isValid(int x, int y)**  **{**  **if (x < 0 || y < 0 || x >= N || y >= N) {**  **return false;**  **}**    **return true;**  **}**    **// Recursive function to perform the knight's tour using backtracking**  **void knightTour(int visited[N][N], int x, int y, int pos)**  **{**  **// mark the current square as visited**  **visited[x][y] = pos;**    **// if all squares are visited, print the solution**  **if (pos >= N\*N)**  **{**  **for (int i = 0; i < N; i++)**  **{**  **for (int j = 0; j < N; j++) {**  **cout << visited[i][j] << " ";**  **}**  **cout << endl;**  **}**  **cout << endl;**    **// backtrack before returning**  **visited[x][y] = 0;**  **return;**  **}**    **// check for all eight possible movements for a knight**  **// and recur for each valid movement**  **for (int k = 0; k < 8; k++)**  **{**  **// get the new position of the knight from the current**  **// position on the chessboard**  **int newX = x + row[k];**  **int newY = y + col[k];**    **// if the new position is valid and not visited yet**  **if (isValid(newX, newY) && !visited[newX][newY]) {**  **knightTour(visited, newX, newY, pos + 1);**  **}**  **}**    **// backtrack from the current square and remove it from the current path**  **visited[x][y] = 0;**  **}**    **int main()**  **{**  **// `visited[][]` serves two purposes:**  **// 1. It keeps track of squares involved in the knight's tour.**  **// 2. It stores the order in which the squares are visited.**  **int visited[N][N];**    **// initialize `visited[][]` by 0 (unvisited)**  **memset(visited, 0, sizeof visited);**    **int pos = 1;**    **// start knight tour from corner square `(0, 0)`**  **knightTour(visited, 0, 0, pos);**    **return 0;**  **}** | | --- |   **59: K–Partition Problem | Printing all partitions**  **In the k–partition problem, we need to partition an array of positive integers into k disjoint subsets that all have an equal sum, and they completely cover the set.  For example, consider set S = { 7, 3, 5, 12, 2, 1, 5, 3, 8, 4, 6, 4 }. 1. S can be partitioned into two partitions, each having a sum of 30. S1 = { 5, 3, 8, 4, 6, 4 } S2 = { 7, 3, 5, 12, 2, 1 }  2. S can be partitioned into three partitions, each having a sum of 20. S1 = { 2, 1, 3, 4, 6, 4 } S2 = { 7, 5, 8 } S3 = { 3, 5, 12 }  3. S can be partitioned into four partitions, each having a sum of 15. S1 = { 1, 4, 6, 4 } S2 = { 2, 5, 8 } S3 = { 12, 3 } S4 = { 7, 3, 5 }  4. S can be partitioned into five partitions, each having a sum of 12. S1 = { 2, 6, 4 } S2 = { 8, 4 } S3 = { 3, 1, 5, 3 } S4 = { 12 } S5 = { 7, 5 }**  **#include <iostream>**  **#include <numeric>**  **using namespace std;**    **// Function to check if all subsets are filled or not**  **bool checkSum(int sumLeft[], int k)**  **{**  **int r = true;**  **for (int i = 0; i < k; i++)**  **{**  **if (sumLeft[i] != 0) {**  **r = false;**  **}**  **}**    **return r;**  **}**    **// Helper function for solving `k` partition problem.**  **// It returns true if there exist `k` subsets with the given sum**  **bool subsetSum(int S[], int n, int sumLeft[], int A[], int k)**  **{**  **// return true if a subset is found**  **if (checkSum(sumLeft, k)) {**  **return true;**  **}**    **// base case: no items left**  **if (n < 0) {**  **return false;**  **}**    **bool result = false;**    **// consider current item `S[n]` and explore all possibilities**  **// using backtracking**  **for (int i = 0; i < k; i++)**  **{**  **if (!result && (sumLeft[i] - S[n]) >= 0)**  **{**  **// mark the current element subset**  **A[n] = i + 1;**    **// add the current item to the i'th subset**  **sumLeft[i] = sumLeft[i] - S[n];**    **// recur for remaining items**  **result = subsetSum(S, n - 1, sumLeft, A, k);**    **// backtrack: remove the current item from the i'th subset**  **sumLeft[i] = sumLeft[i] + S[n];**  **}**  **}**    **// return true if we get a solution**  **return result;**  **}**    **// Function for solving k–partition problem. It prints the subsets if**  **// set `S[0…n-1]` can be divided into `k` subsets with equal sum**  **void partition(int S[], int n, int k)**  **{**  **// base case**  **if (n < k)**  **{**  **cout << "k-partition of set S is not possible";**  **return;**  **}**    **// get the sum of all elements in the set**  **int sum = accumulate(S, S + n, 0);**    **int A[n], sumLeft[k];**    **// create an array of size `k` for each subset and initialize it**  **// by their expected sum, i.e., `sum/k`**  **for (int i = 0; i < k; i++) {**  **sumLeft[i] = sum/k;**  **}**    **// return true if the sum is divisible by `k` and set `S` can**  **// be divided into `k` subsets with equal sum**  **bool result = !(sum % k) && subsetSum(S, n - 1, sumLeft, A, k);**    **if (!result)**  **{**  **cout << "k-partition of set S is not possible";**  **return;**  **}**    **// print all k–partitions**  **for (int i = 0; i < k; i++)**  **{**  **cout << "Partition " << i << " is ";**  **for (int j = 0; j < n; j++)**  **{**  **if (A[j] == i + 1) {**  **cout << S[j] << " ";**  **}**  **}**  **cout << endl;**  **}**  **}**    **int main()**  **{**  **// Input: a set of integers**  **int S[] = { 7, 3, 5, 12, 2, 1, 5, 3, 8, 4, 6, 4 };**    **// total number of items in `S`**  **int n = sizeof(S) / sizeof(S[0]);**  **int k = 5;**    **partition(S, n, k);**    **return 0;**  **}**  **60: Check if a repeated subsequence is present in a string or not**  **Given a string, check if a repeated subsequence is present in it or not. The repeated subsequence should have a length of 2 or more. String XYBAXB has XB(XBXB) as a repeated subsequence String XBXAXB has XX(XXX) as a repeated subsequence String ABCA doesn’t have any repeated subsequence String XYBYAXBY has XB(XBXB), XY(XYXY), YY(YYY), YB(YBYB), and YBY(YBYBY) as repeated subsequences.**   | **#include <iostream>**  **#include <string>**  **#include <unordered\_map>**  **using namespace std;**    **// Recursive function to check if `str[low…high]` is a palindrome or not**  **bool isPalindrome(string str, int low, int high)**  **{**  **// base case**  **if (low >= high) {**  **return true;**  **}**    **return (str[low] == str[high]) &&**  **isPalindrome(str, low + 1, high - 1);**  **}**    **// Function to checks if repeated subsequence is present**  **// in the string**  **bool hasRepeatedSubsequence(string str)**  **{**  **// base case**  **if (str.length() == 0) {**  **return false;**  **}**    **// map to store the frequency of each distinct character**  **// of a given string**  **unordered\_map<char, int> freq;**    **// update map with frequency**  **for (int i = 0; i < str.length(); i++)**  **{**  **// if the frequency of any character becomes 3,**  **// we have found the repeated subsequence**  **if (++freq[str[i]] >= 3) {**  **return true;**  **}**  **}**    **string temp;**    **// consider all repeated elements (frequency 2 or more)**  **// and discard all non-repeating elements (frequency 1)**  **for (int i = 0; i < str.length(); i++)**  **{**  **if (freq[str[i]] >= 2) {**  **temp += str[i];**  **}**  **}**    **// return false if `temp` is a palindrome**  **return !isPalindrome(temp, 0, temp.length() - 1);**  **}**    **int main()**  **{**  **string str = "XYBYAXB"; // 'XB' and 'YB' are repeated subsequences**    **if (hasRepeatedSubsequence(str)) {**  **cout << "Repeated subsequence is present";**  **}**  **else {**  **cout << "No repeated subsequence is present";**  **}**    **return 0;**  **}** | | --- |   **61: Find all possible combinations of words formed from the mobile keypad**  **Given a sequence of numbers between 2 and 9, print all possible combinations of words formed from the mobile keypad which has english alphabets associated with each key.  Input: [2, 3, 4] Output: ADG BDG CDG AEG BEG CEG AFG BFG CFG ADH BDH CDH AEH BEH CEH AFH BFH CFH ADI BDI CDI AEI BEI CEI AFI BFI CFI**   | **#include <iostream>**  **#include <vector>**  **#include <unordered\_set>**  **using namespace std;**    **// Top-down recursive function to find all possible combinations by**  **// replacing key's digits with characters of the corresponding list**  **void findCombinations(auto &keypad, auto const &keys, auto &combinations,**  **string result, int index)**  **{**  **// if we have processed every digit of the key, print the result**  **if (index == -1)**  **{**  **combinations.insert(result);**  **return;**  **}**    **// stores the current digit**  **int digit = keys[index];**    **// get the list corresponding to the current digit and**  **// one by one, replace the digit with each character in the**  **// corresponding list and recur for the next digit**  **for (char c: keypad[digit]) {**  **findCombinations(keypad, keys, combinations, c + result, index - 1);**  **}**  **}**    **unordered\_set<string> findAllCombinations(auto const &keypad, auto const &keys)**  **{**  **// create a set to store all combinations**  **unordered\_set<string> combinations;**    **// invalid input - return empty set**  **if (keypad.size() == 0 || keys.size() == 0) {**  **return combinations;**  **}**    **// find and return all combinations**  **int n = keys.size();**  **findCombinations(keypad, keys, combinations, "", n - 1);**  **return combinations;**  **}**    **int main()**  **{**  **// mobile keypad**  **vector<vector<char>> keypad =**  **{**  **{}, {}, // 0 and 1 digit doesn't have any characters associated**  **{ 'A', 'B', 'C' },**  **{ 'D', 'E', 'F' },**  **{ 'G', 'H', 'I' },**  **{ 'J', 'K', 'L' },**  **{ 'M', 'N', 'O' },**  **{ 'P', 'Q', 'R', 'S'},**  **{ 'T', 'U', 'V' },**  **{ 'W', 'X', 'Y', 'Z'}**  **};**    **// input number in the form of an array (number cannot start from 0 or 1)**  **vector<int> keys = { 2, 3, 4 };**    **// find all combinations**  **unordered\_set<string> combinations = findAllCombinations(keypad, keys);**  **for (string s: combinations) {**  **cout << s << ' ';**  **}**    **return 0;**  **}** | | --- |  44. **Sort binary array in linear time** Given a binary array, sort it in linear time and constant space. The output should print all zeroes, followed by all ones.  For example,  **Input:**  { 1, 0, 1, 0, 1, 0, 0, 1 }    **Output:** { 0, 0, 0, 0, 1, 1, 1, 1 }  [Practice this problem](https://techiedelight.com/practice/?problem=SortBinaryArray)  A simple solution would be to count the total number of 0’s present in the array, say k, and fill the first k indices in the array by 0 and all remaining indices by 1.  Alternatively, we can count the total number of 1’s present in the array k and fill the last k indices in the array by 1 and all remaining indices by 0.   | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | #include <stdio.h>    // Function to sort a binary array in linear time  int sort(int A[], int n)  {  // count number of 0's  int zeros = 0;  for (int i = 0; i < n; i++)  {  if (A[i] == 0) {  zeros++;  }  }    // put 0's at the beginning  int k = 0;  while (zeros--) {  A[k++] = 0;  }    // fill all remaining elements by 1  while (k < n) {  A[k++] = 1;  }  }    int main(void)  {  int A[] = { 0, 0, 1, 0, 1, 1, 0, 1, 0, 0 };  int n = sizeof(A)/sizeof(A[0]);    sort(A, n);    // print the rearranged array  for (int i = 0; i < n; i++) {  printf("%d ", A[i]);  }    return 0;  } | | --- | --- |   We can also solve this problem in linear time by using the [partitioning logic of Quicksort](https://www.techiedelight.com/quicksort/). The idea is to use 1 as a pivot element and make one pass of the partition process.   | #include <stdio.h>    // Utility function to swap elements `A[i]` and `A[j]` in an array  void swap(int A[], int i, int j)  {  int temp = A[i];  A[i] = A[j];  A[j] = temp;  }    // Function to sort a binary array in linear time  int partition(int A[], int n)  {  int pivot = 1;  int j = 0;    // each time we encounter a 0, `j` is incremented, and  // 0 is placed before the pivot  for (int i = 0; i < n; i++)  {  if (A[i] < pivot)  {  swap(A, i, j);  j++;  }  }  }    int main(void)  {  int A[] = { 1, 0, 0, 0, 1, 0, 1, 1 };  int n = sizeof(A)/sizeof(A[0]);    partition(A, n);    // print the rearranged array  for (int i = 0; i < n; i++) {  printf("%d ", A[i]);  }    return 0;  } | | --- |  **45**. **Complete Search** Complete search (aka brute force or recursive backtracking) is a method for solving a problem by traversing the entire search space in search of a solution. During the search we can prune parts of the search space that we are sure do not lead to the required solution. In programming contests, complete search will likely lead to Time Limit Exceeded (TLE), however, it’s a good strategy for small input problems.  #include <cstdlib>  #include <cstdio>  #include <cstring>  using namespace std;  //row[8]: row # for each queen  //TC: traceback counter  //(a, b): 1st queen placement at (r=a, c=b)  int row[8], TC, a, b, line\_counter;  bool place(int r, int c)  {  // check previously placed queens  for (int prev = 0; prev < c; prev++)  {  // check if same row or same diagonal  if (row[prev] == r || (abs(row[prev] — r) == abs(prev — c)))  return false;  }  return true;  }  void backtrack(int c)  {  // candidate solution; (a, b) has 1 initial queen  if (c == 8 && row[b] == a)  {  printf(“%2d %d”, ++line\_counter, row[0] + 1);  for (int j=1; j < 8; j++) {printf(“ %d”, row[j] + 1);}  printf(“\n”);  }  //try all possible rows  for (int r = 0; r < 8; r++)  {  if (place(r, c))  {  row[c] = r; // place a queen at this col and row  backtrack(c + 1); //increment col and recurse  }  }  }  int main()  {  scanf(“%d”, &TC);  while (TC--)  {  scanf(“%d %d”, &a, &b); a--; b--; //0-based indexing  memset(row, 0, sizeof(row)); line\_counter = 0;  printf(“SOLN COLUMN\n”);  printf(“ # 1 2 3 4 5 6 7 8\n\n”);  backtrack(0); //generate all possible 8! candidate solutions  if (TC) printf(“\n”);  }  return 0;  }  For TC=8 and an initial queen position at (a,b) = (1,1) the above code results in the following output:  SOLN COLUMN  # 1 2 3 4 5 6 7 8  1 1 5 8 6 3 7 2 4  2 1 6 8 3 7 4 2 5  3 1 7 4 6 8 2 5 3  4 1 7 5 8 2 4 6 3  which indicates that there are four possible placements given the initial queen position at (r=1,c=1). Notice that the use of recursion allows to more easily prune the search space in comparison to an iterative solution. |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

# **46**. **Greedy Algorithms**

A greedy algorithm takes a locally optimum choice at each step with the hope of eventually reaching a globally optimal solution. Greedy algorithms often rely on a greedy heuristic and one can often find examples in which greedy algorithms fail to achieve the global optimum.

## **Greedy Example: Fractional Knapsack**

A greedy knapsack problem consists of selecting what items to place in a knapsack of limited capacity W so as to maximize the total value of knapsack items, where each item has an associated weight and value. We can define a greedy heuristic to be a ratio of item value to item weight, i.e. we would like to greedily choose items that are simultaneously high value and low weight and sort the items based on this criteria. In the fractional knapsack problem, we are allowed to take fractions of an item (as opposed to 0–1 Knapsack).

#include <iostream>

#include <algorithm>

using namespace std;

struct Item{

int value, weight;

Item(int value, int weight) : value(value), weight(weight) {}

};

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;

}

double fractional\_knapsack(int W, struct Item arr[], int n)

{

sort(arr, arr + n, cmp);

int cur\_weight = 0; double tot\_value = 0.0;

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

{

if (cur\_weight + arr[i].weight <= W)

{

cur\_weight += arr[i].weight;

tot\_value += arr[i].value;

}

else

{ //add a fraction of the next item

int rem\_weight = W — cur\_weight;

tot\_value += arr[i].value \*

((double) rem\_weight / arr[i].weight);

break;

}

}

return tot\_value;

}

int main()

{

int W = 50; // total knapsack weight

Item arr[] = {{60, 10}, {100, 20}, {120, 30}}; //{value, weight}

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

cout << “greedy fractional knapsack” << endl;

cout << “maximum value: “ << fractional\_knapsack(W, arr, n);

cout << endl;

return 0;

}

Since sorting is the most expensive operation, the algorithm runs in O(n log n) time. Given (value, weight) pairs of three items: {(60, 10), (100, 20), (120, 30)}, and the total capacity W=50, the code above produces the following output:

greedy fractional knapsack

maximum value: 240

We can see that the input items are sorted in decreasing ratio of value / cost, after greedily selecting items 1 and 2, we take a 2/3 fraction of item 3 for a total value of 60+100+(2/3)120 = 240.

# 47. **Divide and Conquer**

Divide and Conquer (D&C) is a technique that divides a problem into smaller,*independent* sub-problems and then combines solutions to each of the sub-problems.

Examples of divide and conquer technique include sorting algorithms such as quick sort, merge sort and heap sort as well as binary search.

## **D&C Example: Binary Search**

The classic use of binary search is in searching for a value in a *sorted* array. First, we check the middle of the array to see if if contains what we are looking for. If it does or there are no more items to consider, we stop. Otherwise, we decide whether the answer is to the left or the right of the middle element and continue searching. As the size of the search space is halved after each check, the complexity of the algorithm is O(log n).

#include <algorithm>

#include <vector>

#include <iostream>

using namespace std;

int bsearch(const vector<int> &arr, int l, int r, int q)

{

while (l <= r)

{

int mid = l + (r-l)/2;

if (arr[mid] == q) return mid;

if (q < arr[mid]) { r = mid — 1; }

else { l = mid + 1; }

}

return -1; //not found

}

int main()

{

int query = 10;

int arr[] = {2, 4, 6, 8, 10, 12};

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

vector<int> v(arr, arr + N);

//sort input array

sort(v.begin(), v.end());

int idx;

idx = bsearch(v, 0, v.size(), query);

if (idx != -1)

cout << "custom binary\_search: found at index " << idx;

else

cout << "custom binary\_search: not found";

return 0;

}

The code above produces the following output:

custom binary\_search: found at index 4

# 48. Dynamic Programming

Dynamic Programming (DP) is a technique that divides a problem into smaller *overlapping* sub-problems, computes a solution for each sub-problem and stores it in a DP table. The final solution is read off the DP table.

Key skills in mastering dynamic programming is the ability to determine the problem states (entries of the DP table) and the relationships or transitions between the states. Then, having defined base cases and recursive relationships, one can populate the DP table in a top-down or bottom-up fashion.

In the top-down DP, the table is populated recursively, as needed, starting from the top and going down to smaller sub-problems. In the bottom-up DP, the table is populated iteratively starting from the smallest sub-problems and using their solutions to build-on and arrive at solutions to bigger sub-problems. In both cases, if a sub-problem was already encountered, its solution is simply looked up in the table (as opposed to re-computing the solution from scratch). This dramatically reduces computational cost.

## **DP Example: Binomial Coefficients**

We use binomial coefficients example to illustrate the use of top-down and bottom-up DP. The code below is based on the recursion for binomial coefficients with overlapping sub-problems. Let C(n,k) denote n choose k, then, we have:

Base case: C(n,0) = C(n,n) = 1

Recursion: C(n,k) = C(n-1, k-1) + C(n-1, k)

Notice that we have multiple over-lapping sub-problems. E.g. For C(n=5, k=2) the recursion tree is as follows:

C(5, 2)

/ \

C(4, 1) C(4, 2)

/ \ / \

C(3, 0) C(3, 1) C(3, 1) C(3, 2)

/ \ / \ / \

C(2, 0) C(2, 1) C(2, 0) C(2, 1) C(2, 1) C(2, 2)

/ \ / \ / \

C(1, 0) C(1, 1) C(1, 0) C(1, 1) C(1, 0) C(1, 1)

We can implement top-down and bottom-up DP as follows:

#include <iostream>

#include <cstring>

using namespace std;

#define V 8

int memo[V][V]; //DP table

int min(int a, int b) {return (a < b) ? a : b;}

void print\_table(int memo[V][V])

{

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

{

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

{

printf(" %2d", memo[i][j]);

}

printf("\n");

}

}

int binomial\_coeffs1(int n, int k)

{

// top-down DP

if (k == 0 || k == n) return 1;

if (memo[n][k] != -1) return memo[n][k];

return memo[n][k] = binomial\_coeffs1(n-1, k-1) +

binomial\_coeffs1(n-1, k);

}

int binomial\_coeffs2(int n, int k)

{

// bottom-up DP

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

{

for (int j = 0; j <= min(i, k); ++j)

{

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

{

memo[i][j] = 1;

}

else

{

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

}

}

}

return memo[n][k];

}

int main()

{

int n = 5, k = 2;

printf("Top-down DP:\n");

memset(memo, -1, sizeof(memo));

int nCk1 = binomial\_coeffs1(n, k);

print\_table(memo);

printf("C(n=%d, k=%d): %d\n", n, k, nCk1);

printf("Bottom-up DP:\n");

memset(memo, -1, sizeof(memo));

int nCk2 = binomial\_coeffs2(n, k);

print\_table(memo);

printf("C(n=%d, k=%d): %d\n", n, k, nCk2);

return 0;

}

For C(n=5, k=2), the code above produces the following output:

Top-down DP:

-1 -1 -1 -1 -1 -1 -1 -1

-1 -1 -1 -1 -1 -1 -1 -1

-1 2 -1 -1 -1 -1 -1 -1

-1 3 3 -1 -1 -1 -1 -1

-1 4 6 -1 -1 -1 -1 -1

-1 -1 10 -1 -1 -1 -1 -1

-1 -1 -1 -1 -1 -1 -1 -1

-1 -1 -1 -1 -1 -1 -1 -1

C(n=5, k=2): 10

Bottom-up DP:

1 -1 -1 -1 -1 -1 -1 -1

1 1 -1 -1 -1 -1 -1 -1

1 2 1 -1 -1 -1 -1 -1

1 3 3 -1 -1 -1 -1 -1

1 4 6 -1 -1 -1 -1 -1

1 5 10 -1 -1 -1 -1 -1

-1 -1 -1 -1 -1 -1 -1 -1

-1 -1 -1 -1 -1 -1 -1 -1

C(n=5, k=2): 10

The time complexity is O(n \* k) and the space complexity is O(n \* k). In the case of top-down DP, solutions to sub-problems are stored (memoized) as needed, whereas in the bottom-up DP, the entire table is computed starting from the base case. Note: a small DP table size (V=8) was chosen for printing purposes, a much larger table size is recommended.

# 49. **Box Stacking Problem | DP-22**

* Difficulty Level : [Hard](https://www.geeksforgeeks.org/hard/)
* Last Updated : 30 Jun, 2021

You are given a set of n types of rectangular 3-D boxes, where the i^th box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

**Source:** <http://people.csail.mit.edu/bdean/6.046/dp/>. The link also has a video for an explanation of the solution.



**Method 1 : dynamic programming using tabulation**

**1)** Generate all 3 rotations of all boxes. The size of rotation array becomes 3 times the size of the original array. For simplicity, we consider width as always smaller than or equal to depth.

**2)** Sort the above generated 3n boxes in decreasing order of base area.

**3)** After sorting the boxes, the problem is same as LIS with following optimal substructure property.

MSH(i) = Maximum possible Stack Height with box i at top of stack

MSH(i) = { Max ( MSH(j) ) + height(i) } where j < i and width(j) > width(i) and depth(j) > depth(i).

If there is no such j then MSH(i) = height(i)

**4)** To get overall maximum height, we return max(MSH(i)) where 0 < i < n

Following is the implementation of the above solution.

/\* Dynamic Programming implementation of Box Stacking problem \*/

#include<stdio.h>

#include<stdlib.h>

/\* Representation of a box \*/

struct Box

{

// h --> height, w --> width, d --> depth

int h, w, d; // for simplicity of solution, always keep w <= d

};

// A utility function to get minimum of two integers

int min (int x, int y)

{ return (x < y)? x : y; }

// A utility function to get maximum of two integers

int max (int x, int y)

{ return (x > y)? x : y; }

/\* Following function is needed for library function qsort(). We

use qsort() to sort boxes in decreasing order of base area.

Refer following link for help of qsort() and compare()

<http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/> \*/

int compare (const void \*a, const void \* b)

{

return ( (\*(Box \*)b).d \* (\*(Box \*)b).w ) -

( (\*(Box \*)a).d \* (\*(Box \*)a).w );

}

/\* Returns the height of the tallest stack that can be

formed with give type of boxes \*/

int maxStackHeight( Box arr[], int n )

{

/\* Create an array of all rotations of given boxes

For example, for a box {1, 2, 3}, we consider three

instances{{1, 2, 3}, {2, 1, 3}, {3, 1, 2}} \*/

Box rot[3\*n];

int index = 0;

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

{

// Copy the original box

rot[index].h = arr[i].h;

rot[index].d = max(arr[i].d, arr[i].w);

rot[index].w = min(arr[i].d, arr[i].w);

index++;

// First rotation of box

rot[index].h = arr[i].w;

rot[index].d = max(arr[i].h, arr[i].d);

rot[index].w = min(arr[i].h, arr[i].d);

index++;

// Second rotation of box

rot[index].h = arr[i].d;

rot[index].d = max(arr[i].h, arr[i].w);

rot[index].w = min(arr[i].h, arr[i].w);

index++;

}

// Now the number of boxes is 3n

n = 3\*n;

/\* Sort the array 'rot[]' in non-increasing order

of base area \*/

qsort (rot, n, sizeof(rot[0]), compare);

// Uncomment following two lines to print all rotations

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

// printf("%d x %d x %d\n", rot[i].h, rot[i].w, rot[i].d);

/\* Initialize msh values for all indexes

msh[i] --> Maximum possible Stack Height with box i on top \*/

int msh[n];

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

msh[i] = rot[i].h;

/\* Compute optimized msh values in bottom up manner \*/

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

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

if ( rot[i].w < rot[j].w &&

rot[i].d < rot[j].d &&

msh[i] < msh[j] + rot[i].h

)

{

msh[i] = msh[j] + rot[i].h;

}

/\* Pick maximum of all msh values \*/

int max = -1;

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

if ( max < msh[i] )

max = msh[i];

return max;

}

/\* Driver program to test above function \*/

int main()

{

Box arr[] = { {4, 6, 7}, {1, 2, 3}, {4, 5, 6}, {10, 12, 32} };

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

printf("The maximum possible height of stack is %d\n",

maxStackHeight (arr, n) );

return 0;

}

**Output**

The maximum possible height of stack is 60

In the above program, given input boxes are {4, 6, 7}, {1, 2, 3}, {4, 5, 6}, {10, 12, 32}. Following are all rotations of the boxes in decreasing order of base area.

10 x 12 x 32

12 x 10 x 32

32 x 10 x 12

4 x 6 x 7

4 x 5 x 6

6 x 4 x 7

5 x 4 x 6

7 x 4 x 6

6 x 4 x 5

1 x 2 x 3

2 x 1 x 3

3 x 1 x 2

/\* Dynamic Programming top-down implementation of Box

\* Stacking problem \*/

#include <bits/stdc++.h>

using namespace std;

/\* Representation of a box \*/

class Box {

public:

int length;

int width;

int height;

};

// dp array

int dp[303];

/\*

boxes -> vector of Box

bottom\_box\_index -> index of the bottom box

index -> index of current box

\*/

/\* NOTE: we can use only one variable in place of bottom\_box\_index and index

but it has been avoided to make it simple \*/

int findMaxHeight(vector<Box>& boxes, int bottom\_box\_index, int index)

{

// base case

if (index < 0)

return 0;

if (dp[index] != -1)

return dp[index];

int maximumHeight = 0;

// recurse

for (int i = index; i >= 0; i--) {

// if there is no bottom box

if (bottom\_box\_index == -1

// or if length & width of new box is < that of

// bottom box

|| (boxes[i].length

< boxes[bottom\_box\_index].length

&& boxes[i].width

< boxes[bottom\_box\_index].width))

maximumHeight

= max(maximumHeight,

findMaxHeight(boxes, i, i - 1)

+ boxes[i].height);

}

return dp[index] = maximumHeight;

}

/\* wrapper function for recursive calls which

Returns the height of the tallest stack that can be

formed with give type of boxes \*/

int maxStackHeight(int height[], int width[], int length[],

int types)

{

// creating a vector of type Box class

vector<Box> boxes;

// Initialize dp array with -1

memset(dp, -1, sizeof(dp));

Box box;

/\* Create an array of all rotations of given boxes

For example, for a box {1, 2, 3}, we consider three

instances{{1, 2, 3}, {2, 1, 3}, {3, 1, 2}} \*/

for (int i = 0; i < types; i++) {

// copy original box

box.height = height[i];

box.length = max(length[i], width[i]);

box.width = min(length[i], width[i]);

boxes.push\_back(box);

// First rotation of box

box.height = width[i];

box.length = max(length[i], height[i]);

box.width = min(length[i], height[i]);

boxes.push\_back(box);

// Second rotation of box

box.height = length[i];

box.length = max(width[i], height[i]);

box.width = min(width[i], height[i]);

boxes.push\_back(box);

}

// sort by area in ascending order .. because we will be dealing with this vector in reverse

sort(boxes.begin(), boxes.end(), [](Box b1, Box b2) {

// if area of box1 < area of box2

return (b1.length \* b1.width)

< (b2.length \* b2.width);

});

// Uncomment following two lines to print all rotations

//for (int i = boxes.size() - 1; i >= 0; i-- )

// printf("%d x %d x %d\n", boxes[i].length, boxes[i].width, boxes[i].height);

return findMaxHeight(boxes, -1, boxes.size() - 1);

}

int main()

{

// where length, width and height of a particular box

// are at ith index of the following arrays

int length[] = { 4, 1, 4, 10 };

int width[] = { 6, 2, 5, 12 };

int height[] = { 7, 3, 6, 32 };

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

printf("The maximum possible height of stack is %d\n",

maxStackHeight(height, length, width, n));

return 0;

}

**Output**

The maximum possible height of stack is 60

In the above program, for boxes of dimensions of {4, 6, 7}, {1, 2, 3}, {4, 5, 6}, {10, 12, 32} on giving the input as {4, 1, 4, 10} for length, {6, 2, 5, 12} for width and {7, 3, 6, 32} for height. Following rotations are possible for the boxes in decreasing order of base area.

32 x 12 x 10 <-

32 x 10 x 12

12 x 10 x 32 <-

7 x 6 x 4 <-

6 x 5 x 4 <-

7 x 4 x 6

6 x 4 x 5

6 x 4 x 7

5 x 4 x 6 <-

3 x 2 x 1 <-

3 x 1 x 2

2 x 1 x 3 <-

The maximum possible height of stack is 60

The height 60 is obtained by boxes { {2, 1, 3}, {3, 2, 1}, {5, 4, 6}, {6, 5, 4}, {7, 6, 4}, {12, 10, 32}, {32, 12, 10}}

# 50. **Maximum size rectangle binary sub-matrix with all 1s**

* Difficulty Level : [Hard](https://www.geeksforgeeks.org/hard/)
* Last Updated : 25 Jan, 2022

Given a binary matrix, find the maximum size rectangle binary-sub-matrix with all 1’s.

**Example:**

**Input:**

0 1 1 0

1 1 1 1

1 1 1 1

1 1 0 0

**Output :**

**8**

**Explanation :**

The largest rectangle with only 1's is from

(1, 0) to (2, 3) which is

1 1 1 1

1 1 1 1

**Input:**

0 1 1

1 1 1

0 1 1

**Output:**

**6**

**Explanation :**

The largest rectangle with only 1's is from

(0, 1) to (2, 2) which is

1 1

1 1

1 1

**Approach:** In this post, an interesting method is discussed that uses [largest rectangle under histogram](https://www.geeksforgeeks.org/largest-rectangle-under-histogram/) as a subroutine.

If the height of bars of the histogram is given then the largest area of the histogram can be found. This way in each row, the largest area of bars of the histogram can be found. To get the largest rectangle full of 1’s, update the next row with the previous row and find the largest area under the histogram, i.e. consider each 1’s as filled squares and 0’s with an empty square and consider each row as the base.

**Illustration:**

**Input :**

0 1 1 0

1 1 1 1

1 1 1 1

1 1 0 0

**Step 1:**

0 1 1 0 maximum area = 2

**Step 2:**

row 1 1 2 2 1 area = 4, maximum area becomes 4

row 2 2 3 3 2 area = 8, maximum area becomes 8

row 3 3 4 0 0 area = 6, maximum area remains 8

**Algorithm:**

1. Run a loop to traverse through the rows.
2. Now If the current row is not the first row then update the row as follows, if matrix[i][j] is not zero then matrix[i][j] = matrix[i-1][j] + matrix[i][j].
3. Find the maximum rectangular area under the histogram, consider the ith row as heights of bars of a histogram. This can be calculated as given in this article [Largest Rectangular Area in a Histogram](https://www.geeksforgeeks.org/largest-rectangle-under-histogram/)
4. Do the previous two steps for all rows and print the maximum area of all the rows.

*Note*: It is strongly recommended to refer to [this](https://www.geeksforgeeks.org/largest-rectangle-under-histogram/) post first as most of the code is taken from there.

// C++ program to find largest

// rectangle with all 1s

// in a binary matrix

#include <bits/stdc++.h>

using namespace std;

// Rows and columns in input matrix

#define R 4

#define C 4

// Finds the maximum area under

// the histogram represented

// by histogram. See below article for details.

int maxHist(int row[])

{

// Create an empty stack.

// The stack holds indexes of

// hist[] array/ The bars stored

// in stack are always

// in increasing order of their heights.

stack<int> result;

int top\_val; // Top of stack

int max\_area = 0; // Initialize max area in current

// row (or histogram)

int area = 0; // Initialize area with current top

// Run through all bars of given histogram (or row)

int i = 0;

while (i < C) {

// If this bar is higher than the bar on top stack,

// push it to stack

if (result.empty() || row[result.top()] <= row[i])

result.push(i++);

else {

// If this bar is lower than top of stack, then

// calculate area of rectangle with stack top as

// the smallest (or minimum height) bar. 'i' is

// 'right index' for the top and element before

// top in stack is 'left index'

top\_val = row[result.top()];

result.pop();

area = top\_val \* i;

if (!result.empty())

area = top\_val \* (i - result.top() - 1);

max\_area = max(area, max\_area);

}

}

// Now pop the remaining bars from stack and calculate

// area with every popped bar as the smallest bar

while (!result.empty()) {

top\_val = row[result.top()];

result.pop();

area = top\_val \* i;

if (!result.empty())

area = top\_val \* (i - result.top() - 1);

max\_area = max(area, max\_area);

}

return max\_area;

}

// Returns area of the largest rectangle with all 1s in

// A[][]

int maxRectangle(int A[][C])

{

// Calculate area for first row and initialize it as

// result

int result = maxHist(A[0]);

// iterate over row to find maximum rectangular area

// considering each row as histogram

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

for (int j = 0; j < C; j++)

// if A[i][j] is 1 then add A[i -1][j]

if (A[i][j])

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

// Update result if area with current row (as last

// row) of rectangle) is more

result = max(result, maxHist(A[i]));

}

return result;

}

// Driver code

int main()

{

int A[][C] = {

{ 0, 1, 1, 0 },

{ 1, 1, 1, 1 },

{ 1, 1, 1, 1 },

{ 1, 1, 0, 0 },

};

cout << "Area of maximum rectangle is "

<< maxRectangle(A);

return 0;

}

**Complexity Analysis:**

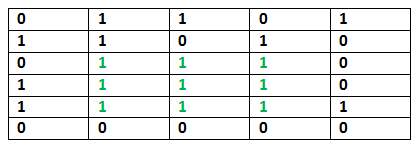
* **Time Complexity:** O(R x C).   
  Only one traversal of the matrix is required, so the time complexity is O(R X C)
* **Space Complexity:** O(C).   
  Stack is required to store the columns, so so space complexity is O(C)

# **51. Maximum size square sub-matrix with all 1s**

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 03 Feb, 2022

Given a binary matrix, find out the maximum size square sub-matrix with all 1s.

For example, consider the below binary matrix.



Algorithm:

Let the given binary matrix be M[R][C]. The idea of the algorithm is to construct an auxiliary size matrix S[][] in which each entry S[i][j] represents the size of the square sub-matrix with all 1s including M[i][j] where M[i][j] is the rightmost and bottom-most entry in sub-matrix.

1) Construct a sum matrix S[R][C] for the given M[R][C].

a) Copy first row and first columns as it is from M[][] to S[][]

b) For other entries, use following expressions to construct S[][]

If M[i][j] is 1 then

S[i][j] = min(S[i][j-1], S[i-1][j], S[i-1][j-1]) + 1

Else /\*If M[i][j] is 0\*/

S[i][j] = 0

2) Find the maximum entry in S[R][C]

3) Using the value and coordinates of maximum entry in S[i], print

sub-matrix of M[][]

For the given M[R][C] in the above example, constructed S[R][C] would be:

0 1 1 0 1

1 1 0 1 0

0 1 1 1 0

1 1 2 2 0

1 2 2 3 1

0 0 0 0 0

The value of the maximum entry in the above matrix is 3 and the coordinates of the entry are (4, 3). Using the maximum value and its coordinates, we can find out the required sub-matrix.

// C++ code for Maximum size square

// sub-matrix with all 1s

#include <bits/stdc++.h>

#define bool int

#define R 6

#define C 5

using namespace std;

void printMaxSubSquare(bool M[R][C])

{

int i,j;

int S[R][C];

int max\_of\_s, max\_i, max\_j;

/\* Set first column of S[][]\*/

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

S[i][0] = M[i][0];

/\* Set first row of S[][]\*/

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

S[0][j] = M[0][j];

/\* Construct other entries of S[][]\*/

for(i = 1; i < R; i++)

{

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

{

if(M[i][j] == 1)

S[i][j] = min({S[i][j-1], S[i-1][j],

S[i-1][j-1]}) + 1; //better of using min in case of arguments more than 2

else

S[i][j] = 0;

}

}

/\* Find the maximum entry, and indexes of maximum entry

in S[][] \*/

max\_of\_s = S[0][0]; max\_i = 0; max\_j = 0;

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

{

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

{

if(max\_of\_s < S[i][j])

{

max\_of\_s = S[i][j];

max\_i = i;

max\_j = j;

}

}

}

cout<<"Maximum size sub-matrix is: \n";

for(i = max\_i; i > max\_i - max\_of\_s; i--)

{

for(j = max\_j; j > max\_j - max\_of\_s; j--)

{

cout << M[i][j] << " ";

}

cout << "\n";

}

}

/\* Driver code \*/

int main()

{

bool M[R][C] = {{0, 1, 1, 0, 1},

{1, 1, 0, 1, 0},

{0, 1, 1, 1, 0},

{1, 1, 1, 1, 0},

{1, 1, 1, 1, 1},

{0, 0, 0, 0, 0}};

printMaxSubSquare(M);

}

// This code is contributed by rathbhupendra

**Output**

Maximum size sub-matrix is:

1 1 1

1 1 1

1 1 1

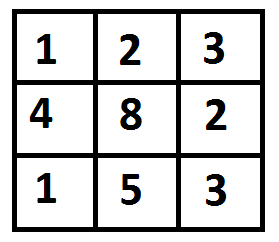
**Time Complexity**: O(m\*n) where m is the number of rows and n is the number of columns in the given matrix. **Auxiliary space:** O(n) where n is the number of columns in the given matrix.

# **52. Min Cost Path | DP-6**

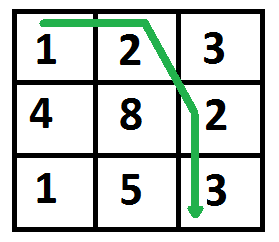
* Difficulty Level : [Easy](https://www.geeksforgeeks.org/easy/)
* Last Updated : 21 Oct, 2021

Given a cost matrix cost[][] and a position (m, n) in cost[][], write a function that returns cost of minimum cost path to reach (m, n) from (0, 0). Each cell of the matrix represents a cost to traverse through that cell. The total cost of a path to reach (m, n) is the sum of all the costs on that path (including both source and destination). You can only traverse down, right and diagonally lower cells from a given cell, i.e., from a given cell (i, j), cells (i+1, j), (i, j+1), and (i+1, j+1) can be traversed. You may assume that all costs are positive integers.

For example, in the following figure, what is the minimum cost path to (2, 2)?



The path with minimum cost is highlighted in the following figure. The path is (0, 0) –> (0, 1) –> (1, 2) –> (2, 2). The cost of the path is 8 (1 + 2 + 2 + 3).



// A Naive recursive implementation

// of MCP(Minimum Cost Path) problem

#include <bits/stdc++.h>

using namespace std;

#define R 3

#define C 3

int min(int x, int y, int z);

// Returns cost of minimum cost path

// from (0,0) to (m, n) in mat[R][C]

int minCost(int cost[R][C], int m, int n)

{

if (n < 0 || m < 0)

return INT\_MAX;

else if (m == 0 && n == 0)

return cost[m][n];

else

return cost[m][n] +

min(minCost(cost, m - 1, n - 1),

minCost(cost, m - 1, n),

minCost(cost, m, n - 1));

}

// A utility function that returns

// minimum of 3 integers

int min(int x, int y, int z)

{

if (x < y)

return (x < z) ? x : z;

else

return (y < z) ? y : z;

}

// Driver code

int main()

{

int cost[R][C] = { { 1, 2, 3 },

{ 4, 8, 2 },

{ 1, 5, 3 } };

cout << minCost(cost, 2, 2) << endl;

return 0;

}

// This code is contributed by nikhilchhipa9

**Output**

8

***Time Complexity:*** *O(m \* n)*

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, there are many nodes which appear more than once. The time complexity of this naive recursive solution is exponential and it is terribly slow.

mC refers to minCost()

mC(2, 2)

/ | \

/ | \

mC(1, 1) mC(1, 2) mC(2, 1)

/ | \ / | \ / | \

/ | \ / | \ / | \

mC(0,0) mC(0,1) mC(1,0) mC(0,1) mC(0,2) mC(1,1) mC(1,0) mC(1,1) mC(2,0)

So the MCP problem has both properties (see [this](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/) and [this](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](https://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of the same subproblems can be avoided by constructing a temporary array tc[][] in a bottom-up manner.

#include <bits/stdc++.h>

#define endl "\n"

using namespace std;

const int row = 3;

const int col = 3;

int minCost(int cost[row][col]) {

// for 1st column

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

cost[i][0] += cost[i-1][0];

}

// for 1st row

for (int j=1 ; j<col ; j++){

cost[0][j] += cost[0][j-1];

}

// for rest of the 2d matrix

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

for (int j=1 ; j<col ; j++) {

cost[i][j] += min(cost[i-1][j-1], min(cost[i-1][j], cost[i][j-1]));

}

}

// returning the value in last cell

return cost[row-1][col-1];

}

int main(int argc, char const \*argv[])

{

int cost[row][col] = { {1, 2, 3},

{4, 8, 2},

{1, 5, 3} };

cout << minCost(cost) << endl;

return 0;

}

// Contributed by Mansimar-Anand TU\_2022 p\_e\_k\_k\_a Task @ Codechef/Codeforces

***Time Complexity:*** *O(row \* col)*

***Auxiliary Space:*** *O(1)*

**Alternate Solution**

# 53.**Coin Change | DP-7**

* Difficulty Level : [Hard](https://www.geeksforgeeks.org/hard/)
* Last Updated : 25 Feb, 2022

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.

**1) Optimal Substructure**

To count the total number of solutions, we can divide all set solutions into two sets.

1) Solutions that do not contain mth coin (or Sm).

2) 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.

**2) Overlapping Subproblems**

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

**3) Approach (Algorithm)**

See, here each coin of a given denomination can come an infinite number of times. (Repetition allowed), this is what we call UNBOUNDED KNAPSACK. We have 2 choices for a coin of a particular denomination, either i) to include, or ii) to exclude. But here, the inclusion process is not for just once; we can include any denomination any number of times until N<0.

Basically, If we are at s[m-1], we can take as many instances of that coin ( unbounded inclusion ) i.e **count(S, m, n – S[m-1] )** ; then we move to s[m-2]. After moving to s[m-2], we can’t move back and can’t make choices for s[m-1] i.e **count(S, m-1, n )**.

Finally, as we have to find the total number of ways, so we will add these 2 possible choices, i.e **count(S, m, n – S[m-1] ) + count(S, m-1, n ) ;** which will be our required answer.

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

using namespace std;

// 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 code

int main()

{

int i, j;

int arr[] = { 1, 2, 3 };

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

cout << " " << count(arr, m, 4);

return 0;

}

// This code is contributed by shivanisinghss2110

**Output**

4

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for S = {1, 2, 3} and n = 5.

The function C({1}, 3) is called two times. If we draw the complete tree, then we can see that there are many subproblems being called more than once.

C() --> count()

C({1,2,3}, 5)

/ \

/ \

C({1,2,3}, 2) C({1,2}, 5)

/ \ / \

/ \ / \

C({1,2,3}, -1) C({1,2}, 2) C({1,2}, 3) C({1}, 5)

/ \ / \ / \

/ \ / \ / \

C({1,2},0) C({1},2) C({1,2},1) C({1},3) C({1}, 4) C({}, 5)

/ \ / \ /\ / \

/ \ / \ / \ / \

. . . . . . C({1}, 3) C({}, 4)

/ \

/ \

. .

Since same subproblems are called again, this problem has Overlapping Subproblems property. So the Coin Change problem has both properties (see [this](https://www.geeksforgeeks.org/archives/12635) and [this](https://www.geeksforgeeks.org/archives/12819)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](https://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of same subproblems can be avoided by constructing a temporary array table[][] in bottom up manner.

54. **Rabin-Karp Algorithm for Pattern Searching**

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 24 Feb, 2022

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

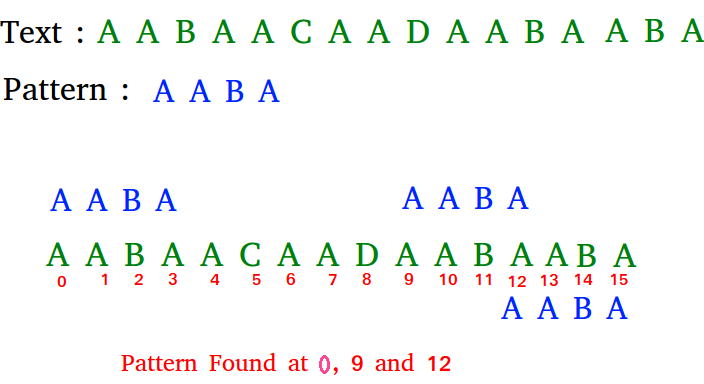
Input: txt[] = "AABAACAADAABAABA"

pat[] = "AABA"

Output: Pattern found at index 0

Pattern found at index 9

Pattern found at index 12



The [Naive String Matching](https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/) algorithm slides the pattern one by one. After each slide, it one by one checks characters at the current shift and if all characters match then prints the match.

Like the Naive Algorithm, Rabin-Karp algorithm also slides the pattern one by one. But unlike the Naive algorithm, Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for following strings.

1) Pattern itself.

2) All the substrings of the text of length m.

Since we need to efficiently calculate hash values for all the substrings of size m of text, we must have a hash function which has the following property.

Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say *hash(txt[s+1 .. s+m])* must be efficiently computable from *hash(txt[s .. s+m-1])* and *txt[s+m]* i.e., *hash(txt[s+1 .. s+m])*= *rehash(txt[s+m], hash(txt[s .. s+m-1]))* and rehash must be O(1) operation.

The hash function suggested by Rabin and Karp calculates an integer value. The integer value for a string is the numeric value of a string.

For example, if all possible characters are from 1 to 10, the numeric value of “122” will be 122. The number of possible characters is higher than 10 (256 in general) and pattern length can be large. So the numeric values cannot be practically stored as an integer. Therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable (can fit in memory words). To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value. Rehashing is done using the following formula.

*hash( txt[s+1 .. s+m] ) = ( d ( hash( txt[s .. s+m-1]) – txt[s]\*h ) + txt[s + m] ) mod q*

*hash( txt[s .. s+m-1] )* : Hash value at shift *s*.

*hash( txt[s+1 .. s+m] )* : Hash value at next shift (or shift *s*+1)

*d*: Number of characters in the alphabet

*q*: A prime number

*h: d^(m-1)*

**How does the** **above expression work?**

This is simple mathematics, we compute decimal value of current window from previous window.

For example pattern length is 3 and string is “23456”

You compute the value of first window (which is “234”) as 234.

How how will you compute value of next window “345”? You will do (234 – 2\*100)\*10 + 5 and get 345.

/\* Following program is a C++ implementation of Rabin Karp

Algorithm given in the CLRS book \*/

#include <bits/stdc++.h>

using namespace std;

// d is the number of characters in the input alphabet

#define d 256

/\* pat -> pattern

txt -> text

q -> A prime number

\*/

void search(char pat[], char txt[], int q)

{

int M = strlen(pat);

int N = strlen(txt);

int i, j;

int p = 0; // hash value for pattern

int t = 0; // hash value for txt

int h = 1;

// The value of h would be "pow(d, M-1)%q"

for (i = 0; i < M - 1; i++)

h = (h \* d) % q;

// Calculate the hash value of pattern and first

// window of text

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

{

p = (d \* p + pat[i]) % q;

t = (d \* t + txt[i]) % q;

}

// Slide the pattern over text one by one

for (i = 0; i <= N - M; i++)

{

// Check the hash values of current window of text

// and pattern. If the hash values match then only

// check for characters one by one

if ( p == t )

{

bool flag = true;

/\* Check for characters one by one \*/

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

{

if (txt[i+j] != pat[j])

{

flag = false;

break;

}

if(flag)

cout<<i<<" ";

}

// if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]

if (j == M)

cout<<"Pattern found at index "<< i<<endl;

}

// Calculate hash value for next window of text: Remove

// leading digit, add trailing digit

if ( i < N-M )

{

t = (d\*(t - txt[i]\*h) + txt[i+M])%q;

// We might get negative value of t, converting it

// to positive

if (t < 0)

t = (t + q);

}

}

}

/\* Driver code \*/

int main()

{

char txt[] = "GEEKS FOR GEEKS";

char pat[] = "GEEK";

// A prime number

int q = 101;

// Function Call

search(pat, txt, q);

return 0;

}

// This is code is contributed by rathbhupendra

**Output:**

Pattern found at index 0

Pattern found at index 10

**Time Complexity:**

The average and best-case running time of the Rabin-Karp algorithm is O(n+m), but its worst-case time is O(nm). Worst case of Rabin-Karp algorithm occurs when all characters of pattern and text are same as the hash values of all the substrings of txt[] match with hash value of pat[]. For example pat[] = “AAA” and txt[] = “AAAAAAA”.

# 55.**C++ Program to Find Transpose of a Matrix**

This program takes a matrix of order r\*c from the user and computes the transpose of the matrix.

In this program, user is asked to entered the number of rows and columns. The value of rows and columns should be less than 10 in this program.

Then, the user is asked to enter elements of the matrix.

The program computes the transpose of the matrix and displays it on the screen.

#include <iostream>

using namespace std;

int main() {

int a[10][10], transpose[10][10], row, column, i, j;

cout << "Enter rows and columns of matrix: ";

cin >> row >> column;

cout << "\nEnter elements of matrix: " << endl;

// Storing matrix elements

for (int i = 0; i < row; ++i) {

for (int j = 0; j < column; ++j) {

cout << "Enter element a" << i + 1 << j + 1 << ": ";

cin >> a[i][j];

}

}

// Printing the a matrix

cout << "\nEntered Matrix: " << endl;

for (int i = 0; i < row; ++i) {

for (int j = 0; j < column; ++j) {

cout << " " << a[i][j];

if (j == column - 1)

cout << endl << endl;

}

}

// Computing transpose of the matrix

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

for (int j = 0; j < column; ++j) {

transpose[j][i] = a[i][j];

}

// Printing the transpose

cout << "\nTranspose of Matrix: " << endl;

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

for (int j = 0; j < row; ++j) {

cout << " " << transpose[i][j];

if (j == row - 1)

cout << endl << endl;

}

return 0;

}

Output:

Enter rows and columns of matrix: 2

3

Enter elements of matrix:

Enter element a11: 1

Enter element a12: 2

Enter element a13: 9

Enter element a21: 0

Enter element a22: 4

Enter element a23: 7

Entered Matrix:

1 2 9

0 4 7

Transpose of Matrix:

1 0

2 4

9 7

# 56. **C++ Program to Calculate Difference Between Two Time Period**

In this example, we will calculate the difference between two time periods provided by the user.

// Computes time difference of two time period

// Time periods are entered by the user

#include <iostream>

using namespace std;

struct TIME

{

int seconds;

int minutes;

int hours;

};

void computeTimeDifference(struct TIME, struct TIME, struct TIME \*);

int main()

{

struct TIME t1, t2, difference;

cout << "Enter start time." << endl;

cout << "Enter hours, minutes and seconds respectively: ";

cin >> t1.hours >> t1.minutes >> t1.seconds;

cout << "Enter stop time." << endl;

cout << "Enter hours, minutes and seconds respectively: ";

cin >> t2.hours >> t2.minutes >> t2.seconds;

computeTimeDifference(t1, t2, &difference);

cout << endl << "TIME DIFFERENCE: " << t1.hours << ":" << t1.minutes << ":" << t1.seconds;

cout << " - " << t2.hours << ":" << t2.minutes << ":" << t2.seconds;

cout << " = " << difference.hours << ":" << difference.minutes << ":" << difference.seconds;

return 0;

}

void computeTimeDifference(struct TIME t1, struct TIME t2, struct TIME \*difference){

if(t2.seconds > t1.seconds)

{

--t1.minutes;

t1.seconds += 60;

}

difference->seconds = t1.seconds - t2.seconds;

if(t2.minutes > t1.minutes)

{

--t1.hours;

t1.minutes += 60;

}

difference->minutes = t1.minutes-t2.minutes;

difference->hours = t1.hours-t2.hours;

}

Output

Enter hours, minutes and seconds respectively: 11

33

52

Enter stop time.

Enter hours, minutes and seconds respectively: 8

12

15

TIME DIFFERENCE: 11:33:52 - 8:12:15 = 3:21:37

# 57. **C++ Program to Solve any Linear Equation in One Variable**

This is a C++ Program to solve linear equation. This problem shows, solution to linear equations of N variable in general.

#if !defined(MATRIX\_H)

#define MATRIX\_H

#include <stdio.h>

#include <iostream>

#include <tchar.h>

#include <math.h>

#include <stdlib.h>

class CMatrix

{

private:

int m\_rows;

int m\_cols;

char m\_name[128];

CMatrix();

public:

double \*\*m\_pData;

CMatrix(const char \*name, int rows, int cols) :

m\_rows(rows), m\_cols(cols)

{

strcpy(m\_name, name);

m\_pData = new double\*[m\_rows];

for (int i = 0; i < m\_rows; i++)

m\_pData[i] = new double[m\_cols];

for (int i = 0; i < m\_rows; i++)

{

for (int j = 0; j < m\_cols; j++)

{

m\_pData[i][j] = 0.0;

}

}

}

CMatrix(const CMatrix &other)

{

strcpy(m\_name, other.m\_name);

m\_rows = other.m\_rows;

m\_cols = other.m\_cols;

m\_pData = new double\*[m\_rows];

for (int i = 0; i < m\_rows; i++)

m\_pData[i] = new double[m\_cols];

for (int i = 0; i < m\_rows; i++)

{

for (int j = 0; j < m\_cols; j++)

{

m\_pData[i][j] = other.m\_pData[i][j];

}

}

}

~CMatrix()

{

for (int i = 0; i < m\_rows; i++)

delete[] m\_pData[i];

delete[] m\_pData;

m\_rows = m\_cols = 0;

}

void SetName(const char \*name)

{

strcpy(m\_name, name);

}

const char\* GetName() const

{

return m\_name;

}

void GetInput()

{

std::cin >> \*this;

}

void FillSimulatedInput()

{

static int factor1 = 1, factor2 = 2;

std::cout << "**\n\n**Enter Input For Matrix : " << m\_name << " Rows: "

<< m\_rows << " Cols: " << m\_cols << "**\n**";

for (int i = 0; i < m\_rows; i++)

{

for (int j = 0; j < m\_cols; j++)

{

std::cout << "Input For Row: " << i + 1 << " Col: " << j

+ 1 << " = ";

int data = ((i + 1) \* factor1) + (j + 1) \* factor2;

m\_pData[i][j] = data / 10.2;

std::cout << m\_pData[i][j] << "**\n**";

factor1 += (rand() % 4);

factor2 += (rand() % 3);

}

std::cout << "**\n**";

}

std::cout << "**\n**";

}

double Determinant()

{

double det = 0;

double \*\*pd = m\_pData;

switch (m\_rows)

{

case 2:

{

det = pd[0][0] \* pd[1][1] - pd[0][1] \* pd[1][0];

return det;

}

break;

case 3:

{

*/\*\*\**

*a b c*

*d e f*

*g h i*

*a b c a b c*

*d e f d e f*

*g h i g h i*

*// det (A) = aei + bfg + cdh - afh - bdi - ceg.*

*\*\*\*/*

double a = pd[0][0];

double b = pd[0][1];

double c = pd[0][2];

double d = pd[1][0];

double e = pd[1][1];

double f = pd[1][2];

double g = pd[2][0];

double h = pd[2][1];

double i = pd[2][2];

double det = (a \* e \* i + b \* f \* g + c \* d \* h); // - a\*f\*h - b\*d\*i - c\*e\*g);

det = det - a \* f \* h;

det = det - b \* d \* i;

det = det - c \* e \* g;

//std::cout << \*this;

//std::cout << "deter: " << det << " \n";

return det;

}

break;

case 4:

{

CMatrix \*temp[4];

for (int i = 0; i < 4; i++)

temp[i] = new CMatrix("", 3, 3);

for (int k = 0; k < 4; k++)

{

for (int i = 1; i < 4; i++)

{

int j1 = 0;

for (int j = 0; j < 4; j++)

{

if (k == j)

continue;

temp[k]->m\_pData[i - 1][j1++]

= this->m\_pData[i][j];

}

}

}

double det = this->m\_pData[0][0] \* temp[0]->Determinant()

- this->m\_pData[0][1] \* temp[1]->Determinant()

+ this->m\_pData[0][2] \* temp[2]->Determinant()

- this->m\_pData[0][3] \* temp[3]->Determinant();

return det;

}

break;

case 5:

{

CMatrix \*temp[5];

for (int i = 0; i < 5; i++)

temp[i] = new CMatrix("", 4, 4);

for (int k = 0; k < 5; k++)

{

for (int i = 1; i < 5; i++)

{

int j1 = 0;

for (int j = 0; j < 5; j++)

{

if (k == j)

continue;

temp[k]->m\_pData[i - 1][j1++]

= this->m\_pData[i][j];

}

}

}

double det = this->m\_pData[0][0] \* temp[0]->Determinant()

- this->m\_pData[0][1] \* temp[1]->Determinant()

+ this->m\_pData[0][2] \* temp[2]->Determinant()

- this->m\_pData[0][3] \* temp[3]->Determinant()

+ this->m\_pData[0][4] \* temp[4]->Determinant();

return det;

}

case 6:

case 7:

case 8:

case 9:

case 10:

case 11:

case 12:

default:

{

int DIM = m\_rows;

CMatrix \*\*temp = new CMatrix\*[DIM];

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

temp[i] = new CMatrix("", DIM - 1, DIM - 1);

for (int k = 0; k < DIM; k++)

{

for (int i = 1; i < DIM; i++)

{

int j1 = 0;

for (int j = 0; j < DIM; j++)

{

if (k == j)

continue;

temp[k]->m\_pData[i - 1][j1++]

= this->m\_pData[i][j];

}

}

}

double det = 0;

for (int k = 0; k < DIM; k++)

{

if ((k % 2) == 0)

det = det + (this->m\_pData[0][k]

\* temp[k]->Determinant());

else

det = det - (this->m\_pData[0][k]

\* temp[k]->Determinant());

}

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

delete temp[i];

delete[] temp;

return det;

}

break;

}

}

CMatrix& operator =(const CMatrix &other)

{

if (this->m\_rows != other.m\_rows || this->m\_cols != other.m\_cols)

{

std::cout

<< "WARNING: Assignment is taking place with by changing the number of rows and columns of the matrix";

}

for (int i = 0; i < m\_rows; i++)

delete[] m\_pData[i];

delete[] m\_pData;

m\_rows = m\_cols = 0;

strcpy(m\_name, other.m\_name);

m\_rows = other.m\_rows;

m\_cols = other.m\_cols;

m\_pData = new double\*[m\_rows];

for (int i = 0; i < m\_rows; i++)

m\_pData[i] = new double[m\_cols];

for (int i = 0; i < m\_rows; i++)

{

for (int j = 0; j < m\_cols; j++)

{

m\_pData[i][j] = other.m\_pData[i][j];

}

}

return \*this;

}

CMatrix CoFactor()

{

CMatrix cofactor("COF", m\_rows, m\_cols);

if (m\_rows != m\_cols)

return cofactor;

if (m\_rows < 2)

return cofactor;

else if (m\_rows == 2)

{

cofactor.m\_pData[0][0] = m\_pData[1][1];

cofactor.m\_pData[0][1] = -m\_pData[1][0];

cofactor.m\_pData[1][0] = -m\_pData[0][1];

cofactor.m\_pData[1][1] = m\_pData[0][0];

return cofactor;

}

else if (m\_rows >= 3)

{

int DIM = m\_rows;

CMatrix \*\*\*temp = new CMatrix\*\*[DIM];

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

temp[i] = new CMatrix\*[DIM];

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

for (int j = 0; j < DIM; j++)

temp[i][j] = new CMatrix("", DIM - 1, DIM - 1);

for (int k1 = 0; k1 < DIM; k1++)

{

for (int k2 = 0; k2 < DIM; k2++)

{

int i1 = 0;

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

{

int j1 = 0;

for (int j = 0; j < DIM; j++)

{

if (k1 == i || k2 == j)

continue;

temp[k1][k2]->m\_pData[i1][j1++]

= this->m\_pData[i][j];

}

if (k1 != i)

i1++;

}

}

}

bool flagPositive = true;

for (int k1 = 0; k1 < DIM; k1++)

{

flagPositive = ((k1 % 2) == 0);

for (int k2 = 0; k2 < DIM; k2++)

{

if (flagPositive == true)

{

cofactor.m\_pData[k1][k2]

= temp[k1][k2]->Determinant();

flagPositive = false;

}

else

{

cofactor.m\_pData[k1][k2]

= -temp[k1][k2]->Determinant();

flagPositive = true;

}

}

}

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

for (int j = 0; j < DIM; j++)

delete temp[i][j];

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

delete[] temp[i];

delete[] temp;

}

return cofactor;

}

CMatrix Adjoint()

{

CMatrix cofactor("COF", m\_rows, m\_cols);

CMatrix adj("ADJ", m\_rows, m\_cols);

if (m\_rows != m\_cols)

return adj;

cofactor = this->CoFactor();

// adjoint is transpose of a cofactor of a matrix

for (int i = 0; i < m\_rows; i++)

{

for (int j = 0; j < m\_cols; j++)

{

adj.m\_pData[j][i] = cofactor.m\_pData[i][j];

}

}

return adj;

}

CMatrix Transpose()

{

CMatrix trans("TR", m\_cols, m\_rows);

for (int i = 0; i < m\_rows; i++)

{

for (int j = 0; j < m\_cols; j++)

{

trans.m\_pData[j][i] = m\_pData[i][j];

}

}

return trans;

}

CMatrix Inverse()

{

CMatrix cofactor("COF", m\_rows, m\_cols);

CMatrix inv("INV", m\_rows, m\_cols);

if (m\_rows != m\_cols)

return inv;

// to find out Determinant

double det = Determinant();

cofactor = this->CoFactor();

// inv = transpose of cofactor / Determinant

for (int i = 0; i < m\_rows; i++)

{

for (int j = 0; j < m\_cols; j++)

{

inv.m\_pData[j][i] = cofactor.m\_pData[i][j] / det;

}

}

return inv;

}

CMatrix operator +(const CMatrix &other)

{

if (this->m\_rows != other.m\_rows || this->m\_cols != other.m\_cols)

{

std::cout

<< "Addition could not take place because number of rows and columns are different between the two matrices";

return \*this;

}

CMatrix result("", m\_rows, m\_cols);

for (int i = 0; i < m\_rows; i++)

{

for (int j = 0; j < m\_cols; j++)

{

result.m\_pData[i][j] = this->m\_pData[i][j]

+ other.m\_pData[i][j];

}

}

return result;

}

CMatrix operator -(const CMatrix &other)

{

if (this->m\_rows != other.m\_rows || this->m\_cols != other.m\_cols)

{

std::cout

<< "Subtraction could not take place because number of rows and columns are different between the two matrices";

return \*this;

}

CMatrix result("", m\_rows, m\_cols);

for (int i = 0; i < m\_rows; i++)

{

for (int j = 0; j < m\_cols; j++)

{

result.m\_pData[i][j] = this->m\_pData[i][j]

- other.m\_pData[i][j];

}

}

return result;

}

CMatrix operator \*(const CMatrix &other)

{

if (this->m\_cols != other.m\_rows)

{

std::cout

<< "Multiplication could not take place because number of columns of 1st Matrix and number of rows in 2nd Matrix are different";

return \*this;

}

CMatrix result("", this->m\_rows, other.m\_cols);

for (int i = 0; i < this->m\_rows; i++)

{

for (int j = 0; j < other.m\_cols; j++)

{

for (int k = 0; k < this->m\_cols; k++)

{

result.m\_pData[i][j] += this->m\_pData[i][k]

\* other.m\_pData[k][j];

}

}

}

return result;

}

bool operator ==(const CMatrix &other)

{

if (this->m\_rows != other.m\_rows || this->m\_cols != other.m\_cols)

{

std::cout

<< "Comparision could not take place because number of rows and columns are different between the two matrices";

return false;

}

CMatrix result("", m\_rows, m\_cols);

bool bEqual = true;

for (int i = 0; i < m\_rows; i++)

{

for (int j = 0; j < m\_cols; j++)

{

if (this->m\_pData[i][j] != other.m\_pData[i][j])

bEqual = false;

}

}

return bEqual;

}

friend std::istream& operator >>(std::istream &is, CMatrix &m);

friend std::ostream& operator <<(std::ostream &os, const CMatrix &m);

};

std::istream& operator >>(std::istream &is, CMatrix &m)

{

std::cout << "**\n\n**Enter Input For Matrix : " << m.m\_name << " Rows: "

<< m.m\_rows << " Cols: " << m.m\_cols << "**\n**";

for (int i = 0; i < m.m\_rows; i++)

{

for (int j = 0; j < m.m\_cols; j++)

{

std::cout << "Input For Row: " << i + 1 << " Col: " << j + 1

<< " = ";

is >> m.m\_pData[i][j];

}

std::cout << "**\n**";

}

std::cout << "**\n**";

return is;

}

std::ostream& operator <<(std::ostream &os, const CMatrix &m)

{

os << "**\n\n**Matrix : " << m.m\_name << " Rows: " << m.m\_rows << " Cols: "

<< m.m\_cols << "**\n\n**";

for (int i = 0; i < m.m\_rows; i++)

{

os << " | ";

for (int j = 0; j < m.m\_cols; j++)

{

char buf[32];

double data = m.m\_pData[i][j];

if (m.m\_pData[i][j] > -0.00001 && m.m\_pData[i][j] < 0.00001)

data = 0;

sprintf(buf, "%10.2lf ", data);

os << buf;

}

os << "|**\n**";

}

os << "**\n\n**";

return os;

}

#endif

int main()

{

CMatrix a("A", 6, 6);

CMatrix b("B", 6, 1);

a.FillSimulatedInput();

b.FillSimulatedInput();

std::cout << a << "**\n** Determinant : ";

std::cout << a.Determinant() << "**\n**";

std::cout << b << "**\n** Determinant : ";

std::cout << b.Determinant() << "**\n**";

CMatrix ainv = a.Inverse();

CMatrix q = a \* ainv;

q.SetName("A \* A'");

std::cout << q << "**\n**";

CMatrix x = ainv \* b;

x.SetName("X");

std::cout << x << "**\n**";

CMatrix y = a \* x; // we will get B

y.SetName("Y");

std::cout << y << "**\n**";

return 0;

}

Output:

$ g++ SolveLinearEquation.cpp

$ a.out

Enter Input For Matrix : A Rows: 6 Cols: 6

Input For Row: 1 Col: 1 = 0.294118

Input For Row: 1 Col: 2 = 0.980392

Input For Row: 1 Col: 3 = 1.86275

Input For Row: 1 Col: 4 = 2.84314

Input For Row: 1 Col: 5 = 3.62745

Input For Row: 1 Col: 6 = 5.58824

Input For Row: 2 Col: 1 = 2.94118

Input For Row: 2 Col: 2 = 4.11765

Input For Row: 2 Col: 3 = 5.88235

Input For Row: 2 Col: 4 = 8.43137

Input For Row: 2 Col: 5 = 10.3922

Input For Row: 2 Col: 6 = 12.3529

Input For Row: 3 Col: 1 = 8.13725

Input For Row: 3 Col: 2 = 9.70588

Input For Row: 3 Col: 3 = 12.0588

Input For Row: 3 Col: 4 = 15.098

Input For Row: 3 Col: 5 = 17.8431

Input For Row: 3 Col: 6 = 20.5882

Input For Row: 4 Col: 1 = 14.902

Input For Row: 4 Col: 2 = 18.2353

Input For Row: 4 Col: 3 = 21.4706

Input For Row: 4 Col: 4 = 24.7059

Input For Row: 4 Col: 5 = 27.549

Input For Row: 4 Col: 6 = 31.1765

Input For Row: 5 Col: 1 = 24.902

Input For Row: 5 Col: 2 = 27.9412

Input For Row: 5 Col: 3 = 32.451

Input For Row: 5 Col: 4 = 36.0784

Input For Row: 5 Col: 5 = 39.7059

Input For Row: 5 Col: 6 = 43.9216

Input For Row: 6 Col: 1 = 36.3725

Input For Row: 6 Col: 2 = 39.6078

Input For Row: 6 Col: 3 = 43.8235

Input For Row: 6 Col: 4 = 47.2549

Input For Row: 6 Col: 5 = 51.3725

Input For Row: 6 Col: 6 = 55.2941

Enter Input For Matrix : B Rows: 6 Cols: 1

Input For Row: 1 Col: 1 = 9.41176

Input For Row: 2 Col: 1 = 15.7843

Input For Row: 3 Col: 1 = 22.7451

Input For Row: 4 Col: 1 = 29.902

Input For Row: 5 Col: 1 = 37.1569

Input For Row: 6 Col: 1 = 44.6078

Matrix : A Rows: 6 Cols: 6

| 0.29 0.98 1.86 2.84 3.63 5.59 |

| 2.94 4.12 5.88 8.43 10.39 12.35 |

| 8.14 9.71 12.06 15.10 17.84 20.59 |

| 14.90 18.24 21.47 24.71 27.55 31.18 |

| 24.90 27.94 32.45 36.08 39.71 43.92 |

| 36.37 39.61 43.82 47.25 51.37 55.29 |

Determinant : -11.9339

Matrix : B Rows: 6 Cols: 1

| 9.41 |

| 15.78 |

| 22.75 |

| 29.90 |

| 37.16 |

| 44.61 |

Determinant : -1.#IND

Matrix : A \* A' Rows: 6 Cols: 6

| 1.00 0.00 0.00 0.00 0.00 0.00 |

| 0.00 1.00 0.00 0.00 0.00 0.00 |

| 0.00 0.00 1.00 0.00 0.00 0.00 |

| 0.00 0.00 0.00 1.00 0.00 0.00 |

| 0.00 0.00 0.00 0.00 1.00 0.00 |

| 0.00 0.00 0.00 0.00 0.00 1.00 |

Matrix : X Rows: 6 Cols: 1

| 0.82 |

| 3.47 |

| -9.38 |

| 7.71 |

| -5.76 |

| 3.98 |

Matrix : Y Rows: 6 Cols: 1

| 9.41 |

| 15.78 |

| 22.75 |

| 29.90 |

| 37.16 |

| 44.61 |

# 58. **C++ Program to Use rand and srand Functions**

1. #include <iostream>
2. #include <stdlib.h>
3. #include <time.h>
5. using namespace std;
7. int main(int argc, char \*\*argv)
8. {
9. cout << "First number: " << rand() % 100;
10. srand(time(NULL));
11. cout << "**\n**Random number: " << rand() % 100;
12. srand(1);
13. cout << "**\n**Again the first number: " << rand() % 100;
14. return 0;
15. }

Output:

$ g++ RandSrandFunctions.cpp

$ a.out

First number: 41

Random number: 98

Again the first number: 41

# 59.**C++ Program to Generate Randomized Sequence of Given Range of Numbers**

1. #include <iostream>
2. #include <cstdlib>
3. #include <time.h>
5. const int LOW = 1;
6. const int HIGH = 32000;
8. using namespace std;
10. int main()
11. {
12. int randomNumber;
13. time\_t seconds;
14. time(&seconds);
15. srand((unsigned int) seconds);
16. for (int i = 0; i < 10; i++)
17. {
18. randomNumber = rand() % (HIGH - LOW + 1) + LOW;
20. cout << randomNumber << " ";
21. }
22. cout << "...";
23. return 0;
24. }

Output:

$ g++ RandomizedSequenceOfNumbers.cpp

$ a.out

312 7423 23444 16008 31816 1823 29315 17424 11753 18384 ...

# 60. **C++ Program to Find Path Between Two Nodes in a Graph**

# This is a C++ Program to check and find if the path between two nodes exists. By running DFS on given graph we can find out whether path exists between two nodes.

# Here is source code of the C++ Program to Find Path Between Two Nodes in a Graph. The C++ program is successfully compiled and run on a Linux system. The program output is also shown below.

#include <iostream>

#include <list>

using namespace std;

// This class represents a directed graph using adjacency list representation

class Graph

{

int V; // No. of vertices

list<int> \*adj; // Pointer to an array containing adjacency lists

public:

Graph(int V); // Constructor

void addEdge(int v, int w); // function to add an edge to graph

bool isReachable(int s, int d); // returns true if there is a path from s to d

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int> [V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

// A BFS based function to check whether d is reachable from s.

bool Graph::isReachable(int s, int d)

{

// Base case

if (s == d)

return true;

// Mark all the vertices as not visited

bool \*visited = new bool[V];

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

visited[i] = false;

// Create a queue for BFS

list<int> queue;

// Mark the current node as visited and enqueue it

visited[s] = true;

queue.push\_back(s);

// it will be used to get all adjacent vertices of a vertex

list<int>::iterator i;

while (!queue.empty())

{

// Dequeue a vertex from queue and print it

s = queue.front();

queue.pop\_front();

// Get all adjacent vertices of the dequeued vertex s

// If a adjacent has not been visited, then mark it visited

// and enqueue it

for (i = adj[s].begin(); i != adj[s].end(); ++i)

{

// If this adjacent node is the destination node, then return true

if (\*i == d)

return true;

// Else, continue to do BFS

if (!visited[\*i])

{

visited[\*i] = true;

queue.push\_back(\*i);

}

}

}

return false;

}

// Driver program to test methods of graph class

int main()

{

// Create a graph given in the above diagram

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Enter the source and destination vertices: (0-3)";

int u, v;

cin >> u >> v;

if (g.isReachable(u, v))

cout << "**\n**There is a path from " << u << " to " << v;

else

cout << "**\n**There is no path from " << u << " to " << v;

int temp;

temp = u;

u = v;

v = temp;

if (g.isReachable(u, v))

cout << "**\n**There is a path from " << u << " to " << v;

else

cout << "**\n**There is no path from " << u << " to " << v;

return 0;

}

Output:

$ g++ PathBetweenNodes.cpp

$ a.out

Enter the source and destination vertices: (0-3)

1 3

There is a path from 1 to 3

There is no path from 3 to 1

Enter the source and destination vertices: (0-3)

2 3

There is a path from 2 to 3

There is no path from 3 to 2

# 61. **C++ Program to Generate N Number of Passwords of Length M Each**

# Output:

$ g++ NPasswordMLength.cpp

$ a.out

Enter the length of the password: 3

1, 7, 4, The Passwords are: 174

147

714

741

471

417

Enter the length of the password: 5

1, 7, 4, 0, 9, The Passwords are: 17409

17490

17049

17094

17904

17940

14709

14790

14079

14097

14907

…

1. #include<iostream>
2. #include<conio.h>
3. #include<stdlib.h>
5. using namespace std;
7. void permute(int \*a, int k, int size)
8. {
9. if (k == size)
10. {
11. for (int i = 0; i < size; i++)
12. {
13. cout << \*(a + i);
14. }
15. cout << endl;
16. }
17. else
18. {
19. for (int i = k; i < size; i++)
20. {
21. int temp = a[k];
22. a[k] = a[i];
23. a[i] = temp;
25. permute(a, k + 1, size);
27. temp = a[k];
28. a[k] = a[i];
29. a[i] = temp;
30. }
31. }
33. }
34. int main(int argc, char \*\*argv)
35. {
36. cout << "Enter the length of the password: ";
37. int m;
38. cin >> m;
39. int a[m];
40. for (int i = 0; i < m; i++)
41. {
42. */\*generates random number between 1 and 10\*/*
43. a[i] = rand() % 10;
44. }
45. for (int i = 0; i < m; i++)
46. {
47. cout << a[i] << ", ";
48. }
49. cout << "The Passwords are: ";
50. permute(a, 0, m);
51. }

# 

# 62. **C++ Program to Perform LU Decomposition of any Matrix**

# This is a C++ Program to perform LU Decomposition of any matrix. In numerical analysis, LU decomposition (where ‘LU’ stands for ‘Lower Upper’, and also called LU factorization) factors a matrix as the product of a lower triangular matrix and an upper triangular matrix. The product sometimes includes a permutation matrix as well. The LU decomposition can be viewed as the matrix form of Gaussian elimination. Computers usually solve square systems of linear equations using the LU decomposition, and it is also a key step when inverting a matrix, or computing the determinant of a matrix

Output:

$ g++ LUDecomposition.cpp

$ a.out

Enter size of 2d array(Square matrix) : 3

Enter values no:0, 0: 1

Enter values no:0, 1: 1

Enter values no:0, 2: -1

Enter values no:1, 0: 2

Enter values no:1, 1: -1

Enter values no:1, 2: 3

Enter values no:2, 0: 3

Enter values no:2, 1: 1

Enter values no:2, 2: -1

L Decomposition

1.000000 0.000000 0.000000

2.000000 -3.000000 0.000000

3.000000 -2.000000 -1.333333

U Decomposition

1.000000 1.000000 -1.000000

0.000000 1.000000 -1.666667

0.000000 0.000000 1.000000

1. #include<iostream>
2. #include<cstdio>
4. using namespace std;
6. int main(int argc, char \*\*argv)
7. {
8. void lu(float[][10], float[][10], float[][10], int n);
9. void output(float[][10], int);
10. float a[10][10], l[10][10], u[10][10];
11. int n = 0, i = 0, j = 0;
12. cout << "Enter size of 2d array(Square matrix) : ";
13. cin >> n;
14. for (i = 0; i < n; i++)
15. {
16. for (j = 0; j < n; j++)
17. {
18. cout << "Enter values no:" << i << ", " << j << ": ";
19. cin >> a[i][j];
20. }
21. }
22. lu(a, l, u, n);
23. cout << "**\n**L Decomposition**\n\n**";
24. output(l, n);
25. cout << "**\n**U Decomposition**\n\n**";
26. output(u, n);
27. return 0;
28. }
29. void lu(float a[][10], float l[][10], float u[][10], int n)
30. {
31. int i = 0, j = 0, k = 0;
32. for (i = 0; i < n; i++)
33. {
34. for (j = 0; j < n; j++)
35. {
36. if (j < i)
37. l[j][i] = 0;
38. else
39. {
40. l[j][i] = a[j][i];
41. for (k = 0; k < i; k++)
42. {
43. l[j][i] = l[j][i] - l[j][k] \* u[k][i];
44. }
45. }
46. }
47. for (j = 0; j < n; j++)
48. {
49. if (j < i)
50. u[i][j] = 0;
51. else if (j == i)
52. u[i][j] = 1;
53. else
54. {
55. u[i][j] = a[i][j] / l[i][i];
56. for (k = 0; k < i; k++)
57. {
58. u[i][j] = u[i][j] - ((l[i][k] \* u[k][j]) / l[i][i]);
59. }
60. }
61. }
62. }
63. }
64. void output(float x[][10], int n)
65. {
66. int i = 0, j = 0;
67. for (i = 0; i < n; i++)
68. {
69. for (j = 0; j < n; j++)
70. {
71. printf("%f ", x[i][j]);
72. }
73. cout << "**\n**";
74. }
75. }

# 63. **C++ Program to Find if a Matrix is Invertible or Not**

# This is a C++ Program to Find if a Matrix is Invertible or Not.

# **Problem Description**

# The program takes a matrix and checks if it is invertible. An invertible matrix is a square matrix whose determinant exists.

# **Problem Solution**

# 1. The program takes a square matrix.

# 2. Using a function, determinant of the matrix is calculated.

# 3. If determinant is equal to 0, it is not an invertible matrix.

# 4. Else it is an invertible matrix.

# 5. The result is printed.

# 6. Exit.

1. #include<iostream>
2. #include<math.h>
3. using namespace std;
4. double det(int n, double mat[10][10])
5. {
6. int subj = 0, subi = 0, i, j, c;
7. double submat[10][10], d = 0;
8. if (n == 2)
9. return ((mat[0][0] \* mat[1][1]) - (mat[1][0] \* mat[0][1]));
10. else
11. {
12. for (c = 0; c < n; c++)
13. {
14. for (i = 1; i < n; i++)
15. {
16. for (j = 0; j < n; j++)
17. {
18. if (j == c)
19. continue;
20. submat[subi][subj] = mat[i][j];
21. subj++;
22. }
23. subi++;
24. }
25. d = d + (pow(-1, c) \* mat[0][c] \* det(n - 1, submat));
26. }
27. }
28. return d;
29. }
30. int main(int argc, char \*\*argv)
31. {
32. int n, i, j;
33. double mat[10][10];
34. cout << "Enter order of the matrix : ";
35. cin >> n;
36. cout << "Enter the elements of the matrix : ";
37. for (i = 0; i < n; i++)
38. for (j = 0; j < n; j++)
39. cin >> mat[j][i];
40. if (det(n, mat) != 0)
41. cout << "The given matrix is invertible.**\n** ";
42. else
43. cout << "The given matrix is not invertible.**\n** ";
44. for (i = 0; i < n; i++)
45. {
46. for (j = 0; j < n; j++)
47. cout << mat[i][j] << " ";
48. cout << "**\n** ";
49. }
50. }

# 64.**C++ Program to Find if a Matrix is Orthogonal**

This is a C++ Program to Find if a Matrix is Orthogonal.

**Problem Description**

The program takes a matrix and checks if it is orthogonal. A square matrix is defined as an orthogonal matrix if its transpose matrix is same as its inverse matrix.

**Problem Solution**

1. The program takes a square matrix.

2. The transpose of the matrix is found.

3. The product of the matrix and transpose are calculated and stored.

4. If the product is an identity matrix, then it is orthogonal.

5. Else it is not an orthogonal matrix.

6. The result is printed.

7. Exit.

1. #include<iostream>
2. using namespace std;
3. int main ()
4. {
5. int m, n, p, i, j, k, sum = 0;
6. int A[10][10], T[10][10], P[10][10];
7. cout << "Enter number of rows and columns : ";
8. cin >> m >> n;
9. if (m != n)
10. {
11. cout << "Matrix is not a square matrix!";
12. exit(0);
13. }
14. cout << "Enter elements of matrix : ";
15. for (i = 0; i < m; i++)
16. for (j = 0; j < n; j++)
17. cin >> A[i][j];
19. for (i = 0; i < m; i++)
20. for (j = 0; j < n; j++)
21. T[j][i] = A[i][j];
23. for (i = 0; i < m; i++)
24. {
25. for (j = 0; j < n; j++)
26. {
27. for ( k = 0 ; k < n ; k++ )
28. sum = sum + A[i][k] \* T[k][j];
29. P[i][j] = sum;
30. sum = 0;
31. }
32. }
33. for (i = 0; i < m; i++)
34. {
35. for (j = 0; j < n; j++)
36. {
37. if (i == j)
38. if (P[i][j] != 1 )
39. break;
40. else
41. {
42. if (P[i][j] != 0 )
43. break;
44. }
45. }
46. if (j != m)
47. break;
48. }
49. if (i != m)
50. cout << "Matrix is not orthogonal.**\n** ";
51. else
52. cout << "Matrix is orthogonal.**\n** ";
53. for (i = 0; i < m; i++)
54. {
55. for (j = 0; j < n; j++)
56. cout << A[i][j] << " ";
57. cout << "**\n** ";
58. }
59. return 0;
60. }

# 65.**C++ Program to Find if an Array is a Sparse Matrix**

This is a C++ Program to Find if an Array is a Sparse Matrix.

**Problem Description**

The program takes an array and checks if it is a sparse matrix. A sparse matrix is a matrix which has maximum elements equal to 0.

**Problem Solution**

1. The program takes the number of rows and columns of the matrix.

2. Then the elements are entered.

3. If the matrix contains maximum number of elements as 0, then it is a sparse matrix.

4. Else not.

5. The result is printed.

6. Exit.

**C++ Program/Source code**

Here is the source code of C++ Program to Find if an Array is a Sparse Matrix. The program output is shown below.

1. #include<iostream>
2. using namespace std;
3. int main ()
4. {
5. int A[10][10], i, j, m, n, count = 0;
6. cout << "Enter number of rows and columns : ";
7. cin >> m >> n;
8. cout << "Enter array elements : ";
9. for (i = 0; i < m; i++)
10. {
11. for (j = 0; j < n; j++)
12. {
13. cin >> A[i][j];
14. if (A[i][j] == 0)
15. count++;
16. }
17. }
18. if (count > ((m \* n) / 2))
19. cout << "The matrix is a sparse matrix.**\n** ";
20. else
21. cout << "The given matrix is not a sparse matrix.**\n** ";
22. for (i = 0; i < m; i++)
23. {
24. for (j = 0; j < n; j++)
25. cout << A[i][j] << " ";
26. cout << "**\n** ";
27. }
28. return 0;
29. }

# 66. **C++ Program to Implement Sparse Matrix**

This C++ Program demonstrates the implementation of Sparse Matrix.

Here is source code of the C++ Program to demonstrate the implementation of Sparse Matrix. The C++ program is successfully compiled and run on a Linux system. The program output is also shown below.

$ **g++** sparse\_matrix.cpp

$ a.out

the sparse Matrix is:

1 NULL 2

NULL 3 NULL

4 6 NULL

The Size of Sparse Matrix is 5

-----

**(**program exited with code: 1**)**

1. */\**
2. *\* C++ Program to Implement Sparse Matrix*
3. *\*/*
4. #include <iostream>
5. #include <iomanip>
6. #include <string>
7. using namespace std;
9. */\**
10. *\* Class List Declaration*
11. *\*/*
12. class List
13. {
14. private:
15. int index;
16. int value;
17. List \*nextindex;
18. public:
19. List(int index)
20. {
21. this->index = index;
22. nextindex = NULL;
23. value = NULL;
24. }
25. List()
26. {
27. index = -1;
28. value = NULL;
29. nextindex = NULL;
30. }
31. void store(int index, int value)
32. {
33. List \*current = this;
34. List \*previous = NULL;
35. List \*node = new List(index);
36. node->value = value;
37. while (current != NULL && current->index < index)
38. {
39. previous = current;
40. current = current->nextindex;
41. }
42. if (current == NULL)
43. {
44. previous->nextindex = node;
45. }
46. else
47. {
48. if (current->index == index)
49. {
50. cout<<"DUPLICATE INDEX"<<endl;
51. return;
52. }
53. previous->nextindex = node;
54. node->nextindex = current;
55. }
56. return;
57. }
59. int fetch(int index)
60. {
61. List \*current = this;
62. int value = NULL;
63. while (current != NULL && current->index != index)
64. {
65. current = current->nextindex;
66. }
67. if (current != NULL)
68. {
69. value = current->value;
70. }
71. else
72. {
73. value = NULL;
74. }
75. return value;
76. }
78. int elementCount()
79. {
80. int elementCount = 0;
81. List \*current = this->nextindex;
82. for ( ; (current != NULL); current = current->nextindex)
83. {
84. elementCount++;
85. }
86. return elementCount;
87. }
88. };
89. */\**
90. *\* Class SpareArray Declaration*
91. *\*/*
92. class SparseArray
93. {
94. private:
95. List \*start;
96. int index;
97. public:
98. SparseArray(int index)
99. {
100. start = new List();
101. this->index = index;
102. }
103. void store(int index, int value)
104. {
105. if (index >= 0 && index < this->index)
106. {
107. if (value != NULL)
108. start->store(index, value);
109. }
110. else
111. {
112. cout<<"INDEX OUT OF BOUNDS"<<endl;
113. }
114. }
115. int fetch(int index)
116. {
117. if (index >= 0 && index < this->index)
118. return start->fetch(index);
119. else
120. {
121. cout<<"INDEX OUT OF BOUNDS"<<endl;
122. return NULL;
123. }
124. }
125. int elementCount()
126. {
127. return start->elementCount();
128. }
129. };
130. */\**
131. *\* Class SparseMatrix Declaration*
132. *\*/*
133. class SparseMatrix
134. {
135. private:
136. int N;
137. SparseArray \*\*sparsearray;
138. public:
139. SparseMatrix(int N)
140. {
141. this->N = N;
142. sparsearray = new SparseArray\* [N];
143. for (int index = 0; index < N; index++)
144. {
145. sparsearray[index] = new SparseArray(N);
146. }
147. }
148. void store(int rowindex, int colindex, int value)
149. {
150. if (rowindex < 0 || rowindex > N)
151. {
152. cout<<"row index out of bounds"<<endl;
153. return;
154. }
155. if (colindex < 0 || colindex > N)
156. {
157. cout<<"col index out of bounds"<<endl;
158. return;
159. }
160. sparsearray[rowindex]->store(colindex, value);
161. }
163. int get(int rowindex, int colindex)
164. {
165. if (rowindex < 0 || colindex > N)
166. {
167. cout<<"row index out of bounds"<<endl;
168. return 0;
169. }
170. if (rowindex < 0 || colindex > N)
171. {
172. cout<<"col index out of bounds"<<endl;
173. return 0;
174. }
175. return (sparsearray[rowindex]->fetch(colindex));
176. }
177. int elementCount()
178. {
179. int count = 0;
180. for (int index = 0; index < N; index++)
181. {
182. count += sparsearray[index]->elementCount();
183. }
184. return count;
185. }
186. };
187. */\**
188. *\* Main*
189. *\*/*
190. int main()
191. {
192. int iarray[3][3];
193. iarray[0][0] = 1;
194. iarray[0][1] = NULL;
195. iarray[0][2] = 2;
196. iarray[1][0] = NULL;
197. iarray[1][1] = 3;
198. iarray[1][2] = NULL;
199. iarray[2][0] = 4;
200. iarray[2][1] = 6;
201. iarray[2][2] = NULL;
202. SparseMatrix sparseMatrix(3);
203. for (int rowindex = 0; rowindex < 3; rowindex++)
204. {
205. for (int colindex = 0; colindex < 3; colindex++)
206. {
207. sparseMatrix.store(rowindex, colindex, iarray[rowindex][colindex]);
208. }
209. }
211. cout<<"the sparse Matrix is: "<<endl;
212. for (int rowindex = 0; rowindex < 3; rowindex++)
213. {
214. for (int colindex = 0; colindex < 3; colindex++)
215. {
216. if (sparseMatrix.get(rowindex, colindex) == NULL)
217. cout<<"NULL"<< "**\t**";
218. else
219. cout<<sparseMatrix.get(rowindex, colindex) << "**\t**";
220. }
221. cout<<endl;
222. }
223. cout<<"The Size of Sparse Matrix is "<<sparseMatrix.elementCount()<<endl;
225. }

# 67. **C++ Program to Compute Determinant of a Matrix**

This C++ program displays the value of the determinant of a particular matrix. A determinant is a value associated with a square matrix. It can be computed from the entries of the matrix by a specific arithmetic expression, while other ways to determine its value exist as well.

Here is the source code of the C++ Program to Compute Determinant of a Matrix. The C++ program is successfully compiled and run on a Linux system. The program output is also shown below.

Output:

$ g++ DeterminantOfMatrix.cpp

$ a.out

Enter the dimension of the matrix:

3

Enter the elements of the matrix:

3 5 2

8 4 8

2 4 7

The determinant of the given matrix is: -164

Enter the dimension of the matrix:

4

Enter the elements of the matrix:

9 5 2 5

9 5 3 7

6 5 4 8

1 5 3 7

The determinant of the given matrix is: 0

1. */\**
2. *\* C++ Program to Find the Determinant of a Given Matrix*
3. *\*/*
4. #include<iostream>
5. #include<math.h>
6. #include<conio.h>
8. using namespace std;
9. double d = 0;
10. double det(int n, double mat[10][10]);
11. double det(int n, double mat[10][10])
12. {
13. double submat[10][10];
14. if (n == 2)
15. return ((mat[0][0] \* mat[1][1]) - (mat[1][0] \* mat[0][1]));
16. else
17. {
18. for (int c = 0; c < n; c++)
19. {
20. int subi = 0; //submatrix's i value
21. for (int i = 1; i < n; i++)
22. {
23. int subj = 0;
24. for (int j = 0; j < n; j++)
25. {
26. if (j == c)
27. continue;
28. submat[subi][subj] = mat[i][j];
29. subj++;
30. }
31. subi++;
33. }
34. d = d + (pow(-1, c) \* mat[0][c] \* det(n - 1, submat));
35. }
36. }
37. return d;
38. }
39. int main(int argc, char \*\*argv)
40. {
42. cout << "Enter the dimension of the matrix:**\n**";
43. int n;
44. cin >> n;
45. double mat[10][10];
46. cout << "Enter the elements of the matrix:**\n**";
47. for (int i = 0; i < n; i++)
48. {
49. for (int j = 0; j < n; j++)
50. {
51. cin >> mat[j][i];
52. }
53. }
54. cout << "The determinant of the given matrix is: " << det(n, mat);
55. return 0;
56. }

\

34. **C++ Program to Implement Interval Tree**

This is a C++ Program to implement interval tree. In computer science, an interval tree is an ordered tree data structure to hold intervals. Specifically, it allows one to efficiently find all intervals that overlap with any given interval or point. It is often used for windowing queries, for instance, to find all roads on a computerized map inside a rectangular viewport, or to find all visible elements inside a three-dimensional scene. A similar data structure is the segment tree.

Output:

$ g++ IntervalTree.cpp

$ a.out

In-order traversal of constructed Interval Tree is

[5, 20] max = 20

[10, 30] max = 30

[12, 15] max = 15

[15, 20] max = 40

[17, 19] max = 40

[30, 40] max = 40

Searching for interval [6,7]

Overlaps with [5, 20]

-----

(program exited with code: 0)

Press return to continue

1. #include <iostream>
3. using namespace std;
5. struct Interval
6. {
7. int low, high;
8. };
10. struct ITNode
11. {
12. Interval \*i; // 'i' could also be a normal variable
13. int max;
14. ITNode \*left, \*right;
15. };
17. // A utility function to create a new Interval Search Tree Node
18. ITNode \* newNode(Interval i)
19. {
20. ITNode \*temp = new ITNode;
21. temp->i = new Interval(i);
22. temp->max = i.high;
23. temp->left = temp->right = NULL;
24. };
26. // A utility function to insert a new Interval Search Tree Node
27. // This is similar to BST Insert. Here the low value of interval
28. // is used tomaintain BST property
29. ITNode \*insert(ITNode \*root, Interval i)
30. {
31. // Base case: Tree is empty, new node becomes root
32. if (root == NULL)
33. return newNode(i);
35. // Get low value of interval at root
36. int l = root->i->low;
38. // If root's low value is smaller, then new interval goes to
39. // left subtree
40. if (i.low < l)
41. root->left = insert(root->left, i);
43. // Else, new node goes to right subtree.
44. else
45. root->right = insert(root->right, i);
47. // Update the max value of this ancestor if needed
48. if (root->max < i.high)
49. root->max = i.high;
51. return root;
52. }
54. // A utility function to check if given two intervals overlap
55. bool doOVerlap(Interval i1, Interval i2)
56. {
57. if (i1.low <= i2.high && i2.low <= i1.high)
58. return true;
59. return false;
60. }
62. // The main function that searches a given interval i in a given
63. // Interval Tree.
64. Interval \*intervalSearch(ITNode \*root, Interval i)
65. {
66. // Base Case, tree is empty
67. if (root == NULL)
68. return NULL;
70. // If given interval overlaps with root
71. if (doOVerlap(\*(root->i), i))
72. return root->i;
74. // If left child of root is present and max of left child is
75. // greater than or equal to given interval, then i may
76. // overlap with an interval is left subtree
77. if (root->left != NULL && root->left->max >= i.low)
78. return intervalSearch(root->left, i);
80. // Else interval can only overlap with right subtree
81. return intervalSearch(root->right, i);
82. }
84. void inorder(ITNode \*root)
85. {
86. if (root == NULL)
87. return;
89. inorder(root->left);
91. cout << "[" << root->i->low << ", " << root->i->high << "]" << " max = "
92. << root->max << endl;
94. inorder(root->right);
95. }
97. int main(int argc, char \*\*argv)
98. {
100. Interval ints[] = { { 15, 20 }, { 10, 30 }, { 17, 19 }, { 5, 20 },
101. { 12, 15 }, { 30, 40 } };
102. int n = sizeof(ints) / sizeof(ints[0]);
103. ITNode \*root = NULL;
104. for (int i = 0; i < n; i++)
105. root = insert(root, ints[i]);
107. cout << "In-order traversal of constructed Interval Tree is**\n**";
108. inorder(root);
110. Interval x = { 6, 7 };
112. cout << "**\n**Searching for interval [" << x.low << "," << x.high << "]";
113. Interval \*res = intervalSearch(root, x);
114. if (res == NULL)
115. cout << "**\n**No Overlapping Interval";
116. else
117. cout << "**\n**Overlaps with [" << res->low << ", " << res->high << "]";
118. }

# 68. **C++ Program for Level Order Traversal of a given Tree using Recursion**

# This is a C++ Program for Level Order Traversal of a Tree using Recursion.

**Problem Description**

Here in this problem we will be traversing the nodes of tree from left to right level by level. First the nodes at level 1 will be printed followed by the level two and so on. This problem is implemented using C++ programming language by recursion method.

Expected Input and Output

**Case 1. Balanced Tree:** When the weight on both the sides of the root node is same.

25

/ \

27 19

/ \ / \

17 91 13 55

Output: 25 27 19 17 91 13 55

**Case 2. Right Skewed Tree:** When the nodes at every level have just a right child.

1

\

2

\

3

\

4

\

5

Output: 1 2 3 4 5

# 

1. */\* C++ Program for level order traversal of a tree using recursion \*/*
2. #include <iostream>
3. #include<stdio.h>
4. using namespace std;
5. struct node
6. {
7. int info;
8. struct node \*left, \*right;
9. };
10. class Tree
11. {
12. public:
13. struct node \*root;
14. struct node \*createnode(int key);
15. int heightoftree(struct node \*newnode);
16. void currentlevel(struct node \*root, int level);
17. Tree()
18. {
19. root = NULL;
20. }
21. };
22. */\**
23. *\* Main Function*
24. *\*/*
25. int main()
26. { Tree b1;
27. */\* Creating First Tree. \*/*
28. struct node \*newnode = b1.createnode(25);
29. newnode->left = b1.createnode(27);
30. newnode->right = b1.createnode(19);
31. newnode->left->left = b1.createnode(17);
32. newnode->left->right = b1.createnode(91);
33. newnode->right->left = b1.createnode(13);
34. newnode->right->right = b1.createnode(55);
35. */\* Sample Tree 1: Balanced Tree*
36. *\* 25*
37. *\* / \*
38. *\* 27 19*
39. *\* / \ / \*
40. *\* 17 91 13 55*
41. *\*/*
42. cout<<"Level Order Traversal of Tree 1 is **\n**";
43. int i;
44. int height = b1.heightoftree(newnode);
45. for(i = 1; i <= height; i++)
46. {
47. b1.currentlevel(newnode,i);
48. }
50. */\* Creating Second Tree \*/*
51. struct node \*node = b1.createnode(1);
52. node->left = b1.createnode(2);
53. node->left->left = b1.createnode(3);
54. node->left->left->left = b1.createnode(4);
55. node->left->left->left->left = b1.createnode(5);
56. */\* Sample Tree 2- Left Skewed Tree*
57. *\* 1*
58. *\* /*
59. *\* 2*
60. *\* /*
61. *\* 3*
62. *\* /*
63. *\* 4*
64. *\* /*
65. *\* 5*
66. *\*/*
67. cout<<"**\n\n**Level Order Traversal of Tree 2 is **\n**";
68. height = b1.heightoftree(node);
69. for(i = 1; i <= height; i++)
70. {
71. b1.currentlevel(node,i);
72. }
74. */\* Creating Third Tree \*/*
75. struct node \*root = b1.createnode(15);
76. */\* Sample Tree 3- Tree having just one root node.*
77. *\* 15*
78. *\*/*
79. cout<<"**\n\n**Level Order Traversal of Tree 3 is **\n**";
80. height = b1.heightoftree(root);
81. for(i = 1; i <= height; i++)
82. {
83. b1.currentlevel(root,i);
84. }
85. return 0;
86. }
87. */\**
88. *\* Function to create nodes*
89. *\*/*
90. struct node \*Tree :: createnode(int key)
91. {
92. struct node \*newnode = new node;
93. newnode->info = key;
94. newnode->left = NULL;
95. newnode->right = NULL;
96. return(newnode);
97. }
98. */\**
99. *\* Function to ascertain the height of a Tree*
100. *\*/*
101. int Tree :: heightoftree(struct node \*root)
102. {
103. int max;
104. if (root!=NULL)
105. {
106. */\* Finding the height of left subtree. \*/*
107. int leftsubtree = heightoftree(root->left);
108. */\* Finding the height of right subtree. \*/*
109. int rightsubtree = heightoftree(root->right);
110. if (leftsubtree > rightsubtree)
111. {
112. max = leftsubtree + 1;
113. return max;
114. }
115. else
116. {
117. max = rightsubtree + 1;
118. return max;
119. }
120. }
121. }
122. */\**
123. *\* Function to print all the nodes left to right of the current level*
124. *\*/*
125. void Tree :: currentlevel(struct node \*root, int level)
126. {
127. if (root != NULL)
128. {
129. if (level == 1)
130. {
131. cout<<root->info<<"**\t**";
132. }
133. else if (level > 1)
134. {
135. currentlevel(root->left, level-1);
136. currentlevel(root->right, level-1);
137. }
138. }
139. }

# 69. **C++ Program For Inorder Tree Traversal without Recursion**

In a binary search tree, the inorder traversal method follows the Left Node Right or Left Root Right policy. In the in-order traversal algorithm, the subtree on the left side of the root node, then the root, and then the subtree on the right side of the root node is traversed.

**Method 1:**

This C++ program, without recursion, displays the nodes of a particular binary tree in an inorder fashion without using recursive traversal.

Output

enter value of root node

6

enter value of node

2

value entered in left

enter value of node

9

value entered in right

enter value of node

3

value entered in right

enter value of node

5

value entered in right

enter value of node

7

value entered in left

printing traversal in inorder

2

3

5

6

7

9

6

1. */\**
2. *\* C++ Program For Inorder Tree Traversal without recursion*
3. *\*/*
4. #include<iostream>
5. using namespace std;
6. #include<conio.h>
7. struct tree
8. {
9. tree \*l, \*r;
10. int data;
11. }\*root = NULL, \*p = NULL, \*s = NULL;
12. struct node
13. {
14. tree \*pt;
15. node \*next;
16. }\*q = NULL, \*top = NULL, \*np = NULL;
17. void create()
18. {
19. int value ,c = 0;
20. while (c < 6)
21. {
22. if (root == NULL)
23. {
24. root = new tree;
25. cout<<"enter value of root node**\n**";
26. cin>>root->data;
27. root->r = NULL;
28. root->l = NULL;
29. }
30. else
31. {
32. p = root;
33. cout<<"enter value of node**\n**";
34. cin>>value;
35. while(true)
36. {
37. if(value < p->data)
38. {
39. if (p->l == NULL)
40. {
41. p->l = new tree;
42. p = p->l;
43. p->data = value;
44. p->l = NULL;
45. p->r = NULL;
46. cout<<"value entered in left**\n**";
47. break;
48. }
49. else if (p->l != NULL)
50. {
51. p = p->l;
52. }
53. }
54. else if (value > p->data)
55. {
56. if (p->r == NULL)
57. {
58. p->r = new tree;
59. p = p->r;
60. p->data = value;
61. p->l = NULL;
62. p->r = NULL;
63. cout<<"value entered in right**\n**";
64. break;
65. }
66. else if(p->r != NULL)
67. {
68. p = p->r;
69. }
70. }
71. }
72. }
73. c++;
74. }
75. }
76. void push(tree \*ptr)
77. {
78. np = new node;
79. np->pt = ptr;
80. np->next = NULL;
81. if (top == NULL)
82. {
83. top = np;
84. }
85. else
86. {
87. q = top;
88. top = np;
89. np->next = q;
90. }
91. }
92. tree \*pop()
93. {
94. if (top == NULL)
95. {
96. cout<<"underflow**\n**";
97. }
98. else
99. {
100. q = top;
101. top = top->next;
102. return(q->pt);
103. delete(q);
104. }
105. }
106. void traverse(tree \*p)
107. {
108. push(p);
109. while (top != NULL)
110. {
111. while (p != NULL)
112. {
113. push(p);
114. p = p->l;
115. }
116. if (top != NULL && p == NULL)
117. {
118. p = pop();
119. cout<<p->data<<endl;
120. p = p->r;
121. }
122. }
123. }
124. int main()
125. {
126. int val;
127. create();
128. cout<<"printing traversal in inorder**\n**";
129. traverse(root);
130. getch();
131. }

# 70.**C++ Program to Implement Array in STL**

This C++ Program demonstrates implementation of Array in STL.

$ **g++** array.cpp

$ a.out

Array Implementation **in** Stl

1.Insert Element into the Array

2.Size of the array

3.Front Element of Array

4.Back Element of Array

5.Display elements of the Array

6.Exit

Enter your Choice: 1

Enter value to be inserted: 2

Array Implementation **in** Stl

1.Insert Element into the Array

2.Size of the array

3.Front Element of Array

4.Back Element of Array

5.Display elements of the Array

6.Exit

Enter your Choice: 1

Enter value to be inserted: 3

Array Implementation **in** Stl

1.Insert Element into the Array

2.Size of the array

3.Front Element of Array

4.Back Element of Array

5.Display elements of the Array

6.Exit

Enter your Choice: 1

Enter value to be inserted: 4

Array Implementation **in** Stl

1.Insert Element into the Array

2.Size of the array

3.Front Element of Array

4.Back Element of Array

5.Display elements of the Array

6.Exit

Enter your Choice: 1

Enter value to be inserted: 5

Array Implementation **in** Stl

1.Insert Element into the Array

2.Size of the array

3.Front Element of Array

4.Back Element of Array

5.Display elements of the Array

6.Exit

Enter your Choice: 1

Enter value to be inserted: 6

Array Implementation **in** Stl

1.Insert Element into the Array

2.Size of the array

3.Front Element of Array

4.Back Element of Array

5.Display elements of the Array

6.Exit

Enter your Choice: 2

Size of the Array: 5

Array Implementation **in** Stl

1.Insert Element into the Array

2.Size of the array

3.Front Element of Array

4.Back Element of Array

5.Display elements of the Array

6.Exit

Enter your Choice: 3

Front Element of the Array: 2

Array Implementation **in** Stl

1.Insert Element into the Array

2.Size of the array

3.Front Element of Array

4.Back Element of Array

5.Display elements of the Array

6.Exit

Enter your Choice: 4

Back Element of the Stack: 6

Array Implementation **in** Stl

1.Insert Element into the Array

2.Size of the array

3.Front Element of Array

4.Back Element of Array

5.Display elements of the Array

6.Exit

Enter your Choice: 5

2 3 4 5 6

Array Implementation **in** Stl

1.Insert Element into the Array

2.Size of the array

3.Front Element of Array

4.Back Element of Array

5.Display elements of the Array

6.Exit

Enter your Choice: 6

-----

**(**program exited with code: 1**)**

Press **return** to **continue**

1. */\**
2. *\* C++ Program to Implement Array in Stl*
3. *\*/*
4. #include <iostream>
5. #include <array>
6. #include <string>
7. #include <cstdlib>
8. using namespace std;
9. int main()
10. {
11. array<int, 5> arr;
12. array<int, 5>::iterator it;
13. int choice, item;
14. arr.fill(0);
15. int count = 0;
16. while (1)
17. {
18. cout<<"**\n**"<<endl;
19. cout<<"Array Implementation in Stl"<<endl;
20. cout<<"**\n**"<<endl;
21. cout<<"1.Insert Element into the Array"<<endl;
22. cout<<"2.Size of the array"<<endl;
23. cout<<"3.Front Element of Array"<<endl;
24. cout<<"4.Back Element of Array"<<endl;
25. cout<<"5.Display elements of the Array"<<endl;
26. cout<<"6.Exit"<<endl;
27. cout<<"Enter your Choice: ";
28. cin>>choice;
29. switch(choice)
30. {
31. case 1:
32. cout<<"Enter value to be inserted: ";
33. cin>>item;
34. arr.at(count) = item;
35. count++;
36. break;
37. case 2:
38. cout<<"Size of the Array: ";
39. cout<<arr.size()<<endl;
40. break;
41. case 3:
42. cout<<"Front Element of the Array: ";
43. cout<<arr.front()<<endl;
44. break;
45. case 4:
46. cout<<"Back Element of the Stack: ";
47. cout<<arr.back()<<endl;
48. break;
49. case 5:
50. for (it = arr.begin(); it != arr.end(); ++it )
51. cout <<" "<< \*it;
52. cout<<endl;
53. break;
54. case 6:
55. exit(1);
56. break;
57. default:
58. cout<<"Wrong Choice"<<endl;
59. }
60. }
61. return 0;
62. }

# 71. **C++ Program to Implement Deque in STL**

This C++ Program demonstrates implementation of Deque in STL.

Output:

$ **g++** deque.cpp

$ a.out

Deque Implementation **in** Stl

1.Insert Element at the End

2.Insert Element at the Front

3.Delete Element at the End

4.Delete Element at the Front

5.Front Element at Deque

6.Last Element at Deque

7.Size of the Deque

8.Display Deque

9.Exit

Enter your Choice: 1

Enter value to be inserted at the end: 9

Deque Implementation **in** Stl

1.Insert Element at the End

2.Insert Element at the Front

3.Delete Element at the End

4.Delete Element at the Front

5.Front Element at Deque

6.Last Element at Deque

7.Size of the Deque

8.Display Deque

9.Exit

Enter your Choice: 2

Enter value to be inserted at the front: 8

Deque Implementation **in** Stl

1.Insert Element at the End

2.Insert Element at the Front

3.Delete Element at the End

4.Delete Element at the Front

5.Front Element at Deque

6.Last Element at Deque

7.Size of the Deque

8.Display Deque

9.Exit

Enter your Choice: 1

Enter value to be inserted at the end: 7

Deque Implementation **in** Stl

1.Insert Element at the End

2.Insert Element at the Front

3.Delete Element at the End

4.Delete Element at the Front

5.Front Element at Deque

6.Last Element at Deque

7.Size of the Deque

8.Display Deque

9.Exit

Enter your Choice: 1

Enter value to be inserted at the end: 6

Deque Implementation **in** Stl

1.Insert Element at the End

2.Insert Element at the Front

3.Delete Element at the End

4.Delete Element at the Front

5.Front Element at Deque

6.Last Element at Deque

7.Size of the Deque

8.Display Deque

9.Exit

Enter your Choice: 1

Enter value to be inserted at the end: 5

Deque Implementation **in** Stl

1.Insert Element at the End

2.Insert Element at the Front

3.Delete Element at the End

4.Delete Element at the Front

5.Front Element at Deque

6.Last Element at Deque

7.Size of the Deque

8.Display Deque

9.Exit

Enter your Choice: 2

Enter value to be inserted at the front: 10

Deque Implementation **in** Stl

1.Insert Element at the End

2.Insert Element at the Front

3.Delete Element at the End

4.Delete Element at the Front

5.Front Element at Deque

6.Last Element at Deque

7.Size of the Deque

8.Display Deque

9.Exit

Enter your Choice: 2

Enter value to be inserted at the front: 7

Deque Implementation **in** Stl

1.Insert Element at the End

2.Insert Element at the Front

3.Delete Element at the End

4.Delete Element at the Front

5.Front Element at Deque

6.Last Element at Deque

7.Size of the Deque

8.Display Deque

9.Exit

Enter your Choice: 7

Size of the Deque: 7

Deque Implementation **in** Stl

1.Insert Element at the End

2.Insert Element at the Front

3.Delete Element at the End

4.Delete Element at the Front

5.Front Element at Deque

6.Last Element at Deque

7.Size of the Deque

8.Display Deque

9.Exit

Enter your Choice: 8

Elements of Deque: 7 10 8 9 7 6 5

Deque Implementation **in** Stl

1.Insert Element at the End

2.Insert Element at the Front

3.Delete Element at the End

4.Delete Element at the Front

5.Front Element at Deque

6.Last Element at Deque

7.Size of the Deque

8.Display Deque

9.Exit

Enter your Choice: 5

Front Element of the Deque: 7

Deque Implementation **in** Stl

1.Insert Element at the End

2.Insert Element at the Front

3.Delete Element at the End

4.Delete Element at the Front

5.Front Element at Deque

6.Last Element at Deque

7.Size of the Deque

8.Display Deque

9.Exit

Enter your Choice: 6

Back Element of the Deque: 5

Deque Implementation **in** Stl

1.Insert Element at the End

2.Insert Element at the Front

3.Delete Element at the End

4.Delete Element at the Front

5.Front Element at Deque

6.Last Element at Deque

7.Size of the Deque

8.Display Deque

9.Exit

Enter your Choice: 3

Element 5 deleted

Deque Implementation **in** Stl

1.Insert Element at the End

2.Insert Element at the Front

3.Delete Element at the End

4.Delete Element at the Front

5.Front Element at Deque

6.Last Element at Deque

7.Size of the Deque

8.Display Deque

9.Exit

Enter your Choice: 8

Elements of Deque: 7 10 8 9 7 6

Deque Implementation **in** Stl

1.Insert Element at the End

2.Insert Element at the Front

3.Delete Element at the End

4.Delete Element at the Front

5.Front Element at Deque

6.Last Element at Deque

7.Size of the Deque

8.Display Deque

9.Exit

Enter your Choice: 4

Element 7 deleted

Deque Implementation **in** Stl

1.Insert Element at the End

2.Insert Element at the Front

3.Delete Element at the End

4.Delete Element at the Front

5.Front Element at Deque

6.Last Element at Deque

7.Size of the Deque

8.Display Deque

9.Exit

Enter your Choice: 7

Size of the Deque: 5

Deque Implementation **in** Stl

1.Insert Element at the End

2.Insert Element at the Front

3.Delete Element at the End

4.Delete Element at the Front

5.Front Element at Deque

6.Last Element at Deque

7.Size of the Deque

8.Display Deque

9.Exit

Enter your Choice: 8

Elements of Deque: 10 8 9 7 6

Deque Implementation **in** Stl

1.Insert Element at the End

2.Insert Element at the Front

3.Delete Element at the End

4.Delete Element at the Front

5.Front Element at Deque

6.Last Element at Deque

7.Size of the Deque

8.Display Deque

9.Exit

Enter your Choice: 9

-----

**(**program exited with code: 1**)**

Press **return** to **continue**

1. */\**
2. *\* C++ Program to Implement Deque in Stl*
3. *\*/*
4. #include <iostream>
5. #include <deque>
6. #include <string>
7. #include <cstdlib>
8. using namespace std;
9. int main()
10. {
11. deque<int> dq;
12. deque<int>::iterator it;
13. int choice, item;
14. while (1)
15. {
16. cout<<"**\n**--------"<<endl;
17. cout<<"Deque Implementation in Stl"<<endl;
18. cout<<"**\n**--------"<<endl;
19. cout<<"1.Insert Element at the End"<<endl;
20. cout<<"2.Insert Element at the Front"<<endl;
21. cout<<"3.Delete Element at the End"<<endl;
22. cout<<"4.Delete Element at the Front"<<endl;
23. cout<<"5.Front Element at Deque"<<endl;
24. cout<<"6.Last Element at Deque"<<endl;
25. cout<<"7.Size of the Deque"<<endl;
26. cout<<"8.Display Deque"<<endl;
27. cout<<"9.Exit"<<endl;
28. cout<<"Enter your Choice: ";
29. cin>>choice;
30. switch(choice)
31. {
32. case 1:
33. cout<<"Enter value to be inserted at the end: ";
34. cin>>item;
35. dq.push\_back(item);
36. break;
37. case 2:
38. cout<<"Enter value to be inserted at the front: ";
39. cin>>item;
40. dq.push\_front(item);
41. break;
42. case 3:
43. item = dq.back();
44. dq.pop\_back();
45. cout<<"Element "<<item<<" deleted"<<endl;
46. break;
47. case 4:
48. item = dq.front();
49. dq.pop\_front();
50. cout<<"Element "<<item<<" deleted"<<endl;
51. break;
52. case 5:
53. cout<<"Front Element of the Deque: ";
54. cout<<dq.front()<<endl;
55. break;
56. case 6:
57. cout<<"Back Element of the Deque: ";
58. cout<<dq.back()<<endl;
59. break;
60. case 7:
61. cout<<"Size of the Deque: "<<dq.size()<<endl;
62. break;
63. case 8:
64. cout<<"Elements of Deque: ";
65. for (it = dq.begin(); it != dq.end(); it++)
66. cout<<\*it<<" ";
67. cout<<endl;
68. break;
69. case 9:
70. exit(1);
71. break;
72. default:
73. cout<<"Wrong Choice"<<endl;
74. }
75. }
76. return 0;
77. }

# 72. **C++ Program to Represent Linear Equations in Matrix Form**

This is a C++ Program to represent a set of linear equations in matrix form.

Output:

$ g++ MatrixRepOfEquations.cpp

$ a.out

Enter the number of variables in the equations: 3

Enter the coefficients of each variable for each equations

ax + by + cz + ... = d

1 2 3 4

1 2 6 8

2 3 9 8

Matrix representation is:

1 2 3 x = 4

1 2 6 y = 8

2 3 9 z = 8

1. #include<iostream>
2. #include<conio.h>
4. using namespace std;
6. int main(void)
7. {
8. char var[] = { 'x', 'y', 'z', 'w' };
9. cout << "Enter the number of variables in the equations: ";
11. int n;
12. cin >> n;
13. cout << "**\n**Enter the coefficients of each variable for each equations";
14. cout << "**\n**ax + by + cz + ... = d";
15. int mat[n][n];
16. int constants[n][1];
17. for (int i = 0; i < n; i++)
18. {
19. for (int j = 0; j < n; j++)
20. {
21. cin >> mat[i][j];
22. }
23. cin >> constants[i][0];
24. }
26. cout << "Matrix representation is: ";
27. for (int i = 0; i < n; i++)
28. {
29. for (int j = 0; j < n; j++)
30. {
31. cout << " " << mat[i][j];
32. }
33. cout << " " << var[i];
34. cout << " = " << constants[i][0];
35. cout << "**\n**";
36. }
37. return 0;
38. }

# 73. **C++ Program to Perform Encoding of a Message Using Matrix Multiplication**

$ g++ EncodingMatrix.cpp

$ a.out

Enter the number of rows and columns for Message Matrix:

2 2

Enter elements for Matrix :::

1 2

3 4

Matrix :

1 2

3 4

-------

Encoded Matrix :

12 12

28 28

1. #include<conio.h>
2. #include<iostream>
3. using namespace std;
4. int main()
5. {
6. int a[10][10], b[10][10], c[10][10];
7. int x, y, i, j;
9. cout << "**\n**Enter the number of rows and columns for Message Matrix:**\n\n**";
10. cin >> x >> y;
12. // x denotes number rows in matrix A
13. // y denotes number columns in matrix A
14. cout << "**\n\n**Enter elements for Matrix :::**\n\n**";
15. for (i = 0; i < x; i++)
16. {
17. for (j = 0; j < y; j++)
18. {
19. cin >> a[i][j];
20. }
21. cout << "**\n**";
22. }
23. cout << "**\n\n**Matrix :**\n\n**";
24. for (i = 0; i < x; i++)
25. {
26. for (j = 0; j < y; j++)
27. {
28. cout << "**\t**" << a[i][j];
29. }
30. cout << "**\n\n**";
31. }


35. for (i = 0; i < y; i++)
36. {
37. for (j = 0; j < x; j++)
38. {
39. b[i][j]=x+y;
40. }
41. cout << "**\n**";
42. }
44. for (i = 0; i < x; i++)
45. {
46. for (j = 0; j < x; j++)
47. {
48. c[i][j] = 0;
49. for (int k = 0; k < y; k++)
50. {
51. c[i][j] = c[i][j] + a[i][k] \* b[k][j];
52. }
53. }
54. }
55. cout
56. << "**\n**-------**\n**";
57. cout << "**\n\n**Encoded Matrix :**\n\n**";
58. for (i = 0; i < x; i++)
59. {
60. for (j = 0; j < x; j++)
61. {
62. cout << "**\t**" << c[i][j];
63. }
64. cout << "**\n\n**";
65. }
66. getch();
67. return 0;
68. }

# 74. **C++ Program to Find Closest Pair of Points in an Array**

This C++ Program Finds the Closest Pair of Points in an Array.

1. */\**
2. *\* Main*
3. *\*/*
4. int main()
5. {
6. Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};
7. int n = sizeof(P) / sizeof(P[0]);
8. cout << "The smallest distance is " << closest(P, n);
9. return 0;
10. }

$ **g++** closest\_pair.cpp

$ a.out

The smallest distance is 1.41421

**(**program exited with code: 1**)**

Press **return** to **continue**

1. */\**
2. *\* C++ Program to Find Closest Pair of Points in an Array*
3. *\*/*
4. #include <iostream>
5. #include <cfloat>
6. #include <cstdlib>
7. #include <cmath>
8. using namespace std;
10. */\**
11. *\* Point Declaration*
12. *\*/*
13. struct Point
14. {
15. int x, y;
16. };
18. */\**
19. *\* sort array of points according to X coordinate*
20. *\*/*
21. int compareX(const void\* a, const void\* b)
22. {
23. Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;
24. return (p1->x - p2->x);
25. }
26. */\**
27. *\* sort array of points according to Y coordinate*
28. *\*/*
29. int compareY(const void\* a, const void\* b)
30. {
31. Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;
32. return (p1->y - p2->y);
33. }
34. */\**
35. *\* find the distance between two points*
36. *\*/*
37. float dist(Point p1, Point p2)
38. {
39. return sqrt((p1.x - p2.x) \* (p1.x - p2.x) + (p1.y - p2.y) \* (p1.y - p2.y));
40. }
41. */\**
42. *\* return the smallest distance between two points*
43. *\*/*
44. float small\_dist(Point P[], int n)
45. {
46. float min = FLT\_MAX;
47. for (int i = 0; i < n; ++i)
48. {
49. for (int j = i + 1; j < n; ++j)
50. {
51. if (dist(P[i], P[j]) < min)
52. min = dist(P[i], P[j]);
53. }
54. }
55. return min;
56. }
57. */\**
58. *\* find the distance beween the closest points of strip of given size*
59. *\*/*
60. float stripClosest(Point strip[], int size, float d)
61. {
62. float min = d;
63. for (int i = 0; i < size; ++i)
64. {
65. for (int j = i + 1; j < size && (strip[j].y - strip[i].y) < min; ++j)
66. {
67. if (dist(strip[i],strip[j]) < min)
68. min = dist(strip[i], strip[j]);
69. }
70. }
71. return min;
72. }
73. */\**
74. *\* find the smallest distance.*
75. *\*/*
76. float closestUtil(Point Px[], Point Py[], int n)
77. {
78. if (n <= 3)
79. return small\_dist(Px, n);
80. int mid = n / 2;
81. Point midPoint = Px[mid];
82. Point Pyl[mid + 1];
83. Point Pyr[n - mid - 1];
84. int li = 0, ri = 0;
85. for (int i = 0; i < n; i++)
86. {
87. if (Py[i].x <= midPoint.x)
88. Pyl[li++] = Py[i];
89. else
90. Pyr[ri++] = Py[i];
91. }
92. float dl = closestUtil(Px, Pyl, mid);
93. float dr = closestUtil(Px + mid, Pyr, n-mid);
94. float d = min(dl, dr);
95. Point strip[n];
96. int j = 0;
97. for (int i = 0; i < n; i++)
98. {
99. if (abs(Py[i].x - midPoint.x) < d)
100. strip[j] = Py[i], j++;
101. }
102. return min(d, stripClosest(strip, j, d));
103. }
104. */\**
105. *\* finds the smallest distance*
106. *\*/*
107. float closest(Point P[], int n)
108. {
109. Point Px[n];
110. Point Py[n];
111. for (int i = 0; i < n; i++)
112. {
113. Px[i] = P[i];
114. Py[i] = P[i];
115. }
116. qsort(Px, n, sizeof(Point), compareX);
117. qsort(Py, n, sizeof(Point), compareY);
118. return closestUtil(Px, Py, n);
119. }

# 75. **C++ Program to Optimize Wire Length in Electrical Circuit**

This is a C++ Program to optimize wire length in electix circuit. This problem can be reduced to finding the shortest path between two components.

Output:

$ g++ OptimizeWireLength.cpp

$ a.out

Enter the starting component: 1

Component Distance from other component

0 4

1 0

2 8

3 15

4 22

5 12

6 12

7 11

8 10

Enter the starting component: 6

Component Distance from other component

0 9

1 12

2 6

3 13

4 12

5 2

6 0

7 1

8 6

1. #include <stdio.h>
2. #include <limits.h>
3. #include <iostream>
5. using namespace std;
7. // Number of components in the graph
8. #define V 9
10. // A utility function to find the component with minimum distance value, from
11. // the set of components not yet included in shortest path tree
12. int minDistance(int dist[], bool sptSet[])
13. {
14. // Initialize min value
15. int min = INT\_MAX, min\_index;
17. for (int v = 0; v < V; v++)
18. if (sptSet[v] == false && dist[v] <= min)
19. min = dist[v], min\_index = v;
21. return min\_index;
22. }
24. // A utility function to print the constructed distance array
25. void printSolution(int dist[], int n)
26. {
27. cout << "Component**\t**Distance from other component**\n**";
28. for (int i = 0; i < V; i++)
29. printf("%d**\t\t**%d**\n**", i, dist[i]);
30. }
32. // Funtion that implements Dijkstra's single source shortest path algorithm
33. // for a graph represented using adjacency matrix representation
34. void optimizeLength(int graph[V][V], int src)
35. {
36. int dist[V]; // The output array. dist[i] will hold the shortest
37. // distance from src to i
39. bool sptSet[V]; // sptSet[i] will true if component i is included in shortest
40. // path tree or shortest distance from src to i is finalized
42. // Initialize all distances as INFINITE and stpSet[] as false
43. for (int i = 0; i < V; i++)
44. dist[i] = INT\_MAX, sptSet[i] = false;
46. // Distance of source component from itself is always 0
47. dist[src] = 0;
49. // Find shortest path for all components
50. for (int count = 0; count < V - 1; count++)
51. {
52. // Pick the minimum distance component from the set of components not
53. // yet processed. u is always equal to src in first iteration.
54. int u = minDistance(dist, sptSet);
56. // Mark the picked component as processed
57. sptSet[u] = true;
59. // Update dist value of the adjacent components of the picked component.
60. for (int v = 0; v < V; v++)
62. // Update dist[v] only if is not in sptSet, there is an edge from
63. // u to v, and total weight of path from src to v through u is
64. // smaller than current value of dist[v]
65. if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u]
66. + graph[u][v] < dist[v])
67. dist[v] = dist[u] + graph[u][v];
68. }
70. // print the constructed distance array
71. printSolution(dist, V);
72. }
74. // driver program to test above function
75. int main()
76. {
77. */\* Let us create the example graph discussed above \*/*
78. int graph[V][V] =
79. { { 0, 4, 0, 0, 0, 0, 0, 8, 0 }, { 4, 0, 8, 0, 0, 0, 0, 11, 0 }, {
80. 0, 8, 0, 7, 0, 4, 0, 0, 2 },
81. { 0, 0, 7, 0, 9, 14, 0, 0, 0 }, { 0, 0, 0, 9, 0, 10, 0, 0,
82. 0 }, { 0, 0, 4, 0, 10, 0, 2, 0, 0 }, { 0, 0, 0, 14,
83. 0, 2, 0, 1, 6 }, { 8, 11, 0, 0, 0, 0, 1, 0, 7 }, {
84. 0, 0, 2, 0, 0, 0, 6, 7, 0 } };
86. cout << "Enter the starting component: ";
87. int s;
88. cin >> s;
89. optimizeLength(graph, s);
91. return 0;
92. }

# 76. **C++ Program to Perform Optimal Parenthesization Using Dynamic Programming**

This is a C++ Program to perform optimal paranthesization using DP.

Output:

$ g++ OptimalParanthesizationDP.cpp

$ a.out

Enter the array p[], which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i]Enter the total length:4

Enter the dimensions: 2 4 3 5

Minimum number of multiplications is: 54

1. #include<stdio.h>
2. #include<limits.h>
3. #include<iostream>
5. using namespace std;
7. // Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
9. int MatrixChainOrder(int p[], int n)
10. {
11. */\* For simplicity of the program, one extra row and one extra column are*
12. *allocated in m[][]. 0th row and 0th column of m[][] are not used \*/*
13. int m[n][n];
14. int s[n][n];
15. int i, j, k, L, q;
17. */\* m[i,j] = Minimum number of scalar multiplications needed to compute*
18. *the matrix A[i]A[i+1]...A[j] = A[i..j] where dimention of A[i] is*
19. *p[i-1] x p[i] \*/*
21. // cost is zero when multiplying one matrix.
22. for (i = 1; i < n; i++)
23. m[i][i] = 0;
25. // L is chain length.
26. for (L = 2; L < n; L++)
27. {
28. for (i = 1; i <= n - L + 1; i++)
29. {
30. j = i + L - 1;
31. m[i][j] = INT\_MAX;
32. for (k = i; k <= j - 1; k++)
33. {
34. // q = cost/scalar multiplications
35. q = m[i][k] + m[k + 1][j] + p[i - 1] \* p[k] \* p[j];
36. if (q < m[i][j])
37. {
38. m[i][j] = q;
39. s[i][j] = k;
40. }
41. }
42. }
43. }
45. return m[1][n - 1];
46. }
47. int main()
48. {
49. cout
50. << "Enter the array p[], which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i]";
51. cout << "Enter the total length:";
52. int n;
53. cin >> n;
54. int array[n];
55. cout << "Enter the dimensions: ";
56. for (int var = 0; var < n; ++var)
57. {
58. cin >> array[var];
59. }
60. cout << "Minimum number of multiplications is: " << MatrixChainOrder(array,
61. n);
62. return 0;
63. }

# 77. **C++ Program to Implement Strassen’s Algorithm**

This is a C++ Program to implement Strassen’s algorithm for matrix multiplication. In the mathematical discipline of linear algebra, the Strassen algorithm, named after Volker Strassen, is an algorithm used for matrix multiplication. It is faster than the standard matrix multiplication algorithm and is useful in practice for large matrices, but would be slower than the fastest known algorithms for extremely large matrices.

Output:

$ g++ StrassenMulitplication.cpp

$ a.out

A = {

1.2, 0.83, 0.39, 0.41,

1.8, 1.9, 0.49, 0.23,

0.38, 0.72, 1.8, 1.9,

0.13, 1.8, 0.48, 0.82,

}

B = {

1.2, 1.6, 1.4, 1.6,

0.27, 0.63, 0.3, 0.79,

0.58, 1.2, 1.1, 0.07,

2, 1.9, 0.47, 0.47,

}

C = {

2.7, 3.7, 2.6, 2.9,

3.4, 5, 3.7, 4.5,

5.3, 6.7, 3.6, 2.2,

2.5, 3.5, 1.6, 2.1,

}

1. #include <assert.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <time.h>
6. #define M 2
7. #define N (1<<M)
9. typedef double datatype;
10. #define DATATYPE\_FORMAT "%4.2g"
11. typedef datatype mat[N][N]; // mat[2\*\*M,2\*\*M] for divide and conquer mult.
12. typedef struct
13. {
14. int ra, rb, ca, cb;
15. } corners; // for tracking rows and columns.
16. // A[ra..rb][ca..cb] .. the 4 corners of a matrix.
18. // set A[a] = I
19. void identity(mat A, corners a)
20. {
21. int i, j;
22. for (i = a.ra; i < a.rb; i++)
23. for (j = a.ca; j < a.cb; j++)
24. A[i][j] = (datatype) (i == j);
25. }
27. // set A[a] = k
28. void set(mat A, corners a, datatype k)
29. {
30. int i, j;
31. for (i = a.ra; i < a.rb; i++)
32. for (j = a.ca; j < a.cb; j++)
33. A[i][j] = k;
34. }
36. // set A[a] = [random(l..h)].
37. void randk(mat A, corners a, double l, double h)
38. {
39. int i, j;
40. for (i = a.ra; i < a.rb; i++)
41. for (j = a.ca; j < a.cb; j++)
42. A[i][j] = (datatype) (l + (h - l) \* (rand() / (double) RAND\_MAX));
43. }
45. // Print A[a]
46. void print(mat A, corners a, char \*name)
47. {
48. int i, j;
49. printf("%s = {**\n**", name);
50. for (i = a.ra; i < a.rb; i++)
51. {
52. for (j = a.ca; j < a.cb; j++)
53. printf(DATATYPE\_FORMAT ", ", A[i][j]);
54. printf("**\n**");
55. }
56. printf("}**\n**");
57. }
59. // C[c] = A[a] + B[b]
60. void add(mat A, mat B, mat C, corners a, corners b, corners c)
61. {
62. int rd = a.rb - a.ra;
63. int cd = a.cb - a.ca;
64. int i, j;
65. for (i = 0; i < rd; i++)
66. {
67. for (j = 0; j < cd; j++)
68. {
69. C[i + c.ra][j + c.ca] = A[i + a.ra][j + a.ca] + B[i + b.ra][j
70. + b.ca];
71. }
72. }
73. }
75. // C[c] = A[a] - B[b]
76. void sub(mat A, mat B, mat C, corners a, corners b, corners c)
77. {
78. int rd = a.rb - a.ra;
79. int cd = a.cb - a.ca;
80. int i, j;
81. for (i = 0; i < rd; i++)
82. {
83. for (j = 0; j < cd; j++)
84. {
85. C[i + c.ra][j + c.ca] = A[i + a.ra][j + a.ca] - B[i + b.ra][j
86. + b.ca];
87. }
88. }
89. }
91. // Return 1/4 of the matrix: top/bottom , left/right.
92. void find\_corner(corners a, int i, int j, corners \*b)
93. {
94. int rm = a.ra + (a.rb - a.ra) / 2;
95. int cm = a.ca + (a.cb - a.ca) / 2;
96. \*b = a;
97. if (i == 0)
98. b->rb = rm; // top rows
99. else
100. b->ra = rm; // bot rows
101. if (j == 0)
102. b->cb = cm; // left cols
103. else
104. b->ca = cm; // right cols
105. }
107. // Multiply: A[a] \* B[b] => C[c], recursively.
108. void mul(mat A, mat B, mat C, corners a, corners b, corners c)
109. {
110. corners aii[2][2], bii[2][2], cii[2][2], p;
111. mat P[7], S, T;
112. int i, j, m, n, k;
114. // Check: A[m n] \* B[n k] = C[m k]
115. m = a.rb - a.ra;
116. assert(m==(c.rb-c.ra));
117. n = a.cb - a.ca;
118. assert(n==(b.rb-b.ra));
119. k = b.cb - b.ca;
120. assert(k==(c.cb-c.ca));
121. assert(m>0);
123. if (n == 1)
124. {
125. C[c.ra][c.ca] += A[a.ra][a.ca] \* B[b.ra][b.ca];
126. return;
127. }
129. // Create the 12 smaller matrix indexes:
130. // A00 A01 B00 B01 C00 C01
131. // A10 A11 B10 B11 C10 C11
132. for (i = 0; i < 2; i++)
133. {
134. for (j = 0; j < 2; j++)
135. {
136. find\_corner(a, i, j, &aii[i][j]);
137. find\_corner(b, i, j, &bii[i][j]);
138. find\_corner(c, i, j, &cii[i][j]);
139. }
140. }
142. p.ra = p.ca = 0;
143. p.rb = p.cb = m / 2;
145. #define LEN(A) (sizeof(A)/sizeof(A[0]))
146. for (i = 0; i < LEN(P); i++)
147. set(P[i], p, 0);
149. #define ST0 set(S,p,0); set(T,p,0)
151. // (A00 + A11) \* (B00+B11) = S \* T = P0
152. ST0;
153. add(A, A, S, aii[0][0], aii[1][1], p);
154. add(B, B, T, bii[0][0], bii[1][1], p);
155. mul(S, T, P[0], p, p, p);
157. // (A10 + A11) \* B00 = S \* B00 = P1
158. ST0;
159. add(A, A, S, aii[1][0], aii[1][1], p);
160. mul(S, B, P[1], p, bii[0][0], p);
162. // A00 \* (B01 - B11) = A00 \* T = P2
163. ST0;
164. sub(B, B, T, bii[0][1], bii[1][1], p);
165. mul(A, T, P[2], aii[0][0], p, p);
167. // A11 \* (B10 - B00) = A11 \* T = P3
168. ST0;
169. sub(B, B, T, bii[1][0], bii[0][0], p);
170. mul(A, T, P[3], aii[1][1], p, p);
172. // (A00 + A01) \* B11 = S \* B11 = P4
173. ST0;
174. add(A, A, S, aii[0][0], aii[0][1], p);
175. mul(S, B, P[4], p, bii[1][1], p);
177. // (A10 - A00) \* (B00 + B01) = S \* T = P5
178. ST0;
179. sub(A, A, S, aii[1][0], aii[0][0], p);
180. add(B, B, T, bii[0][0], bii[0][1], p);
181. mul(S, T, P[5], p, p, p);
183. // (A01 - A11) \* (B10 + B11) = S \* T = P6
184. ST0;
185. sub(A, A, S, aii[0][1], aii[1][1], p);
186. add(B, B, T, bii[1][0], bii[1][1], p);
187. mul(S, T, P[6], p, p, p);
189. // P0 + P3 - P4 + P6 = S - P4 + P6 = T + P6 = C00
190. add(P[0], P[3], S, p, p, p);
191. sub(S, P[4], T, p, p, p);
192. add(T, P[6], C, p, p, cii[0][0]);
194. // P2 + P4 = C01
195. add(P[2], P[4], C, p, p, cii[0][1]);
197. // P1 + P3 = C10
198. add(P[1], P[3], C, p, p, cii[1][0]);
200. // P0 + P2 - P1 + P5 = S - P1 + P5 = T + P5 = C11
201. add(P[0], P[2], S, p, p, p);
202. sub(S, P[1], T, p, p, p);
203. add(T, P[5], C, p, p, cii[1][1]);
205. }
206. int main()
207. {
208. mat A, B, C;
209. corners ai = { 0, N, 0, N };
210. corners bi = { 0, N, 0, N };
211. corners ci = { 0, N, 0, N };
212. srand(time(0));
213. // identity(A,bi); identity(B,bi);
214. // set(A,ai,2); set(B,bi,2);
215. randk(A, ai, 0, 2);
216. randk(B, bi, 0, 2);
217. print(A, ai, "A");
218. print(B, bi, "B");
219. set(C, ci, 0);
220. // add(A,B,C, ai, bi, ci);
221. mul(A, B, C, ai, bi, ci);
222. print(C, ci, "C");
223. return 0;
224. }

# 78. **C++ Program to Implement Fisher-Yates Algorithm for Array Shuffling**

This is a C++ Program to shuffle array using Fisher-Yates algorithm. The Fisher–Yates shuffle (named after Ronald Fisher and Frank Yates), also known as the Knuth shuffle (after Donald Knuth), is an algorithm for generating a random permutation of a finite set—in plain terms, for randomly shuffling the set. A variant of the Fisher–Yates shuffle, known as Sattolo’s algorithm, may be used to generate random cycles of length n instead. The Fisher–Yates shuffle is unbiased, so that every permutation is equally likely. The modern version of the algorithm is also rather efficient, requiring only time proportional to the number of items being shuffled and no additional storage space.

Output:

$ g++ Fisher-YatesShuffling.cpp

$ a.out

Enter the array size: 7

Enter the array elements: 12 23 34 45 56 67 78

78 23 67 45 34 12 56

#include <iostream>

#include <stdlib.h>

using namespace std;

void fisherYatesShuffling(int \*arr, int n)

{

int a[n];

int ind[n];

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

ind[i] = 0;

int index;

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

{

do

{

index = rand() % n;

}

while (ind[index] != 0);

ind[index] = 1;

a[i] = \*(arr + index);

}

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

{

cout << a[i] << " ";

}

}

int main(int argc, char \*\*argv)

{

cout << "Enter the array size: ";

int n;

cin >> n;

cout << "Enter the array elements: ";

int a[n];

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

{

cin >> a[i];

}

fisherYatesShuffling(a, n);

}

# 79. **C++ Program to Implement Park-Miller Random Number Generation Algorithm**

This is a C++ Program to generate random numbers using Park-Miller algorithm. A general formula of a random number generator (RNG) of this type is:

X\_{k+1} = g X(k) mod n

where the modulus n is a prime number or a power of a prime number, the multiplier g is an element of high multiplicative order modulo n (e.g., a primitive root modulo n), and the seed X0 is co-prime to n.

Output:

$ g++ ParkMillerRandomNumbers.cpp

$ a.out

Random numbers are:

1.08525e+009

5.0826e+008

1.35229e+009

1.56324e+009

8.90733e+008

1.81003e+009

1.50959e+009

8.62973e+008

1.85299e+009

6.77684e+008

#include <iostream>

#include <math.h>

#include <stdlib.h>

using namespace std;

const long m = 2147483647L;

const long a = 48271L;

const long q = 44488L;

const long r = 3399L;

static long r\_seed = 12345678L;

double uniform()

{

long hi = r\_seed / q;

long lo = r\_seed - q \* hi;

long t = a \* lo - r \* hi;

if (t > 0)

r\_seed = t;

else

r\_seed = t + m;

return r\_seed;

}

int main(int argc, char \*\*argv)

{

double A[10];

for (int i = 0; i < 10; i++)

A[i] = uniform();

cout<<"Random numbers are:**\n**";

for (int i = 0; i < 10; i++)

cout << A[i]<<endl;

}

# 80. **C++ Program to Implement Naor-Reingold Pseudo Random Function**

Output:

This is a C++ Program to genrate random numbers using Naor-Reingold random function. Moni Naor and Omer Reingold described efficient constructions for various cryptographic primitives in private key as well as public-key cryptography. Their result is the construction of an efficient pseudorandom function. Let p and l be prime numbers with l |p-1. Select an element g ? {\mathbb F\_p}^\* of multiplicative order l. Then for each n-dimensional vector a = (a1, …, an)? (\mathbb F\_{l})^{n} they define the function

f\_{a}(x) = g^{a\_{1}^{x\_{1}} a\_{2}^{x\_{2}}…a\_{n}^{x\_{n}}} \in \mathbb F\_p

where x = x1 … xn is the bit representation of integer x, 0 = x = 2^n-1, with some extra leading zeros if necessary.

$ g++ Naor-Reingold.cpp

$ a.out

The Random numbers are:

2 4 16 4 2 4 16 16 4 2

#include <iostream>

#include <math.h>

#include <stdlib.h>

using namespace std;

int main(int argc, char \*\*argv)

{

int p = 7, l = 3, g = 2, n = 4, x;

int a[] = { 1, 2, 2, 1 };

int bin[4];

cout << "The Random numbers are: ";

for (int i = 0; i < 10; i++)

{

x = rand() % 16;

for (int j = 3; j >= 0; j--)

{

bin[j] = x % 2;

x /= 2;

}

int mul = 1;

for (int k = 0; k < 4; k++)

mul \*= pow(a[k], bin[k]);

cout << pow(g, mul)<<" ";

}

}

# 81. **C++ Program to Implement wheel Sieve to Generate Prime Numbers Between Given Range**

Output:

$ g++ WheelSeive.cpp

$ a.out

1st prime is: 2

2nd prime is: 3

3rd prime is: 5

4th prime is: 7

5th prime is: 11

6th prime is: 13

7th prime is: 17

8th prime is: 19

9th prime is: 23

10th prime is: 29

11th prime is: 31

12th prime is: 37

13th prime is: 41

14th prime is: 43

15th prime is: 47

#include <iostream>

#include <math.h>

#include <stdlib.h>

using namespace std;

#define MAX\_NUM 50

// array will be initialized to 0 being global

int primes[MAX\_NUM];

void gen\_sieve\_primes(void)

{

for (int p = 2; p < MAX\_NUM; p++) // for all elements in array

{

if (primes[p] == 0) // it is not multiple of any other prime

primes[p] = 1; // mark it as prime

// mark all multiples of prime selected above as non primes

int c = 2;

int mul = p \* c;

for (; mul < MAX\_NUM;)

{

primes[mul] = -1;

c++;

mul = p \* c;

}

}

}

void print\_all\_primes()

{

int c = 0;

for (int i = 0; i < MAX\_NUM; i++)

{

if (primes[i] == 1)

{

c++;

if (c < 4)

{

switch (c)

{

case 1:

cout << c << "st prime is: " << i << endl;

break;

case 2:

cout << c << "nd prime is: " << i << endl;

break;

case 3:

cout << c << "rd prime is: " << i << endl;

break;

default:

break;

}

}

else

cout << c << "th prime is: " << i << endl;

}

}

}

int main()

{

gen\_sieve\_primes();

print\_all\_primes();

return 0;

}

# 82. **C++ Program to Implement Sieve of Eratosthenes**

This C++ program to implement Sieve of Eratosthenes. The program initializes an integer array with all the elements initialized to 0. Then the algorithm follows where the each non-prime element’s index is marked as 1 inside the nested loops. The prime numbers are those whose value of index is marked as 0.

$ a.out

1 2 3 5 7 11 13 17 19 23

29 31 37 41 43 47 53 59 61 67

71 73 79 83 89 97

*/\**

*\* C++ Program to implement Sieve of Eratosthenes*

*\*/*

#include <iostream>

const int len = 100;

int main()

{

int arr[100] = {0};

for (int i = 2; i < 100; i++)

{

for (int j = i \* i; j < 100; j+=i)

{

arr[j - 1] = 1;

}

}

for (int i = 1; i < 100; i++)

{

if (arr[i - 1] == 0)

std::cout << i << "**\t**";

}

}

# 83.**C++ Program to Implement Sieve of Atkins**

This C++ Program demonstrates the implementation of Sieve of Atkins.

$ **g++** sieve\_atkins.cpp

$ a.out

Following are the prime numbers below 300

2 3 5 7 11 13 17 19 23 29

31 37 41 43 47 53 59 61 67 71

73 79 83 89 97 101 103 107 109 113

127 131 137 139 149 151 157 163 167 173

179 181 191 193 197 199 211 223 227 229

233 239 241 251 257 263 269 271 277 281

283 293

*\**

*\* C++ Program to Implement Sieve of Atkins*

*\*/*

#include <iostream>

#include <cmath>

#include <vector>

#define ll long long

using namespace std;

*/\**

*\* Sieve of Atkins*

*\*/*

void sieve\_atkins(ll int n)

{

vector<bool> is\_prime(n + 1);

is\_prime[2] = true;

is\_prime[3] = true;

for (ll int i = 5; i <= n; i++)

{

is\_prime[i] = false;

}

ll int lim = ceil(sqrt(n));

for (ll int x = 1; x <= lim; x++)

{

for (ll int y = 1; y <= lim; y++)

{

ll int num = (4 \* x \* x + y \* y);

if (num <= n && (num % 12 == 1 || num % 12 == 5))

{

is\_prime[num] = true;

}

num = (3 \* x \* x + y \* y);

if (num <= n && (num % 12 == 7))

{

is\_prime[num] = true;

}

if (x > y)

{

num = (3 \* x \* x - y \* y);

if (num <= n && (num % 12 == 11))

{

is\_prime[num] = true;

}

}

}

}

for (ll int i = 5; i <= lim; i++)

{

if (is\_prime[i])

{

for (ll int j = i \* i; j <= n; j += i)

{

is\_prime[j] = false;

}

}

}

for (ll int i = 2; i <= n; i++)

{

if (is\_prime[i])

{

cout<<i<<"**\t**";

}

}

}

*/\**

*\* Main*

*\*/*

int main()

{

ll int n;

n = 300;

cout<<"Following are the prime numbers below "<<n<<endl;

sieve\_atkins(n);

cout<<endl;

return 0;

}

# 84. **C++ Program to Implement Segmented Sieve**

This C++ Program demonstrates the implementation of Segmented Sieve.

$ **g++** sieve\_segemented.cpp

$ a.out

Enter Lower Bound: 7

Enter Upper Bound: 600

Number of primes between 7 and 600: 106

-----

**(**program exited with code: 1**)**

Press **return** to **continue**

*/\**

*\* C++ Program to Implement Segmented Sieve*

*\*/*

#include <iostream>

#include <cstring>

#define MAX 46656

#define LMT 216

#define LEN 4830

#define RNG 100032

#define sq(x) ((x)\*(x))

#define mset(x,v) memset(x, v , sizeof(x))

#define chkC(x,n) (x[n >> 6] & (1 << ((n >> 1) & 31)))

#define setC(x,n) (x[n >> 6] |= (1 << ((n >> 1) & 31)))

using namespace std;

unsigned base[MAX/64], segment[RNG/64], primes[LEN];

*/\**

*\* Generates all the necessary prime numbers and marks them in base[]*

*\*/*

void sieve()

{

unsigned i, j, k;

for (i = 3; i < LMT; i += 2)

{

if (!chkC(base, i))

{

for (j = i \* i, k = i << 1; j < MAX; j += k)

setC(base, j);

}

}

for (i = 3, j = 0; i < MAX; i += 2)

{

if (!chkC(base, i))

primes[j++] = i;

}

}

*/\**

*\* Returns the prime-count within range [a,b] and marks them in segment[]*

*\*/*

int segmented\_sieve(int a, int b)

{

unsigned i, j, k, cnt = (a <= 2 && 2 <=b )? 1 : 0;

if (b < 2)

return 0;

if (a < 3)

a = 3;

if (a % 2 == 0)

a++;

mset (segment, 0);

for (i = 0; sq(primes[i]) <= b; i++)

{

j = primes[i] \* ((a + primes[i] - 1) / primes[i]);

if (j % 2 == 0) j += primes[i];

for (k = primes[i] << 1; j <= b; j += k)

{

if (j != primes[i])

setC(segment, (j - a));

}

}

for (i = 0; i <= b - a; i += 2)

{

if (!chkC(segment, i))

cnt++;

}

return cnt;

}

*/\**

*\* Main*

*\*/*

int main()

{

sieve();

int a, b;

cout<<"Enter Lower Bound: ";

cin>>a;

cout<<"Enter Upper Bound: ";

cin>>b;

cout<<"Number of primes between "<<a<<" and "<<b<<": ";

cout<<segmented\_sieve(a, b)<<endl;

return 0;

}

# 85.**C++ Program to Implement Solovay-Strassen Primality Test**

This C++ Program demonstrates the implementation of Solovay-Strassen Primality Test.

Output:

$ **g++** solovay\_strassen.cpp

$ a.out

Enter integr to **test** primality: 219891801103773

219891801103773 is not prime

**(**program exited with code: 1**)**

Press **return** to **continue**

*/\**

*\* C++ Program to Implement Solovay-Strassen Primality Test*

*\*/*

#include <cstring>

#include <iostream>

#include <cstdlib>

#define ll long long

using namespace std;

*/\**

*\* modular exponentiation*

*\*/*

ll modulo(ll base, ll exponent, ll mod)

{

ll x = 1;

ll y = base;

while (exponent > 0)

{

if (exponent % 2 == 1)

x = (x \* y) % mod;

y = (y \* y) % mod;

exponent = exponent / 2;

}

return x % mod;

}

*/\**

*\* calculates Jacobian(a/n) n>0 and n is odd*

*\*/*

int calculateJacobian(ll a,ll n)

{

if (!a)

return 0;

int ans = 1;

ll temp;

if (a < 0)

{

a = -a;

if (n % 4 == 3)

ans=-ans;

}

if (a == 1)

return ans;

while (a)

{

if (a < 0)

{

a = -a;

if (n % 4 == 3)

ans = -ans;

}

while (a % 2 == 0)

{

a = a / 2;

if (n % 8 == 3 || n % 8 == 5)

ans = -ans;

}

swap(a, n);

if (a % 4 == 3 && n % 4 == 3)

ans = -ans;

a = a % n;

if (a > n / 2)

a = a - n;

}

if (n == 1)

return ans;

return 0;

}

*/\**

*\* Solovay-Strassen Primality Test*

*\* Iterations determine the accuracy of the test*

*\*/*

bool Solovoy(ll p, int iteration)

{

if (p < 2)

return false;

if (p != 2 && p % 2 == 0)

return false;

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

{

ll a = rand() % (p - 1) + 1;

ll jacobian = (p + calculateJacobian(a, p)) % p;

ll mod = modulo(a, (p - 1) / 2, p);

if (!jacobian || mod != jacobian)

{

return false;

}

}

return true;

}

//Main

int main()

{

int iteration = 50;

ll num;

cout<<"Enter integr to test primality: ";

cin>>num;

if (Solovoy(num, iteration))

cout<<num<<" is prime"<<endl;

else

cout<<num<<" is not prime"<<endl;

return 0;

}

# 86. **C++ Program to Find the GCD and LCM of n Numbers**

This is a C++ Program to find GCD and LCM of given two numbers.

Output:

$ g++ GCDLCM.cpp

$ a.out

Enter the two numbers:

5

8

The GCD of two numbers is: 1

The LCM of two numbers is: 40

Enter the two numbers:

100

50

The GCD of two numbers is: 50

The LCM of two numbers is: 100

#include<iostream>

#include<conio.h>

#include<stdlib.h>

using namespace std;

int gcd(int x, int y)

{

int r = 0, a, b;

a = (x > y) ? x : y; // a is greater number

b = (x < y) ? x : y; // b is smaller number

r = b;

while (a % b != 0)

{

r = a % b;

a = b;

b = r;

}

return r;

}

int lcm(int x, int y)

{

int a;

a = (x > y) ? x : y; // a is greater number

while (true)

{

if (a % x == 0 && a % y == 0)

return a;

++a;

}

}

int main(int argc, char \*\*argv)

{

cout << "Enter the two numbers: ";

int x, y;

cin >> x >> y;

cout << "The GCD of two numbers is: " << gcd(x, y) << endl;

;

cout << "The LCM of two numbers is: " << lcm(x, y) << endl;

;

return 0;

}

# 87. **C++ Program to Generate All Unique Partitions of an Integer**

This is a C++ Program to find the unique partitions of a given integer such that the addition of a partition result an integer. Given a positive integer n, generate all possible unique ways to represent n as a sum of positive integers.

$ **g++** unique\_partitions.cpp

$ a.out

Enter an Integer**(**0 to **exit)**: 2

All Unique Partitions of 2

2

1 1

Enter an Integer**(**0 to **exit)**: 3

All Unique Partitions of 3

3

2 1

1 1 1

Enter an Integer**(**0 to **exit)**: 4

All Unique Partitions of 4

4

3 1

2 2

2 1 1

1 1 1 1

Enter an Integer**(**0 to **exit)**: 5

All Unique Partitions of 5

5

4 1

3 2

3 1 1

2 2 1

2 1 1 1

1 1 1 1 1

Enter an Integer**(**0 to **exit)**: 7

All Unique Partitions of 7

7

6 1

5 2

5 1 1

4 3

4 2 1

4 1 1 1

3 3 1

3 2 2

3 2 1 1

3 1 1 1 1

2 2 2 1

2 2 1 1 1

2 1 1 1 1 1

1 1 1 1 1 1 1

Enter an Integer**(**0 to **exit)**: 0

***/\****

***\* C++ Program to Generate All Unique Partitions of an Integer***

***\*/***

**#include<iostream>**

**using namespace std;**

***/\****

***\* print an array p[] of size 'n'***

***\*/***

**void printArray(int p[], int n)**

**{**

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

**cout << p[i] << " ";**

**cout << endl;**

**}**

**void printAllUniqueParts(int n)**

**{**

**int p[n]; // An array to store a partition**

**int k = 0; // Index of last element in a partition**

**p[k] = n; // Initialize first partition as number itself**

**// This loop first prints the current partition then generates the next partition.**

**// The loop stops when the current partition has all 1s**

**while (true)**

**{**

**// print current partition**

**printArray(p, k + 1);**

**// Find the rightmost non-one value in p[]. Also, update the rem\_val**

**// So that we know how much value can be accommodated**

**int rem\_val = 0;**

**while (k >= 0 && p[k] == 1)**

**{**

**rem\_val += p[k];**

**k--;**

**}**

**// if k < 0, all the values are 1 so there are no more partitions**

**if (k < 0)**

**return;**

**// Decrease the p[k] found above and adjust the rem\_val**

**p[k]--;**

**rem\_val++;**

**// If rem\_val is more, then the sorted order is violeted.**

**// Divide rem\_val in differnt values of size p[k]**

**// Copy these values at different positions after p[k]**

**while (rem\_val > p[k])**

**{**

**p[k+1] = p[k];**

**rem\_val = rem\_val - p[k];**

**k++;**

**}**

**// Copy rem\_val to next position and increment position**

**p[k+1] = rem\_val;**

**k++;**

**}**

**}**

***/\****

***\* Main***

***\*/***

**int main()**

**{**

**int value;**

**while(1)**

**{**

**cout<<"Enter an Integer(0 to exit): ";**

**cin>>value;**

**if (value == 0)**

**break;**

**cout << "All Unique Partitions of "<<value<<endl;**

**printAllUniqueParts(value);**

**cout<<endl;**

**}**

**return 0;**

**}**

# 88. **C++ Program to Generate Prime Numbers Between a Given Range Using the Sieve of Sundaram**

Output:

$ g++ SieveOfSundaram.cpp

$ a.out

Welcome to the Sieve of Sundaram

Input a positive integer to find all the prime numbers up to and

including that number: 10

2 3 5 7

Number of Primes: 4

Welcome to the Sieve of Sundaram

Input a positive integer to find all the prime numbers up to and

including that number: 100

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Number of Primes: 25

#include <iostream>

using namespace std;

int main()

{

cout << "Welcome to the Sieve of Sundaram**\n**" << endl;

int arraySize;

int numberPrimes = 0;

cout << "Input a positive integer to find all the prime numbers up to and "

<< "**\n**including that number: ";

cin >> arraySize;

int n = arraySize / 2;

*/\* array to start off with that will eventually get*

*all the composite numbers removed and the remaining*

*ones output to the screen \*/*

int isPrime[arraySize + 1];

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

{

isPrime[i] = i;

}

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

{

for (int j = i; j <= (n - i) / (2 \* i + 1); j++)

{

isPrime[i + j + 2 \* i \* j] = 0;*/\*From this list, remove all*

*numbers of the form i + j + 2ij \*/*

}

}

int TheseArePrime = 0;

if (arraySize > 2)

{

isPrime[TheseArePrime++] = 2;*/\*this IF statement adds 2 to the output \*/*

}

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

{

if (isPrime[i] != 0)

{

isPrime[TheseArePrime++] = i \* 2 + 1;

}

}

int size = sizeof isPrime / sizeof(int);//total size of array/size of array data type

for (int x = 0; x <= size; x++)

{

if (isPrime[x] != 0)

{

cout << isPrime[x] << "**\t**";//outputs all prime numbers found

numberPrimes++;// the counter of the number of primes found

}

else

{

break;

}

}

cout << "**\n**Number of Primes: " << numberPrimes << endl;

return 0;

}

# 89. **C++ Program to Implement Fermat Primality Test**

Output:

$ **g++** fermat\_primality.cpp

$ a.out

Enter integer to **test** primality: 479001599

479001599 is prime

*/\**

*\* C++ Program to Implement Fermat Primality Test*

*\*/*

#include <cstring>

#include <iostream>

#include <cstdlib>

#define ll long long

using namespace std;

*/\**

*\* modular exponentiation*

*\*/*

ll modulo(ll base, ll exponent, ll mod)

{

ll x = 1;

ll y = base;

while (exponent > 0)

{

if (exponent % 2 == 1)

x = (x \* y) % mod;

y = (y \* y) % mod;

exponent = exponent / 2;

}

return x % mod;

}

*/\**

*\* Fermat's test for checking primality*

*\*/*

bool Fermat(ll p, int iterations)

{

if (p == 1)

{

return false;

}

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

{

ll a = rand() % (p - 1) + 1;

if (modulo(a, p - 1, p) != 1)

{

return false;

}

}

return true;

}

*/\**

*\* Main*

*\*/*

int main()

{

int iteration = 50;

ll num;

cout<<"Enter integer to test primality: ";

cin>>num;

if (Fermat(num, iteration))

cout<<num<<" is prime"<<endl;

else

cout<<num<<" is not prime"<<endl;

return 0;

}

# 90. **C++ Program to Perform Addition Operation Using Bitwise Operators**

This is a C++ Program to perform addition using bitwise operators. Using AND and XOR operators addition can be done, where carry is given by AND between two operands and result can be given by XOR between two operands.

Output:

$ g++ BitWiseAddition.cpp

$ a.out

Enter the numbers to be added:23 24

The Summation is: 47

#include<iostream>

#include<conio.h>

#include<stdlib.h>

using namespace std;

int add(int x, int y)

{

int carry;

while (y != 0)

{

carry = x & y;

x = x ^ y;

y = carry << 1;

}

return x;

}

int main(int argc, char \*\*argv)

{

cout << "Enter the numbers to be added:";

int x, y;

cin >> x >> y;

cout << "The Summation is: " << add(x, y);

}

# 91. **C++ Program to Implement Booth’s Multiplication Algorithm for Multiplication of 2 signed Numbers**

This is a C++ Program to multiply two signed numbers using booth’s algorithm. Booth’s multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two’s complement notation. Booth used desk calculators that were faster at shifting than adding and created the algorithm to increase their speed. Booth’s algorithm is of interest in the study of computer architecture.

Output:

$ g++ BoothsMultiplication.cpp

$ a.out

--Enter the multiplicand and multipier in signed 2's complement form if negative--

Number of multiplicand bit=5

Multiplicand=1 0 1 1 1

Number of multiplier bit=5

Multiplier=1 0 0 1 1

qn q[n+1] BR AC QR sc

initial 00000 10011 5

1 0 subtracting BR 01001

ashr 00100 11001 4

1 1 ashr 00010 01100 3

0 1 adding BR 11001

ashr 11100 10110 2

0 0 ashr 11110 01011 1

1 0 subtracting BR 00111

ashr 00011 10101 0

#include<iostream>

#include<conio.h>

using namespace std;

void add(int a[], int x[], int qrn);

void complement(int a[], int n)

{

int i;

int x[8] = { NULL };

x[0] = 1;

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

{

a[i] = (a[i] + 1) % 2;

}

add(a, x, n);

}

void add(int ac[], int x[], int qrn)

{

int i, c = 0;

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

{

ac[i] = ac[i] + x[i] + c;

if (ac[i] > 1)

{

ac[i] = ac[i] % 2;

c = 1;

}

else

c = 0;

}

}

void ashr(int ac[], int qr[], int &qn, int qrn)

{

int temp, i;

temp = ac[0];

qn = qr[0];

cout << "**\t\t**ashr**\t\t**";

for (i = 0; i < qrn - 1; i++)

{

ac[i] = ac[i + 1];

qr[i] = qr[i + 1];

}

qr[qrn - 1] = temp;

}

void display(int ac[], int qr[], int qrn)

{

int i;

for (i = qrn - 1; i >= 0; i--)

cout << ac[i];

cout << " ";

for (i = qrn - 1; i >= 0; i--)

cout << qr[i];

}

int main(int argc, char \*\*argv)

{

int mt[10], br[10], qr[10], sc, ac[10] = { 0 };

int brn, qrn, i, qn, temp;

cout

<< "**\n**--Enter the multiplicand and multipier in signed 2's complement form if negative--";

cout << "**\n** Number of multiplicand bit=";

cin >> brn;

cout << "**\n**multiplicand=";

for (i = brn - 1; i >= 0; i--)

cin >> br[i]; //multiplicand

for (i = brn - 1; i >= 0; i--)

mt[i] = br[i]; // copy multipier to temp array mt[]

complement(mt, brn);

cout << "**\n**No. of multiplier bit=";

cin >> qrn;

sc = qrn; //sequence counter

cout << "Multiplier=";

for (i = qrn - 1; i >= 0; i--)

cin >> qr[i]; //multiplier

qn = 0;

temp = 0;

cout << "qn**\t**q[n+1]**\t\t**BR**\t\t**AC**\t**QR**\t\t**sc**\n**";

cout << "**\t\t\t**initial**\t\t**";

display(ac, qr, qrn);

cout << "**\t\t**" << sc << "**\n**";

while (sc != 0)

{

cout << qr[0] << "**\t**" << qn;

if ((qn + qr[0]) == 1)

{

if (temp == 0)

{

add(ac, mt, qrn);

cout << "**\t\t**subtracting BR**\t**";

for (i = qrn - 1; i >= 0; i--)

cout << ac[i];

temp = 1;

}

else if (temp == 1)

{

add(ac, br, qrn);

cout << "**\t\t**adding BR**\t**";

for (i = qrn - 1; i >= 0; i--)

cout << ac[i];

temp = 0;

}

cout << "**\n\t**";

ashr(ac, qr, qn, qrn);

}

else if (qn - qr[0] == 0)

ashr(ac, qr, qn, qrn);

display(ac, qr, qrn);

cout << "**\t**";

sc--;

cout << "**\t**" << sc << "**\n**";

}

cout << "Result=";

display(ac, qr, qrn);

}

# 92. **C++ Program to Implement the Schonhage-Strassen Algorithm for Multiplication of Two Numbers**

# Output:

# $ g++ Schonhage-StrassenAlgorithm.cpp

# $ a.out

# 

# Enter the numbers:3452 1245

# The Linear Convolution is: ( 3 10 25 43 44 33 10 )

# Product of the numbers is: 4297740

# 

#include <iostream>

using namespace std;

int noOfDigit(long a)

{

int n = 0;

while (a > 0)

{

a /= 10;

n++;

}

return n;

}

void schonhageStrassenMultiplication(long x, long y, int n, int m)

{

int linearConvolution[n + m - 1];

for (int i = 0; i < (n + m - 1); i++)

linearConvolution[i] = 0;

long p = x;

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

{

x = p;

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

{

linearConvolution[i + j] += (y % 10) \* (x % 10);

x /= 10;

}

y /= 10;

}

cout << "The Linear Convolution is: ( ";

for (int i = (n + m - 2); i >= 0; i--)

{

cout << linearConvolution[i] << " ";

}

cout << ")";

long product = 0;

int nextCarry = 0, base = 1;

;

for (int i = 0; i < n + m - 1; i++)

{

linearConvolution[i] += nextCarry;

product = product + (base \* (linearConvolution[i] % 10));

nextCarry = linearConvolution[i] / 10;

base \*= 10;

}

cout << "**\n**The Product of the numbers is: " << product;

}

int main(int argc, char \*\*argv)

{

cout << "Enter the numbers:";

long a, b;

cin >> a >> b;

int n = noOfDigit(a);

int m = noOfDigit(b);

schonhageStrassenMultiplication(a, b, n, m);

}

# 93. **C++ Program to Implement Russian Peasant Multiplication**

This C++ Program demonstrates the implementation of Russian Peasant Multiplication.

int main()

{

cout << russianPeasant(15, 5) << endl;

cout << russianPeasant(13, 6) << endl;

return 0;

}

Output

$ **g++** russian\_peasant.cpp

$ a.out

75

78

*/\**

*\* C++ Program to Implement Russian Peasant Multiplication*

*\*/*

#include <iostream>

using namespace std;

*/\**

*\* multiply two numbers using Russian Peasant method*

*\*/*

unsigned int russianPeasant(unsigned int a, unsigned int b)

{

int res = 0;

while (b > 0)

{

if (b & 1)

res = res + a;

a = a << 1;

b = b >> 1;

}

return res;

}

*/\**

*\* Main*

*\*/*

int main()

{

cout << russianPeasant(15, 5) << endl;

cout << russianPeasant(13, 6) << endl;

return 0;

}

# 94. **C++ Program to Implement Fermat’s Little Theorem**

Output

$ **g++** fermat\_little.cpp

$ a.out

Enter number to **find** modular multiplicative inverse: 1111

Enter Modular Value: 331

216

*/\**

*\* C++ Program to Implement Fermat's Little Theorem*

*\*/*

#include <iostream>

using namespace std;

*/\* calculates (a^b)%MOD \*/*

int pow(int a, int b, int MOD)

{

int x = 1, y = a;

while (b > 0)

{

if (b % 2 == 1)

{

x = (x \* y);

if (x > MOD)

x %= MOD;

}

y = (y \* y);

if (y > MOD)

y %= MOD;

b /= 2;

}

return x;

}

int modInverse(int a, int m)

{

return pow(a, m - 2, m);

}

//Main

int main()

{

int a, m;

cout<<"Enter number to find modular multiplicative inverse: ";

cin>>a;

cout<<"Enter Modular Value: ";

cin>>m;

cout<<modInverse(a, m)<<endl;

}

# 95. **C++ Program to Solve the 0-1 Knapsack Problem**

This is a C++ Program to solve 0-1 knapsack problem. The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

Output:

$ g++ 0-1Knapsack.cpp

$ a.out

Enter the number of items in a Knapsack:5

Enter value and weight for item 0:11 111

Enter value and weight for item 1:22 121

Enter value and weight for item 2:33 131

Enter value and weight for item 3:44 141

Enter value and weight for item 4:55 151

Enter the capacity of knapsack 300

99

#include<stdio.h>

#include<conio.h>

#include<iostream>

using namespace std;

// A utility function that returns maximum of two integers

int max(int a, int b)

{

return (a > b) ? a : b;

}

// Returns the maximum value that can be put in a knapsack of capacity W

int knapSack(int W, int wt[], int val[], int n)

{

// Base Case

if (n == 0 || W == 0)

return 0;

// If weight of the nth item is more than Knapsack capacity W, then

// this item cannot be included in the optimal solution

if (wt[n - 1] > W)

return knapSack(W, wt, val, n - 1);

// Return the maximum of two cases: (1) nth item included (2) not included

else

return max(val[n - 1] + knapSack(W - wt[n - 1], wt, val, n - 1),

knapSack(W, wt, val, n - 1));

}

// Driver program to test above function

int main()

{

cout << "Enter the number of items in a Knapsack:";

int n, W;

cin >> n;

int val[n], wt[n];

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

{

cout << "Enter value and weight for item " << i << ":";

cin >> val[i];

cin >> wt[i];

}

// int val[] = { 60, 100, 120 };

// int wt[] = { 10, 20, 30 };

// int W = 50;

cout << "Enter the capacity of knapsack";

cin >> W;

cout << knapSack(W, wt, val, n);

return 0;

}

# 96. **C++ Perform to a 2D FFT Inplace Given a Complex 2D Array**

# 

Output:

$ g++ TwoDFFT.cpp

$ a.out

Enter the size:

2

Enter the 2D elements

2 3

4 2

2.5 + 0.0 i

5.5 + 0.0 i

-0.5 + -1.8369701987210297E-16 i

0.5 + -3.0616169978683826E-16 i

2.5 + 0.0 i

-0.5 + -3.6739403974420594E-16 i

-0.5 + -1.8369701987210297E-16 i

-1.5 + -1.8369701987210297E-16 i

#include <iostream>

#include <math.h>

using namespace std;

#define PI 3.14159265

int n;

int main(int argc, char \*\*argv)

{

cout << "Enter the size: ";

cin >> n;

double inputData[n][n];

cout << "Enter the 2D elements ";

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

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

cin >> inputData[i][j];

double realOut[n][n];

double imagOut[n][n];

double amplitudeOut[n][n];

int height = n;

int width = n;

// Two outer loops iterate on output data.

for (int yWave = 0; yWave < height; yWave++)

{

for (int xWave = 0; xWave < width; xWave++)

{

// Two inner loops iterate on input data.

for (int ySpace = 0; ySpace < height; ySpace++)

{

for (int xSpace = 0; xSpace < width; xSpace++)

{

// Compute real, imag, and ampltude.

realOut[yWave][xWave] += (inputData[ySpace][xSpace] \* cos(

2 \* PI \* ((1.0 \* xWave \* xSpace / width) + (1.0

\* yWave \* ySpace / height)))) / sqrt(

width \* height);

imagOut[yWave][xWave] -= (inputData[ySpace][xSpace] \* sin(

2 \* PI \* ((1.0 \* xWave \* xSpace / width) + (1.0

\* yWave \* ySpace / height)))) / sqrt(

width \* height);

amplitudeOut[yWave][xWave] = sqrt(

realOut[yWave][xWave] \* realOut[yWave][xWave]

+ imagOut[yWave][xWave]

\* imagOut[yWave][xWave]);

}

cout << realOut[yWave][xWave] << " + " << imagOut[yWave][xWave]

<< " i (" << amplitudeOut[yWave][xWave] << ")**\n**";

}

}

}

}

# 97. **C Program to Accepts two Strings & Compare them**

Problem Description

This program accepts two strings as input and compares them.

Problem Solution

1. Take two strings as input.

2. Compare the two strings and display the result whether both are equal, or first string is greater than the second or the first string is less than the second string

3. Exit.

#include <stdio.h>

int main ()

{

int count1 = 0, count2 = 0, flag = 0, i;

char string1[30], string2[30];

printf ("Enter the First string**\n**");

gets (string1);

printf ("Enter the Second string**\n**");

gets (string2);

while (string1[count1] != '**\0**')

count1 ++;

while (string2[count2] != '**\0**')

count2 ++;

i = 0;

while (string1[i] == string2[i] && string1[i] != '**\0**')

{

i ++;

}

if (string1[i] > string2[i])

printf ("First string is greater than Second string**\n**");

else if (string1[i] < string2[i])

printf("Second string is greater than First string**\n**");

else

printf ("Both strings are EQUAL**\n**");

return 0;

}

# 

# 98. **C Program to Find out the Roots of a Quadratic Equation**

**Problem Description**

This C Program calculates the roots of a quadratic equation.

**Problem Solution**

First it finds discriminant using the formula : disc = b \* b – 4 \* a \* c. There are 3 types of roots. They are complex, distinct & equal roots. We have to find the given equation belongs to which type of root.

*/\**

*\* C program to find out the roots of a quadratic equation*

*\* for non-zero coefficients. In case of errors the program*

*\* should report suitable error message.*

*\*/*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

void main()

{

float a, b, c, root1, root2;

float realp, imagp, disc;

printf("Enter the values of a, b and c **\n**");

scanf("%f %f %f", &a, &b, &c);

*/\* If a = 0, it is not a quadratic equation \*/*

if (a == 0 || b == 0 || c == 0)

{

printf("Error: Roots cannot be determined **\n**");

exit(1);

}

else

{

disc = b \* b - 4.0 \* a \* c;

if (disc < 0)

{

printf("Imaginary Roots**\n**");

realp = -b / (2.0 \* a) ;

imagp = sqrt(abs(disc)) / (2.0 \* a);

printf("Root1 = %f +i %f**\n**", realp, imagp);

printf("Root2 = %f -i %f**\n**", realp, imagp);

}

else if (disc == 0)

{

printf("Roots are real and equal**\n**");

root1 = -b / (2.0 \* a);

root2 = root1;

printf("Root1 = %f**\n**", root1);

printf("Root2 = %f**\n**", root2);

}

else if (disc > 0 )

{

printf("Roots are real and distinct **\n**");

root1 =(-b + sqrt(disc)) / (2.0 \* a);

root2 =(-b - sqrt(disc)) / (2.0 \* a);

printf("Root1 = %f **\n**", root1);

printf("Root2 = %f **\n**", root2);

}

}

}

# 99.C Program to Find the Frequency of the Word ‘the’ in a given Sentence

**Problem Description**

This program takes the sentence as input and finds the frequency of the word ‘the’ in a given sentence.

**Problem Solution**

1. Take any sentence as input.

2. Check for the word ‘the’ in the input sentence.

3. Use a variable to keep the count of number of ‘the’ in the sentence.

*/\**

*\* C program to accept a string and find the number of times the word*

*\* 'the' appears in that string*

*\*/*

#include <stdio.h>

void main()

{

int count = 0, i, times = 0, t, h, e, space;

char string[100];

puts("Enter a string:");

gets(string);

*/\* Traverse the string to count the number of characters \*/*

while (string[count] != '**\0**')

{

count++;

}

*/\* Finding the frequency of the word 'the' \*/*

for (i = 0; i <= count - 3; i++)

{

t =(string[i] == 't' || string[i] == 'T');

h =(string[i + 1] == 'h' || string[i + 1] == 'H');

e =(string[i + 2] == 'e'|| string[i + 2] == 'E');

space =(string[i + 3] == ' ' || string[i + 3] == '**\0**');

if ((t && h && e && space) == 1)

times++;

}

printf("Frequency of the word 'the' is %d**\n**", times);

}

# 100. C Program to Find First N Fibonacci Numbers

This C Program calculate the Fibonacci numbers in the series. The first two numbers in the Fibonacci sequence are 0 and 1 and each subsequent number is the sum of the previous two. The formula for this program is: Fn = Fn-1 + Fn-2

$ cc pgm10.c

$ a.out

Enter the value of num

15

First 15 FIBONACCI numbers are ...

0

1

1

2

3

5

8

13

21

34

55

89

144

233

377

*/\**

*\* C program to generate and print first N FIBONACCI numbers*

*\* in the series.*

*\*/*

#include <stdio.h>

void main()

{

int fib1 = 0, fib2 = 1, fib3, num, count = 0;

printf("Enter the value of num **\n**");

scanf("%d", &num);

printf("First %d FIBONACCI numbers are ...**\n**", num);

printf("%d**\n**", fib1);

printf("%d**\n**", fib2);

count = 2; */\* fib1 and fib2 are already used \*/*

while (count < num)

{

fib3 = fib1 + fib2;

count++;

printf("%d**\n**", fib3);

fib1 = fib2;

fib2 = fib3;

}

}

# 101. **C Program to Calculate the Sum of cos(x) Series**

# **Problem Description**

# This C Program calculates the sum of cos(x) series.

# **Problem Solution**

# Take input from the user and perform operations as shown in the program below.

*/\**

*\* C program to find the sum of cos(x) series*

*\*/*

#include <stdio.h>

#include <math.h>

void main()

{

int n, x1, i, j;

float x, sign, cosx, fact;

printf("Enter the number of the terms in a series**\n**");

scanf("%d", &n);

printf("Enter the value of x(in degrees)**\n**");

scanf("%f", &x);

x1 = x;

*/\* Degrees to radians \*/*

x = x \* (3.142 / 180.0);

cosx = 1;

sign = -1;

for (i = 2; i <= n; i = i + 2)

{

fact = 1;

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

{

fact = fact \* j;

}

cosx = cosx + (pow(x, i) / fact) \* sign;

sign = sign \* (-1);

}

printf("Sum of the cosine series = %7.2f**\n**", cosx);

printf("The value of cos(%d) using library function = %f**\n**", x1,

cos(x));

}

# 102. **C Program to Read a Grade & Display the Equivalent Description**

**Problem Description**

This program takes a grade as input and displays its equivalent description.

**Problem Solution**

1. Take the grade as input.

2. Use switch statement to verify the grade.

3. Print the output and exit.

**Program Explanation**

1. Take the letter as input and store it in the variable grade.

2. Convert the input letter into its uppercase using function toupper().

3. Using switch statement, verify the input letter.

4. If the letter is S, then copy the string ” SUPER” into the variable remark and break.

5. If the letter is A, then copy the string ” VERY GOOD” into the variable remark and break.

6. If the letter is B, then copy the string ” FAIR” into the variable remark and break.

7. If the letter is Y, then copy the string ” ABSENT” into the variable remark and break.

8. If the letter is F , then copy the string ” FAILS” into the variable remark and break.

9. In the default case, copy the string ” ERROR IN GRADE” into the variable remark and break.

10. Print the variable remark as output and exit.

Note: Join free Sanfoundry classes at [Telegram](https://t.me/sanfoundryclasses/) or [Youtube](https://www.youtube.com/c/SanfoundryOfficial)

*/\**

*\* C Program to accept a grade and declare the equivalent description*

*\* if code is S, then print SUPER*

*\* if code is A, then print VERY GOOD*

*\* if code is B, then print FAIR*

*\* if code is Y, then print ABSENT*

*\* if code is F, then print FAILS*

*\*/*

#include <stdio.h>

#include <ctype.h>

#include <string.h>

void main()

{

char remark[15];

char grade;

printf("Enter the grade **\n**");

scanf("%c", &grade);

*/\* lower case letter to upper case \*/*

grade = toupper(grade);

switch(grade)

{

case 'S':

strcpy(remark, " SUPER");

**break**;

case 'A':

strcpy(remark, " VERY GOOD");

**break**;

case 'B':

strcpy(remark, " FAIR");

**break**;

case 'Y':

strcpy(remark, " ABSENT");

**break**;

case 'F':

strcpy(remark, " FAILS");

**break**;

default :

strcpy(remark, "ERROR IN GRADE **\n**");

**break**;

}

printf("RESULT : %s**\n**", remark);

}

# 

# 103. C Program to Read a String and find the Sum of all Digits in the String

**Problem Description**

This program takes a string containing both digits and alphabet as input and finds the sum of all digits in the string.

**Problem Solution**

1. Take the string as input.

2. Check for the digits in the string.

3. Count the number of digits and add all the digits to get the sum.

# **Program Explanation**

# 1. Take the string containing both digits and alphabet as input and store it in the array string[].

# 2. Using for loop and if statement check for the digits in the array. If it is, then increment the variable nc by 1 and increment the variable sum with the current digit.

# 3. Do the step-2 till the end of the input string.

# 4. Variable nc gives the count of number of digits in the array and variable sum gives the sum of all the digits in the array.

# 

3. */\**
4. *\* C program to find the sum of all digits present in the string*
5. *\*/*
6. #include <stdio.h>
7. void main()
8. {
9. char string[80];
10. int count, nc = 0, sum = 0;
12. printf("Enter the string containing both digits and alphabet **\n**");
13. scanf("%s", string);
14. for (count = 0; string[count] != '**\0**'; count++)
15. {
16. if ((string[count] >= '0') && (string[count] <= '9'))
17. {
18. nc += 1;
19. sum += (string[count] - '0');
20. }
21. }
22. printf("NO. of Digits in the string = %d**\n**", nc);
23. printf("Sum of all digits = %d**\n**", sum);
24. }

# 104. **C++ Program to Implement Hash Tables Chaining with Binary Trees**

$ **g++** xor\_list.cpp

$ a.out

Operations on XOR Linked List

1.Insert Element at First

2.Display List

3.Quit

Enter your Choice: 1

Enter value to be inserted: 100

Operations on XOR Linked List

1.Insert Element at First

2.Display List

3.Quit

Enter your Choice: 2

Elements of XOR Linked List:

100

Operations on XOR Linked List

1.Insert Element at First

2.Display List

3.Quit

Enter your Choice: 1

Enter value to be inserted: 200

Operations on XOR Linked List

1.Insert Element at First

2.Display List

3.Quit

Enter your Choice: 2

Elements of XOR Linked List:

200 100

Operations on XOR Linked List

1.Insert Element at First

2.Display List

3.Quit

Enter your Choice: 1

Enter value to be inserted: 300

Operations on XOR Linked Lis

1.Insert Element at First

2.Display List

3.Quit

Enter your Choice: 2

Elements of XOR Linked List:

300 200 100

Operations on XOR Linked List

1.Insert Element at First

2.Display List

3.Quit

Enter your Choice: 1

Enter value to be inserted: 400

Operations on XOR Linked List

1.Insert Element at First

2.Display List

3.Quit

Enter your Choice: 2

Elements of XOR Linked List:

400 300 200 100

Operations on XOR Linked List

1.Insert Element at First

2.Display List

3.Quit

Enter your Choice: 1

Enter value to be inserted: 500

Operations on XOR Linked List

1.Insert Element at First

2.Display List

3.Quit

Enter your Choice: 2

Elements of XOR Linked List:

500 400 300 200 100

Operations on XOR Linked List

1.Insert Element at First

2.Display List

3.Quit

Enter your Choice: 3

**(**program exited with code: 1**)**

Press **return** to **continue**

1. */\**
2. *\* C++ Program to Implement Hash Tables Chaining with Binary Trees*
3. *\*/*
4. #include <iostream>
5. #include <string>
6. #include <vector>
7. #include <cstdlib>
8. using namespace std;
10. */\**
11. *\* Node Class Declaration*
12. *\*/*
13. template <class T>
14. class BSTNode
15. {
16. private:
17. T value;
18. BSTNode \*left, \*right;
19. public:
20. BSTNode(T value)
21. {
22. this->value = value;
23. left = NULL;
24. right = NULL;
25. }
26. */\**
27. *\* Adding element to a node*
28. *\*/*
29. void add(T& value)
30. {
31. if (value < this->value)
32. {
33. if (left)
34. left->add(value);
35. else
36. left = new BSTNode(value);
37. }
38. else if (this->value < value)
39. {
40. if (right)
41. right->add(value);
42. else
43. right = new BSTNode(value);
44. }
45. }
46. */\**
47. *\* Check if node contains element*
48. *\*/*
49. bool contains(T& value)
50. {
51. if (value < this->value)
52. {
53. if (left)
54. return left->contains(value);
55. else
56. return false;
57. }
58. else if (this->value < value)
59. {
60. if (right)
61. return right->contains(value);
62. else
63. return false;
64. }
65. else
66. {
67. return true;
68. }
69. }
70. friend class BSTHashtable;
71. };
73. */\**
74. *\* Table Class Declaration*
75. *\*/*
76. class BSTHashtable
77. {
78. private:
79. int size;
80. vector<BSTNode<string>\*>\* bucket;
81. int hash(string& s)
82. {
83. unsigned int hashVal = 0;
84. for (unsigned int i = 0; i < s.length(); i++)
85. hashVal ^= (hashVal << 5) + (hashVal >> 2) + s[i];
86. return hashVal % size;
87. }
88. public:
89. BSTHashtable(int vectorSize)
90. {
91. size = vectorSize;
92. bucket = new vector<BSTNode<string>\*>(size);
93. }
94. */\**
95. *\* Adding string in the table*
96. *\*/*
97. void add(string& s)
98. {
99. int index = hash(s);
100. if (bucket->at(index) == NULL)
101. bucket->at(index) = new BSTNode<string>(s);
102. else
103. bucket->at(index)->add(s);
104. }
105. */\**
106. *\* Check if table contains string*
107. *\*/*
108. bool contains(string& s)
109. {
110. int index = hash(s);
111. if (bucket->at(index) == NULL)
112. return false;
113. cout<<"String **\"**"<<s<<"**\"** found at index: "<<index<<endl;
114. return bucket->at(index)->contains(s);
115. }
116. */\**
117. *\* Display Table*
118. *\*/*
119. void display()
120. {
121. for (unsigned int i = 0; i < bucket->size(); i++)
122. {
123. cout <<"[" << i << "] ";
124. if (bucket->at(i) != NULL)
125. {
126. BSTNode<string> \*node = bucket->at(i);
127. display\_bst(node);
128. }
129. cout << endl;
130. }
131. }
132. */\**
133. *\* Display BST*
134. *\*/*
135. void display\_bst(BSTNode<string> \*node)
136. {
137. if (node!=NULL)
138. {
139. display\_bst(node->left);
140. cout << node->value << " ";
141. display\_bst(node->right);
142. }
143. }
144. };
146. */\**
147. *\* Main Contains Menu*
148. *\*/*
149. int main()
150. {
151. BSTHashtable table(10);
152. string str;
153. int choice;
154. while (1)
155. {
156. cout<<"**\n**----------------------"<<endl;
157. cout<<"Operations on BST Hash Table"<<endl;
158. cout<<"**\n**----------------------"<<endl;
159. cout<<"1.Insert element into the table"<<endl;
160. cout<<"2.Find element in the table"<<endl;
161. cout<<"3.Display Table Chained with Binary Tree"<<endl;
162. cout<<"4.Exit"<<endl;
163. cout<<"Enter your choice: ";
164. cin>>choice;
165. switch(choice)
166. {
167. case 1:
168. cout<<"Enter String to be inserted: ";
169. cin>>str;
170. table.add(str);
171. break;
172. case 2:
173. cout<<"Enter String to search: ";
174. cin>>str;
175. if (table.contains(str) == 0)
176. {
177. cout<<"String **\"**"<<str<<"**\"** not found in the table"<<endl;
178. continue;
179. }
180. break;
181. case 3:
182. cout<<"Displaying Table Chained with Binary Tree: "<<endl;
183. table.display();
184. break;
185. case 4:
186. exit(1);
187. default:
188. cout<<"**\n**Enter correct option**\n**";
189. }
190. }
191. return 0;
192. }

# 105. **C++ Program to Implement Hash Tables chaining with Singly Linked Lists**

# This C++ Program demonstrates operations on Hash Tables chaining with Singly Linked Lists.

1. */\**
2. *\* C++ Program to Implement Hash Tables chaining*
3. *\* with Singly Linked Lists*
4. *\*/*
5. #include<iostream>
6. #include<cstdlib>
7. #include<string>
8. #include<cstdio>
9. using namespace std;
10. const int TABLE\_SIZE = 128;
12. */\**
13. *\* HashNode Class Declaration*
14. *\*/*
15. class HashNode
16. {
17. public:
18. . int key;
19. int value;
20. HashNode\* next;
21. HashNode(int key, int value)
22. {
23. this->key = key;
24. this->value = value;
25. this->next = NULL;
26. }
27. };
29. */\**
30. *\* HashMap Class Declaration*
31. *\*/*
32. class HashMap
33. {
34. private:
35. HashNode\*\* htable;
36. public:
37. HashMap()
38. {
39. htable = new HashNode\*[TABLE\_SIZE];
40. for (int i = 0; i < TABLE\_SIZE; i++)
41. htable[i] = NULL;
42. }
43. ~HashMap()
44. {
45. for (int i = 0; i < TABLE\_SIZE; ++i)
46. {
47. HashNode\* entry = htable[i];
48. while (entry != NULL)
49. {
50. HashNode\* prev = entry;
51. entry = entry->next;
52. delete prev;
53. }
54. }
55. delete[] htable;
56. }
57. */\**
58. *\* Hash Function*
59. *\*/*
60. int HashFunc(int key)
61. {
62. return key % TABLE\_SIZE;
63. }
65. */\**
66. *\* Insert Element at a key*
67. *\*/*
68. void Insert(int key, int value)
69. {
70. int hash\_val = HashFunc(key);
71. HashNode\* prev = NULL;
72. HashNode\* entry = htable[hash\_val];
73. while (entry != NULL)
74. {
75. prev = entry;
76. entry = entry->next;
77. }
78. if (entry == NULL)
79. {
80. entry = new HashNode(key, value);
81. if (prev == NULL)
82. {
83. htable[hash\_val] = entry;
84. }
85. else
86. {
87. prev->next = entry;
88. }
89. }
90. else
91. {
92. entry->value = value;
93. }
94. }
95. */\**
96. *\* Remove Element at a key*
97. *\*/*
98. void Remove(int key)
99. {
100. int hash\_val = HashFunc(key);
101. HashNode\* entry = htable[hash\_val];
102. HashNode\* prev = NULL;
103. if (entry == NULL || entry->key != key)
104. {
105. cout<<"No Element found at key "<<key<<endl;
106. return;
107. }
108. while (entry->next != NULL)
109. {
110. prev = entry;
111. entry = entry->next;
112. }
113. if (prev != NULL)
114. {
115. prev->next = entry->next;
116. }
117. delete entry;
118. cout<<"Element Deleted"<<endl;
119. }
120. */\**
121. *\* Search Element at a key*
122. *\*/*
123. int Search(int key)
124. {
125. bool flag = false;
126. int hash\_val = HashFunc(key);
127. HashNode\* entry = htable[hash\_val];
128. while (entry != NULL)
129. {
130. if (entry->key == key)
131. {
132. cout<<entry->value<<" ";
133. flag = true;
134. }
135. entry = entry->next;
136. }
137. if (!flag)
138. return -1;
139. }
140. };
141. */\**
142. *\* Main Contains Menu*
143. *\*/*
144. int main()
145. {
146. HashMap hash;
147. int key, value;
148. int choice;
149. while (1)
150. {
151. cout<<"**\n**----------------------"<<endl;
152. cout<<"Operations on Hash Table"<<endl;
153. cout<<"**\n**----------------------"<<endl;
154. cout<<"1.Insert element into the table"<<endl;
155. cout<<"2.Search element from the key"<<endl;
156. cout<<"3.Delete element at a key"<<endl;
157. cout<<"4.Exit"<<endl;
158. cout<<"Enter your choice: ";
159. cin>>choice;
160. switch(choice)
161. {
162. case 1:
163. cout<<"Enter element to be inserted: ";
164. cin>>value;
165. cout<<"Enter key at which element to be inserted: ";
166. cin>>key;
167. hash.Insert(key, value);
168. break;
169. case 2:
170. cout<<"Enter key of the element to be searched: ";
171. cin>>key;
172. cout<<"Element at key "<<key<<" : ";
173. if (hash.Search(key) == -1)
174. {
175. cout<<"No element found at key "<<key<<endl;
176. continue;
177. }
178. break;
179. case 3:
180. cout<<"Enter key of the element to be deleted: ";
181. cin>>key;
182. hash.Remove(key);
183. break;
184. case 4:
185. exit(1);
186. default:
187. cout<<"**\n**Enter correct option**\n**";
188. }
189. }
190. return 0;
191. }

# 106. **C++ Program to Implement Radix Sort**

# **roblem Description**

# 1. In this algorithm sorting of data is done from least significant digit to most significant digit.

# 2. Here we need 10 different spaces labeled 0 to 9.

# 3. Assume we have ‘n’ number of inputs.

# 4. Let ‘d’ be the maximum number of digit the input data has.

# 5. The time complexity for radix sort is O(n\*d).

# 6. Radix sort solves the problem of card sorting by sorting on the least significant digit first.

# **Problem Solution**

# 1. Get the maximum value from the input array which has ‘d’ digits.

# 2. Starting from least significant digit, sort the data.

# 3. Take this data as input for next significant digit.

# 4. Run the iteration till d digit.

# 5. Display the result.

# 6. Exit.

# **Program Explanation**

# 1. Take input of data.

# 2. Get the maximum of input data.

# 3. Run the countSort() till (m/exp) > 0.

# 4. Sort the data on the basis of the digit at (exp)th place.

# 5. Assign the sorted data back to arr[] array.

# 6. Check the condition in step 3.

# 7. If false, print the sorted output.

# 8. Exit.

#include <iostream>

using namespace std;

// Get maximum value from array.

int getMax(int arr[], int n)

{

int max = arr[0];

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

if (arr[i] > max)

max = arr[i];

return max;

}

// Count sort of arr[].

void countSort(int arr[], int n, int exp)

{

// Count[i] array will be counting the number of array values having that 'i' digit at their (exp)th place.

int output[n], i, count[10] = {0};

// Count the number of times each digit occurred at (exp)th place in every input.

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

count[(arr[i] / exp) % 10]++;

// Calculating their cumulative count.

for (i = 1; i < 10; i++)

count[i] += count[i-1];

// Inserting values according to the digit '(arr[i] / exp) % 10' fetched into count[(arr[i] / exp) % 10].

for (i = n - 1; i >= 0; i--)

{

output[count[(arr[i] / exp) % 10] - 1] = arr[i];

count[(arr[i] / exp) % 10]--;

}

// Assigning the result to the arr pointer of main().

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

arr[i] = output[i];

}

// Sort arr[] of size n using Radix Sort.

void radixsort(int arr[], int n)

{

int exp, m;

m = getMax(arr, n);

// Calling countSort() for digit at (exp)th place in every input.

for (exp = 1; m/exp > 0; exp \*= 10)

countSort(arr, n, exp);

}

int main()

{

int n, i;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>n;

int arr[n];

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

{

cout<<"Enter element "<<i+1<<": ";

cin>>arr[i];

}

radixsort(arr, n);

// Printing the sorted data.

cout<<"**\n**Sorted Data ";

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

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

return 0;

}

# 

# 107.**C++ Program to Implement Bucket Sort**

# **Problem Description**

# 1. We should implement Bucket Sort on uniformly distributed data over a range by splitting the range into equal parts.

# 2. Assign those parts as buckets and each bucket ‘i’ will be having ‘Ni’ number of the elements.

# 3. Selecting these Bucket for inserting will cost time complexity of O(N) where N is a total number of elements.

# 4. Sort them separately. We have used insertion sort which has a time complexity of summation of O(Ni^2).

# 5. Complexity for bucket sort is O(N + summation of(Ni^2)).

# 6. It is better than the other sorting algorithms (insertion sort, bubble sort, etc) with complexities O(N^2).

# **Problem Solution**

# 1. Divide the range into equal parts and assign a bucket to each part.

# 2. Split the data and insert them into the corresponding bucket using insertion sort.

# 3. Merge all the buckets into one.

# 4. Display the result.

# 5. Exit.

**Program Explanation**

1. Create Node, Bucket and BucketList structures.

2. Take range and data, which should be uniformly distributed over the range.

3. Allocate memory to their objects accordingly.

4. Insert the data into the list according to their value which comes out to dividing data into the bucket.

5. Sort data using insertion sort in the linked lists.

6. Display the result.

7. Exit.

#include <iostream>

using namespace std;

// A structure to represent a node.

struct Node

{

int value;

struct Node\* next;

};

// A structure to represent a Head Bucket Node of the bucket list.

struct Bucket

{

// Pointer to head node of Bucket.

struct Node \*head;

};

struct BucketList

{

int V;

struct Bucket \* array;

};

// A utility function to create a new node for a particular entry in a bucket.

struct Node\* newNode(int value)

{

struct Node\* newnode = new Node;

newnode->value = value;

newnode->next = NULL;

return newnode;

}

// A utility function that creates a list of the bucket over the range of input data.

struct BucketList\* createBucket(int V)

{

int i;

struct BucketList\* bl = new BucketList;

bl->V = V;

bl->array = new Bucket[V];

// Initialize each Bucket list as empty by making head as NULL.

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

bl->array[i].head = NULL;

return bl;

}

// A function to Insert the nodes to corresponding Buckets.

void addNode(struct BucketList\* bl, int bckt, int value)

{

// Creating new data node.

struct Node \*newnode = newNode(value);

struct Node \*temp = new Node;

if(bl->array[bckt].head != NULL)

{

temp = bl->array[bckt].head;

// Sorting.

// If the head node value is lesser than the newnode value, then add node at beginning.

if(temp->value > newnode->value)

{

newnode->next = bl->array[bckt].head;

bl->array[bckt].head = newnode;

}

else

{

// Search for the node whose value is more than the newnode value.

while(temp->next != NULL)

{

if((temp->next)->value > newnode->value)

break;

temp = temp->next;

}

// Insert newnode after temp node.

newnode->next = temp->next;

temp->next = newnode;

}

}

else

{

// Assign head of the Bucket as newnode since bucket head is NULL.

bl->array[bckt].head = newnode;

}

}

// A function to print the result as sorted Data.

void printBuckets(struct BucketList \*bl)

{

int v;

struct Node\* pCrawl = new Node;

for(v = 0; v < bl->V; v++)

{

// To view the data in individual bucket remove next line from comment.

// cout<<"\n\t bucket "<<v+1;

pCrawl = bl->array[v].head;

while (pCrawl != NULL)

{

cout<<"->"<< pCrawl->value;

pCrawl = pCrawl->next;

}

}

}

int main()

{

// Create the BucketLists for the data and set 10 as default number of Buckets.

int V = 10, range, NOE, i;

struct BucketList\* mybucket = createBucket(V);

cout<<"**\n\n**Enter the upper limit in the power of 10 (10 or 100 or 1000 ..) to create Bucket: ";

cin>>range;

// Dividing range into 10 parts so it will have 10 buckets as default.

range = range/10;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>NOE;

int arr[NOE];

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

{

cout<<"Enter element "<<i+1<<" : ";

cin>>arr[i];

addNode(mybucket, arr[i]/range, arr[i]);

}

// Print the adjacency list representation of the BucketList i.e the sorted Output.

cout<<"**\n**Sorted Data ";

printBuckets(mybucket);

return 0;

}

# 108.**C++ Program to Implement Bubble Sort**

# **Problem Description**

# 1. Bubble sort algorithm sort data by comparing two consecutive numbers.

# 2. The time complexity of this algorithm is O(n^2).

# **Problem Solution**

# 1. Compare two consecutive number.

# 2. Switch values if the number with higher index value is smaller.

# 3. Display the result.

# 4. Exit.

**Program Explanation**

1. Take input of data.

2. Call BubbleSort() function with ‘arr’ the array of data and ‘n’ the number of values, in the argument list.

3. Implement Sorting algorithm using nested for loop.

4. The first loop will run on ‘i’ from 0 to n-1.

5. The second loop will run on ‘j’ from 0 to n-i-1.

6. Compare two consecutive values.

7. Switch the values if arr[j+1] <arr[j].

8. Return to main and display the result.

9. Exit.

#include <iostream>

using namespace std;

// Sort arr[] of size n using Bubble Sort.

void BubbleSort (int arr[], int n)

{

int i, j;

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

{

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

{

// Comparing consecutive data and switching values if value at j > j+1.

if (arr[j] > arr[j+1])

{

arr[j] = arr[j]+arr[j+1];

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

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

}

}

// Value at n-i-1 will be maximum of all the values below this index.

}

}

int main()

{

int n, i;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>n;

int arr[n];

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

{

cout<<"Enter element "<<i+1<<": ";

cin>>arr[i];

}

BubbleSort(arr, n);

// Display the sorted data.

cout<<"**\n**Sorted Data ";

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

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

return 0;

}

# 109.**C++ Program to Implement Heap Sort**

**Problem Description**

1. Heap sort is a comparison based algorithm.

2. It is improved version of selection sort.

3. The time complexity is O(n\*log(n)).

**Problem Solution**

1. Build a max heap using the given data element.

2. Delete the root node repeatedly.

3. Store the node at the end of the array.

4. Display the result.

5. Exit.

**Program Explanation**

1. Take input of data.

2. Call Build\_MaxHeap() function with ‘arr’ the array of data and ‘n-1’ the number of values, in the argument list.

3. After building the max heap call HeapSort().

4. Switch the root value of heap with the last index value of array since root value is highest among all.

5. Decrement the last index value.

6. Repeat it for all the element.

7. Return to main and display the result.

8. Exit.

#include <iostream>

using namespace std;

// A function to heapify the array.

void MaxHeapify(int a[], int i, int n)

{

int j, temp;

temp = a[i];

j = 2\*i;

while (j <= n)

{

if (j < n && a[j+1] > a[j])

j = j+1;

// Break if parent value is already greater than child value.

if (temp > a[j])

break;

// Switching value with the parent node if temp < a[j].

else if (temp <= a[j])

{

a[j/2] = a[j];

j = 2\*j;

}

}

a[j/2] = temp;

return;

}

void HeapSort(int a[], int n)

{

int i, temp;

for (i = n; i >= 2; i--)

{

// Storing maximum value at the end.

temp = a[i];

a[i] = a[1];

a[1] = temp;

// Building max heap of remaining element.

MaxHeapify(a, 1, i - 1);

}

}

void Build\_MaxHeap(int a[], int n)

{

int i;

for(i = n/2; i >= 1; i--)

MaxHeapify(a, i, n);

}

int main()

{

int n, i;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>n;

n++;

int arr[n];

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

{

cout<<"Enter element "<<i<<": ";

cin>>arr[i];

}

// Building max heap.

Build\_MaxHeap(arr, n-1);

HeapSort(arr, n-1);

// Printing the sorted data.

cout<<"**\n**Sorted Data ";

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

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

return 0;

}

# 110.**C++ Program to Implement Merge Sort**

**Problem Description**

1. Merge-sort is based on an algorithmic design pattern called divide-and-conquer.

2. It forms tree structure.

3. The height of the tree will be log(n).

4. we merge n element at every level of the tree.

5. The time complexity of this algorithm is O(n\*log(n)).

**Problem Solution**

1. Split the data into two equal half until we get at most one element in both half.

2. Merge Both into one making sure the resulting sequence is sorted.

3. Recursively split them and merge on the basis of constraint given in step 1.

4. Display the result.

5. Exit.

**Program Explanation**

1. Take input of data.

2. Call MergeSort() function.

3. Recursively split the array into two equal parts.

4. Split them until we get at most one element in both half.

5. Combine the result by invoking Merge().

6. It combines the individually sorted data from low to mid and mid+1 to high.

7. Return to main and display the result.

8. Exit.

#include <iostream>

using namespace std;

// A function to merge the two half into a sorted data.

void Merge(int \*a, int low, int high, int mid)

{

// We have low to mid and mid+1 to high already sorted.

int i, j, k, temp[high-low+1];

i = low;

k = 0;

j = mid + 1;

// Merge the two parts into temp[].

while (i <= mid && j <= high)

{

if (a[i] < a[j])

{

temp[k] = a[i];

k++;

i++;

}

else

{

temp[k] = a[j];

k++;

j++;

}

}

// Insert all the remaining values from i to mid into temp[].

while (i <= mid)

{

temp[k] = a[i];

k++;

i++;

}

// Insert all the remaining values from j to high into temp[].

while (j <= high)

{

temp[k] = a[j];

k++;

j++;

}

// Assign sorted data stored in temp[] to a[].

for (i = low; i <= high; i++)

{

a[i] = temp[i-low];

}

}

// A function to split array into two parts.

void MergeSort(int \*a, int low, int high)

{

int mid;

if (low < high)

{

mid=(low+high)/2;

// Split the data into two half.

MergeSort(a, low, mid);

MergeSort(a, mid+1, high);

// Merge them to get sorted output.

Merge(a, low, high, mid);

}

}

int main()

{

int n, i;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>n;

int arr[n];

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

{

cout<<"Enter element "<<i+1<<": ";

cin>>arr[i];

}

MergeSort(arr, 0, n-1);

// Printing the sorted data.

cout<<"**\n**Sorted Data ";

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

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

return 0;

}

# 111.**C++ Program to Implement Selection Sort**

**Problem Description**

1. Selection sort algorithm sort data by comparing one element to every other element and decide its position.

2. The time complexity of this algorithm is O(n^2).

**Problem Solution**

1. Starting from the beginning pick one number.

2. Compare it with others one by one.

3. replace if the other number is lesser than this one.

4. Display the result.

5. Exit.

#include <iostream>

using namespace std;

// Sort arr[] of size n using Selection Sort.

void SelectionSort (int arr[], int n)

{

int i, j;

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

{

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

{

// Comparing consecutive data and switching values if value at i > j.

if (arr[i] > arr[j])

{

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

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

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

}

}

// Value at i will be minimum of all the values above this index.

}

}

int main()

{

int n, i;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>n;

int arr[n];

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

{

cout<<"Enter element "<<i+1<<": ";

cin>>arr[i];

}

SelectionSort(arr, n);

// Display the sorted data.

cout<<"**\n**Sorted Data ";

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

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

return 0;

}

**Program Explanation**

1. Take input of data.

2. Call SelectionSort() function with ‘arr’ the array of data and ‘n’ the number of values, in the argument list.

3. Implement Sorting algorithm using nested for loop.

4. The first loop will run on ‘i’ from 0 to n-1.

5. The second loop will run on ‘j’ from i+1 to n-1.

6. Compare value at i with value at j.

7. Switch the values if arr[j+1] < arr[j]. 8. Return to main and display the result. 9. Exit.

# 112.**C++ Program to Implement Insertion Sort**

**Problem Description**

1. Insertion sort algorithm sort data by inserting them one by one into the list.

2. The time complexity of this algorithm is O(n^2).

**Problem Solution**

1. This algorithm is based on sorting playing cards by picking and inserting them one by one.

2. Here we take data element and place it in sorted list.

3. It should be placed so that list remains sorted.

4. Display the result.

5. Exit.

#include <iostream>

using namespace std;

// A structure to represent a node.

struct list

{

int data;

list \*next;

};

// Function implementing insertion sort.

list\* InsertinList(list \*head, int n)

{

// Creating newnode and temp node.

list \*newnode = new list;

list \*temp = new list;

// Using newnode as the node to be inserted in the list.

newnode->data = n;

newnode->next = NULL;

// If head is null then assign new node to head.

if(head == NULL)

{

head = newnode;

return head;

}

else

{

temp = head;

// If newnode->data is lesser than head->data, then insert newnode before head.

if(newnode->data < head->data)

{

newnode->next = head;

head = newnode;

return head;

}

// Traverse the list till we get value more than newnode->data.

while(temp->next != NULL)

{

if(newnode->data < (temp->next)->data)

break;

temp=temp->next;

}

// Insert newnode after temp.

newnode->next = temp->next;

temp->next = newnode;

return head;

}

}

int main()

{

int n, i, num;

// Declaring head of the linked list.

list \*head = new list;

head = NULL;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>n;

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

{

cout<<"Enter element "<<i+1<<": ";

cin>>num;

// Inserting num in the list.

head = InsertinList(head, num);

}

// Display the sorted data.

cout<<"**\n**Sorted Data ";

while(head != NULL)

{

cout<<"->"<<head->data;

head = head->next;

}

return 0;

}

**Program Explanation**

1. Create a head node of the list structure.

2. Take input of data and simultaneously insert it into a list using InsertinList().

3. Assign the new element as newnode.

4. If the head is null then assign newnode to head.

5. Otherwise, insert the newnode so that list remains sorted.

6. Return head to main.

7. Display the result.

8. Exit.

# 113.**C++ Program to Implement Quick Sort Using Randomization**

**Problem Description**

1. Quick sort is based on an algorithmic design pattern called divide-and-conquer.

2. Unlike Merge Sort it doesn’t require extra memory space.

3. The average time complexity is O(n\*log(n)) but the worst case complexity is O(n^2).

4. To reduce the chances of the worst case we have implemented Quicksort using randomization.

5. Here we will be selecting the pivot randomly.

**Problem Solution**

1. Randomly select pivot value from the given subpart of the array.

2. Partition that subpart so that the values left of the pivot are smaller and to the right are greater from the pivot.

3. Consider both as new sub-array and repeat step 1 until only one element left in subpart.

4. Display the result.

5. Exit.

#include<iostream>

#include<cstdlib>

using namespace std;

// Swapping two values.

void swap(int \*a, int \*b)

{

int temp;

temp = \*a;

\*a = \*b;

\*b = temp;

}

// Partitioning the array on the basis of values at high as pivot value.

int Partition(int a[], int low, int high)

{

int pivot, index, i;

index = low;

pivot = high;

// Getting index of pivot.

for(i=low; i < high; i++)

{

if(a[i] < a[pivot])

{

swap(&a[i], &a[index]);

index++;

}

}

// Swapping value at high and at the index obtained.

swap(&a[pivot], &a[index]);

return index;

}

// Random selection of pivot.

int RandomPivotPartition(int a[], int low, int high)

{

int pvt, n, temp;

n = rand();

// Randomizing the pivot value in the given subpart of array.

pvt = low + n%(high-low+1);

// Swapping pvt value from high, so pvt value will be taken as pivot while partitioning.

swap(&a[high], &a[pvt]);

return Partition(a, low, high);

}

// Implementing QuickSort algorithm.

int QuickSort(int a[], int low, int high)

{

int pindex;

if(low < high)

{

// Partitioning array using randomized pivot.

pindex = RandomPivotPartition(a, low, high);

// Recursively implementing QuickSort.

QuickSort(a, low, pindex-1);

QuickSort(a, pindex+1, high);

}

return 0;

}

int main()

{

int n, i;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>n;

int arr[n];

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

{

cout<<"Enter element "<<i+1<<": ";

cin>>arr[i];

}

QuickSort(arr, 0, n-1);

// Printing the sorted data.

cout<<"**\n**Sorted Data ";

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

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

return 0;

}

# 114.**C++ Program to Implement Shell Sort**

**Problem Description**

1. Shell sort is an improvement over insertion sort.

2. It compares the element separated by a gap of several positions.

3. A data element is sorted with multiple passes and with each pass gap value reduces.

4. The worst case time complexity is O(n\*log(n)).

**Problem Solution**

1. Assign gap value as half the length of the array.

2. Compare element present at a difference of gap value.

3. Sort them and reduce the gap value to half and repeat.

4. Display the result.

5. Exit.

#include<iostream>

using namespace std;

// A function implementing Shell sort.

void ShellSort(int a[], int n)

{

int i, j, k, temp;

// Gap 'i' between index of the element to be compared, initially n/2.

for(i = n/2; i > 0; i = i/2)

{

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

{

for(k = j-i; k >= 0; k = k-i)

{

// If value at higher index is greater, then break the loop.

if(a[k+i] >= a[k])

break;

// Switch the values otherwise.

else

{

temp = a[k];

a[k] = a[k+i];

a[k+i] = temp;

}

}

}

}

}

int main()

{

int n, i;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>n;

int arr[n];

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

{

cout<<"Enter element "<<i+1<<": ";

cin>>arr[i];

}

ShellSort(arr, n);

// Printing the sorted data.

cout<<"**\n**Sorted Data ";

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

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

return 0;

}

# 115.**C++ Program to Implement Sorting of Less than 100 Numbers in O(n) Complexity**

**Problem Description**

1. We can implement the task best using Radix sort.

2. In this algorithm sorting of data is done from least significant digit to most significant digit.

3. Here we need 10 different spaces labeled 0 to 9.

4. Assume we have ‘n’ number of inputs.

5. Let ‘d’ be the maximum number of digit the input data has, here d can have values either 1 or 2 or 3.

6. The time complexity for radix sort is O(n\*3) that is O(n).

7. Radix sort solves the problem of card sorting by sorting on the least significant digit first.

**Problem Solution**

1. Get the maximum value from the input array which has ‘d’ digits.

2. Starting from least significant digit, sort the data.

3. Take this data as input for next significant digit.

4. Run the iteration till d digit.

5. Display the result.

6. Exit.

#include <iostream>

using namespace std;

// Get maximum value from array.

int getMax(int arr[], int n)

{

int max = arr[0];

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

if (arr[i] > max)

max = arr[i];

return max;

}

// Count sort of arr[].

void countSort(int arr[], int n, int exp)

{

// Count[i] array will be counting the number of array values having that 'i' digit at their (exp)th place.

int output[n], i, count[10] = {0};

// Count the number of times each digit occurred at (exp)th place in every input.

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

count[(arr[i] / exp) % 10]++;

// Calculating their cumulative count.

for (i = 1; i < 10; i++)

count[i] += count[i-1];

// Inserting values according to the digit '(arr[i] / exp) % 10' fetched into count[(arr[i] / exp) % 10].

for (i = n - 1; i >= 0; i--)

{

output[count[(arr[i] / exp) % 10] - 1] = arr[i];

count[(arr[i] / exp) % 10]--;

}

// Assigning the result to the arr pointer of main().

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

arr[i] = output[i];

}

// Sort arr[] of size n using Radix Sort.

void radixsort(int arr[], int n)

{

int exp, m;

m = getMax(arr, n);

// Calling countSort() for digit at (exp)th place in every input.

for (exp = 1; m/exp > 0; exp \*= 10)

countSort(arr, n, exp);

}

int main()

{

int n, i;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>n;

int arr[n];

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

{

cout<<"Enter element "<<i+1<<": ";

cin>>arr[i];

}

radixsort(arr, n);

// Printing the sorted data.

cout<<"**\n**Sorted Data ";

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

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

return 0;

}

# 116.**C++ Program to Implement Merge Sort using Linked List**

**Problem Description**

1. Merge-sort is based on an algorithmic design pattern called divide-and-conquer.

2. It forms tree structure.

3. The height of the tree will be log(n).

4. we merge n element at every level of the tree.

5. The time complexity of this algorithm is O(n\*log(n)).

**Problem Solution**

1. Split the data into two equal half until we get at most one element in both half.

2. Merge Both into one making sure the resulting sequence is sorted.

3. Recursively split them and merge on the basis of constraint given in step 1.

4. Display the result.

5. Exit.

#include<iostream>

using namespace std;

// A structure representing a node of a linked list.

struct node

{

int data;

node \*next;

};

// A function creating a new node.

node\* NewNode(int d)

{

struct node \*temp = new node;

temp->data = d;

temp->next = NULL;

// Returning temp as the new node.

return temp;

}

// A function adding the given data at the end of the linked list.

node\* AddToList(node \*tail, int data)

{

struct node \*newnode;

newnode = NewNode(data);

if(tail == NULL)

{

tail = newnode;

}

// If tail is not null assign newnode to the next of tail.

else

{

tail->next = newnode;

// Shift tail pointer to the added node.

tail = tail->next;

}

return tail;

}

node\* Merge(node\* h1, node\* h2)

{

node \*t1 = new node;

node \*t2 = new node;

node \*temp = new node;

// Return if the first list is empty.

if(h1 == NULL)

return h2;

// Return if the Second list is empty.

if(h2 == NULL)

return h1;

t1 = h1;

// A loop to traverse the second list, to merge the nodes to h1 in sorted way.

while (h2 != NULL)

{

// Taking head node of second list as t2.

t2 = h2;

// Shifting second list head to the next.

h2 = h2->next;

t2->next = NULL;

// If the data value is lesser than the head of first list add that node at the beginning.

if(h1->data > t2->data)

{

t2->next = h1;

h1 = t2;

t1 = h1;

continue;

}

// Traverse the first list.

flag:

if(t1->next == NULL)

{

t1->next = t2;

t1 = t1->next;

}

// Traverse first list until t2->data more than node's data.

else if((t1->next)->data <= t2->data)

{

t1 = t1->next;

goto flag;

}

else

{

// Insert the node as t2->data is lesser than the next node.

temp = t1->next;

t1->next = t2;

t2->next = temp;

}

}

// Return the head of new sorted list.

return h1;

}

// A function implementing Merge Sort on linked list using reference.

void MergeSort(node \*\*head)

{

node \*first = new node;

node \*second = new node;

node \*temp = new node;

first = \*head;

temp = \*head;

// Return if list have less than two nodes.

if(first == NULL || first->next == NULL)

{

return;

}

else

{

// Break the list into two half as first and second as head of list.

while(first->next != NULL)

{

first = first->next;

if(first->next != NULL)

{

temp = temp->next;

first = first->next;

}

}

second = temp->next;

temp->next = NULL;

first = \*head;

}

// Implementing divide and conquer approach.

MergeSort(&first);

MergeSort(&second);

// Merge the two part of the list into a sorted one.

\*head = Merge(first, second);

}

int main()

{

int n, i, num;

struct node \*head = new node;

struct node \*tail = new node;

head = NULL;

tail = NULL;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>n;

// Create linked list.

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

{

cout<<"Enter element "<<i+1<<": ";

cin>>num;

tail = AddToList(tail, num);

if(head == NULL)

head = tail;

}

// Send reference of head into MergeSort().

MergeSort(&head);

// Printing the sorted data.

cout<<"**\n**Sorted Data ";

while(head != NULL)

{

cout<<".."<<head->data;

head=head->next;

}

return 0;

}

# 117.**C++ Program to Implement Sorting of Less than 100 Numbers in O(n) Complexity**

**Problem Description**

1. We can implement the task best using Radix sort.

2. In this algorithm sorting of data is done from least significant digit to most significant digit.

3. Here we need 10 different spaces labeled 0 to 9.

4. Assume we have ‘n’ number of inputs.

5. Let ‘d’ be the maximum number of digit the input data has, here d can have values either 1 or 2 or 3.

6. The time complexity for radix sort is O(n\*3) that is O(n).

7. Radix sort solves the problem of card sorting by sorting on the least significant digit first.

**Problem Solution**

1. Get the maximum value from the input array which has ‘d’ digits.

2. Starting from least significant digit, sort the data.

3. Take this data as input for next significant digit.

4. Run the iteration till d digit.

5. Display the result.

6. Exit.

#include <iostream>

using namespace std;

// Get maximum value from array.

int getMax(int arr[], int n)

{

int max = arr[0];

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

if (arr[i] > max)

max = arr[i];

return max;

}

// Count sort of arr[].

void countSort(int arr[], int n, int exp)

{

// Count[i] array will be counting the number of array values having that 'i' digit at their (exp)th place.

int output[n], i, count[10] = {0};

// Count the number of times each digit occurred at (exp)th place in every input.

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

count[(arr[i] / exp) % 10]++;

// Calculating their cumulative count.

for (i = 1; i < 10; i++)

count[i] += count[i-1];

// Inserting values according to the digit '(arr[i] / exp) % 10' fetched into count[(arr[i] / exp) % 10].

for (i = n - 1; i >= 0; i--)

{

output[count[(arr[i] / exp) % 10] - 1] = arr[i];

count[(arr[i] / exp) % 10]--;

}

// Assigning the result to the arr pointer of main().

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

arr[i] = output[i];

}

// Sort arr[] of size n using Radix Sort.

void radixsort(int arr[], int n)

{

int exp, m;

m = getMax(arr, n);

// Calling countSort() for digit at (exp)th place in every input.

for (exp = 1; m/exp > 0; exp \*= 10)

countSort(arr, n, exp);

}

int main()

{

int n, i;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>n;

int arr[n];

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

{

cout<<"Enter element "<<i+1<<": ";

cin>>arr[i];

}

radixsort(arr, n);

// Printing the sorted data.

cout<<"**\n**Sorted Data ";

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

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

return 0;

}

**Program Explanation**

1. Take input of data.

2. Get the maximum of input data.

3. Run the countSort() till (m/exp) > 0.

4. Sort the data on the basis of the digit at (exp)th place.

5. Assign the sorted data back to arr[] array.

6. Check the condition in step 3.

7. If false, print the sorted output.

8. Exit.

# 118.**C++ Program to Implement Merge Sort using Linked List**

**Problem Description**

1. Merge-sort is based on an algorithmic design pattern called divide-and-conquer.

2. It forms tree structure.

3. The height of the tree will be log(n).

4. we merge n element at every level of the tree.

5. The time complexity of this algorithm is O(n\*log(n)).

**Problem Solution**

1. Split the data into two equal half until we get at most one element in both half.

2. Merge Both into one making sure the resulting sequence is sorted.

3. Recursively split them and merge on the basis of constraint given in step 1.

4. Display the result.

5. Exit.

#include<iostream>

using namespace std;

// A structure representing a node of a linked list.

struct node

{

int data;

node \*next;

};

// A function creating a new node.

node\* NewNode(int d)

{

struct node \*temp = new node;

temp->data = d;

temp->next = NULL;

// Returning temp as the new node.

return temp;

}

// A function adding the given data at the end of the linked list.

node\* AddToList(node \*tail, int data)

{

struct node \*newnode;

newnode = NewNode(data);

if(tail == NULL)

{

tail = newnode;

}

// If tail is not null assign newnode to the next of tail.

else

{

tail->next = newnode;

// Shift tail pointer to the added node.

tail = tail->next;

}

return tail;

}

node\* Merge(node\* h1, node\* h2)

{

node \*t1 = new node;

node \*t2 = new node;

node \*temp = new node;

// Return if the first list is empty.

if(h1 == NULL)

return h2;

// Return if the Second list is empty.

if(h2 == NULL)

return h1;

t1 = h1;

// A loop to traverse the second list, to merge the nodes to h1 in sorted way.

while (h2 != NULL)

{

// Taking head node of second list as t2.

t2 = h2;

// Shifting second list head to the next.

h2 = h2->next;

t2->next = NULL;

// If the data value is lesser than the head of first list add that node at the beginning.

if(h1->data > t2->data)

{

t2->next = h1;

h1 = t2;

t1 = h1;

continue;

}

// Traverse the first list.

flag:

if(t1->next == NULL)

{

t1->next = t2;

t1 = t1->next;

}

// Traverse first list until t2->data more than node's data.

else if((t1->next)->data <= t2->data)

{

t1 = t1->next;

goto flag;

}

else

{

// Insert the node as t2->data is lesser than the next node.

temp = t1->next;

t1->next = t2;

t2->next = temp;

}

}

// Return the head of new sorted list.

return h1;

}

// A function implementing Merge Sort on linked list using reference.

void MergeSort(node \*\*head)

{

node \*first = new node;

node \*second = new node;

node \*temp = new node;

first = \*head;

temp = \*head;

// Return if list have less than two nodes.

if(first == NULL || first->next == NULL)

{

return;

}

else

{

// Break the list into two half as first and second as head of list.

while(first->next != NULL)

{

first = first->next;

if(first->next != NULL)

{

temp = temp->next;

first = first->next;

}

}

second = temp->next;

temp->next = NULL;

first = \*head;

}

// Implementing divide and conquer approach.

MergeSort(&first);

MergeSort(&second);

// Merge the two part of the list into a sorted one.

\*head = Merge(first, second);

}

int main()

{

int n, i, num;

struct node \*head = new node;

struct node \*tail = new node;

head = NULL;

tail = NULL;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>n;

// Create linked list.

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

{

cout<<"Enter element "<<i+1<<": ";

cin>>num;

tail = AddToList(tail, num);

if(head == NULL)

head = tail;

}

// Send reference of head into MergeSort().

MergeSort(&head);

// Printing the sorted data.

cout<<"**\n**Sorted Data ";

while(head != NULL)

{

cout<<".."<<head->data;

head=head->next;

}

return 0;

}

**Program Explanation**

1. Take input of data and create a linked list.

2. Call MergeSort() function with reference of head pointer in the argument list.

3. Recursively split the linked into two equal parts.

4. Split them until we get at most one element in both halves.

5. Combine lists into one linked list by invoking Merge() with the head of both the halves in the argument list.

6. Inside Merge(), take each of the nodes of the second list and insert it into the first list, according to the data part of the node.

7. Assign the head of the merged list to the head in MergeSort().

8. Return to main and display the result.

9. Exit.

# 119.**C++ Program to Implement Counting Sort**

**Problem Description**

1. Counter sort implements on a given finite range (k) of the integers.

2. It counts the occurrence of each element.

3. Since it maintains the counter of each integer in the range space complexity is O(k).

4. The time complexity is O(n+k).

**Problem Solution**

1. count the number of occurrences of each element.

2. Store it in the array of size same as the range of data input.

3. Use the element value to refer the counter index.

4. Display the result.

5. Exit.

#include<iostream>

using namespace std;

// A function implementing Counter sort.

void CounterSort(int a[], int n, int r, int lower)

{

int i, j = 0, counter[r] = {0};

// Counting the number occurrence of each element.

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

counter[a[i]-lower]++;

i=0;

// placing the elements back into array.

while(i < r)

{

flag:

a[j] = lower+i;

j++;

counter[i]--;

// place the same element until its counter is zero.

if(counter[i] > 0)

goto flag;

i++;

}

}

int main()

{

int n, i, range, ulimit, llimit;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>n;

cout<<"**\n**Enter the lower and upper limit of the data to be entered: ";

cin>>llimit>>ulimit;

// Range of the input data.

range = ulimit-llimit+1;

int arr[n];

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

{

cout<<"Enter element "<<i+1<<": ";

cin>>arr[i];

}

CounterSort(arr, n, range, llimit);

// Printing the sorted data.

cout<<"**\n**Sorted Data ";

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

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

return 0;

}

# 120.**C++ Program to Perform the Shaker Sort**

**Problem Description**

1. Shaker sort, unlike bubble sort, orders the array in both directions.

2. The worst case time complexity is O(n^2).

**Problem Solution**

1. In each iteration, sorting is done in two parts.

2. Firstly set the highest value to the highest index and decrement the index.

3. Then lowest value to the lowest index and increment the index.

4. Display the result.

5. Exit.

#include<iostream>

using namespace std;

// A function to swap values using call by reference.

void swap(int \*a, int \*b)

{

int temp;

temp = \*a;

\*a = \*b;

\*b = temp;

}

// A function implementing shaker sort.

void ShakerSort(int a[], int n)

{

int i, j, k;

for(i = 0; i < n;)

{

// First phase for ascending highest value to the highest unsorted index.

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

{

if(a[j] < a[j-1])

swap(&a[j], &a[j-1]);

}

// Decrementing highest index.

n--;

// Second phase for descending lowest value to the lowest unsorted index.

for(k = n-1; k > i; k--)

{

if(a[k] < a[k-1])

swap(&a[k], &a[k-1]);

}

// Incrementing lowest index.

i++;

}

}

int main()

{

int n, i;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>n;

int arr[n];

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

{

cout<<"Enter element "<<i+1<<": ";

cin>>arr[i];

}

ShakerSort(arr, n);

// Printing the sorted data.

cout<<"**\n**Sorted Data ";

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

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

return 0;

}

# 121.**C++ Program to Perform Stooge Sort**

**Problem Description**

1. Stooge sort is a recursive sorting algorithm.

2. It divides the array into two overlapping parts, 2/3 each.

3. Sort the array in three steps by sorting I then II and again I part.

4. It is fairly inefficient algorithm with worst case time complexity O(n^2.7095).

**Problem Solution**

1. Recursively divide the divide the array into two parts of size 2/3 of array length.

2. Sort the first part.

3. Sort second part.

4. Again sort the first part.

5. Display the result.

6. Exit.

**Program/Source Code**

C++ program to implement Shaker Sort.

This program is successfully run on Dev-C++ using TDM-GCC 4.9.2 MinGW compiler on a Windows system.

#include<iostream>

using namespace std;

// A function implementing stooge sort.

void StoogeSort(int a[],int start, int end)

{

int temp;

// Further breaking the array if the Subpart's length is more than 2.

if(end-start+1 > 2)

{

temp = (end-start+1)/3;

StoogeSort(a, start, end-temp);

StoogeSort(a, start+temp, end);

StoogeSort(a, start, end-temp);

}

// swapping the element at start and end.

if(a[end] < a[start])

{

temp = a[start];

a[start] = a[end];

a[end] = temp;

}

}

int main()

{

int n, i;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>n;

int arr[n];

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

{

cout<<"Enter element "<<i+1<<": ";

cin>>arr[i];

}

StoogeSort(arr, 0, n-1);

// Printing the sorted data.

cout<<"**\n**Sorted Data ";

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

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

return 0;

}

# 122.**C++ Program to Count Inversion in an Array**

**Problem Description**

1. The number of switches required to make an array sorted is termed as inversion count.

2. Its value varies with the sorting algorithms.

3. The time complexity is O(n^2).

**Problem Solution**

1. Compare the values of the element with each other.

2. Increment the counter if the value at lower index is higher.

3. Display the result.

4. Exit.

**Program/Source Code**

C++ program to count the inversion in an array.

This program is successfully run on Dev-C++ using TDM-GCC 4.9.2 MinGW compiler on a Windows system.

#include<iostream>

using namespace std;

int CountInversion(int a[], int n)

{

int i, j, count = 0;

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

{

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

if(a[i] > a[j])

count++;

}

return count;

}

int main()

{

int n, i;

cout<<"**\n**Enter the number of data element: ";

cin>>n;

int arr[n];

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

{

cout<<"Enter element "<<i+1<<": ";

cin>>arr[i];

}

// Printing the number of inversion in the array.

cout<<"**\n**The number of inversion in the array: "<<CountInversion(arr, n);

return 0;

}

**Program Explanation**

1. Take input of data.

2. Call CountInversion() function with ‘arr’ the array of data and ‘n’ the number of values, in the argument list.

3. Using nested loops compare all the element with each other.

4. Increment the counter if a[i] > a[j] where i < j. 5. Return to main and display the result. 6. Exit.

Case 1:

Enter the number of data element: 10

Enter element 1: 9

Enter element 2: 3

Enter element 3: 2

Enter element 4: 6

Enter element 5: 8

Enter element 6: 4

Enter element 7: 5

Enter element 8: 7

Enter element 9: 0

Enter element 10: 1

The number of inversion in the array: 29

# 123.**C++ Program to Compare Binary and Sequential Search**

**Problem Description**

1. Implement both binary and sequential search.

2. The time complexity of Binary search is O(log(n)).

3. The time complexity of Linear search is O(n).

**Problem Solution**

1. Implement the binary search and count the number of iteration to compute the result.

2. Implement the linear search and count the number of iteration to compute the result.

3. Compare the number of iteration and show the better algorithm.

4. Exit.

**Program/Source Code**

C++ program to compare Binary and Sequential Search.

This program is successfully run on Dev-C++ using TDM-GCC 4.9.2 MinGW compiler on a Windows system.

#include<iostream>

using namespace std;

// A function implementing Binary search on a sorted array.

int BinarySearch(int a[], int start, int end, int item, int iter)

{

int i, mid;

// Every time this function called, counted as a iteration of binary search.

cout<<"**\n**iteration "<<iter+1;

iter++;

// Assigning middle of the array.

mid = start + (end-start+1)/2;

// If value is less than value at start index more than end index then item is not in the array.

if(item > a[end] || item < a[start] || mid == end)

{

cout<<"**\n**Not found";

return iter;

}

// Return if item found at mid index.

else if(item == a[mid])

{

cout<<"**\n** item found at "<<mid<<" index.";

return iter;

}

// Return if item found at start index.

else if(item == a[start])

{

cout<<"**\n** item found at "<<start<<" index.";

return iter;

}

// Return if item found at end index.

else if(item == a[end])

{

cout<<"**\n** item found at "<<end<<" index.";

return iter;

}

// According to the item value choose the partion to proceed further.

else if(item > a[mid])

BinarySearch(a, mid, 19, item, iter);

else

BinarySearch(a, start, mid, item, iter);

}

// A function implementing Binary search on a sorted array.

int LinearSearch(int a[], int n, int item)

{

int i;

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

{

cout<<"**\n**iteration "<<i+1;

// Directly comparing the item with the array element sequentially.

if(a[i] == item)

{

cout<<"**\n** item found at "<<i<<" index.";

// Returning the number of iteration taken place.

return i+1;

}

}

cout<<"**\n**Not found";

}

int main()

{

int n, i, Biter, Liter, a[20]={1, 9, 18, 24, 27, 35, 38, 41, 49, 53, 55, 66, 67, 72, 75, 77, 81, 89, 90, 97};

cout<<"**\n**Enter the element to be searched: ";

cin>>n;

cout<<"**\n\n\t\t\t**Binary Search :";

cout<<"**\n\t\t\t**\*\*\*\*\*\*\*\*\*\*\*\*\*";

Biter = BinarySearch(a, 0, 19, n, 0);

cout<<"**\n\n\t\t\t**Linear Search :";

cout<<"**\n\t\t\t**\*\*\*\*\*\*\*\*\*\*\*\*\*";

Liter = LinearSearch(a, 20, n);

// Comparing the number of iteration and printing the better approach for this search.

if(Liter > Biter)

cout<<"**\n\n**Binary search is better for this search.";

else if(Liter < Biter)

cout<<"**\n\n**Linear search is better for this search.";

else

cout<<"**\n\n**Both are equally efficient for this search.";

return 0;

}

**Program Explanation**

1. Assign the data to the array in a sorted manner.

2. Call BinarySearch() function with ‘arr’ the array of data and ‘n’ the number of values, start and end index, iteration count and element to be searched in the argument list.

3. Increment the iteration counter and compare the item value with the a[mid].

4. If item < a[mid] choose first half otherwise second half to proceed further.

5. Return iteration value to main.

6. Call the LinearSearch() function with ‘arr’ the array of data and ‘n’ the number of values, iteration count and element to be searched in the argument list.

7. Sequentially search for the item in the array.

8. Return iteration value to main.

9. Compare the number of iteration and show the better algorithm.

10. Exit.

# 124.**C++ Program to Implement a Binary Search Algorithm for a Specific Search Sequence**

**Problem Description**

1. Implement binary search to find the existence of a search sequence in an array.

2. The time complexity of Binary search is O(log(n)).

**Problem Solution**

1. Implement the binary search to find the first value of search sequence.

2. If it is there, then compare the remaining item sequentially.

3. Otherwise, the sequence is not there.

4. Exit.

**Program/Source Code**

C++ program to compare Binary and Sequential Search.

This program is successfully run on Dev-C++ using TDM-GCC 4.9.2 MinGW compiler on a Windows system.

#include<iostream>

using namespace std;

// A function implementing Binary search on a sorted array.

int BinarySearch(int a[], int start, int end, int item, int iter)

{

int i, mid;

iter++;

// Assigning middle of the array.

mid = start + (end-start+1)/2;

// If value is less than value at start index more than end index then item is not in the array.

if(item > a[end] || item < a[start] || mid == end)

{

cout<<"**\n**Not found";

return -1;

}

// Return the mid index.

else if(item == a[mid])

{

return mid;

}

// Return the start index.

else if(item == a[start])

{

return start;

}

// Return the end index.

else if(item == a[end])

{

return end;

}

// According to the item value choose the partion to proceed further.

else if(item > a[mid])

BinarySearch(a, mid, 19, item, iter);

else

BinarySearch(a, start, mid, item, iter);

}

int main()

{

int n, i, flag=0, Bindex, a[20]={1, 9, 18, 24, 27, 35, 38, 41, 49, 53, 55, 66, 67, 72, 75, 77, 81, 89, 90, 97};

cout<<"**\n**Enter the number of element in the search sequence: ";

cin>>n;

int s[n];

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

cin>>s[i];

// Get the index of the first value in the search sequence found in data set array.

Bindex = BinarySearch(a, 0, 19, s[0], 0);

if(Bindex == -1)

{

// if return index is -1 then not found.

cout<<"**\n**Not found.";

return 0;

}

else

{

// If first value found then check for others sequentially.

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

if(a[i] != s[i-Bindex])

flag = 5;

if(flag == 5)

cout<<"**\n**Not found.";

else

cout<<"**\n**Sequence found between index "<<Bindex<<" and "<<Bindex+n<<".";

}

return 0;

}

**Program Explanation**

1. Assign the data to the array in a sorted manner.

2. Call BinarySearch() function with ‘arr’ the array of data and ‘n’ the number of values, start and end index, iteration count and s[0] be the element to be searched in the argument list.

3. Increment the iteration counter and compare the item value with the a[mid].

4. If item < a[mid] choose first half otherwise second half to proceed further. 5. Return index value to main. 6. In main(), sequentially compare the remaining items of search sequence to next items in the array. 7. Print the index range of the sequence found. 8. Exit.

# 125.**C++ Program to Implement Interpolation Search Algorithm**

**Problem Description**

1. It is a better approach for uniform data.

2. Implement binary search using interpolation approach.

3. The time complexity is O(log(n)).

**Problem Solution**

1. Implement the binary search on a sorted array.

2. Calculate mid value using interpolation formula.

3. Exit.

**Program Explanation**

1. Assign the data to the array in a sorted manner.

2. Take the input of the element to be searched.

3. Call InterpolationSearch() function.

4. Calculate mid value using ‘start+((item-a[start])\*(end-start))/(a[end]-a[start])’ expression.

5. If the item is equal to the value at mid index, print result and return to main.

6. If the item is lesser than the value at mid index, proceed with the left sub-array.

7. If the item is more than the value at mid index, proceed with the right sub-array.

8. If the calculated mid value is equal to either start or end then the item is not found in the array.

9. Return to main and ask for user’s choice to search more.

10. Exit.

#include<iostream>

using namespace std;

// A function implementing Interpolation search on a sorted array.

void InterpolationSearch(int a[], int start, int end, int item)

{

int mid;

// Assigning middle of the array.

mid = start+((item-a[start])\*(end-start))/(a[end]-a[start]);

if(item == a[mid])

{

cout<<"**\n**Item found at "<<mid<<" index.";

return;

}

// Return if item found at start index.

else if(item == a[start])

{

cout<<"**\n**Item found at "<<start<<" index.";

return;

}

// Return if item found at end index.

else if(item == a[end])

{

cout<<"**\n**Item found at "<<end<<" index.";

return;

}

else if(mid == start || mid == end)

{

cout<<"**\n**Element not found";

return;

}

// According to the item value choose the partition to proceed further.

else if(item > a[mid])

InterpolationSearch(a, mid, 19, item);

else

InterpolationSearch(a, start, mid, item);

}

int main()

{

int n, i, biter, a[20]={1, 9, 18, 24, 27, 35, 38, 41, 49, 53, 55, 66, 67, 72, 75, 77, 81, 89, 90, 97};

char ch;

up:

cout<<"**\n**Enter the Element to be searched: ";

cin>>n;

// Implement Interpolation search.

InterpolationSearch(a, 0, 19, n);

// Ask user to enter choice for further searching.

cout<<"**\n\n\t**Do you want to search more...enter choice(y/n)?";

cin>>ch;

if(ch == 'y' || ch == 'Y')

goto up;

return 0;

}

# 126.**C++ Program to Search Sorted Sequence Using Divide and Conquer with the Aid of Fibonacci Numbers**

**Problem Description**

1. It implements a Divide and Conquer approach using Fibonacci numbers.

2. Using Fibonacci numbers we calculate mid of data array to search the data item.

3. The time complexity is O(log(n)).

**Problem Solution**

1. Calculate the mid of the array.

2. Divide the array into two sub-array.

3. Compare the item with mid element and proceed accordingly.

4. Exit.

**Program Explanation**

1. Assign the data to the array in a sorted manner.

2. Take input of the element to be searched.

3. Call FibonacciSearch() function.

4. Calculate the mid value using ‘start+fab[index-2]’ expression.

5. If the item is equal to the value at mid index, print result and return to main.

6. If the item is lesser than the value at mid index, proceed with the left sub-array.

7. If the item is more than the value at mid index, proceed with the right sub-array.

8. If the calculated mid value is equal to either start or end then the item is not found in the array.

9. Return to main and ask for user’s choice to search more.

10. Exit.

#include<iostream>

using namespace std;

void FibonacciSearch(int \*a, int start, int end, int \*fab, int index, int item)

{

int i, mid;

// Assigning middle of the array using Fibonacci element.

mid = start+fab[index-2];

// Return if item found at mid index.

if(item == a[mid])

{

cout<<"**\n** item found at "<<mid<<" index.";

return;

}

// Return if item found at start index.

else if(item == a[start])

{

cout<<"**\n** item found at "<<start<<" index.";

return;

}

// Return if item found at end index.

else if(item == a[end])

{

cout<<"**\n** item found at "<<end<<" index.";

return;

}

// If mid becomes start or end of the sub-array then element not found.

else if(mid == start || mid == end)

{

cout<<"**\n**Element not found";

return;

}

// According to the item value choose the partion to proceed further.

else if(item > a[mid])

FibonacciSearch(a, mid, end, fab, index-1, item);

else

FibonacciSearch(a, start, mid, fab, index-2, item);

}

main()

{

int n, i, biter, fab[20], a[20]={1, 9, 18, 24, 27, 35, 38, 41, 49, 53, 55, 66, 67, 72, 75, 77, 81, 89, 90, 97};

char ch;

fab[0] = 0;

fab[1] = 1;

i = 1;

while(fab[i] < 20)

{

i++;

fab[i] = fab[i-1]+fab[i-2];

}

up:

cout<<"**\n**Enter the Element to be searched: ";

cin>>n;

// Implement Fibonacci search.

FibonacciSearch(a, 0, 19, fab, i, n);

// Ask user to enter choice for further searching.

cout<<"**\n\n\t**Do you want to search more...enter choice(y/n)?";

cin>>ch;

if(ch == 'y' || ch == 'Y')

goto up;

return 0;

}

# 127.**C++ Program to Perform Uniform Binary Search**

**Problem Description**

1. Implement binary search using a lookup table.

2. It is an improvement in binary search since table lookup is faster than a shift and addition.

3. The time complexity is O(log(n)).

**Problem Solution**

1. Implement the binary search on a sorted array.

2. For the mid index value of any of the sub-array, instead of calculating refer lookup table.

3. Exit.

**Program Explanation**

1. Assign the data to the array in a sorted manner.

2. Calculate the maximum length of lookup array and declare a new array ‘del’.

3. Assign the values to the lookup array as n/2, n/4 and so on till ‘0’, where n is the length of the data array.

4. Call UniBinarySearch() function.

5. Assign mid to the value at ‘0’ index of ‘del’ array and compare key to the value at mid index.

6. If the key is equal then return the index value to the main.

7. If index value in ‘del’ is zero then the element is not there, return -1 to main.

8. If lesser, subtract next value stored in ‘del’ array and shift the pointer to next value in ‘del’ array.

9. If greater, add next value stored in ‘del’ array and shift the pointer to next value in ‘del’ array.

10. print the index value returned by the function and ask for user’s choice to search more.

11. Exit.

#include<iostream>

using namespace std;

// A function to create lookup array.

void MakeDelta(int \*delta, int N)

{

int power = 1, i = 0;

do

{

int half = power;

power \*= 2;

delta[i] = (N+half)/power;

i++;

}while (delta[i-1] != 0);

}

// A function implementing Uniform Binary search.

int UniBinarySearch(int \*a, int \*del, int key)

{

int i = del[0]-1, d = 0;

flag:

// If item found at mid index return to main.

if (key == a[i])

return i;

// If lookup table vlaue is 0 then no more sub-part of array remain to search hence element not found.

else if (del[d] == 0)

return -1;

else

{

// Shift to left subpart using lookup array 'del'.

if (key < a[i])

{

i -= del[++d];

goto flag;

}

// Shift to right subpart using lookup array 'del'.

else

{

i += del[++d];

goto flag;

}

}

}

int main(void)

{

int i, n = 20,d = 0, pow = 1, index;

char ch;

int a[20] = {1, 2, 5, 8, 10, 12, 14, 15, 17, 19, 43, 65, 67, 71, 75, 79, 83, 90, 94, 99};

// Determine the size of delta array.

while(pow <= n)

{

pow \*=2;

d++;

}

int del[d];

// Create lookup array.

MakeDelta(del, n);

up:

cout<<"**\n**Enter the Element to be searched: ";

cin>>n;

// Implement Uniform Binary search and get index value.

index = UniBinarySearch(a, del, n);

if(index == -1)

cout<<"**\n**Item not found";

else

cout<<"**\n**Item "<<n<<" found at "<<index+1<<" position";

// Ask user to enter choice for further searching.

cout<<"**\n\n\t**Do you want to search more...enter choice(y/n)?";

cin>>ch;

if(ch == 'y' || ch == 'Y')

goto up;

return 0;

}

# 128.**C++ Program to Find kth Largest Element in a Sequence**

**Problem Description**

1. Extract the Kth largest element from a sequence.

2. By selectively sorting the array to get Kth largest element it has the complexity of O(k\*n).

2. We can improve the time complexity by approaching the problem using max-heap.

3. The time complexity is O(n+k\*log(n)).

**Problem Solution**

1. Approach the solution using max heap technique.

2. Build the max heap k times.

3. In each iteration pop max of the heap out of the sequence.

4. Display the Kth max of the heap.

5. Exit.

**Program Explanation**

1. Take input of data.

2. Call Build\_MaxHeap() function with ‘arr’ the array of data and ‘n-1’ the number of values, in the argument list.

3. Send the max of the heap to the end of the sequence.

4. Heapify the remaining sequence.

5. Repeat the process for ‘k’ times.

6. Display the final state of the array.

7. Display the max from the heap extracted from kth iteration as a result.

8. Exit.

#include <iostream>

using namespace std;

// A function to heapify the array.

void MaxHeapify(int a[], int i, int n)

{

int j, temp;

temp = a[i];

j = 2\*i;

while (j <= n)

{

if (j < n && a[j+1] > a[j])

j = j+1;

// Break if parent value is already greater than child value.

if (temp > a[j])

break;

// Switching value with the parent node if temp < a[j].

else if (temp <= a[j])

{

a[j/2] = a[j];

j = 2\*j;

}

}

a[j/2] = temp;

return;

}

// A function to build max heap from the initial array by checking all non-leaf node to satisfy the condition.

void Build\_MaxHeap(int a[], int n)

{

int i;

for(i = n/2; i >= 1; i--)

MaxHeapify(a, i, n);

}

int main()

{

int n, i, temp, k;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>n;

n++;

int arr[n];

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

{

cout<<"Enter element "<<i<<": ";

cin>>arr[i];

}

cout<<"**\n**Enter the k value: ";

cin>>k;

Build\_MaxHeap(arr, n-1);

// Build max-heap k times, extract the maximum and store it in the end of the array.

for(i = n-1; i >= n-k; i--)

{

temp = arr[i];

arr[i] = arr[1];

arr[1] = temp;

MaxHeapify(arr, 1, i - 1);

}

// Printing the array state.

cout<<"**\n**After max-heapify the given array "<<k<<" times the array state is: ";

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

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

// The Kth largest element.

cout<<"**\n\n**The "<<k<<"th largest element is: "<<arr[n-k];

return 0;

}

# 129.**C++ Program to Find Second Smallest of n Elements with Given Complexity Constraint**

**Problem Description**

1. Traverse the data array linearly and find the second smallest element.

2. The time complexity of the algorithm is O(n).

**Problem Solution**

1. Linearly traverse the data array.

2. Keep track of the smallest number.

3. Simultaneously keep updating the second smallest number also.

4. Exit.

**Program Explanation**

1. Assign the data to the array.

2. Call SecondSmallest() function with ‘arr’ the array of data and number of elements, in the argument list.

3. Assign variable ‘s’ and ‘ss’ as the first element of the array.

4. Traverse the array.

5. Check if the value of s is larger than current array element, update the variable ‘s’.

6. Otherwise, check if the value of ss is larger than current array element, update the variable ‘ss’.

7. Return ss as the second smallest element, to the main().

8. Print the result.

9. Exit.

#include<iostream>

using namespace std;

// A function to calculate second.

int SecondSmallest(int \*a, int n)

{

int s, ss, i;

// A variable 's' keeping track of smallest number.

s = a[0];

// A variable 'ss' keeping track of second smallest number.

ss = a[0];

// Traverse the data array.

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

{

// If array element is lesser than current 's' value then update.

if(s > a[i])

{

ss = s;

s = a[i];

}

// Otherwise the number can be second smallest number so check for the condition and update 'ss'.

else if(ss > a[i])

{

ss = a[i];

}

}

// Return second smallest number.

return ss;

}

int main()

{

int n, i;

cout<<"Enter the number of element in dataset: ";

cin>>n;

int a[n];

// Take input.

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

{

cout<<"Enter "<<i+1<<"th element: ";

cin>>a[i];

}

// Print the result.

cout<<"**\n\n**The second Smallest number of the given data array is: "<<SecondSmallest(a, n);

return 0;

}

# 130.**C++ Program to Implement Quick Sort Using Randomization**

**Problem Description**

1. Quick sort is based on an algorithmic design pattern called divide-and-conquer.

2. Unlike Merge Sort it doesn’t require extra memory space.

3. The average time complexity is O(n\*log(n)) but the worst case complexity is O(n^2).

4. To reduce the chances of the worst case we have implemented Quicksort using randomization.

5. Here we will be selecting the pivot randomly.

**Problem Solution**

1. Randomly select pivot value from the given subpart of the array.

2. Partition that subpart so that the values left of the pivot are smaller and to the right are greater from the pivot.

3. Consider both as new sub-array and repeat step 1 until only one element left in subpart.

4. Display the result.

5. Exit.

**Program Explanation**

1. Take input of data.

2. Call QuickSort() function.

3. Through RandomPivotPartition(), select pivot randomly.

4. Create a partition of the array on the basis of the pivot.

5. Recursively insert the partitions into QuickSort() and repeat step 2 until low is lesser than high.

6. Return to main and display the result.

7. Exit.

#include<iostream>

#include<cstdlib>

using namespace std;

// Swapping two values.

void swap(int \*a, int \*b)

{

int temp;

temp = \*a;

\*a = \*b;

\*b = temp;

}

// Partitioning the array on the basis of values at high as pivot value.

int Partition(int a[], int low, int high)

{

int pivot, index, i;

index = low;

pivot = high;

// Getting index of pivot.

for(i=low; i < high; i++)

{

if(a[i] < a[pivot])

{

swap(&a[i], &a[index]);

index++;

}

}

// Swapping value at high and at the index obtained.

swap(&a[pivot], &a[index]);

return index;

}

// Random selection of pivot.

int RandomPivotPartition(int a[], int low, int high)

{

int pvt, n, temp;

n = rand();

// Randomizing the pivot value in the given subpart of array.

pvt = low + n%(high-low+1);

// Swapping pvt value from high, so pvt value will be taken as pivot while partitioning.

swap(&a[high], &a[pvt]);

return Partition(a, low, high);

}

// Implementing QuickSort algorithm.

int QuickSort(int a[], int low, int high)

{

int pindex;

if(low < high)

{

// Partitioning array using randomized pivot.

pindex = RandomPivotPartition(a, low, high);

// Recursively implementing QuickSort.

QuickSort(a, low, pindex-1);

QuickSort(a, pindex+1, high);

}

return 0;

}

int main()

{

int n, i;

cout<<"**\n**Enter the number of data element to be sorted: ";

cin>>n;

int arr[n];

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

{

cout<<"Enter element "<<i+1<<": ";

cin>>arr[i];

}

QuickSort(arr, 0, n-1);

// Printing the sorted data.

cout<<"**\n**Sorted Data ";

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

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

return 0;

}

# 131.**C++ Program to Find k Numbers Closest to Median of S, Where S is a Set of n Numbers**

**Problem Description**

1. We need to find k numbers which have a minimum difference with the median of the data set.

2. It includes sorting using quick sort and then printing k numbers as a result.

3. The time complexity will be O(n\*log(n)+k).

**Problem Solution**

1. Take input of the data.

2. Sort the data using the quick-sort algorithm.

3. Starting from the median index, using two variable moves towards the end of the array.

4. compare each value to the median and print those which are closer to it.

5. Repeat this k time printing the k numbers closest to the median.

6. Exit.

**Program Explanation**

1. Take input of the data.

2. Call QuickSort() function to sort the data.

3. Check if the number of the data element are odd then assign the middle index to low and the index next to it to high and calculate median.

4. Run a loop for k times and print the element which his closer to the median.

5. Otherwise, the number of the data element are even then the median will be an average of two middle values so, it can be fractional so use floating variable.

6. Run a loop for k times and print the element which his closer to the median.

7. Exit.

#include<iostream>

using namespace std;

// Swapping two values.

void swap(int \*a, int \*b)

{

int temp;

temp = \*a;

\*a = \*b;

\*b = temp;

}

// Partitioning the array on the basis of values at high as pivot value.

int Partition(int a[], int low, int high)

{

int pivot, index, i;

index = low;

pivot = high;

// Getting index of pivot.

for(i=low; i < high; i++)

{

if(a[i] < a[pivot])

{

swap(&a[i], &a[index]);

index++;

}

}

// Swapping value at high and at the index obtained.

swap(&a[pivot], &a[index]);

return index;

}

// Implementing QuickSort algorithm.

int QuickSort(int a[], int low, int high)

{

int pindex;

if(low < high)

{

// Partitioning array using randomized pivot.

pindex = Partition(a, low, high);

// Recursively implementing QuickSort.

QuickSort(a, low, pindex-1);

QuickSort(a, pindex+1, high);

}

return 0;

}

int main()

{

int n, i, high, low, k;

double d1,d2, median;

cout<<"Enter the number of element in dataset: ";

cin>>n;

int a[n];

// Taking input of the data set.

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

{

cout<<"**\n**Enter "<<i+1<<"th element: ";

cin>>a[i];

}

cout<<"**\n**Enter the number of element nearest to the median required: ";

cin>>k;

// Sort the data.

QuickSort(a, 0, n-1);

//Print the result.

cout<<"**\t**The K element nearest to the median are: ";

// Check the number of data element to be even or odd and proceed accordingly.

if(n%2 == 1)

{

median = a[n/2];

high = n/2+1;

low = n/2;

// Loop to search for the next element generating minimum difference from median.

while(k > 0)

{

// If difference from the first half element is minimum.

if((median-a[low] <= a[high]-median) && low >= 0)

{

cout<<" "<<a[low];

low--;

k--;

}

// If difference from the Second half element is minimum.

else if((median-a[low] > a[high]-median) && high <= n-1)

{

cout<<" "<<a[high];

high++;

k--;

}

}

}

else

{

// Need to use floating variable since the median can be in the fractional form.

d1 = a[n/2];

d2 = a[n/2-1];

median = (d1+d2)/2;

high = n/2;

low = n/2-1;

while(k > 0)

{

d1 = a[low];

d2 = a[high];

// If difference from the first half element is minimum.

if((median-d2 <= d1-median) && low >= 0)

{

cout<<" "<<a[low];

low--;

k--;

}

// If difference from the Second half element is minimum.

else if((median-d2 > d1-median) && high <= n-1)

{

cout<<" "<<a[high];

high++;

k--;

}

}

}

return 0;

}

# 132.**C++ Program to Find Median of Two Sorted Arrays**

**Problem Description**

1. We need to find combined median of two different data set.

2. It includes sorting of both arrays using quick sort and then printing the median by simultaneously traversing both arrays.

3. The time complexity will be O(n\*log(n)+m\*log(m)+m+n).

**Problem Solution**

1. Take input of both the data set of length m and n respectively.

2. Sort the data sets using the quick-sort algorithm.

3. Calculate the median, considering both arrays as a single array.

4. Exit.

**Program Explanation**

1. Take input of the first data set in the array ‘a’ of length ‘n’ and the second data set as ‘b’ of length ‘m’.

2. Call QuickSort() function to sort both the data set.

3. Check if m+n is odd, then assign k as ‘(m+n)/2 + 1’ and skip first m+n smallest element from both arrays to reach the combined median.

4. Otherwise, m+n is even, then assign k as ‘(m+n)/2 + 1’ and skip first m+n-1 smallest element from both arrays.

5. Take the average of next two smallest elements as the combined average.

6. Print the median.

7. Exit.

#include<iostream>

using namespace std;

// Swapping two values.

void swap(int \*a, int \*b)

{

int temp;

temp = \*a;

\*a = \*b;

\*b = temp;

}

// Partitioning the array on the basis of values at high as pivot value.

int Partition(int a[], int low, int high)

{

int pivot, index, i;

index = low;

pivot = high;

// Getting index of pivot.

for(i=low; i < high; i++)

{

if(a[i] < a[pivot])

{

swap(&a[i], &a[index]);

index++;

}

}

// Swapping value at high and at the index obtained.

swap(&a[pivot], &a[index]);

return index;

}

// Implementing QuickSort algorithm.

int QuickSort(int a[], int low, int high)

{

int pindex;

if(low < high)

{

// Partitioning array using randomized pivot.

pindex = Partition(a, low, high);

// Recursively implementing QuickSort.

QuickSort(a, low, pindex-1);

QuickSort(a, pindex+1, high);

}

return 0;

}

int main()

{

int n, m, bi, ai, i, k;

double median;

cout<<"Enter the number of element in the first data set: ";

cin>>n;

int a[n];

// Take input of first sequence.

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

{

cout<<"Enter "<<i+1<<"th element: ";

cin>>a[i];

}

cout<<"**\n**Enter the number of element in the second data set: ";

cin>>m;

int b[m];

// Take input of second sequence.

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

{

cout<<"Enter "<<i+1<<"th element: ";

cin>>b[i];

}

// Sort the data of both arrays.

QuickSort(a, 0, n-1);

QuickSort(b, 0, m-1);

//Print the result.

cout<<"**\t**The Median from these data set is: ";

ai = 0;

bi = 0;

// If the m+n is odd then one median will be there otherwise average of two will be taken as median.

if((m+n)%2 == 1)

{

// K is the number of element present upto the median from the beginning of the data array.

k =(n+m)/2+1;

while(k > 0)

{

// Compare current element of array 'a' and 'b' and skip next the smaller one.

if(a[ai] <= b[bi] && ai < n)

{

k--;

// Print if we have skipped k element.

if(k == 0)

cout<<a[ai];

ai++;

}

else if(a[ai] > b[bi] && bi < m)

{

k--;

// Print if we have skipped k element.

if(k == 0)

cout<<b[bi];

bi++;

}

}

}

else

{

k = (n+m)/2+1;

while(k > 0)

{

// Compare current element of array 'a' and 'b' and skip next the smaller one.

if(a[ai] <= b[bi] && ai < n)

{

k--;

// Add the last two numbers so as we can calculate average.

if(k <= 1)

median += a[ai];

ai++;

}

else if(a[ai] > b[bi] && bi < m)

{

k--;

// Add the last two numbers so as we can calculate average.

if(k <= 1)

median += b[bi];

bi++;

}

}

// Take average.

cout<<median/2;

}

}

# 133.**C++ Program to Find ith Largest Number from a Given List Using Order-Statistic Algorithm**

**Problem Description**

1. Implements Order-Statistic tree.

2. It is an improvement in BST by adding two more key functions- rank() and select().

3. The time complexity of Order-statistic tree generation is O(n+n\*log(n)).

4. Once the tree is constructed, this algorithm takes O(log(n)) to find Kth largest number.

**Problem Solution**

1. Construct Order-Statistic tree for the given unsorted data array.

2. Using the select function get the kth largest number from the given data set.

3. Print the result.

4. Exit.

**Program Explanation**

1. Construct Order-Statistic tree for the given unsorted data array by inserting data into tree one by one.

2. Using insert(), it will create a binary search tree where the rank of each node is zero initially.

3. Traverse the tree inorder and assign the rank using a static integer variable.

4. For kth largest number use select() with the root of the tree and N-k in its argument list.

5. Inside select(), a temporary variable traverses the tree and compares N-k to the rank of the current node.

6. If found to be equal, return to main and display the result.

7. If it is greater then shift the temporary variable to the right child.

8. If it is smaller then shift the temporary variable to the left child.

9. After returning to main display the result.

10. Exit.

#include<iostream>

using namespace std;

static int count = 0;

// A structure representing a node of a tree.

struct node

{

int data;

int rank;

node \*left;

node \*right;

};

// A function creating new node of tree and assigning the data.

node\* CreateNode(int data)

{

node \*newnode = new node;

newnode->data = data;

newnode->rank = 0;

newnode->left = NULL;

newnode->right = NULL;

return newnode;

}

// A function to create binary search tree.

node\* Insert(node\* root, int data)

{

// Create node using data from argument list.

node \*temp = CreateNode(data);

node \*t = new node;

t = root;

// If root is null, assign it to the node created.

if(root == NULL)

root = temp;

else

{

// Find the position for the new node to be inserted.

while(t != NULL)

{

if(t->data < data )

{

if(t->right == NULL)

{

// If current node is NULL then insert the node.

t->right = temp;

break;

}

// Shift pointer to the left.

t = t->right;

}

else if(t->data > data)

{

if(t->left == NULL)

{

// If current node is NULL then insert the node.

t->left = temp;

break;

}

// Shift pointer to the left.

t = t->left;

}

}

}

return root;

}

// A function to assign a rank to each node of the tree.

void AssignRank(node \*root)

{

if(root->left != NULL)

AssignRank(root->left);

root->rank = count;

count++;

if(root->right != NULL)

AssignRank(root->right);

}

// A function to search Kth smallest element from the data stored in the tree.

int Select(node\* root, int k)

{

// Search for the entered rank and shift the pointer accordingly, if rank not matched.

if(root->rank == k)

return root->data;

else if(root->rank > k)

return Select(root->left, k);

else

return Select(root->right, k);

}

// A function to take an inorder traversal of the tree and print the tree data of each node.

void print(node \*root)

{

if(root->left != NULL)

print(root->left);

cout<<"**\n** data: "<<root->data<<" rank: "<<root->rank;

if(root->right != NULL)

print(root->right);

}

int main()

{

char ch;

int n, i, k, a[20]={40, 53, 95, 1, 9, 67, 72, 66, 75, 77, 18, 24, 35, 90, 38, 41, 49, 81, 27, 97};

node \*root = new node;

root = NULL;

// Construct the BST.

for(i = 0; i < 20; i++)

root = Insert(root, a[i]);

cout<<"Enter the k value: ";

cin>>k;

// Assign rank to each of the nodes of the Binary Search tree.

AssignRank(root);

// Inorder traversal of the tree and displaying the data and the rank corresponding to that data.

cout<<"**\n**Rank associated to each node:-";

print(root);

// Print the result.

cout<<"**\n\n**The kth Largest element is: "<<Select(root, 20-k);

return 0;

}

# 134.**C++ Program to Find kth Smallest Element by the Method of Partitioning the Array**

**Problem Description**

1. Implement partitioning to find the Kth smallest number from a dataset of n element.

2. The time complexity of this algorithm is O(n\*log(n)).

**Problem Solution**

1. Take the input of the data set.

2. Use the partition algorithm.

3. As we place the pivot at the (k-1)th index it will be the kth smallest number.

3. Exit.

**Program Explanation**

1. Take the input of the data set.

2. Take the input of the value of k.

3. Call Partition().

4. Inside Partition(), rearrange the array according to the pivot using CreatePartition().

5. Get the new index of the pivot as ‘pindex’ and compare it with (k-1).

6. If both the values are equal then return the value as a result to the main().

7. If pindex > k-1, recursively call Partition() for the part before the pivot value.

8. If pindex < k-1, recursively call Partition() for the part after the pivot value. 9. Inside main(), print the kth smallest element. 10. Exit.

Output:

Case 1:

Enter the number of data element: 10

Enter element 1: 9

Enter element 2: 4

Enter element 3: 0

Enter element 4: 3

Enter element 5: 7

Enter element 6: 8

Enter element 7: 1

Enter element 8: 5

Enter element 9: 6

Enter element 10: 2

Enter the k for the kth smallest element: 4

The kth smallest element: 3

#include<iostream>

using namespace std;

// Swapping two values.

void swap(int \*a, int \*b)

{

int temp;

temp = \*a;

\*a = \*b;

\*b = temp;

}

// Partitioning the array on the basis of values at high as pivot value.

int CreatePartition(int a[], int low, int high)

{

int pivot, index, i;

index = low;

pivot = high;

// Getting index of pivot.

for(i=low; i < high; i++)

{

if(a[i] < a[pivot])

{

swap(&a[i], &a[index]);

index++;

}

}

// Swapping value at high and at the index obtained.

swap(&a[pivot], &a[index]);

return index;

}

// Implementing Partition.

int Partition(int a[], int low, int high, int k)

{

int pindex;

if(low < high)

{

// Partitioning array using last element as a pivot.

// Recursively implementing partitioning in the direction to place the pivot at (k-1)th pivot.

pindex = CreatePartition(a, low, high);

if(pindex == k-1)

return k-1;

else if(pindex > k-1)

Partition(a, low, pindex-1, k);

else

Partition(a, pindex+1, high, k);

}

}

int main()

{

int n, i, k, kk;

cout<<"**\n**Enter the number of data element: ";

cin>>n;

int arr[n];

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

{

cout<<"Enter element "<<i+1<<": ";

cin>>arr[i];

}

cout<<"**\n**Enter the k for the kth smallest element: ";

cin>>k;

kk = Partition(arr, 0, n-1, k);

// Printing the result.

cout<<"**\n**The kth smallest element: "<<arr[kk];

return 0;

}