

Algorithms Handbook

Contents

1. **Introduction Topics**
 1. Sieve of Eratosthenes
 2. Primality Testing
2. **Data Structures**
 1. Heaps
 2. Segment Trees
3. **Divide and Conquer**
 1. Master Theorem
 2. Mergesort
 3. Fast Fourier Transform
 4. Problems
4. **Greedy Algorithms**
 1. Elements of Greedy Strategy
 2. Huffman Codes
 3. Set Cover Problem
 4. Problems
5. **Dynamic Programming**
 1. Analysis of methods using Fibonacci Numbers
 2. The Edit Distance Problem
 3. Chain Matrix Multiplication
 4. Knapsack
 5. Problems
6. **Graph Algorithms**
 1. Breadth First Search
 2. Depth First Search
 3. Minimum Spanning Trees
 1. Prim's Algorithm
 2. Kruskal's Algorithm
 4. Shortest Path in DAGs
 5. Dijkstra's Algorithm
 6. Floyd Warshall Algorithm
 7. Toposort
 8. Problems

Work By:

Hitesh Goel

2020115003

UG2

CHAPTER 1: INTRODUCTION TOPICS

Sieve of Eratosthenes

Sieve of Eratosthenes is a very efficient algorithms which calculates all the prime numbers between 1 to n (n is any natural number). The algorithm was invented by the Greek scientist and mathematician in 4th century B.C. He is also famous for making the first measurement of the size of the Earth.

The Algorithm:

1. Write all numbers between 2 to n.
2. Mark all the proper multiples of 2 as composite, since 2 is the only even prime number.
3. Iterate from 2 and find the next number that has not yet been marked (3). 3 is prime, and now mark all the proper multiples of 3 as composite.
4. Continuing with the algorithm, the next number will be 5, hence it is prime and now mark all its proper multiples as composite.
5. Continue till all the numbers till n have been processed.



Code

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

typedef long long int ll;

using VB = vector<bool>;

VB sieveOfEratosthenes(int n)
{
```

```

VB v(n + 1, true);
v[0] = v[1] = false;
for (int i = 2; i <= n; i++)
{
    if (v[i] && 1l(i * i) <= n)
    {
        for (int j = i * i; j <= n; j += i)
        {
            v[j] = false;
        }
    }
}

return v;
}

int main()
{
    cout << "enter n" << endl;
    int n;
    cin >> n;

    VB isprime = sieveOfEratosthenes(n);

    for (int i = 0; i < isprime.size(); i++)
    {
        if (isprime[i])
        {
            cout << i << " ";
        }
    }
    cout << endl;
}

```

Complexity: $O(n * \log(\log n))$

Examples

Example 1

Solution:

```

#include <bits/stdc++.h>
using namespace std;

int sieve[100005];

```

```

int main()
{
    int i, n, j;
    cin >> n;
    for (i = 2; i <= n + 1; i++)
    {
        if (!sieve[i])
            for (j = 2 * i; j <= n + 1; j += i)
                sieve[j] = 1;
    }

    if (n > 2)
        cout << "2" << endl;
    else
        cout << "1" << endl;

    for (i = 2; i <= n + 1; i++)
    {
        if (!sieve[i])
            cout << "1 ";
        else
            cout << "2 ";
    }

    return 0;
}

```

Practice Problems (Click on the problem link to view the problem)

The problems are sorted by difficulty.

Problem 1

Problem 2

Problem 3

Primality Testing

Now we will see various techniques to tell whether a number is prime or not. This is widely used not only in various competitive programming problems, but also has many cryptographic based applications.

Trial Division

This is a simple algorithm which uses a theorem from Discrete Structures which says that for a composite number, there must be at least one divisor which is lesser than or equal to \sqrt{n} .

Code

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

bool isPrime(int x)
{
    for (int d = 2; d * d <= x; d++)
    {
        if (x % d == 0)
            return false;
    }
    return true;
}

int main()
{
    cout << "enter n: ";
    int n;
    cin >> n;

    if(isPrime(n)){
        cout << "number is prime" << endl;
    }
    else{
        cout << "number is composite" << endl;
    }

    return 0;
}
```

Complexity: $O(\sqrt{n})$

Fermat Primality Test

Fermat's little theorem states, that for a prime number p and a coprime integer a the following equation holds: $a^x \equiv 1 \pmod{p}$ where $x = p - 1$.

This theorem can be used to check if a number is prime or not. Let us say, we pick any random natural number, we check if the equation holds. If it does not hold, the number is composite. However, it is also possible that the equation holds for a composite number. In such a case it is impossible for us to tell whether the number is prime or not.

A way around this is to run the algorithm sufficient number of times for different, random values of a , and if the equation holds for all the iterations, we can say that the number is *most likely prime*.

Miller Rabin Primality Test

It is an extension of the ideas of Fermat's Theorem.

for an odd number n ,
 $(n-1)$ is even.

$$\Rightarrow (n-1) = 2^s \cdot d \quad ; \quad (d \% 2 = 1)$$

Factorising the equation of Fermat's little Theorem:

$$a^{n-1} \equiv 1 \pmod{n}$$

$$\Rightarrow a^{2^s d} - 1 \equiv 0 \pmod{n}$$

$$\Rightarrow (a^{(2^{s-1})d} + 1)(a^{2^{s-1}d} - 1) \equiv 0 \pmod{n}$$

\downarrow
 this term can be further
 factorised using the same formula:

$$a^2 - b^2 = (a+b)(a-b)$$

$$\Rightarrow (a^{2^{s-1}d} + 1) \dots (a^d + 1)(a^d - 1) \equiv 0 \pmod{n}$$

If n is prime, the n has to divide one of the factors mentioned in the final equation.

Code

```

import random

def miller_rabin_method(number: int, rounds: int = 40)-> bool:
    if number == 1:
        return False
    if number == 2:
        return True
    if number == 3:
        return True

    # Factor out the powers of 2 from {number - 1} and save the result
    d = number - 1
    r = 0
    while not d & 1:
        d = d >> 1
        r += 1

    # Cycle at most {round} times
    for i in range(rounds + 1):
        a = random.randint(2, number - 2)
        x = pow(a, d, number)
        if x == 1 or x == number - 1:
            continue
        # Cycle at most {r - 1} times
        for e in range(r):
            x = x * x % number
            if x == number - 1:
                break
        if x == number - 1:
            continue
        return False
    return True

if __name__ == "__main__":
    count = 0
    upper_bound = 1000
    print(f"Prime numbers lower than {upper_bound}:")
    for i in range(1, 1000):
        if miller_rabin_method(i):
            print(f"\t{i}")
            count += 1
    print(f"Total: {count}")

```

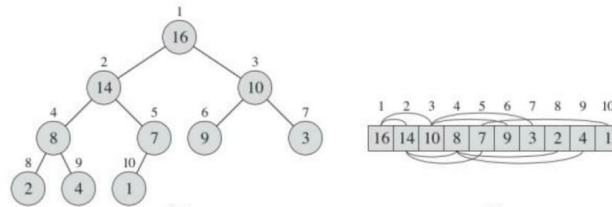

Practice Problem:

Problem 1

CHAPTER 2: DATA STRUCTURES

Heaps

The binary heap data structure is an array object that we can view as a nearly complete binary tree. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array A that represents a heap is an object with two attributes: $A.length$, which (as usual) gives the number of elements in the array, and $A.heap - size$, which represents how many elements in the heap are stored within array A . The root of the tree is $A[1]$

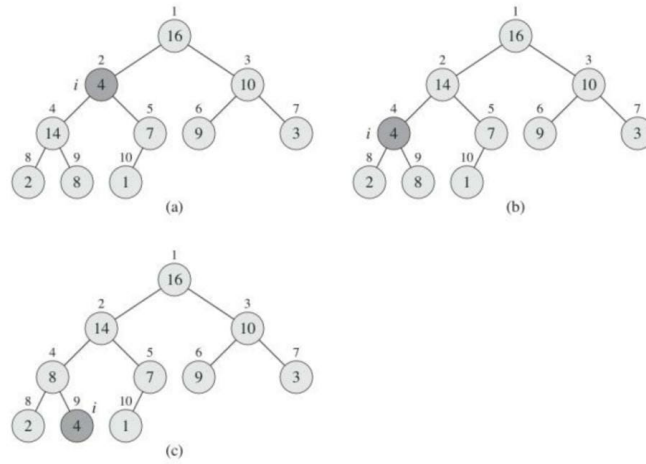


Max Heap: For every node i , $A[parent(i)] \geq A[i]$

Min Heap: For every node i , $A[parent(i)] \leq A[i]$

Maintaining the heap property (Heapify)

For a Max Heap: Its inputs are an array A and an index i into the array. When it is called, it assumes that the binary trees rooted at $L\ LEFT(i)$ and $R\ RIGHT(i)$ are max-heaps, but that $A[i]$ might be smaller than its children, thus violating the max-heap property. The algorithm lets the value at $A[i]$ “float down” in the max-heap so that the subtree rooted at index i obeys the max-heap property.

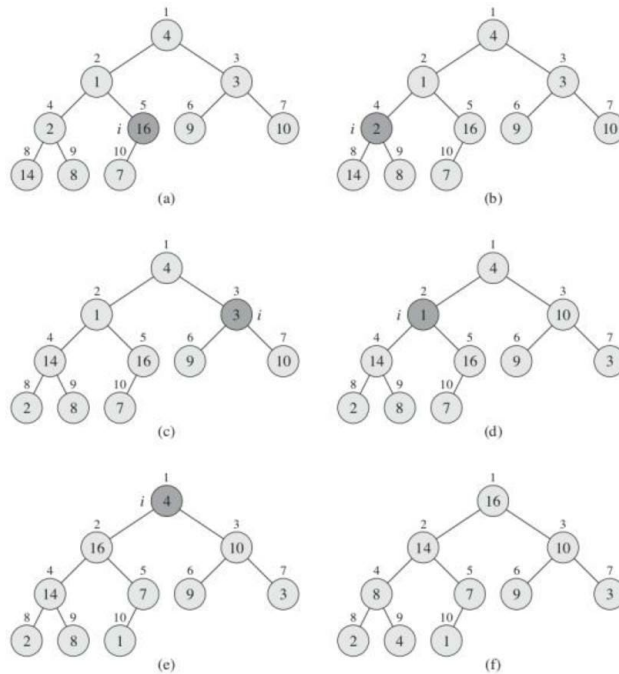


Building a Max Heap

We can use the procedure of heapify in a bottom-up manner to convert an array $A[1..n]$, where $n = A.length$, into a max-heap.

A

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---



Implementation:

```
#include <bits/stdc++.h>
#include <iostream>

int heap[100000];
int num_elements = 0;

int get_left_child(int a)
{
    return 2 * a;
}

int get_right_child(int a)
{
    return 2 * a + 1;
}

int get_parent(int a)
{
    return a / 2;
}

void insert(int a)
{
    num_elements++;
    heap[num_elements] = a;
    int position = num_elements;
    while (true)
    {
        int par = get_parent(position);
        if (par != 0 && heap[par] < heap[position])
        {
            swap(heap[par], heap[position]);
        }
        else
            break;
        position = par;
    }
}

void del_top()
{
    if (num_elements == 0)
        return;
    swap(heap[1], heap[num_elements]);
```

```

num_elements--;
int position = 1;
while (true)
{
    int a = get_left_child(position);
    int b = get_right_child(position);
    int final_index;
    if (a > num_elements)
        break;
    else if (b > num_elements)
        final_index = a;
    else
        final_index = (heap[a] > heap[b]) ? a : b;
    if (heap[final_index] > heap[position])
        swap(heap[position], heap[final_index]);
    else
        break;
    position = final_index;
}
}

void heapify()
{
    for (int i = num_elements; i >= 1; i--)
    {
        int position = i;
        while (true)
        {
            int a = get_left_child(position);
            int b = get_right_child(position);
            int final_index;
            if (a > num_elements)
                break;
            else if (b > num_elements)
                final_index = a;
            else
                final_index = (heap[a] > heap[b]) ? a : b;
            if (heap[final_index] > heap[position])
                swap(heap[position], heap[final_index]);
            else
                break;
            position = final_index;
        }
    }
}

```

```

int get_top()
{
    if (num_elements)
    {
        return heap[1];
    }
    else
    {
        return -1;
    }
}

void solve()
{
    int n;
    cin >> n;
    num_elements = n;
    for (int i = 0; i < n; i++)
    {
        cin >> heap[i + 1];
    }
    heapify();
    for (ll i = 1; i < n + 1; i++)
    {
        cout << heap[i] << " ";
    }
}

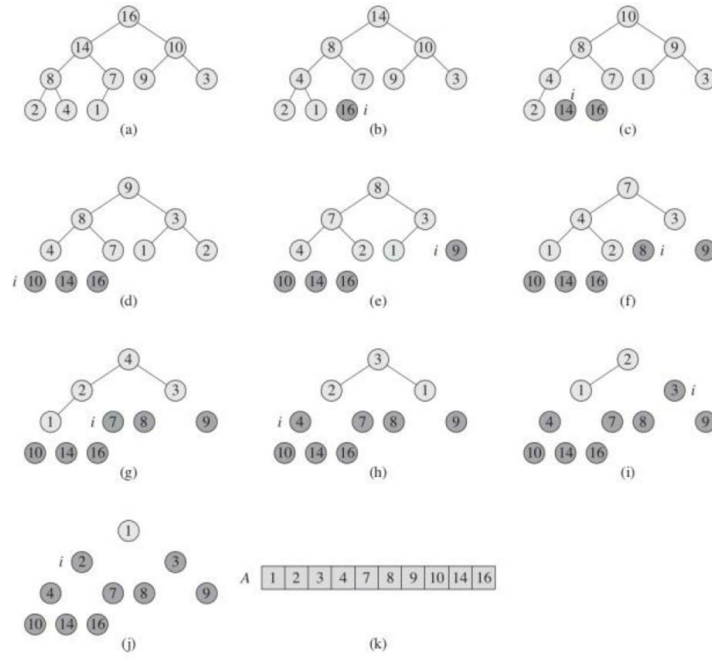
int main(void)
{
    solve();
    return 0;
}

```

Heapsort

The heapsort algorithm starts by building a max-heap on the input array $A[1\dots n]$, where $n = A.length$. Since the maximum element of the array is stored at the root $A[1]$, we can put it into its correct final position by exchanging it with $A[n]$. If we now discard node n from the heap—and we can do so by simply decrementing $A.heap - size$ we observe that the children of the root remain max-heaps, but the new root element might violate the max-heap property. All we need to do to restore the max-heap property, however, is call the `heapify` function, which leaves a max-heap in $A[1\dots n-1]$. The heapsort algorithm then repeats this process for the max-heap of size $n-1$ down to a heap of size 2.

Complexity: $O(n * \log(n))$



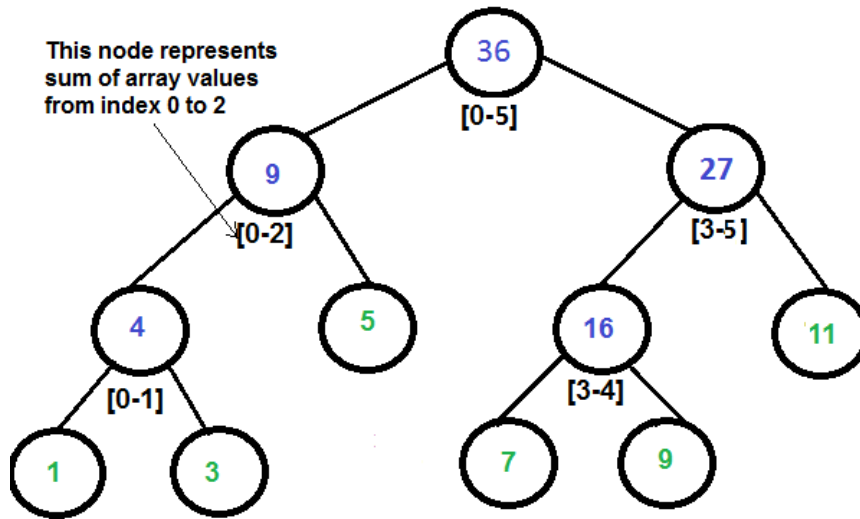
Segment Trees

Let us say we have an array a of n elements. Now we need to process queries $sum(l, r)$ where $0 \leq l \leq r$ and $update(i, x)$ i.e. $a[i] = x$. The naive method for sum would be to traverse over the array and add the elements, and for update we can do it in constant time.

Hence, $O(n)$ complexity for $sum(l, r)$ and $O(1)$ complexity for $update(i, x)$.

However, in case there are a large number of sum queries, this method would be extremely inefficient. This is where **Segment Trees** come in. They provide a much more efficient way of processing the $sum(l, r)$ queries. We shall first look at the construct of the data structure and then analyse the complexities.

Following is the pictorial representation of a segment tree:



Segment Tree for input array {1, 3, 5, 7, 9, 11}

Construction

We compute and store the sum of the elements of the whole array, i.e. the sum of the segment $a[0 \dots n-1]$. We then split the array into two halves $a[0 \dots n/2]$ and $a[n/2+1 \dots n-1]$ and compute the sum of each half and store them. Each of these two halves in turn also split in half, their sums are computed and stored. And this process repeats until all segments reach size 1. In other words we start with the segment $a[0 \dots n-1]$, split the current segment in half (if it has not yet become a segment containing a single element), and then calling the same procedure for both halves. For each such segment we store the sum of the numbers on it.

We can say, that these segments form a binary tree: the root of this tree is the segment $a[0...n-1]$, and each vertex (except leaf vertices) has exactly two child vertices.

Height of the tree: $\log n$

Number of vertices required:

$$2^0 + 2^1 + \dots + 2^h \leq 4n \text{ where } h = \log n$$

Before constructing the segment tree, we need to decide:

1. The value that gets stored at each node of the segment tree. For example, in a sum segment tree, a node would store the sum of the elements in its range $[l, r]$.
2. The merge operation that merges two siblings in a segment tree. For example, in a sum segment tree, the two nodes corresponding to the ranges $a[l1...r1]$ and $a[l2...r2]$ would be merged into a node corresponding to the range $a[l1...r2]$ by adding the values of the two nodes.

Now, for construction of the segment tree, we start at the bottom level (the leaf vertices) and assign them their respective values. On the basis of these values, we can compute the values of the previous level, using the *merge* function. And on the basis of those, we can compute the values of the previous, and repeat the procedure until we reach the root vertex.

It is convenient to describe this operation recursively in the other direction, i.e., from the root vertex to the leaf vertices. The construction procedure, if called on a non-leaf vertex, does the following:

1. Recursively construct the values of the two child vertices
2. Merge the computed values of these children. We start the construction at the root vertex, and hence, we are able to compute the entire segment tree.

The time complexity of this construction is $O(n)$, assuming that the merge operation is constant time (the merge operation gets called n times, which is equal to the number of internal nodes in the segment tree).

Sum Queries

For now we are going to answer sum queries. As an input we receive two integers l and r , and we have to compute the sum of the segment $a[l...r]$ in $O(\log n)$ time.

To do this, we will traverse the Segment Tree and use the precomputed sums of the segments. Let's assume that we are currently at the vertex that covers the segment $a[tl...tr]$. There are three possible cases.

The easiest case is when the segment $a[l...r]$ is equal to the corresponding segment of the current vertex (i.e. $a[l...r] = a[tl...tr]$), then we are finished and can return the precomputed sum that is stored in the vertex.

Alternatively the segment of the query can fall completely into the domain of either the left or the right child. Recall that the left child covers the segment $a[tl...tm]$ and the right vertex covers the segment $a[tm + 1...tr]$ with $tm = (tl + tr)/2$. In this case we can simply go to the child vertex, which corresponding segment covers the query segment, and execute the algorithm described here with that vertex.

And then there is the last case, the query segment intersects with both children. In this case we have no other option as to make two recursive calls, one for each child. First we go to the left child, compute a partial answer for this vertex (i.e. the sum of values of the intersection between the segment of the query and the segment of the left child), then go to the right child, compute the partial answer using that vertex, and then combine the answers by adding them. In other words, since the left child represents the segment $a[tl...tm]$ and the right child the segment $a[tm + 1...tr]$, we compute the sum query $a[l...tm]$ using the left child, and the sum query $a[tm + 1...r]$ using the right child.

So processing a sum query is a function that recursively calls itself once with either the left or the right child (without changing the query boundaries), or twice, once for the left and once for the right child (by splitting the query into two subqueries). And the recursion ends, whenever the boundaries of the current query segment coincides with the boundaries of the segment of the current vertex. In that case the answer will be the precomputed value of the sum of this segment, which is stored in the tree.

In other words, the calculation of the query is a traversal of the tree, which spreads through all necessary branches of the tree, and uses the precomputed sum values of the segments in the tree.

Obviously we will start the traversal from the root vertex of the Segment Tree.

Complexity: $O(\log n)$

Update Queries

Now we want to modify a specific element in the array, let's say we want to do the assignment $a[i] = x$. And we have to rebuild the Segment Tree, such that it correspond to the new, modified array.

This query is easier than the sum query. Each level of a Segment Tree forms a partition of the array. Therefore an element $a[i]$ only contributes to one segment from each level. Thus only $O(\log n)$ vertices need to be updated.

It is easy to see, that the update request can be implemented using a recursive function. The function gets passed the current tree vertex, and it recursively calls itself with one of the two child vertices (the one that contains $a[i]$ in its segment), and after that recomputes its sum value, similar how it is done in the build method (that is as the sum of its two children).

Implementation

```
void build(int node, int start, int end)
{
    if(start == end)
    {
        // Leaf node will have a single element
        tree[node] = A[start];
    }
    else
    {
        int mid = (start + end) / 2;
        // Recurse on the left child
        build(2*node, start, mid);
        // Recurse on the right child
        build(2*node+1, mid+1, end);
        // Internal node will have the sum of both of its children
        tree[node] = tree[2*node] + tree[2*node+1];
    }
}

void update(int node, int start, int end, int idx, int val)
{
    if(start == end)
    {
        // Leaf node
        A[idx] += val;
        tree[node] += val;
    }
    else
    {
        int mid = (start + end) / 2;
        if(start <= idx and idx <= mid)
        {
            // If idx is in the left child, recurse on the left child
            update(2*node, start, mid, idx, val);
        }
        else
        {
            // if idx is in the right child, recurse on the right child
            update(2*node+1, mid+1, end, idx, val);
        }
        // Internal node will have the sum of both of its children
        tree[node] = tree[2*node] + tree[2*node+1];
    }
}
```

```

int sum(int node, int start, int end, int l, int r)
{
    if(r < start or end < l)
    {
        // range represented by a node is completely outside the given range
        return 0;
    }
    if(l <= start and end <= r)
    {
        // range represented by a node is completely inside the given range
        return tree[node];
    }
    // range represented by a node is partially inside and partially outside the given range
    int mid = (start + end) / 2;
    int p1 = query(2*node, start, mid, l, r);
    int p2 = query(2*node+1, mid+1, end, l, r);
    return (p1 + p2);
}

```

Practice Problems

Problem 1

Problem 2

Problem 3

CHAPTER 3: DIVIDE AND CONQUER

A divide and conquer algorithm breaks down a bigger problem into smaller subproblems. We solve these simpler subproblems, and combine the solutions to these subproblems to get the final solution. It is used in many algorithms such as sorting (mergesort), fast fourier transform, etc. We shall now be looking at these algorithms to understand the concept of divide and conquer algorithms.

First we will have a look at the Master Theorem which is useful in finding the complexity of the algorithms.

Master Theorem

Given the recurrence relation, this theorem helps us in finding the complexity of our solution.

Given the recurrence relation: $T(n) = a * T(n/b) + O(n^d)$

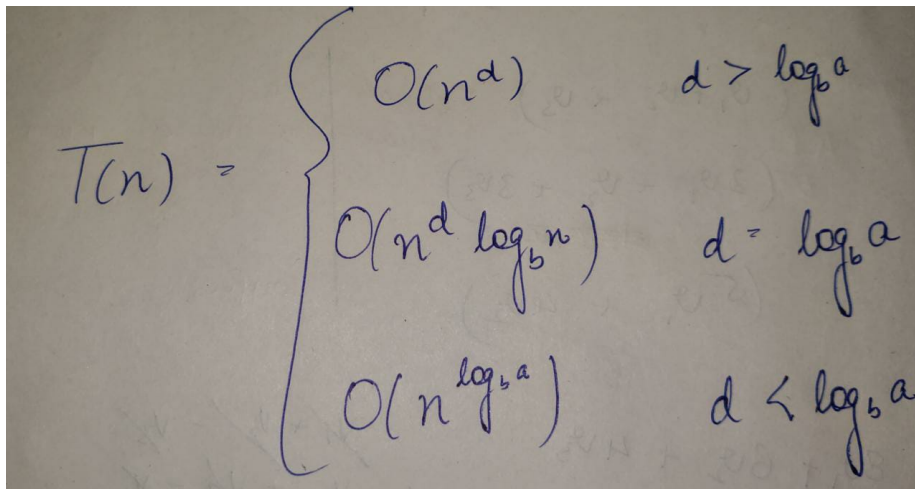
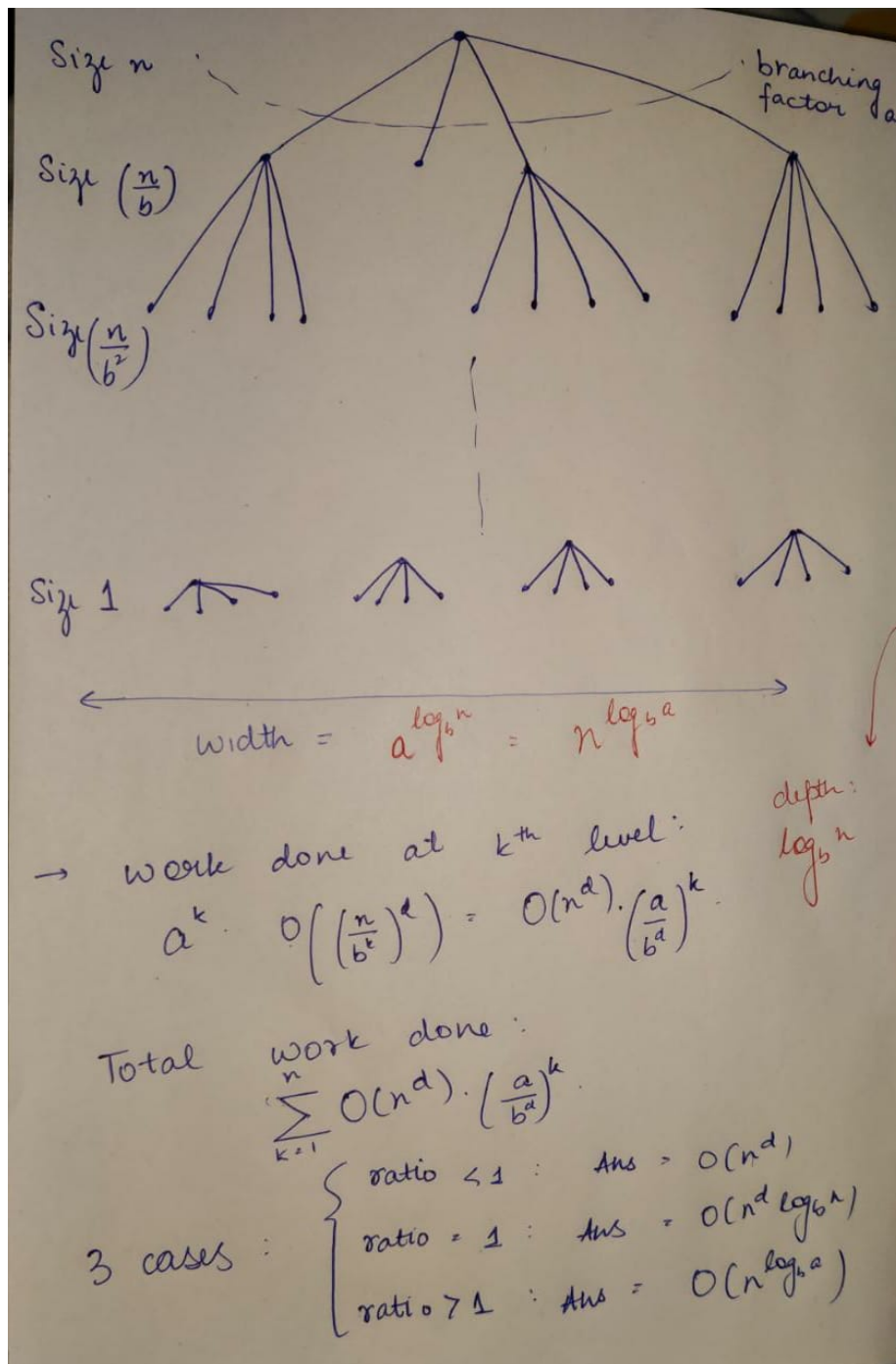

$$T(n) = \begin{cases} O(n^d) & d > \log_b a \\ O(n^d \log_b n) & d = \log_b a \\ O(n^{\log_b a}) & d < \log_b a \end{cases}$$

Figure 1: Solution

Proof:



Mergesort

- Split the array into two equal halves
- Sort them recursively
- Merge the two sorted halves

Complexity: $T(n) = 2T(n/2) + O(n)$ > By Master Theorem, complexity = $O(n \log n)$

Code

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

void printArray(int A[], int n)
{
    for (int i = 0; i < n; i++)
    {
        cout << A[i] << " ";
    }
    cout << endl;
}

void merge(int A[], int start, int end)
{
    int mid = (end + start) / 2;

    int *B = new int[end - start + 1];

    int index = 0;

    int i = start;
    int j = mid + 1;

    int n1 = mid + 1;
    int n2 = end + 1;

    while (i < n1 && j < n2)
    {
        if (A[i] <= A[j])
        {
            B[index] = A[i];
            i++;
        }
        else
        {
            B[index] = A[j];
            j++;
        }
        index++;
    }

    while (i < n1)
        B[index++] = A[i++];

    while (j < n2)
        B[index++] = A[j++];

    for (int i = start; i <= end; i++)
        A[i] = B[i - start];
}
```

```

        j++;
    }
    index++;
}

while (i < n1)
{
    B[index] = A[i];
    i++;
    index++;
}
while (j < n2)
{
    B[index] = A[j];
    j++;
    index++;
}

for (int k = 0; k < index; k++)
{
    A[start] = B[k];
    start++;
}

delete[] B;
}

void mergeSort2(int A[], int start, int end)
{
    if (start >= end)
        return;

    int mid = (end + start) / 2;

    mergeSort2(A, start, mid);
    mergeSort2(A, mid + 1, end);

    // printArray(A, (end - start + 1));

    merge(A, start, end);
}

void mergeSort(int input[], int len)
{
    mergeSort2(input, 0, len - 1);
}

```



```

}

int main()
{
    int length;
    cin >> length;
    int *input = new int[length];
    for (int i = 0; i < length; i++)
        cin >> input[i];
    mergeSort(input, length);
    for (int i = 0; i < length; i++)
    {
        cout << input[i] << " ";
    }
}

```

Why $\Omega(n \log n)$ is the lower bound for comparison based sort

Any comparison based sorting algorithm has to make at least $n * \log n$ comparisons.

- There are at max $n!$ permutations, so the answer can be any 1 of these permutations
- Lets say we have an algorithm that did k comparisons. So there can be 2^k comparison strings.
- 2^k should at least equal to $n!$, otherwise we may miss out on the solution.
- This implies that minimum $n \log n$ comparisons are needed.

Matrix multiplication

Complexity of the Naive Algo: $O(n^3)$

Naive is $O(n^3)$ because it takes $O(n)$ to populate one element in the product matrix.

Divide and conquer method

Let $X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$

$Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$

be $n \times n$ matrices. we can
 divide this into 4 $\frac{n}{2} \times \frac{n}{2}$ matrices.
 They become (A) (B) (C) ... (H).
 multiplication: $XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$

$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$

$T(n) = O(n^3)$

Complexity: $O(n^3)$

The Fast Fourier Transform Algorithm

func FFT(A,w):

input: coefficient representation of a polynomial $A(x)$ of degree $\leq n-1$, where n is a power of 2.

$w(\omega)$, is the n th root of unity.

output: value representation of the polynomial $(A(w^0), A(w^1), \dots, A(w^{n-1}))$

if $w = 1$: return $A(1)$;

express $A(x)$ as $A_e(x^2) + x(A_o)(x^2)$

call $\text{FFT}(A_e, w^2)$; to evaluate A_e at even powers of w

call $\text{FFT}(A_o, w^2)$; to evaluate A_o at even powers of w

for $j=0$ to $n-1$:

compute $A(w^j) = [Ae(w^2j) + (w^j)Ao(w^2j)]$

return $A(w^0), \dots, A(w^{n-1})$;

Complexity: $O(d \log d)$; for multiplying 2 d-degree numbers

$$\langle \text{value} \rangle = \text{FFT}(\langle \text{coeff} \rangle, \omega)$$

$$\langle \text{coeff} \rangle = \frac{1}{n} \text{FFT}(\langle \text{value} \rangle, \omega)$$

Proof:

$$\begin{pmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{pmatrix} = \begin{pmatrix} 1 & x_0 & \dots & x_0^{n-1} \\ 1 & x_1 & \dots & x_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & \dots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

The FFT matrix:

$$M_n(\omega) = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{pmatrix}$$

$$M_n(\omega)^{-1} = \frac{1}{n} M_n(\omega^{-1})$$

Example Problem

Question

Solution

```
#include<bits/stdc++.h>
using namespace std;
long long int cnt(long long int temp) //returns the length of final list
{
    long long int x=1;
    while(temp>1)
    {
        temp/=2;
        x*=2;
    }
    return x;
}
int is_one(long long int pos,long long int target,long long int num)
{
    if(num<2)
        return num;
    if(pos+1==2*target)
    {
        return num%2;
    }
    num/=2;
    pos/=2;
    if(target>pos+1)
        target-=(pos+1);
    return is_one(pos,target,num);
}

int main()
{
    long long int l,r,n,x,ans=0,i;
    cin>>n>>l>>r;
    x=cnt(n);
    x=2*x-1;
    for(i=l; i<=r; i++)    ans+=is_one(x,i,n);
    cout<<ans<<endl;
    return 0;
}
```

Practice Problems

The questions are sorted by difficulty.

Problem 1

Problem 2

Problem 3

CHAPTER 4: GREEDY ALGORITHMS

A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

Note that it is not necessary that a greedy algorithm will necessarily give the optimal solution. Sometimes, it may fail.

Elements of the greedy strategy

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur in either order.)
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

Greedy Choice Property A ingredient is the greedy-choice property: we can assemble a globally optimal solution by making locally optimal (greedy) choices. In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems.

Optimal Substructure A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms.

A major application of these algorithms is the Huffman codes.

Huffman Codes

We know that generally we allocate 1 bit to store one character and 4 bits i.e. 1 byte to store an int. However, such an allocation of memory may not always be optimal. An optimal solution would be to allocate memory according to the frequency of the occurrence of characters in a document, allocating lesser memory to more frequently occurring characters, and vice versa. Huffman Codes is an algorithm to do just that.

Huffman codes compress data very effectively: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string.

Now we shall consider an example and see how Huffman encoding works. Then we shall also look at the implementation of the algorithm.

Example: | Symbol | Frequency | |——|———| | A | 70 | | B | 3 | | C | 20 |
| D | 37 |

One possible method is to use 2 bits for each symbol (since there are 4 symbols in total).

Memory needed for implementation: $2 * (70 + 3 + 20 + 37) = 260$ bytes

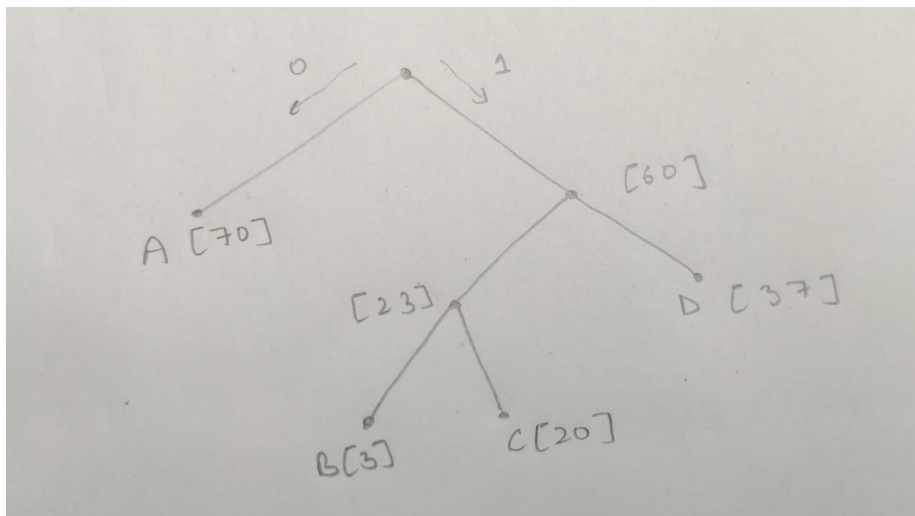
But can we do better? YES!

Variable length encoding:

Symbol	Codeword
A	0
B	100
C	101
D	11

For the following encoding, cost = \$ $(170) + (33) + (320) + (237) = 213$ bytes.

Hence this is an improvement over the initial usage of memory.



How did we arrive at this encoding?

By Greedy Method, of course!

1. The Greedy Choice: The 2 symbols with the smallest frequencies must be at the bottom of the lowest internal node. If this was not the case, swapping these 2 symbols with whatever is lowest in the tree would improve our coding.
2. Optimum Substructure: The frequency of any internal node to be the sum of the frequencies of its descendent leaves. The cost of the trees is the sum of the frequencies of all leaves and internal nodes, except the root.

Algorithm for Huffman Coding (in python):

```
# reading from a file and storing the text:
with open('sample.txt', 'r') as f:
    text = f.read()

# empty lists for alphabets and their frequencies
freq = []
chars = set()

# initialise freq list:
for i in range(26):
    freq.append(0)

# calculating frequencies of characters:
for char in range(0, len(text)):
    index = -1
    ch = ''
    if text[char] >= 'A' and text[char] <= 'Z':
        index = ord(text[char]) - 65
        ch = chr(ord(text[char]) + 32)
    elif text[char] >= 'a' and text[char] <= 'z':
        index = ord(text[char]) - 97
        ch = text[char]
    freq[index] += 1
    if(ch != ''):
        chars.add(ch)

chars = list(chars)
chars.sort()

for i in list(freq):
    if i == 0:
        freq.remove(i)

# A Huffman Tree Node
class node:
```



```

def __init__(self, freq, symbol, left=None, right=None):
    self.freq = freq
    self.symbol = symbol
    self.left = left
    self.right = right
    self.huff = ''

my_dict = {}
def makeDict(node, val=''):
    newVal = val + str(node.huff)

    if(node.left):
        makeDict(node.left, newVal)
    if(node.right):
        makeDict(node.right, newVal)

    if(not node.left and not node.right):
        my_dict[node.symbol] = newVal

# list containing unused nodes
nodes = []

# converting characters and frequencies
# into huffman tree nodes
for x in range(len(chars)):
    nodes.append(node(freq[x], chars[x]))

while len(nodes) > 1:
    nodes = sorted(nodes, key=lambda x: x.freq)

    left = nodes[0]
    right = nodes[1]

    left.huff = 0
    right.huff = 1

    newNode = node(left.freq+right.freq, left.symbol+right.symbol, left, right)

    nodes.remove(left)
    nodes.remove(right)
    nodes.append(newNode)

# Huffman Tree is ready!
makeDict(nodes[0])
print(my_dict.items())

```

Now let us look at another problem statement that makes use of the Greedy strategy. This time we shall not necessarily obtain the optimal solution but the solution would still be preferred keeping in mind the better time complexity of greedy algorithms.

The Set Cover Problem

Input: A set of elements B; sets S_1, S_2, \dots, S_n are a subset of B

Output: A selection of the S_i whose union is B. Cost: Number of sets picked.

Example:

Consider each of the following words as a set of letters:

$\{shun, dear, deer, doctor, mine, their\}$

To cover the Set B:

$\{a, d, e, i, m, n, r, s, t, u\}$

A Greedy Algorithm:

Repeat until all sets of B are covered. Pick the set S_i with the largest number of uncovered elements.

For our example:

- First pick: dear (4 uncovered letters)
- uncovered: $\{i, m, n, s, t, u\}$
- Second pick: shun (3 uncovered letters)
- uncovered: $\{i, m, t\}$
- Third pick: mine (2 uncovered letters)
- uncovered letters: $\{t\}$
- Fourth pick: their

However such a method may NOT always be optimum.

Counter Example:

To cover set $B = \{1, 2, 3, 4, 5, 6\}$ Set family: $\{1, 2, 3, 4\}$, $\{5, 6\}$, $\{2, 4, 6\}$

- Our greedy algorithm would pick all 3 sets, while the optimum solution is 2 (2nd and 3rd set).

However, we would still prefer the Greedy solution because our set is NP-complete i.e. no optimal solution has yet been found for such a problem. Hence, it is advisable to give priority to better Time Complexity over the correct solution.

Claim: Let set B contains n elements and that the optimal cover of k sets. Then our greedy algorithm will use at most $k \ln(n)$ sets.

n_t : number of elements still not covered after t -iterations.

& $n_0 = n$ (because at $t = 0$, all elements are uncovered)

→ Since we assume that all the n elements are covered by some optimal k -sets, ~~there~~

least possible sets = $\frac{n}{k}$.

$$\Rightarrow n_{t+1} = n_t - \frac{n_t}{k} = n_t \left(1 - \frac{1}{k}\right)$$

$$\therefore n_t \leq n_0 \text{ (always)}$$

$$\Rightarrow n_{t+1} \leq n_0 \left(1 - \frac{1}{k}\right)^t \quad \left[\begin{array}{l} \text{for} \\ t\text{-iterations} \end{array} \right]$$

Now, we know,

$$(1-x) \leq e^{-x}.$$

$$\Rightarrow n_t \leq n_0 (e^{-1/k})^t$$

$$n_t \leq n_0 e^{-t/k}$$

At , $t = k \ln(n)$,

$$RHS = 1$$

→ hence, this is our upper bound.

$t \leq k \ln(n)$

hence, proved.

Other Examples (Competitive Programming)

Problem 1

Solution

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;

void solve()
{
    int s; cin >> s;
    int ans = 1, num=1;
    s-=1;

    while(s>0){
        if(s>=num+2){
            num+=2;
            s-=num;
            ans++;
        }
        else{
            ans++;
            break;
        }
    }
}
```

```

    }
    cout << ans << endl;
}

```

```

int main()
{
    int t;
    cin >> t;
    while (t--)
    {
        solve();
    }
}

```

Problem 2

Solution

```

#include <iostream>
#include <bits/stdc++.h>
using namespace std;

typedef long long int ll;

void solve()
{
    ll n, k;
    cin >> n >> k;
    ll ans = 0;
    ll pwssofar = 0;
    --n;
    while (n)
    {
        ll pw = (1LL << pwssofar);
        if (pw > k)
        {
            break;
        }
        if (pw >= n)
        {
            n = 0;
            ++ans;
            break;
        }
        n -= (1LL << pwssofar);
    }
}

```

```

        ++ans;
        pwssofar++;
    }

    ll need = ceil(n / (long double)k);
    ans = ans + need;
    cout << ans << '\n';
}

int main()
{
    int t;
    cin >> t;
    while (t--)
    {
        solve();
    }

    return 0;
}

```

Problem 3

Solution

```

#include <iostream>
#include <bits/stdc++.h>
#include <assert.h>
using namespace std;

#define quick \
    cin.tie(0); \
    ios_base::sync_with_stdio(false);

int grid[25][25];
void tick(int x, int y, int d)
{
    if (x < d || y < d)
        return;
    for (int i = 0; i <= d; i++)
    {
        if (!grid[x - i][y - i] || !grid[x - i][y + i])
            return;
    }
    for (int i = 0; i <= d; i++)
    {

```

```

        ++grid[x - i][y - i];
        ++grid[x - i][y + i];
    }
}

void solve()
{
    int n, m, k;
    cin >> n >> m >> k;
    for (int i = 0; i < n; i++)
    {
        string s;
        cin >> s;
        for (int j = 0; j < m; j++)
            grid[i][j] = s[j] == '*';
    }
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            for (int d = k; j + d < m; d++)
                tick(i, j, d);
    bool ok = true;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            if (grid[i][j] == 1)
                ok = false;
    cout << (ok ? "YES" : "NO") << endl;
}

int main()
{
    quick

    int t;
    cin >> t;
    while (t--)
    {
        solve();
    }
}

```

Practice Problems

Problem 1

Problem 2

Problem 3

CHAPTER 5: DYNAMIC PROGRAMMING

Let us consider the problem of calculating Fibonacci numbers. For fibonacci numbers:

$$\text{fib}(0) = 0$$

$$\text{fib}(1) = 1$$

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \text{ for all } n \geq 2$$

Now, let us try to write code for calculating them. One method is to use recursion.

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

int fibo(int n)
{
    if (n < 2)
    {
        return n;
    }

    int a = fibo(n - 1);
    int b = fibo(n - 2);

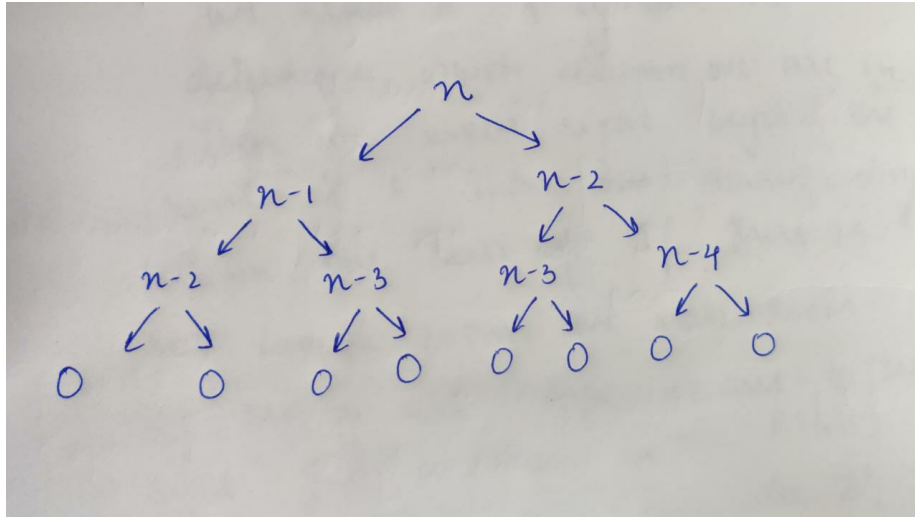
    return a + b;
}

int main()
{
    int n;
    cin >> n;

    cout << fibo(n) << endl;
}
```

Here, for every n , we need to make a recursive call to $f(n-1)$ and $f(n-2)$.

For $f(n-1)$, we will again make the recursive call to $f(n-2)$ and $f(n-3)$. Similarly, for $f(n-2)$, recursive calls are made on $f(n-3)$ and $f(n-4)$ until we reach the base case. The recursive call diagram will look something like shown below:



At every recursive call, we are doing constant work(k)(addition of previous outputs to obtain the current one). At every level, we are doing $2^n K$ work (where $n = 0, 1, 2, \dots$). Since reaching 1 from n will take n calls, therefore, at the last level, we are doing $2^x k$ ($x = n - 1$) work.

$$\text{Total Work} = 2^n k$$

$$\text{Time Complexity: } O(2^n)$$

Clearly, the time complexity of this algorithm is very bad, and it very quickly ($n=40$) starts giving a time limit error as the program runs infinitely. Hence we now need to look at ways of improving our algorithm.

Important Observation: We can observe that repeated recursive calls are being made. For example in the given figure, $\text{fib}(n-2)$ and $\text{fib}(n-3)$ is being called twice. This is causing the time complexity to worsen exponentially. To overcome this problem, we will store the output of previously encountered value. Next time whenever we will be making the recursive calls over these values, we will directly consider their already stored outputs and then use these in our calculations instead of calculating them over again. This way, we can improve the running time of our code. This process of storing each recursive call's output and then using them for further calculations preventing the code from calculating these again is called *memoization*.

Let us look at the implementation:

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

int helper(int n, int *ans)
{
    if (n <= 1)
```

```

    {
        return n;
    }

    //check if output already exists
    if (ans[n] != -1)
    {
        return ans[n];
    }

    int x = helper(n - 1, ans);
    int y = helper(n - 2, ans);

    //saving output
    ans[n] = x + y;
    //returning output
    return ans[n];
}

int fibo(int n)
{
    int *ans = new int[n + 1];

    for (int i = 0; i <= n; i++)
    {
        ans[i] = -1;
    }

    return helper(n, ans);
}

int main()
{
    int n;
    cin >> n;

    cout << fibo(n) << endl;
}

```

Memoization is a top-down approach where we save the previous answers so that they can be used to calculate the future answers and improve the time complexity to a greater extent.

Finally, what we are doing is making a recursive call to every index of the answer array and calculating the value for it using previous outputs stored. Recursive calls terminate over the base case, which means we are already aware of the answers that should be stored in the base case's indexes. In cases of

Fibonacci numbers, these indexes are 0 and 1 as $f(0) = 0$ and $f(1) = 1$. So we can directly allot these two values to our answer array and then use these to calculate $f(2)$, which is $f(1) + f(0)$, and so on for every other index. This can be simply done iteratively by running a loop from $i = (2 \text{ to } n)$. Finally, we will get our answer at the 5th index of the answer array as we already know that the i -th index contains the answer to the i -th value. Simply, we are first trying to figure out the dependency of the current value on the previous values and then using them calculating our new value. Now, we are looking for those values which do not depend on other values, which means they are independent (the base case's values as these are the smallest problems about which we are already aware of). Finally, following a bottom-up approach to reach the desired index. This approach of converting recursion into iteration is known as **Dynamic programming(DP)**.

Now let us look at the DP solution for Fibonacci numbers.

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

int fibo(int n)
{
    int *ans = new int[n + 1];

    ans[0] = 0;
    ans[1] = 1;

    for (int i = 2; i <= n; i++)
    {
        ans[i] = ans[i - 1] + ans[i - 2];
    }
    return ans[n];
}

int main()
{
    int n;
    cin >> n;

    cout << fibo(n) << endl;

    return 0;
}
```

The Edit Distance Problem:

Q. Between 2 words, calculate the minimum number of insert/delete/overwrite operations to convert one word to another. Or, find the cost of aligning the

2 words, where the cost is equal to the number of columns where the letters differ. The edit distance equals such best possible alignment.

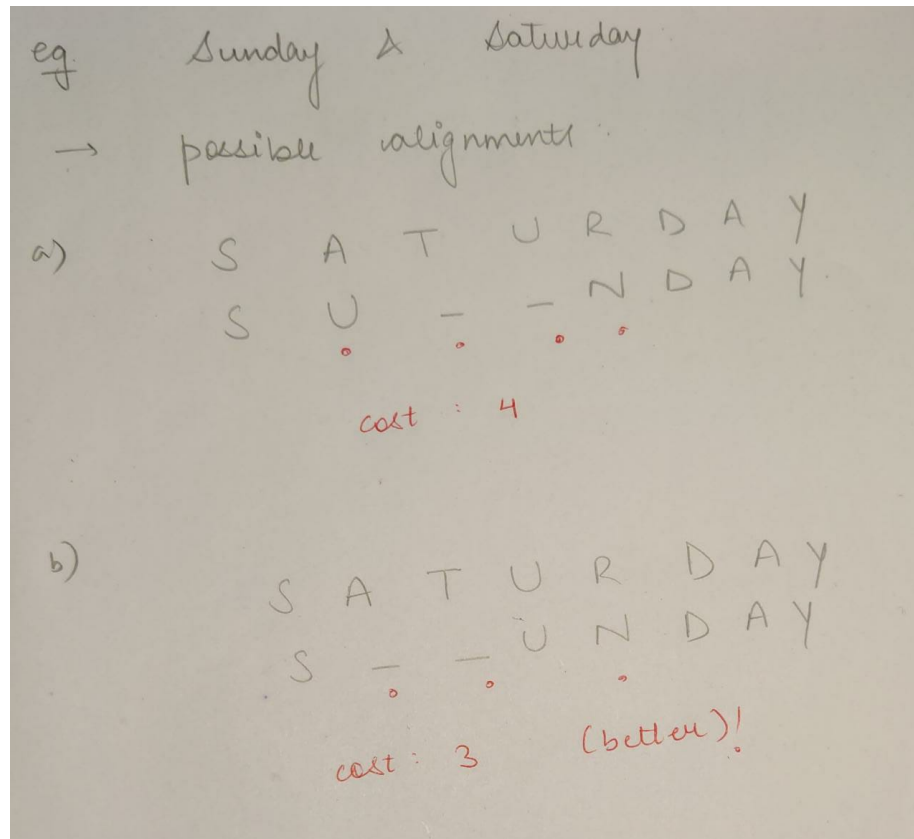


Figure 1: example

Solution using dynamic programming:

1. First we need to figure out what our subproblems are. Clearly, for any 2 substrings of length a and b , with $a < s1.size()$ and $b < s2.size()$, we need to find the edit distance between them.
2. For the rightmost column, we can have the following 3 cases:

$$\begin{array}{c}
 x[i] \quad \text{OR} \quad y[j] \quad \text{OR} \quad x[i] \\
 \quad \quad \quad \quad \quad \quad \quad \quad \quad y[j] \quad \quad \quad \quad y[j]
 \end{array}$$

$$E(i, j) = \min \left(\begin{array}{l} 1 + E(i-1, j) \\ 1 + E(i, j-1) \\ \text{diff}(i, j) + E(i-1, j-1) \end{array} \right)$$

$$\text{diff}(i, j) = \begin{cases} 0, & \text{if } x[i] = y[j] \\ 1, & \text{otherwise} \end{cases}$$

Algorithm:

- run a loop over all possible substrings of string s1 and s2.
- Maintain a 2d array dp[m+1][n+1], to store the answers, m: length of string 1 and n: length of string 2.
- Initialise dp[i][0] = i; dp[0][j] = j
- Then fill the other values according to the formula given above. dp[m][n] will be our final answer.

Code:

```

#include <iostream>
#include <bits/stdc++.h>
using namespace std;

int min(int x, int y, int z)
{
    return min(x, min(y, z));
}

int Algo(string str1, string str2, int m, int n)
{
    int dp[m + 1][n + 1];

    for (int i = 0; i <= m; i++)
    {
        for (int j = 0; j <= n; j++)
        {
            if (i == 0)
                dp[i][j] = j;
            else if (j == 0)

```

```

        dp[i][j] = i;
    else if (str1[i - 1] == str2[j - 1])
        dp[i][j] = dp[i - 1][j - 1];
    else
        dp[i][j] = 1 + min(dp[i][j - 1], dp[i - 1][j], dp[i - 1][j - 1]);
    }
}

return dp[m][n];
}

int main()
{
    string str1, str2;
    cin >> str1 >> str2;

    cout << Algo(str1, str2, str1.length(), str2.length());

    return 0;
}

```

Complexity: $O(m*n)$

DP is used extensively in many graph algorithms for optimisation and we shall be looking at those later on in the book. For now, let us look at some of the problems that make use of DP for practice purposes.

Problem 1

Solution

```

#include <iostream>
using namespace std;

#include <bits/stdc++.h>
#define mod 1000000007

int balancedBTs(int h)
{
    vector<long long> dp(h+1);

    dp[0] = 1;
    dp[1] = 1;
    dp[2] = 3;

    if(h==1 || h==2)
        return dp[h];
}

```

```

        for(int i=3;i<=h;i++){
            long long x = dp[i-1] % mod;
            long long y = dp[i-2]%mod;

            long long a = (x*x)%mod;
            long long b = (2*x*y)%mod;
            long long ans = (a+b)%mod;
            dp[i]=ans;
        }
        return dp[h];
    }

int main() {
    int n;
    cin >> n;
    cout << balancedBTs(n);
}

```

Problem 2

Solution

I am giving the solution using all 3 techniques i.e. Recursion, Memoisation, and DP.

```

#include <bits/stdc++.h>
using namespace std;

int helper(int **arr, int m, int n, int i, int j)
{
    if (i == m - 1 && j == n - 1)
    {
        return arr[m - 1][n - 1];
    }

    int x = INT_MAX, y = INT_MAX, z = INT_MAX;

    if (i < m - 1)
        x = helper(arr, m, n, i + 1, j);

    if (j < n - 1)
        y = helper(arr, m, n, i, j + 1);

    if (i < m - 1 && j < n - 1)
        z = helper(arr, m, n, i + 1, j + 1);
}

```

```

        return min(min(x, y), z) + arr[i][j];
    }

    int minCostPath(int **input, int m, int n)
    {
        //Write your code here
        return helper(input, m, n, 0, 0);
    }

    int memo_helper(int **arr, int m, int n, int i, int j, int **ans)
    {
        if (i == m - 1 && j == n - 1)
        {
            return arr[i][j];
        }

        if (i >= m || j >= n)
        {
            return INT_MAX;
        }

        if (ans[i][j] != -1)
        {
            return ans[i][j];
        }

        int x = helper(arr, m, n, i + 1, j);
        int y = helper(arr, m, n, i, j + 1);
        int z = helper(arr, m, n, i + 1, j + 1);

        ans[i][j] = min(min(x, y), z) + arr[i][j];
        return ans[i][j];
    }

    int memoisation(int **input, int m, int n)
    {
        int **ans = new int *[m];
        for (int i = 0; i < m; ++i)
            ans[i] = new int[n];

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

```



```

        ans[i][j] = -1;
    }
}
return memo_helper(input, m, n, 0, 0, ans);
}

int dp_minCost(int **input, int m, int n)
{
    int **ans = new int * [m];
    for (int i = 0; i < m; i++)
    {
        ans[i] = new int [n];
    }

    //filling the last cell
    ans[m - 1][n - 1] = input[m - 1][n - 1];

    //fill last row (right --> left)
    for (int j = n - 2; j >= 0; j--)
    {
        ans[m - 1][j] = ans[m - 1][j + 1] + input[m - 1][j];
    }

    //fill last column (down --> up)
    for (int j = m - 2; j >= 0; j--)
    {
        ans[j][n - 1] = ans[j + 1][n - 1] + input[j][n - 1];
    }

    //fill remaining cells
    for (int i = m - 2; i >= 0; i--)
    {
        for (int j = n - 2; j >= 0; j--)
        {
            ans[i][j] = min(min(ans[i + 1][j], ans[i][j + 1]), ans[i + 1][j + 1]) + input[i][j];
        }
    }

    return ans[0][0];
}

int main()
{
    int **arr, n, m;
    cin >> n >> m;
    arr = new int * [n];

```

```

for (int i = 0; i < n; i++)
{
    arr[i] = new int[m];
}
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < m; j++)
    {
        cin >> arr[i][j];
    }
}
cout << minCostPath(arr, n, m) << endl;
cout << memoisation(arr, m, n) << endl;
cout << dp_minCost(arr, m, n) << endl;
}

```

Other interesting problems

Chain Matrix Multiplication

We want to multiply 4 matrices A,B,C,D of dimensions 50x20, 20x1, 1x10, 10x100.

Multiplying $m \times n$ matrix with $n \times p$ matrix takes mnp multiplications.

We can visualise this problem as binary trees, where the children of the same parent node imply that those 2 matrices are being multiplied.

Now, for the binary tree to be optimum, the subtrees must also be optimum.

Code:

```

#include <bits/stdc++.h>
using namespace std;
int dp[100][100];

// Function for matrix chain multiplication
int helper(vector<int> p, int i, int j)
{
    if (i == j)
    {
        return 0;
    }
    if (dp[i][j] != -1)
    {
        return dp[i][j];
    }
}

```

```

        dp[i][j] = INT_MAX;
        for (int k = i; k < j; k++)
        {
            dp[i][j] = min(
                dp[i][j], helper(p, i, k) + helper(p, k + 1, j) + p[i - 1] * p[k] * p[j]);
        }
        return dp[i][j];
    }
    int Algo(vector<int> v, int n)
    {
        int i = 1, j = n - 1;
        return helper(v, i, j);
    }

    int main()
    {
        int n;
        cin >> n;
        vector<int> v(n);
        for (int i = 0; i < n; i++)
        {
            cin >> v[i];
        }

        cout << Algo(v, n) << endl;
    }

```

Knapsack

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

Solution: - we will either choose a weight or we won't. - so we maintain an array and update its value accordingly to find the max number of weights that can be stored, by comparing its weight and value.

Code::

```

#include <iostream>
#include <climits>

using namespace std;

int knapsack(int *weights, int *values, int n, int maxWeight)
{
    int dp[maxWeight + 1] = {0};

```

```

    for (int i = 0; i < n; i++)
    {
        for (int j = maxWeight; j >= weights[i]; j--)
        {
            dp[j] = max(dp[j], values[i] + dp[j - weights[i]]);
        }
    }
    return dp[maxWeight];
}

int main()
{
    int n;
    cin >> n;

    int *weights = new int[n];
    int *values = new int[n];

    for (int i = 0; i < n; i++)
    {
        cin >> weights[i];
    }

    for (int i = 0; i < n; i++)
    {
        cin >> values[i];
    }

    int maxWeight;
    cin >> maxWeight;

    cout << knapsack(weights, values, n, maxWeight) << endl;

    delete[] weights;
    delete[] values;
}

```

Practice Problems

Problem 1

Problem 2

Problem 3

Problem 4

Problem 5

CHAPTER 6: GRAPH ALGORITHMS

As we know, Graphs can either be constructed using an adjacency matrix (2D array) or an adjacency list. For the algorithms, I will be using an Adjacency List with the help of vectors in c++. Let us start by looking at some of the basic algorithms.

Breadth First Search (BFS)

Breadth-first search is one of the simplest algorithms for searching a graph. Given a graph $G = (V, E)$ and a distinguished source vertex s , breadth-first search systematically explores the edges of G to “discover” every vertex that is reachable from s . It computes the distance (smallest number of edges) from s to each reachable vertex.

To keep track of progress, breadth-first search colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is discovered the first time it is encountered during the search, at which time it becomes nonwhite. Gray and black vertices, therefore, have been discovered, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner. If $(u, v) \in E$ and vertex u is black, then vertex v is either gray or black; that is, all vertices adjacent to black vertices have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

Applications of DFS

- Find the shortest path from a source to other vertices in an unweighted graph.
- Find all connected components in an undirected graph in $O(n + m)$ time.
- Finding a solution to a problem or a game with the least number of moves, if each state of the game can be represented by a vertex of the graph, and the transitions from one state to the other are the edges of the graph.
- Finding the shortest cycle in a directed unweighted graph: Start a breadth-first search from each vertex. As soon as we try to go from the current vertex back to the source vertex, we have found the shortest cycle containing the source vertex. At this point we can stop the BFS, and start a new BFS from the next vertex. From all such cycles (at most one from each BFS) choose the shortest.

Implementation

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

//directed unweighted graph
```

```

void bfs(int i, bool *visited, vector<vector<int>> graph)
{
    queue<int> q;
    q.push(i);
    visited[i] = true;

    while (!q.empty())
    {
        int node = q.front();
        cout << node << ' ';
        q.pop();
        visited[node] = true;

        for (int x = 0; x < graph[node].size(); x++)
        {
            if (!visited[graph[node][x]])
            {
                q.push(graph[node][x]);
                visited[graph[node][x]] = true;
            }
        }
    }
}

int main()
{
    int n, m;

    cin >> n >> m; //vertices and edges

    vector<vector<int>> graph(n + 1);
    bool *visited = new bool[n];
    for (int i = 0; i < n; i++)
    {
        visited[i] = false;
    }

    for (int i = 0; i < m; i++)
    {
        int a, b;
        cin >> a >> b;

        graph[a].push_back(b);
        graph[b].push_back(a);
    }
}

```

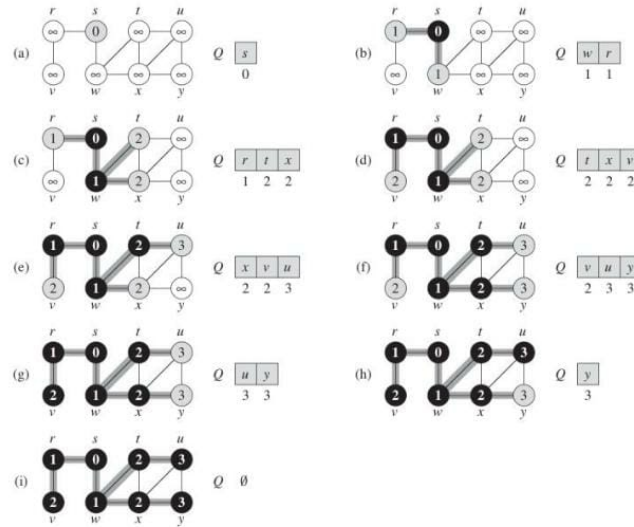
```

for (int i = 1; i <= n; i++)
{
    if (!visited[i])
    {
        bfs(i, visited, graph);
    }
}

//PRINTING THE GRAPH:
for (int i = 1; i < graph.size(); i++)
{
    for (int j = 0; j < graph[i].size(); j++)
    {
        cout << i << "->" << graph[i][j] << ' ';
    }
    cout << endl;
}

delete[] visited;
}

```



Analysis

Let graph $G = (V, E)$

The operations of enqueueing and dequeuing take $O(1)$ time, and so the total

time devoted to queue operations is $O(V)$. Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all the adjacency lists is $\theta(E)$, the total time spent in scanning adjacency lists is $O(E)$. The overhead for initialization is $(O(V))$, and thus the total running time of the BFS procedure is $O(E + V)$. Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G .

Practice Problems

Problem 1

Problem 2

Problem 3

Problem 4

Depth First Search (DFS)

The strategy followed by depth-first search is, as its name implies, to search “deeper” in the graph whenever possible. Depth-first search explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v ’s edges have been explored, the search “backtracks” to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex.

Applications of DFS

- Find any path in the graph from source vertex u to all vertices.
- Find lexicographical first path in the graph from source u to all vertices
- Topological sorting
- Check if a vertex in a tree is an ancestor of some other vertex
- Find strongly connected components in a directed graph

Algorithm

The idea behind DFS is to go as deep into the graph as possible, and backtrack once you are at a vertex without any unvisited adjacent vertices.

It is very easy to describe / implement the algorithm recursively: We start the search at one vertex. After visiting a vertex, we further perform a DFS for each adjacent vertex that we haven’t visited before. This way we visit all vertices that are reachable from the starting vertex.

Implementation:

```

#include <iostream>
#include <bits/stdc++.h>
using namespace std;

void dfs(vector<vector<int>> graph, int i, bool *visited)
{
    visited[i] = true;
    cout << i << endl;

    for (int x = 0; x < graph[i].size(); x++)
    {
        if (!visited[graph[i][x]])
        {
            dfs(graph, graph[i][x], visited);
        }
    }
}

int main()
{
    int n, m;

    cin >> n >> m; //vertices and edges

    vector<vector<int>> graph(n + 1);
    bool *visited = new bool[n];
    for (int i = 0; i < n; i++)
    {
        visited[i] = false;
    }

    for (int i = 0; i < m; i++)
    {
        int a, b;
        cin >> a >> b;

        graph[a].push_back(b);
        graph[b].push_back(a);
    }

    for (int i = 1; i <= n; i++)
    {
        if (!visited[i])
        {
            dfs(graph, i, visited);
        }
    }
}

```

```

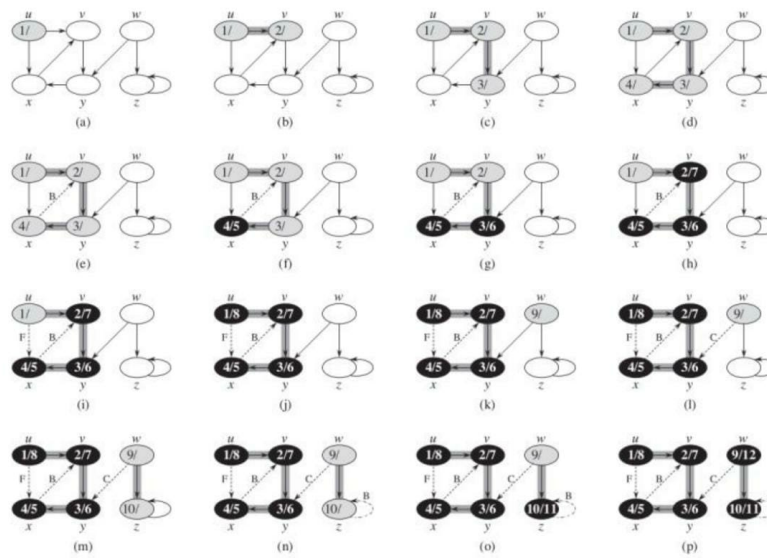
}

//PRINTING THE GRAPH:
for (int i = 1; i < graph.size(); i++)
{
    for (int j = 0; j < graph[i].size(); j++)
    {
        cout << i << "->" << graph[i][j] << ' ';
    }
    cout << endl;
}

delete[] visited;
}

```

Operation of DFS depicted pictorially:



Complexity: $O(V + E)$

Practice Problems

Problem 1

Problem 2

Problem 3

Problem 4

Problem 5

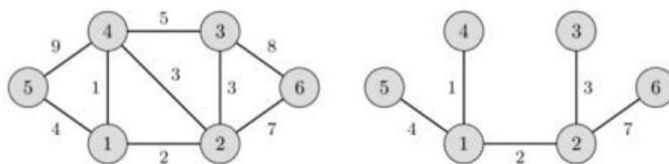
Minimum Spanning Trees

Given a weighted, undirected graph G with n vertices and m edges. You want to find a spanning tree of this graph which connects all vertices and has the least weight (i.e. the sum of weights of edges is minimal). A spanning tree is a set of edges such that any vertex can reach any other by exactly one simple path. The spanning tree with the least weight is called a minimum spanning tree.

Properties of an MST

- A minimum spanning tree of a graph is unique, if the weight of all the edges are distinct. Otherwise, there may be multiple minimum spanning trees. (Specific algorithms typically output one of the possible minimum spanning trees).
- Minimum spanning tree is also the tree with minimum product of weights of edges. (It can be easily proved by replacing the weights of all edges with their logarithms)
- In a minimum spanning tree of a graph, the maximum weight of an edge is the minimum possible from all possible spanning trees of that graph. (This follows from the validity of Kruskal's algorithm).
- The maximum spanning tree (spanning tree with the sum of weights of edges being maximum) of a graph can be obtained similarly to that of the minimum spanning tree, by changing the signs of the weights of all the edges to their opposite and then applying any of the minimum spanning tree algorithm.

MIT 6.034



Prim's Algorithm

This algorithm was originally discovered by the Czech mathematician Vojtěch Jarník in 1930. However this algorithm is mostly known as Prim's algorithm after the American mathematician Robert Clay Prim, who rediscovered and republished it in 1957. Additionally Edsger Dijkstra published this algorithm in 1959.

Algorithm

Here we describe the algorithm in its simplest form. The minimum spanning tree is built gradually by adding edges one at a time. At first the spanning

tree consists only of a single vertex (chosen arbitrarily). Then the minimum weight edge outgoing from this vertex is selected and added to the spanning tree. After that the spanning tree already consists of two vertices. Now select and add the edge with the minimum weight that has one end in an already selected vertex (i.e. a vertex that is already part of the spanning tree), and the other end in an unselected vertex. And so on, i.e. every time we select and add the edge with minimal weight that connects one selected vertex with one unselected vertex. The process is repeated until the spanning tree contains all vertices (or equivalently until we have $n - 1$ edges).

In the end the constructed spanning tree will be minimal. If the graph was originally not connected, then there doesn't exist a spanning tree, so the number of selected edges will be less than $n - 1$.

Implementation:

```
#include <bits/stdc++.h>
using namespace std;

void prims(vector<vector<pair<int, int>>> &graph, int n)
{
    int v = 1, w = 0, sum = 0, cnt = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    int visited[n + 1] = {0};

    pq.push({w, v});

    while (!pq.empty())
    {
        pair<int, int> top = pq.top();
        pq.pop();

        v = top.second;
        w = top.first;

        if (visited[v] == 1)
            continue;
        visited[v] = 1;

        sum += w;
        cnt++;

        cout << v << " " << w << "\n";

        if (cnt >= n)
            break;
    }
}
```

```

        for (int i = 0; i < graph[v].size(); i++)
        {
            int u = graph[v][i].second, w = graph[v][i].first;

            if (visited[u] == 1)
                continue;

            pq.push({w, u});
        }
    }

    cout << sum << "\n";
}

int main()
{
    int n, m;
    cin >> n >> m;

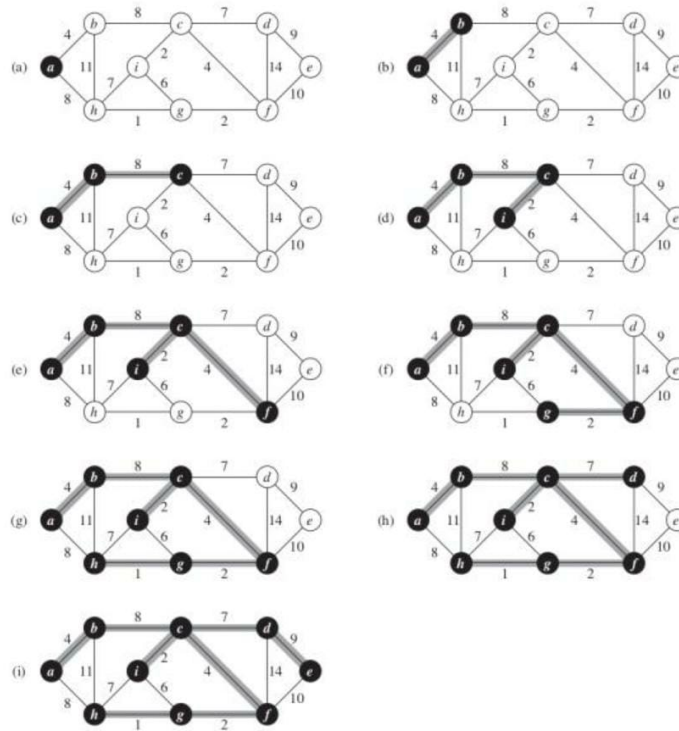
    vector<vector<pair<int, int>>> graph(n + 1);

    for (int i = 0; i < m; i++)
    {
        int u, v, w;
        cin >> u >> v >> w;
        graph[u].push_back({w, v});
        graph[v].push_back({w, u});
    }

    prims(graph, n);
}

```

Complexity: $O(m * n)$



Kruskal's Algorithm

This algorithm was described by Joseph Bernard Kruskal, Jr. in 1956.

Kruskal's algorithm initially places all the nodes of the original graph isolated from each other, to form a forest of single node trees, and then gradually merges these trees, combining at each iteration any two of all the trees with some edge of the original graph. Before the execution of the algorithm, all edges are sorted by weight (in non-decreasing order). Then begins the process of unification: pick all edges from the first to the last (in sorted order), and if the ends of the currently picked edge belong to different subtrees, these subtrees are combined, and the edge is added to the answer. After iterating through all the edges, all the vertices will belong to the same sub-tree, and we will get the answer.

Implementation:

```
#include <bits/stdc++.h>
using namespace std;

int find(vector<int> &parent, int u)
{
    if (parent[u] != u)
```

```

    {
        parent[u] = find(parent, parent[u]);
    }
    return parent[u];
}

void Union(vector<int> &rank, vector<int> &parent, int rootu, int rootv)
{
    if (rank[rootu] < rank[rootv])
    {
        parent[rootu] = rootv;
    }
    else if (rank[rootv] < rank[rootu])
    {
        parent[rootv] = rootu;
    }
    else
    {
        parent[rootv] = rootu;
        rank[rootu]++;
    }
}

void kruskal(vector<pair<int, pair<int, int>>> &edges, int n)
{
    vector<int> parent(n + 1);
    vector<int> rank(n + 1);

    for (int i = 1; i <= n; i++)
    {
        parent[i] = i;
        rank[i] = 0;
    }

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

    int sum = 0, cnt = 0;

    for (int i = 0; i < edges.size(); i++)
    {
        int w = edges[i].first, u = edges[i].second.first, v = edges[i].second.second;

        int rootu = find(parent, u);
        int rootv = find(parent, v);

        if (rootu != rootv)

```



```

        {
            Union(rank, parent, rootu, rootv);
            sum += w;
            cnt++;

            cout << u << " " << v << " " << w << "\n";

            if (cnt >= n - 1)
                break;
        }
    }

    cout << sum << "\n";
}

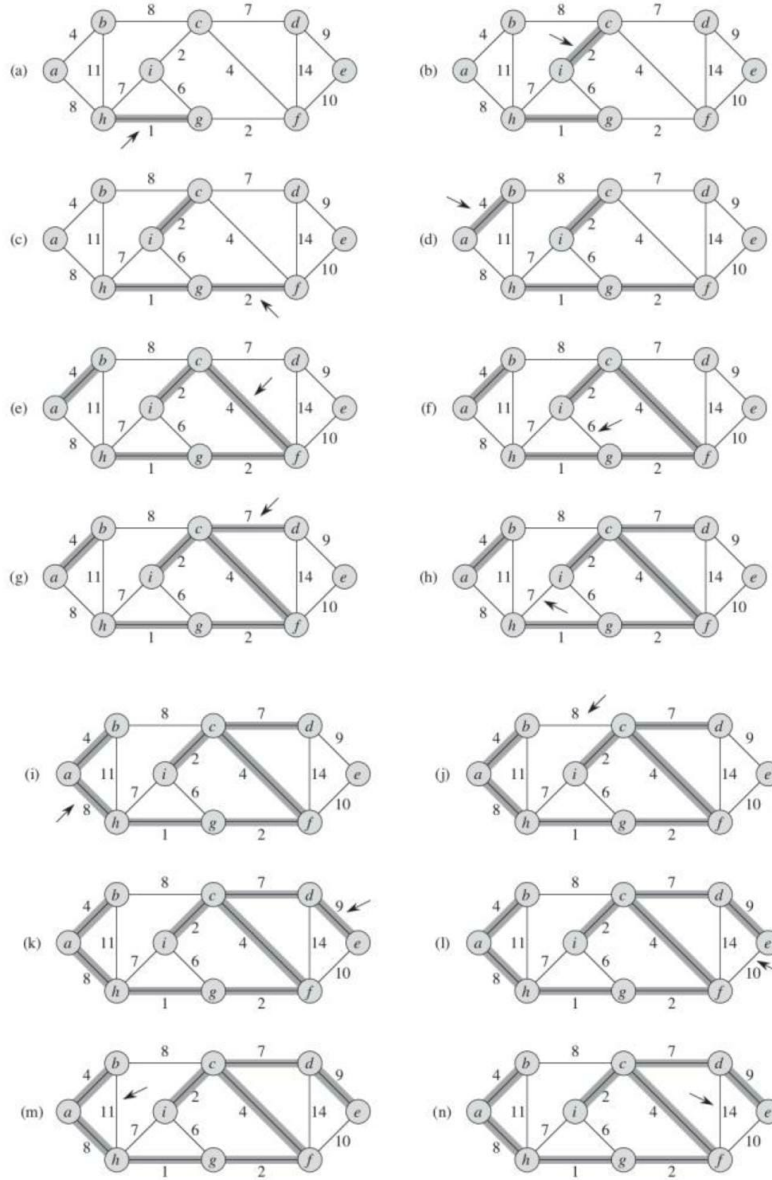
int main()
{
    int n, m;
    cin >> n >> m;

    vector<pair<int, pair<int, int>>> edges;

    for (int i = 0; i < m; i++)
    {
        int u, v, w;
        cin >> u >> v >> w;
        edges.push_back({w, {u, v}});
    }

    kruskal(edges, n);
}

```



Shortest Path in DAGs

Q. We have a directed acyclic graph (no cycles will be formed), with given edge weights and we need to find the shortest path from a source node to all the other reachable nodes of the graph.

The main idea for the algorithm is to create a topological ordering for the graph

and calculate the minimum cost/ shortest path between 2 edges.

We will set the initial distance of source node = 0, and all the other nodes to be infinity. And then while traversing the graph and by using dynamic programming, we will update the values if we can find a better path to a given node.

Implementation:

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

// 1 is the source node

void printShortestPaths(int *a, int n)
{
    for (int i = 1; i < n + 1; i++)
    {
        cout << a[i] << " ";
    }
    cout << endl;
}

void Algo(bool *visited, vector<vector<pair<int, int>>> &graph, int *a)
{
    a[1] = 0;
    visited[1] = true;
    queue<int> q;
    q.push(1);

    while (!q.empty())
    {
        int node = q.front();
        q.pop();
        visited[node] = true;
        for (int i = 0; i < graph[node].size(); i++)
        {
            int x = graph[node][i].first;
            int y = graph[node][i].second;
            if (!visited[x])
            {
                q.push(x);

                a[x] = min(a[x], a[node] + y); // DYNAMICALLY UPDATING THE VALUES
            }
        }
    }
}
```

```

    }
}

int main()
{
    // n: number of vertices; m: number of edges
    int n, m;
    cin >> n >> m;

    vector<vector<pair<int, int>>> graph(n + 1);
    int *a = new int[n + 1];
    bool *visited = new bool[n + 1];
    for (int i = 0; i < n + 1; i++)
    {
        a[i] = INT_MAX;
        visited[i] = false;
    }

    for (int i = 0; i < m; i++)
    {
        int s, e, w;
        cin >> s >> e >> w;
        pair<int, int> p(e, w);
        graph[s].push_back(p);
    }

    Algo(visited, graph, a);
    printShortestPaths(a, n);

    delete[] a;
    delete[] visited;
}

```

Longest path in a general graph is an NP-Hard problem but can be found easily for DAG's. We multiply all the edge weights by -1, and apply the same algorithm as for the shortest path. Then we again multiply by -1 while returning the answer!

Dijkstra's Algorithm

You are given a directed or undirected weighted graph with n vertices and m edges. The weights of all edges are non-negative. You are also given a starting vertex s . This article discusses finding the lengths of the shortest paths from a starting vertex s to all other vertices, and output the shortest paths themselves.

This problem is also called *single-source shortest paths problem*.

Algorithm

Here is an algorithm described by the Dutch computer scientist Edsger W. Dijkstra in 1959.

Let's create an array $d[]$ where for each vertex v we store the current length of the shortest path from s to v in $d[v]$. Initially $d[s] = 0$, and for all other vertices this length equals infinity. In the implementation a sufficiently large number (which is guaranteed to be greater than any possible path length) is chosen as infinity.

$$d[v] = \infty$$

$$v \neq s$$

In addition, we maintain a Boolean array $u[]$ which stores for each vertex v whether it's marked. Initially all vertices are unmarked: $u[v] = false$

The Dijkstra's algorithm runs for n iterations. At each iteration a vertex v is chosen as unmarked vertex which has the least value $d[v]$:

Evidently, in the first iteration the starting vertex s will be selected.

The selected vertex v is marked. Next, from vertex v relaxations are performed: all edges of the form (v, to) are considered, and for each vertex to the algorithm tries to improve the value $d[to]$. If the length of the current edge equals len , the code for relaxation is:

$$d[to] = \min(d[to], d[v] + len)$$

After all such edges are considered, the current iteration ends. Finally, after n iterations, all vertices will be marked, and the algorithm terminates. We claim that the found values $d[v]$ are the lengths of shortest paths from s to all vertices v .

Note that if some vertices are unreachable from the starting vertex s , the values $d[v]$ for them will remain infinite. Obviously, the last few iterations of the algorithm will choose those vertices, but no useful work will be done for them. Therefore, the algorithm can be stopped as soon as the selected vertex has infinite distance to it.

Implementation:

```
#include <bits/stdc++.h>
using namespace std;
#define INF 1e9

vector<int> dijkstra(vector<vector<pair<int, int>>> &graph, int s)
{
    int visited[graph.size()] = {0};

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
```

```

vector<int> dist(graph.size(), INF);

dist[s] = 0;
pq.push({dist[s], s});

while (!pq.empty())
{
    pair<int, int> top = pq.top();
    pq.pop();

    int u = top.second;

    if (visited[u] == 1)
        continue;
    visited[u] = 1;

    for (int i = 0; i < graph[u].size(); i++)
    {
        int v = graph[u][i].second, w = graph[u][i].first;

        if (visited[v] == 1)
            continue;

        if (dist[v] > dist[u] + w)
        {
            dist[v] = dist[u] + w;
            pq.push({dist[v], v});
        }
    }
}

return dist;
}

int main()
{
    int n, m;
    cin >> n >> m;

    vector<vector<pair<int, int>>> graph(n + 1);

    for (int i = 0; i < m; i++)
    {
        int u, v, w;

```

```

        cin >> u >> v >> w;
        graph[u].push_back({w, v});
    }

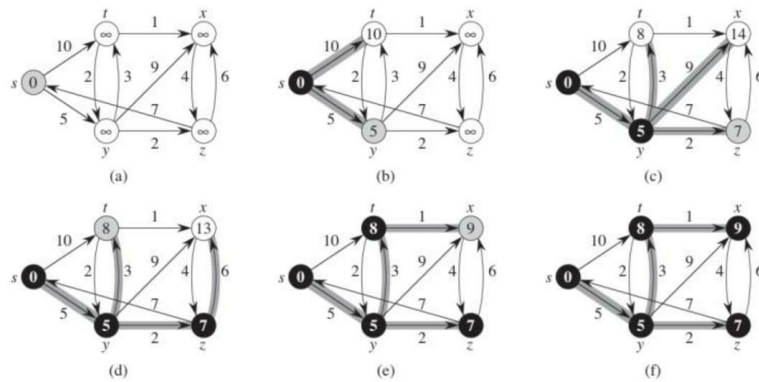
    int s;
    cin >> s;

    vector<int> dist = dijkstra(graph, s);

    for (int i = 1; i <= n; i++)
    {
        cout << "Distance from " << s << "to " << i << " is: " << dist[i] << "\n";
    }
}

```

Complexity: $O(n^2 + m)$



Practice Problems

Problem 1

Problem 2

Problem 3

Problem 4

Floyd Warshall Algorithm

Given a directed or an undirected weighted graph G with n vertices. The task is to find the length of the shortest path dij between each pair of vertices i and j .

The graph may have negative weight edges, but no negative weight cycles.

If there is such a negative cycle, you can just traverse this cycle over and over, in each iteration making the cost of the path smaller. So you can make certain paths arbitrarily small, or in other words that shortest path is undefined. That automatically means that an undirected graph cannot have any negative weight edges, as such an edge forms already a negative cycle as you can move back and forth along that edge as long as you like.

This algorithm can also be used to detect the presence of negative cycles. The graph has a negative cycle if at the end of the algorithm, the distance from a vertex v to itself is negative.

This algorithm has been simultaneously published in articles by Robert Floyd and Stephen Warshall in 1962. However, in 1959, Bernard Roy published essentially the same algorithm, but its publication went unnoticed.

Algorithm

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices $0, 1, 2, \dots, k-1$ as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

1. k is not an intermediate vertex in shortest path from i to j . We keep the value of $dist[i][j]$ as it is.
2. k is an intermediate vertex in shortest path from i to j . We update the value of $dist[i][j]$ as $dist[i][k] + dist[k][j]$ if $dist[i][j] > dist[i][k] + dist[k][j]$

Implementation:

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

typedef long long int ll;
ll INF = 1e9 + 13;

bool check(vector<vector<ll>> &dp, int i, int j, int k)
{
    if (dp[i][j] > dp[i][k] + dp[k][j] && dp[i][k] != INF && dp[k][j] != INF)
    {
        return true;
    }
    return false;
}
```



```

int main()
{
    cout << "Enter the number of nodes and edges of graph: " << endl;
    int n, m;
    cin >> n >> m;

    vector<vector<ll>> v(n, vector<ll>(n, INF));

    cout << "Enter the edges and their weights: " << endl;
    for (int i = 0; i < m; i++)
    {
        int s, e, w;
        cin >> s >> e >> w;
        v[s][e] = w;
    }

    cout << "Enter the 2 nodes between which you want the shortest distance: " << endl;
    int x, y;
    cin >> x >> y;

    vector<vector<ll>> dp(n, vector<ll>(n, INF));
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            dp[i][j] = v[i][j];
        }
    }

    // iterating the value of k:
    for (int k = 0; k < n; k++)
    {
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (check(dp, i, j, k))
                {
                    dp[i][j] = dp[i][k] + dp[j][k];
                }
            }
        }
    }

    cout << "Answer: " << dp[x][y] << endl;
}

```

```

    return 0;
}

```

Complexity: $O(n^3)$

Toposort

Q. You are given a directed graph with n vertices and m edges. You have to number the vertices so that every edge leads from the vertex with a smaller number assigned to the vertex with a larger one.

In other words, you want to find a permutation of the vertices (*topological order*) which corresponds to the order defined by all edges of the graph.

Topological order can be non-unique (for example, if the graph is empty; or if there exist three vertices a, b, c for which there exist paths from a to b and from a to c but not paths from b to c or from c to b).

Topological order may not exist at all if the graph contains cycles (because there is a contradiction: there is a path from a to b and vice versa).

A common problem in which topological sorting occurs is the following. There are n variables with unknown values. For some variables we know that one of them is less than the other. You have to check whether these constraints are contradictory, and if not, output the variables in ascending order (if several answers are possible, output any of them). It is easy to notice that this is exactly the problem of finding topological order of a graph with n vertices.

Algorithm

To solve this problem we will use DFS.

Let's assume that the graph is acyclic, i.e. there is a solution. What does the depth-first search do? When started from some vertex v , it tries to run along all edges outgoing from v . It fails to run along the edges for which the opposite ends have been visited previously, and runs along the rest of the edges and starts from their ends.

Thus, by the time of the call $dfs(v)$ is ended, all vertices that are reachable from v either directly (via one edge) or indirectly are already visited by the search. Therefore, if at the time of exit from $dfs(v)$ we add vertex v to the beginning of a certain list, in the end this list will store a topological ordering of all vertices.

These explanations can also be presented in terms of time of exit from DFS routine. Exit time for vertex v is the time at which $dfs(v)$ finished work (the times can be numbered from 1 to n). It is easy to understand that exit time of any vertex v is always greater than exit time of any vertex reachable from it (since they were visited either before the call $dfs(v)$ or during it). Thus, the desired topological ordering is sorting vertices in descending order of their exit times.

Implementation:

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int n, m;
    cin >> n >> m;

    vector<vector<int>>> graph(n + 1);
    vector<int> indegree(n + 1);
    for (int i = 1; i < n + 1; i++)
    {
        indegree[i] = 0;
    }

    for (int i = 0; i < m; i++)
    {
        int a, b;
        cin >> a >> b;
        graph[a].push_back(b);
        indegree[b]++;
    }

    queue<int> q;
    for (int i = 1; i < indegree.size(); i++)
    {
        if (indegree[i] == 0)
        {
            q.push(i);
        }
    }

    while (!q.empty())
    {
        int node = q.front();
        q.pop();
        cout << node << ' ';

        for (int i = 0; i < graph[node].size(); i++)
        {
            int x = graph[node][i];
            indegree[x]--;
            if (indegree[x] == 0)
            {
                q.push(x);
            }
        }
    }
}
```

```
        {
            q.push(x);
        }
    }
    cout << endl;
}
```

Practice Problems

Problem 1

Problem 2