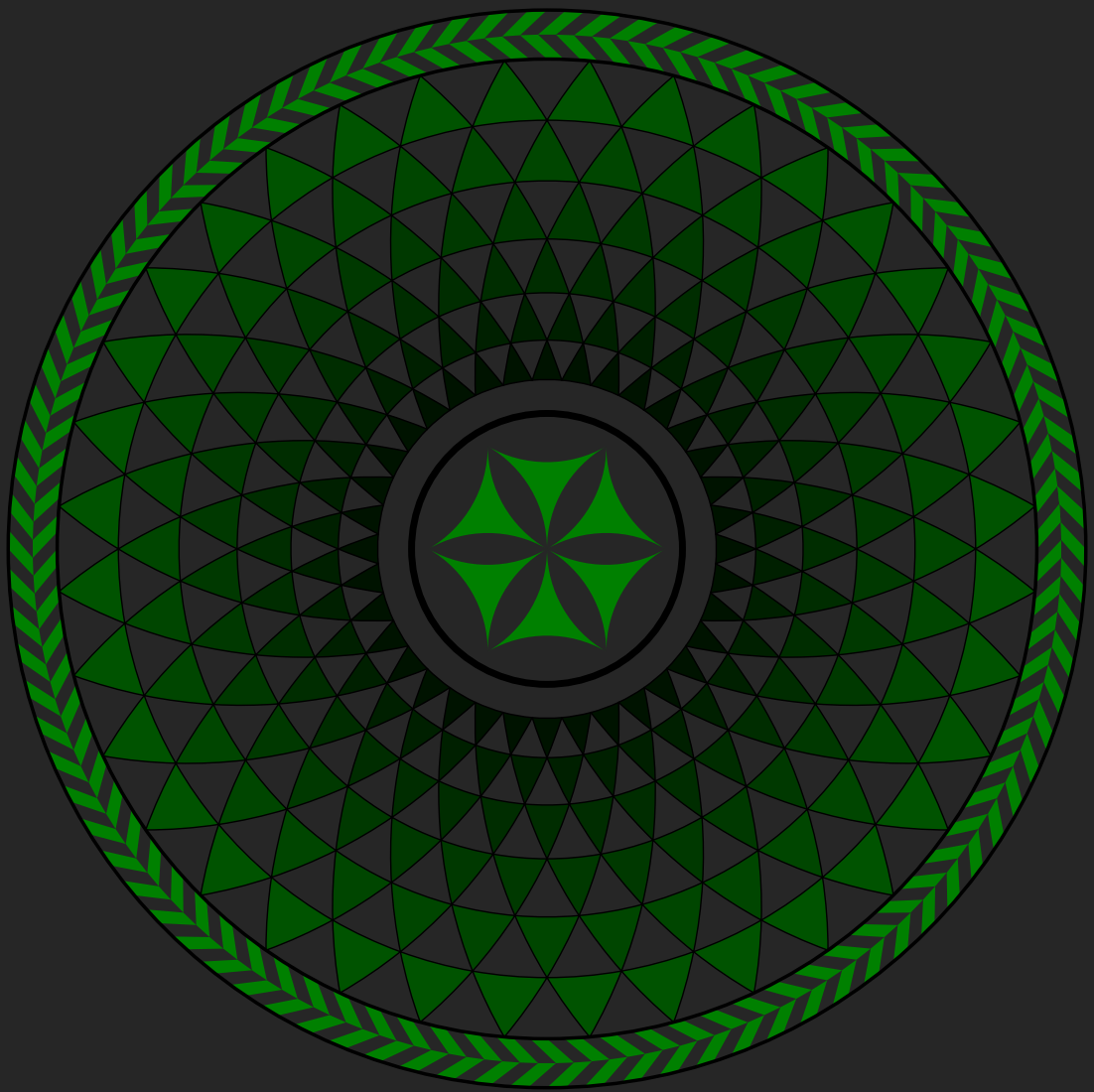
Competitive Programming Notebook

Salil Gokhale

Draft April 20, 2019



This work consists of this book, and it's source code.

You can read this book online by scanning this QR code:

Contents

3 . 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6 4 3 3 8 3 2 7 9 5 0 2 8 8 4 1 9 7 1 6
9 3 9 9 3 7 5 1 0 5 8 2 0 9 7 4 9 4 4 5 9 2 3 0 7 8 1 6 4 0 6 2 8 6 2 0 8 9 9 8 6 2 8 0 3 4 8
2 5 3 4 2 1 1 7 0 6 7 9 8 2 1 4 8 0 8 6 5 1 3 2 8 2 3 0 6 6 4 7 0 9 3 8 4 4 6 0 9 5 5 0 5 8 2 2 3
1 7 2 5 3 5 9 4 0 8 1 2 8 4 8 1 1 1 7 4 5 0 2 8 4 1 0 2 7 0 1 9 3 8 5 2 1 1 0 5 5 5 9 6 4 4 6 2 2 9
4 8 9 5 4 9 3 0 3 8 1 9 6 4
4 2 8 8 1 0 9 7 5 6 6
5 9 3 3 4 4 6 1 2 8
4 7 5 6 4 8 2 3 3 7
8 6 7 8 3 1 6 5 2
7 1 2 0 1 9 0 9 1
4 5 6 4 8 5 6 6
9 2 3 4 6 0 3 4
8 6 1 0 4 5 4 3
2 6 6 4 8 2 1 3
3 9 3 6 0 7 2 6
0 2 4 9 1 4 1 2 7
3 7 2 4 5 8 7 0 0 6
6 0 6 3 1 5 5 8 8 1
7 4 8 8 1 5 2 0 9 2
0 9 6 2 8 2 9 2 5 4 0
9 1 7 1 5 3 6 4 3 6 7
8 9 2 5 9 0 3 6 0 0 1 1
3 3 0 5 3 0 5 4 8 8 2 0
4 6 6 5 2 1 3 8 4 1 4 6 9
5 1 9 4 1 5 1 1 6 0 9 4 3
3 0 5 7 2 7 0 3 6 5 7 5 9 5
9 1 9 5 3 0 9 2 1 8 6 1 1 7
3 8 1 9 3 2 6 1 1 7 9 3 1 0 5
1 1 8 5 4 8 0 7 4 4 6 2 3 7 9
9 6 2 7 4 9 5 6 7 3 5 1 8 8 5 7
5 2 7 2 4 8 9 1 2 2 7 9 3 8
1 8 3 0 1 1 9 4 9 1

π for good luck!

Part I

Basic techniques

Chapter 1

Code Template

Here is my code template for competitive programming:

```
#include<bits/stdc++.h>
using namespace std;
//macro for debugging. Run watch(n) if you want to debug n;
#define watch(x) cerr << "\n" << (#x) << " is " << (x) << endl
const double PI = 3.14159265358979323846; //value of pi
const int MOD = 1000000007; //used in many problems

/*
$alil03
URL: url of problem
Solution Begins here
*/

int main()
{
    //FAST I/O
        ios_base::sync_with_stdio(false);
        cin.tie(0);
        cout.tie(0);
}
```

You can remove the code comments and use it.

My IDE of choice: Sublime Text 3

If you are also using Sublime Text 3, please consider using the Getcode package which can be found here: `https://packagecontrol.io/packages/GetCode`. It will make your life easier.

Chapter 2

C++ Tricks

2.1 Special functions

2.1.1 Copy Elements

```
copy_n(ar, 6, ar1); // copy 6 elements from ar to ar1
```

2.1.2 Ceiling Division $\lceil \frac{x}{y} \rceil$

```
int ceilingdivision(int x,int y)
{
    return (x + y - 1) / y;
}
```

2.1.3 Ceiling and Floor round-off

```
//Works only for floats
cout << "Floor is : " << floor(2.3); //2
cout << "Floor is : " << floor(-2.3); //-3
cout << " Ceil is : " << ceil(2.3); //3
cout << " Ceil is : " << ceil(-2.3); //-2
```

2.1.4 Use `emplace back()`

We can use emplace back instead of push back like this:

```
myvector.emplace_back(4); //adds 4 to end of vector
```

2.1.5 Iota

It fills a vector (or some container) with increasing values starting with x

```
iota(v.begin(), v.end(), x) //fills vector with increasing values
```

2.1.6 Insert value in middle of vector

```
v.insert(1, 42); // Insert 42 after the first index(second value)
```

2.1.7 Accurate value of π

```
const double pi = 2 * acos(0.0)
```

2.1.8 GCD of Two Numbers

```
__gcd(value1, value2); //gcd of two numbers.
```

2.1.9 Range Queries

```
// are all of the elements positive?
all_of(first, first+n, ispositive());

// is there at least one positive element?
any_of(first, first+n, ispositive());

// are none of the elements positive?
none_of(first, first+n, ispositive());
```

2.2 Input and output

Input and output is sometimes a bottleneck in the program. The following lines at the beginning of the code make input and output more efficient:

```
ios::sync_with_stdio(0);
cin.tie(0);
```

Sometimes the program should read a whole line from the input, possibly containing spaces. This can be accomplished by using the `getline` function:

```
string s;
getline(cin, s);
```

If the amount of data is unknown, the following loop is useful:

```
while (cin >> x) {
    // code
}
```

This loop reads elements from the input one after another, until there is no more data available in the input.

In some contest systems, files are used for input and output. An easy solution for this is to write the code as usual using standard streams, but add the following lines to the beginning of the code:

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

After this, the program reads the input from the file "input.txt" and writes the output to the file "output.txt".

Chapter 3

Sorting Algorithms

3.1 Problem Statement of Sorting Algorithms

***Problem*:** Given an array that contains n elements, your task is to sort the elements in increasing order.

Example. For example, the array

1	3	8	2	9	2	5	6

will be as follows after sorting:

1	2	2	3	5	6	8	9

3.2 Properties of Sorting Algorithms

3.2.1 Types of Sorting Algorithms

There are two broad types of sorting algorithms: integer sorts and comparison sorts.

Comparison Sorts

Comparison sorts compare elements at each step of the algorithm to determine if one element should be to the left or right of another element.

Comparison sorts are usually more straightforward to implement than integer sorts, but comparison sorts are limited by a lower bound of $\mathcal{O}(n \log n)$, meaning that, on average, comparison sorts cannot be faster than $\mathcal{O}(n \log n)$.

Integer Sorts

Integer sorts are sometimes called counting sorts (though there is a specific integer sort algorithm called counting sort). Integer sorts do not make comparisons. Integer sorts determine for each element x - how many elements are less than x. For example, if there are 14 elements that are less than x, then x will be placed in the 15^{th} slot.

This information is used to place each element into the correct slot immediately, so there is no need to rearrange lists.

3.2.2 Stability

Stable sorting algorithms maintain the relative order of records with equal keys (i.e. values). That is, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list.

3.2.3 Adaptability

It means whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

3.2.4 Memory Usage

Some sorting algorithms are called "in-place". Strictly, an in-place sort needs only $\mathcal{O}(1)$ memory beyond the items being sorted; sometimes $\mathcal{O}(\log n)$ additional memory is considered "in-place".

3.2.5 Computational Complexity

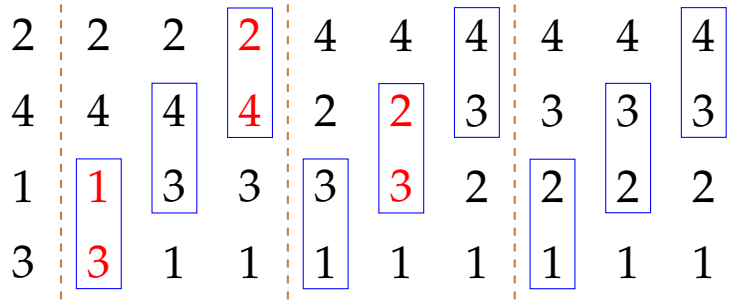
For typical serial sorting algorithms good behaviour is $\mathcal{O}(n \log n)$, with parallel sort in $\mathcal{O}(\log^2 n)$, and bad behaviour is $\mathcal{O}(n^2)$.

3.3 Common $\mathcal{O}(n^2)$ Algorithms

These algorithms are arguably easy to implement but generally not used due to high time complexity.

3.3.1 Bubble Sort

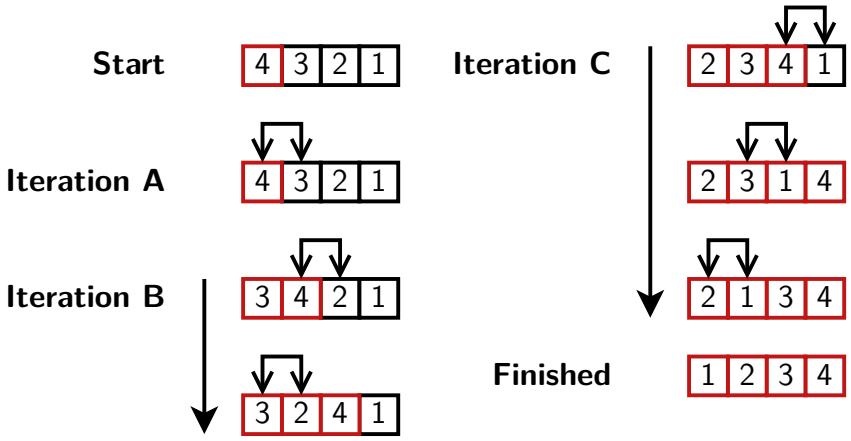
Bubble sort is a simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. It is rarely used to sort large, unordered data sets.



Example of Bubble Sort

3.3.2 Insertion Sort

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and is often used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one.



Insertion Sort Example

3.4 Common $\mathcal{O}(n \log n)$ Algorithms

3.4.1 Merge Sort

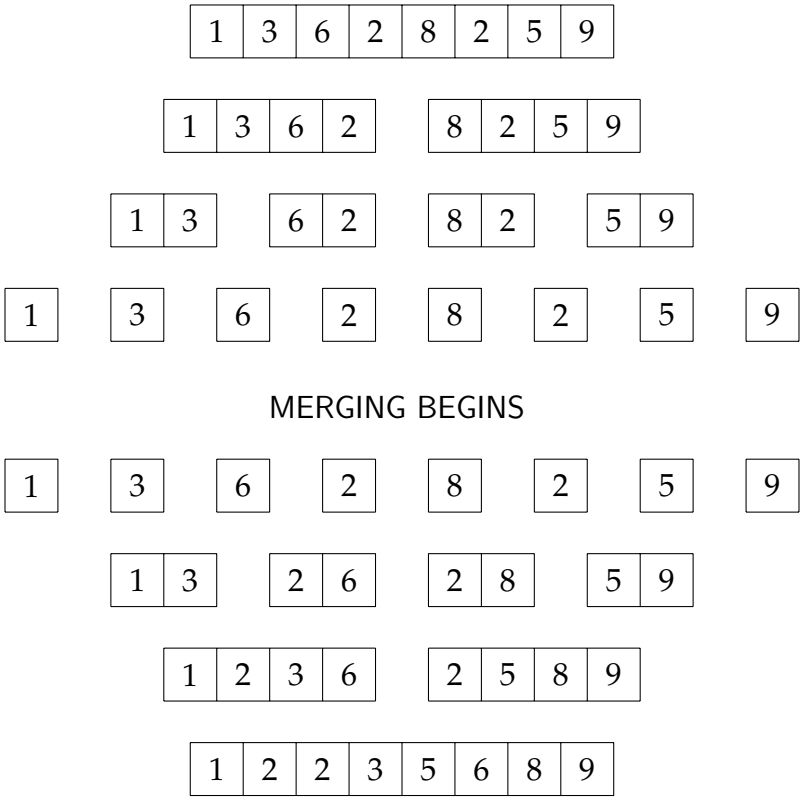
Merge sort sorts a subarray $\texttt{array}[a \ldots b]$ as follows:

1. If $a = b$, do not do anything, because the subarray is already sorted.
2. Calculate the position of the middle element: $k = \lfloor (a+b)/2 \rfloor$.
3. Recursively sort the subarray $\texttt{array}[a \ldots k]$.
4. Recursively sort the subarray $\texttt{array}[k+1 \ldots b]$.

5. *Merge* the sorted subarrays $\texttt{array}[a \ldots k]$ and $\texttt{array}[k+1 \ldots b]$ into a sorted subarray $\texttt{array}[a \ldots b]$.

Merge sort is an efficient algorithm, because it halves the size of the subarray at each step. The recursion consists of $\mathcal{O}(\log n)$ levels, and processing each level takes $O(n)$ time. Merging the subarrays $\texttt{array}[a \ldots k]$ and $\texttt{array}[k+1 \ldots b]$ is possible in linear time, because they are already sorted.

For example, consider sorting the following array:



3.4.2 Quick Sort

Quicksort uses divide and conquer to sort an array. Divide and conquer is a technique used for breaking algorithms down into subproblems, solving the subproblems, and then combining the results back together to solve the original problem. It can be helpful to think of this method as divide, conquer, and combine.

Here are the divide, conquer, and combine steps that quicksort uses:

Divide:

1. Pick a pivot element, $A[q]$. Picking a good pivot is the key for a fast implementation of quicksort; however, it is difficult to determine what a good pivot might be.

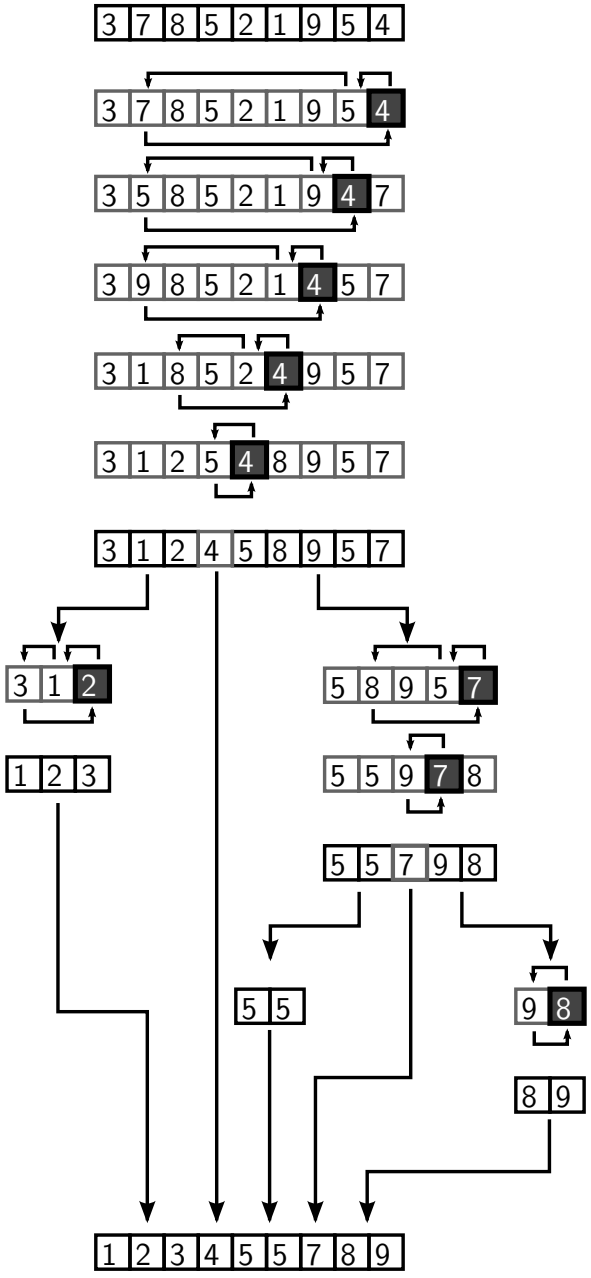
2. Partition, or rearrange, the array into two subarrays: $A[p,...,q-1]$ such that all elements are less than $A[q]$, and $A[q+1,...,r]$ such that all elements are greater than or equal to $A[q]$.

Conquer:

1. Sort the subarrays $A[p,...,q-1]$ and $A[q+1,...,r]$ recursively with quicksort.

Combine:

1. No work is needed to combine the arrays because they are already sorted.



Quicksort Example

3.4.3 Heap Sort

Heapsort is a much more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree.

3.4.4 Counting Sort

Counting sort assumes that each of the n input elements in a list has a key value ranging from 0 to k, for some integer k. For each element in the list, counting sort determines the number of elements that are less than it. Counting sort can use this information to place the element directly into the correct slot of the output array.

Counting sort uses three lists: the input list, $A[0,1,...,n]$, the output list, $B[0,1,...,n]$, and a list that serves as temporary memory, $C[0,1,...,k]$. Note that A and B have n slots (a slot for each element), while C contains k slots (a slot for each key value).

3.5 Comparison of Sorting Algorithms

Algorithm	Best Case	Worst Case	Average Case	Space Usage	Stable?
Bubble Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Yes
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Yes
Merge Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	Yes
Quicksort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	Usually Not
Heapsort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	No
Counting Sort	$\mathcal{O}(k+n)$	$\mathcal{O}(k+n)$	$\mathcal{O}(k+n)$	$\mathcal{O}(k+n)$	Yes

3.6 Sorting in C++

The C++ standard library contains the function `sort` that can be easily used for sorting arrays and other data structures.

There are many benefits in using a library function. First, it saves time because there is no need to implement the function. Second, the library implementation is certainly correct and efficient: it is not probable that a home-made sorting function would be better.

In this section we will see how to use the C++ `sort` function. The following code sorts a vector in increasing order:

```
vector<int> v = {4,2,5,3,5,8,3};
sort(v.begin(),v.end());
```

After the sorting, the contents of the vector will be $[2,3,3,4,5,5,8]$. The default sorting order is increasing, but a reverse order is possible as follows:

```
sort(v.rbegin(),v.rend());
```

An ordinary array can be sorted as follows:

```
int n = 7; // array size
int a[] = {4,2,5,3,5,8,3};
sort(a, a+n);
```

The following code sorts the string s:

```
string s = "monkey";
sort(s.begin(), s.end());
```

Sorting a string means that the characters of the string are sorted. For example, the string "monkey" becomes "ekmnoy".

Integer containers can be sorted in decreasing order as follows:

```
int n = 7; // array size
int a[] = {4,2,5,3,5,8,3};
sort(a, a+n, greater<int>());// greater<int>() as third parameter
// final array is int a[] = {8,5,5,4,3,3,2};
```

Chapter 4

Bit Manipulation

4.1 Tricks

4.1.1 Multiply and divide by 2^i

```
n = n << i;  // Multiply n with 2^i
n = n >> 1;  // Divide n by 2^i
```

4.1.2 Checking if a number is odd or even

```
if (num & 1)
   cout << "ODD";
else
   cout << "EVEN";
```

4.1.3 Swapping of 2 numbers using XOR

This method is fast and doesn't require the use of 3rd variable.

```
a ^= b;
b ^= a;
a ^= b;
```

4.1.4 Compute XOR from 1 to n (direct method)

Algorithm: Compute XOR of numbers from 1 to n

Input: n

Output: XOR of all numbers from 1 to n

1 Find the remainder of n by moduling it with 4.

2 Check,

(I) If rem = 0, then xor will be same as n.

(II) If rem = 1, then xor will be 1.

(II) If rem = 1, then xor will be 1.

(II) If rem = 3 ,then xor will be 0.

```
int computeXOR(int n)
{
    if (n % 4 == 0)
        return n;
    if (n % 4 == 1)
        return 1;
    if (n % 4 == 2)
        return n + 1;
    else
        return 0;
}
```

How does this work?

When we do XOR of numbers, we get 0 as XOR value just before a multiple of 4. This keeps repeating before every multiple of 4.

```
Number Binary-Repr XOR-from-1-to-n
1         1          [0001]
2        10          [0011]
3        11          [0000] <----- We get a 0
4       100          [0100] <----- Equals to n
5       101          [0001]
6       110          [0111]
7       111          [0000] <----- We get 0
8      1000          [1000] <----- Equals to n
9      1001          [0001]
10     1010          [1011]
11     1011          [0000] <------ We get 0
12     1100          [1100] <------ Equals to n
```

4.1.5 Check if a number is a power of 2

```
bool poweroftwo(int x)
{
    return x & (x-1) == 0;
}
```

4.1.6 Change case of English alphabet

```
ch |= ' '; //Upper to Lower
ch &= '_' ; //Lower to Upper
```

4.1.7 Find $\log_2 x$ of integer

```
int logarithm(int x)
{
    int res = 0;
    while (x >>= 1)
        res++;
    return res;
}
```

4.2 Bit Shift

The left bit shift x<<k appends k zero bits to the number, and the right bit shift x>>k removes the k last bits from the number.

Note that x<<k corresponds to multiplying x by 2^k, and x>>k corresponds to dividing x by 2^k rounded down to an integer

4.2.1 Application

Check if k^{th} bit is set

The kth bit of a number is one exactly when x $(1$<<$k)$ is not zero. The following code prints the bit representation of an `int` number x:

```
for (int i = 31; i >= 0; i--)
{
    if (x&(1<<i)) cout << "1"; //check if ith bit is 1
    else cout << "0";
}
```

Set the k^{th} bit

```
x |= (1 < < k) //sets the kth bit of x to one
```

Unset the k^{th} bit

```
x &= ~(1 << k) //unsets the kth bit of x to zero
```

Invert the k^{th} bit

```
x ^= (1 << k) //Inverts the kth bit of x
```

To get the Least Significant Bit

```
T = (S & (-S))
//T is a power of two with only one bit set which is the LSB.
```

To turn on all bits of a number

```
~(x & 0) //x&0 is 0 and ~ inverts all bits to 1
```

To turn on all bits till n

```
S = (1 << n) - 1 //in case n =3 , s = 7 = 8-1
```

Get $n \mod d$ where d is a power of 2

```
// This function will return n % d.
// d must be one of: 1, 2, 4, 8, 16, 32, ...
unsigned int getModulo(unsigned int n, unsigned int d)
{
return ( n & (d - 1) );
}
```

Trivia

The formula x & $(x-1)$ sets the last one bit of x to zero. The formula $x \mid (x-1)$ inverts all the bits after the last one bit.

4.3 C++ Special Functions

The g++ compiler provides the following functions for counting bits:

- `__builtin_clz`(x): the number of zeros at the beginning of the number
- `__builtin_ctz`(x): the number of zeros at the end of the number
- `__builtin_popcount`(x): the number of ones in the number
- `__builtin_parity`(x): the parity (even or odd) of the number of ones

The functions can be used as follows:

```
int x = 5328; // 00000000000000000001010011010000
cout << __builtin_clz(x) << "\n"; // 19
cout << __builtin_ctz(x) << "\n"; // 4
cout << __builtin_popcount(x) << "\n"; // 5
cout << __builtin_parity(x) << "\n"; // 1
```

While the above functions only support `int` numbers, there are also `long long` versions of the functions available with the suffix `ll` like `__builtin_popcountll`(x).

4.4 Set Representation

Every subset of a set $\{0,1,2,\ldots,n-1\}$ can be represented as an n bit integer whose one bits indicate which elements belong to the subset. This is an efficient way to represent sets, because every element requires only one bit of memory, and set operations can be implemented as bit operations.

For example, since `int` is a 32-bit type, an `int` number can represent any subset of the set $\{0,1,2,\ldots,31\}$. The bit representation of the set $\{1,3,4,8\}$ is

$$00000000000000000000000100011010,$$

which corresponds to the number $2^8+2^4+2^3+2^1=282$.

4.4.1 Set implementation

The following code declares an `int` variable x that can contain a subset of $\{0,1,2,\ldots,31\}$. After this, the code adds the elements 1, 3, 4 and 8 to the set and prints the size of the set.

```
int x = 0;
x |= (1<<1);
x |= (1<<3);
x |= (1<<4);
x |= (1<<8);
cout << __builtin_popcount(x) << "\n"; // 4
```

Then, the following code prints all elements that belong to the set:

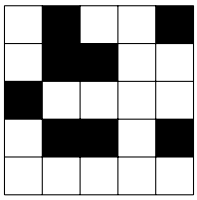
```
for (int i = 0; i < 32; i++) {
    if (x&(1<<i)) cout << i << " ";
}
// output: 1 3 4 8
```

4.5 Example Problems

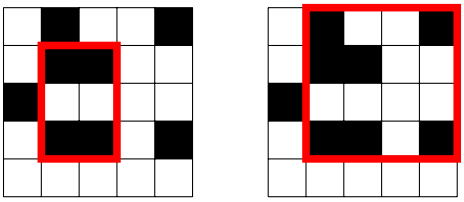
4.5.1 Counting Grids with Black Corners

Problem: Given an $n \times n$ grid whose each square is either black (1) or white (0), calculate the number of subgrids whose all corners are black.

Example. For example, the grid



contains two such subgrids:



Solution

Naive Solution of time complexity $\mathcal{O}(n^3)$:
Go through all $O(n^2)$ pairs of rows and for each pair (a,b) calculate the number of columns that contain a black square in both rows in $O(n)$ time. The following code assumes that $\texttt{color}[y][x]$ denotes the color in row y and column x:

```
int count = 0;
for (int i = 0; i < n; i++) {
    if (color[a][i] == 1 && color[b][i] == 1) count++;
}
```

Then, those columns account for $\texttt{count}(\texttt{count}-1)/2$ subgrids with black corners, because we can choose any two of them to form a subgrid.

Solution

Optimized solution:
Divide the grid into blocks of columns such that each block consists of N consecutive columns. Then, each row is stored as a list of N-bit numbers that describe the colors of the squares. Now we can process N columns at the same time using bit operations. In the following code, $\texttt{color}[y][k]$ represents a block of N colors as bits.

```
int count = 0;
for (int i = 0; i <= n/N; i++) {
    count += __builtin_popcount(color[a][i]&color[b][i]);
}
```

Time Complexity is $O(n^3/N)$ time.

Chapter 5

Brute-Force Algorithms

5.1 Generating Subsets

***Problem*:** Print all the subsets of a set of size n.

Example. For example, the subsets of $\{0,1,2\}$ are $\emptyset$, $\{0\}$, $\{1\}$, $\{2\}$, $\{0,1\}$, $\{0,2\}$, $\{1,2\}$ and $\{0,1,2\}$.

Solution

We can use the bit representation of numbers to generate subsets.
Let's say, set s has n elements. We will use the bits of numbers to show the presence of element in set. If x^{th} bit in number is SET, then x^{th} element in s is present in current subset. We will loop from 0 to $2^n - 1$, and for each number, we will check among the first n bits, the SET bits in it and take corresponding elements. In each iteration, we will have one subset.

```
int arr[n]; //array of n elements
for(int j = 0; j< (1 << n); j++)
{   // (1<<n) is equal to 2^n - 1
    for(int i = 0; i<n; i++)
    {   //checking first n bits
        if(j&(1<<i))
        {   //print the corresponding element
            //first subset is always null set
            cout << arr[i] << " ";
        }
    }
    cout << "\n";
}
```

The time complexity is $\mathcal{O}(n2^n)$ which is exponential.

5.2 Generating Permutations

***Problem*:** Print all the permutations of a set of size n.

Example. For example, the permutations of $\{0,1,2\}$ are $(0,1,2)$, $(0,2,1)$, $(1,0,2)$, $(1,2,0)$, $(2,0,1)$ and $(2,1,0)$.

Solution

We can use the built-in C++ function `next_permutation` which rearranges the array into the next lexicographically greater permutation. It returns 1 if this is possible and 0 otherwise.

```
int arr[n]; //array of n elements
do
{//we use do while loop to also print the original array itself
    for(int i : arr)
    {
        cout << i << " ";
    }
    cout << "\n";
}
while(next_permutation(arr, arr+n));
```

Chapter 6

Searching

6.1 Binary Search

Binary search is the most popular search algorithm. It is efficient and also one of the most commonly used techniques that is used to solve problems.

We will explain this technique using a problem

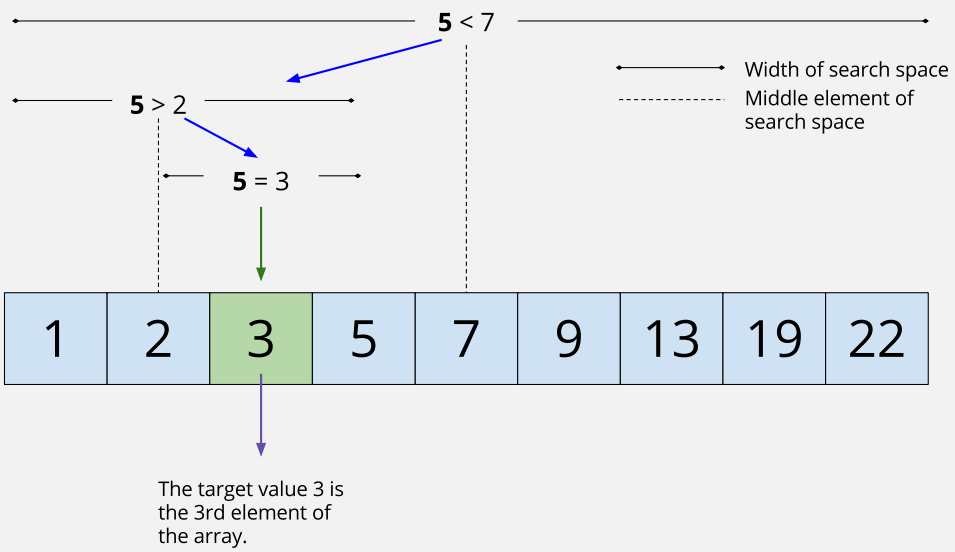
***Problem*:** Given a sorted array $A[]$ of n elements, write a function to search a given element x in $A[]$.

Constraints: The array contains about 10^{10} elements. Each element is less than 10^9

Solution

Naive Approach: A simple approach is to do a linear search(checking every element). The time complexity would be $\mathcal{O}(n)$

Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.
Since we are halving the array at each step, the time complexity becomes $\mathcal{O}(\log n)$




```
int search(int low,int high,int key)
{
   while(low<=high)
   {
     int mid=(low+high)/2;
     if(a[mid]<key)
     {
         low=mid+1; //search upper half
     }
     else if(a[mid]>key)
     {
         high=mid-1; //search lower half
     }
     else
     {
         return mid; //key found
     }
   }
   return -1;       //key not found
 }
```

6.1.1 Binary Search in C++ STL

C++ STL has built-in binary search algorithms for a container

- `binary_search()` : it returns true if value is present in array and false otherwise.
- `lower_bound()` : it returns an iterator pointing to the first element in the array which is greater than or equal to value.
- `upper_bound()` : it returns an iterator pointing to the first element in the array which is greater than value.

6.2 Two Pointer Technique

Two pointer is really an easy and effective technique which is typically used for searching pairs in a sorted array. Binary search is a kind of optimisation on the number of trials taken to reach the optimal position and so is the two pointer technique. The approach relies on the sequence following one specific property on which our pointers can move.

We will explain this technique using an example problem.

***Problem*:** Given a sorted array A, having N integers. You need to find any pair(i,j) having sum as given number X.

Constraints: Array A contains about 10^5 integers with each having values around 10^9.

Solution

Naive Solution: We iterate over every i and j and find out the pair which sums up to X. This approach is to slow because it leads to a time complexity of $\mathcal{O}(n^2)$

2 Pointer Technique: Now let's see how the two pointer technique works. We take two pointers, one representing the first element and other representing the last element of the array, and then we add the values kept at both the pointers. If their sum is smaller than X then we shift the left pointer to right or if their sum is greater than X then we shift the right pointer to left, in order to get closer to the sum. We keep moving the pointers until we get the sum as X or the left pointer meets the right pointer and we don't find the value X. The time complexity is $\mathcal{O}(n)$ if the array is already sorted.

```
bool isPairSum(A[], N, X)
{
    int i = 0; //left pointer
    int j = N - 1; //right pointer

    while (i < j)
    {
        // If we find a pair
        if (A[i] + A[j] == X)
            return true;

        // If sum of elements at current
        // pointers is less, we move towards
        // higher values by doing i++
        else if (A[i] + A[j] < X)
            i++;

        // If sum of elements at current
        // pointers is more, we move towards
        // lower values by doing i++
        else
            j--;
    }
    return false; //i crosses j
}
```

6.3 Sliding window minimum

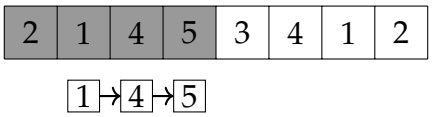
A **sliding window** is a constant-size subarray that moves from left to right through the array. At each window position, we want to calculate some information about the elements inside the window. In this section, we focus on the problem of maintaining the **sliding window minimum**, which means that we should report the smallest value inside each window.

The sliding window minimum can be calculated using a similar idea that we used to calculate the nearest smaller elements. We maintain a queue where each element is larger than the previous element, and the first element always corresponds to the minimum element inside the window. After each window move, we remove elements from the end of the queue until the last queue element is smaller than the new window element, or the queue becomes empty. We also remove the first queue element if it is not inside the window anymore. Finally, we add the new window element to the end of the queue.

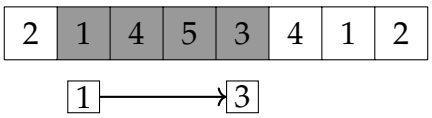
As an example, consider the following array:

2	1	4	5	3	4	1	2

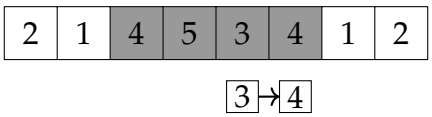
Suppose that the size of the sliding window is 4. At the first window position, the smallest value is 1:



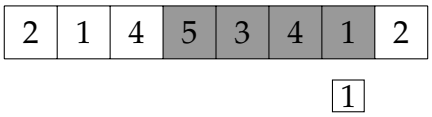
Then the window moves one step right. The new element 3 is smaller than the elements 4 and 5 in the queue, so the elements 4 and 5 are removed from the queue and the element 3 is added to the queue. The smallest value is still 1.



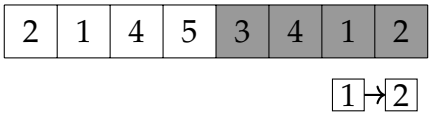
After this, the window moves again, and the smallest element 1 does not belong to the window anymore. Thus, it is removed from the queue and the smallest value is now 3. Also the new element 4 is added to the queue.



The next new element 1 is smaller than all elements in the queue. Thus, all elements are removed from the queue and it will only contain the element 1:



Finally the window reaches its last position. The element 2 is added to the queue, but the smallest value inside the window is still 1.



Since each array element is added to the queue exactly once and removed from the queue at most once, the algorithm works in $O(n)$ time.

Part II

Dynamic Programming

Chapter 7

Common DP Problems

7.1 Largest Sum Contiguous Subarray

Problem: Given an integer array A, find the sum of the contiguous subarray (containing at least one number) which has the largest sum. Formally, the task is to find indices i and j with $1 \leq i \leq j \leq n$, such that the sum

$$\sum_{x=i}^{j} A[x]$$

is as large as possible

Example. Input: `[-2,1,-3,4,-1,2,1,-5,4]`,
Output: 6
Explanation: `[4,-1,2,1]` has the largest sum $= 6$.

Solution

This is called Kadane's algorithm.
The key idea of Kadane's algorithm is to keep a running sum of the integers seen so far and greedily reset that to 0 if the running sum dips below 0. This is because re-starting from 0 is always better than continuing from a negative running sum.

```
int sum = 0, ans = 0; // important, ans must be initialized to 0
for (int i = 0; i < n; i++)
{ // linear scan, O(n)
    sum += A[i]; // we greedily extend this running sum
    ans = max(ans, sum); // we keep the maximum RSQ overall
    if (sum < 0) sum = 0; // but we reset the running sum
}                         // if it ever dips below 0
cout << ans; //ans is the final answer
```

Time Complexity is $\mathcal{O}(n)$

7.2 Rod Cutting

Problem (DP)**:** Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces.

Example. If length of the rod is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6). Price array $=$ `[1,5,8,9,10,17,17,20]`

Solution

We fix the last piece of rod with length i and iterate over i

```
int cutRod(int price[], int n)
{
   int val[n+1]; //dp table with optimal value for rod of length 0...n
   val[0] = 0; //base case

   for (int i = 1; i<=n; i++)
   {
       int max_val = INT_MIN;
       for (int j = 0; j < i; j++)
       {
            max_val = max(max_val, price[j] + val[i-j-1]);
       }
       val[i] = max_val; //answer for i in range 0...n
   }
   return val[n]; //answer
}
```

Time Complexity is $\mathcal{O}(n^2)$

7.3 Longest Increasing Subsequence(LIS)

Problem (DP)**:** Find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order.

Example. Sum of LIS for `[10, 22, 9, 33, 21, 50, 41, 60, 80]` is 6 and LIS is `[10, 22, 33, 50, 60, 80]`.

Solution

Let `arr[0..n-1]` be the input array and `L(i)` be the length of the LIS ending at index i such that `arr[i]` is the last element of the LIS.
Then, `L(i)` can be recursively written as:
`L(i) = 1 + max(L(j))` where $0 \leq j \leq i$ and `arr[j] < arr[i]`; or
L(i) $= 1$, if no such j exists.
To find the LIS for a given array, we need to return `max(L(i))` where $0 \leq i \leq n$.

```
int dp[n] = {0}; //dp table
for(int i =0; i<n; i++)
{
  dp[i] = 1;
  for(int k= i-1; k>=0; k--)
  {
    if(arr[k] < arr[i]) //all k such that arr[k] < arr[i]
    {
      dp[i] = max(dp[i], dp[k]+1);//include k or not
    }
  }
}
cout << dp[n-1];
```

Time Complexity is $\mathcal{O}(n^2)$

7.4 Longest Common Subsequence(LCS)

Problem: Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous

Example. LCS for input Sequences "ABCDGH" and "AEDFHR" is "ADH" of length 3.

Example. LCS for input Sequences "AGGTAB" and "GXTXAYB" is "GTAB" of length 4.

Solution

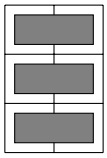
```
/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs(string x, string y, int m, int n )
{
    int L[m+1][n+1]; //2D dp array

/* Following steps build L[m+1][n+1] in bottom up fashion. Note
that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] */
    for (int i=0; i<=m; i++)
    {
      for (int j=0; j<=n; j++)
      {
      if (i == 0 || j == 0) // base case
        L[i][j] = 0;

      else if (X[i-1] == Y[j-1]) //if two chars are equal
        L[i][j] = L[i-1][j-1] + 1;

      else //if two chars are not equal
        L[i][j] = max(L[i-1][j], L[i][j-1]);
      }
    }

    /* L[m][n] contains length of LCS for X[0..n-1] and Y[0..m-1] */
    return L[m][n];
}
```

Time Complexity is $\mathcal{O}(nm)$

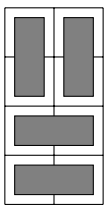
7.5 Tiling Problem

***Problem*:** Given a $2 \times n$ board and tiles of size 2×1, count the number of ways to tile the given board using the 2×1 tiles. A tile can either be placed horizontally i.e., as a 1×2 tile or vertically i.e., as 2×1 tile.

Example. Input n = 3
Output: 3
Explanation:
We need 3 tiles to tile the board of size 2 x 3.
We can tile the board using following ways
1) Place all 3 tiles vertically.
2) Place first tile vertically and remaining 2 tiles horizontally.
3) Place first 2 tiles horizontally and remaining tiles vertically
The first solution is shown below



Example. Input n = 4
Output: 5
Explanation:
For a 2 x 4 board, there are 5 ways
1) All 4 vertical
2) All 4 horizontal
3) First 2 vertical, remaining 2 horizontal
4) First 2 horizontal, remaining 2 vertical
5) Corner 2 vertical, middle 2 horizontal
The third solution is shown below:



Solution

Tiling Problem is nothing but Fibonacci sequence
Method 1:

```
int fib(int n)
{
    //Space optimized Fibonacci
    int a = 1, b = 1, c, i;
    if( n == 1 || n==2)
      return n;
    for (i = 2; i <= n; i++)
    {
      c = a + b;
      a = b;
      b = c;
    }
    return b;
}
```

Time Complexity is $\mathcal{O}(n)$
Method 2:
In this method we directly implement the formula for n^{th} term in the Fibonacci series. Time Complexity : $\mathcal{O}(1)$. Space Complexity: $\mathcal{O}(1)$

$$F_n = \frac{(\frac{(\sqrt{5}+1)}{2})^n}{\sqrt{5}}$$

```
int fib(int n)
{
    double phi = (1 + sqrt(5)) / 2;
    return round(pow(phi, n) / sqrt(5));
}
```

7.6 Coin - 1

Problem: Given a value n, if we want to make change for n cents, and we have infinite supply of each of `S = [`$S_1, S_2, .., S_m$`]` valued coins, how many ways can we make the change? The order of coins does not matter.

Example. $n = 4$ and `S = [1,2,3]` and answer $= 4$
There are four solutions: `[1,1,1,1];[1,1,2];[2,2];[1,3]`

Example. $n = 10$ and `S = [2,5,3,6]` and answer $= 5$
There are five solutions: `[2,2,2,2,2];[2,2,3,3];[2,2,6];[2,3,5];[5,5]`

Solution

Recursive Equation:

$$\text{change}(x) = \begin{cases} \text{x} = 0 & 1 \\ \text{x} > 0 & \sum_{c \in \text{coins}} \text{change}(x - c) \end{cases}$$

```
int coin(int S[], int m, int n)
{
    // table[i] will be storing the number of solutions for
    // value i
    int table[n+1];
    // Initialize all table values as 0
    memset(table, 0, sizeof(table));
    table[0] = 1; //1 way to make 0 coins. First hand experience :)
    // Pick all coins one by one and update the table[] values
    // after the index greater than or equal to the value of the
    // picked coin
    for(int i: S) //for each coin is S
        for(int j=i; j<=n; j++)
            table[j] += table[j-i];

    return table[n]; //answer
}
```

Time Complexity is $\mathcal{O}(nm)$

7.7 Coin - 2

Problem: Given a value n, if we want to make change for n cents, and we have infinite supply of each of `C = [` $C_1, C_2, .., C_m$`]` valued coins, what is the minimum number of coins to make the change?

Example. `coins[] = [25, 10, 5]`, $n = 30$
Output: Minimum 2 coins required
We can use one coin of 25 cents and one of 5 cents

Example. `coins[] = [9, 6, 5, 1]`, $n = 11$
Output: Minimum 2 coins required
We can use one coin of 6 cents and 1 coin of 5 cents

Solution

```
// m is size of coins array (number of different coins)
int minCoins(int coins[], int n)
{
    // table[i] will be storing the minimum number of coins
    // required for i value. So table[n] will have result
    int table[n+1];

    // Base case
    table[0] = 0; //Again first hand experience :)


    // Compute minimum coins required for all
    // values from 1 to n
    for (int i=1; i<=n; i++)
    {
        table[i] = INT_MAX; //initialize all values to Infinity
        // Go through all coins smaller than i
        for (int c : coins)
        {
            if (i-c >= 0)
            {
                table[i] = min(table[i], table[i-c]+1);
            }
        }
    }
    return table[n];
}
```

Time Complexity is $\mathcal{O}(nm)$

7.8 0-1 Knapsack Part 1

Problem: Given n items, each with its own value V_i and weight W_i, $\forall i \in [0..n-1]$, and a maximum knapsack of size S, compute the maximum value of the items that we can carry, if we can either ignore or take a particular item (hence the term 0-1 for ignore/take).

Example. `Value = [60,100,120]`

`Weight = [10,20,30]`

`Knapsack size = 50`

Output $= 220$

Explanation: The best choice is to pick item 2 and item 3

Solution

```
int knapSack(int w, int weight[], int value[], int n)
{
   int knap[n+1][w+1]; //dp table

   // Build table K[][] in bottom up manner
   for (int i = 0; i<=n; i++)
   {
       for (int j = 0; j<=w; j++)
       {
            if(i == 0 || j == 0)
            {
                knap[i][j] = 0;
            }
            else if(weight[i-1] <= j)
            {
knap[i][j] = max(value[i-1]+ knap[i-1][j-weight[i-1]], knap[i-1][j]);
            }
            else
            {
                knap[i][j] = knap[i-1][j];
            }
       }
   }

   return knap[n][w]; //answer
}
```

Time Complexity is $\mathcal{O}(nW)$ where n is the number of items and W is the capacity of knapsack.

7.9 0 - 1 Knapsack Part 2

Problem: Given a list of weights $[w_1, w_2, \ldots, w_n]$, determine all sums that can be constructed using the weights

Example. If the weights are $[1,3,3,5]$, the following sums are possible:

0	1	2	3	4	5	6	7	8	9	10	11	12
X	X		X	X	X	X	X	X	X		X	X

In this case, all sums between $0 \ldots 12$ are possible, except 2 and 10. For example, the sum 7 is possible because we can select the weights $[1,3,3]$.

Solution

Let $\texttt{possible}(x,k) = \text{true}$ if we can construct a sum x using the first k weights, and otherwise $\texttt{possible}(x,k) = \text{false}$. The recursive relation is as follows:

$$\texttt{possible}(x,k) = \texttt{possible}(x - w_k, k-1) \vee \texttt{possible}(x, k-1)$$

The formula is based on the fact that we can either use or not use the weight w_k in the sum. If we use w_k, the remaining task is to form the sum $x - w_k$ using the first $k-1$ weights, and if we do not use w_k, the remaining task is to form the sum x using the first $k-1$ weights. The base cases are,

$$\texttt{possible}(x,0) = \begin{cases} \text{true} & x = 0 \\ \text{false} & x \neq 0 \end{cases}$$

```
// Let "total" denote the total sum of the weights.
w[n]; //array of weights
possible[total+1];
possible[0] = true;
for (int k = 1; k <= n; k++)
{
    for (int x = total; x >= 0; x--)
    {
        if (possible[x]) possible[x+w[k]] = true;
    }
}
for(int i = 0; i<= total; i++)
{
    if(possible[i])
        cout << i << "\n"; //output the values which are possible
}
```

Time Complexity is $\mathcal{O}(n \times total)$

7.10 Maximum Sum Path in a Grid

Problem: Find a path from the upper-left corner to the lower-right corner of an $n \times n$ grid, such that we only move down and right. Each square contains a positive integer, and the path should be constructed so that the sum of the values along the path is as large as possible.

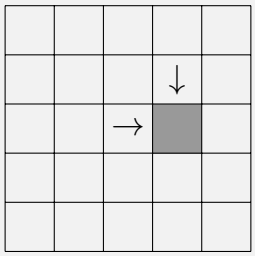
Example.

3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

The sum of the values on the path is 67, and this is the largest possible sum on a path from the upper-left corner to the lower-right corner.

Solution

The formula is based on the observation that a path that ends at square (y,x) can come either from square $(y,x-1)$ or square $(y-1,x)$:



```
value[n+1][n+1]; //grid values
int sum[n][n]; //dp array
for (int y = 1; y <= n; y++)
{
    for (int x = 1; x <= n; x++)
    {
        sum[y][x] = max(sum[y][x-1],sum[y-1][x])+value[y][x];
    }
}
```

Time Complexity is $\mathcal{O}(n^2)$

7.11 Edit Distance

Problem: Given two strings `str1` and `str2` and below operations that can performed on `str1`. Find minimum number of edits (operations) required to convert `str1` into `str2`.

- insert a character (e.g. ABC $\rightarrow$ ABCA)
- remove a character (e.g. ABC $\rightarrow$ AC)
- modify a character (e.g. ABC $\rightarrow$ ADC)

All of the above operations are of equal cost.

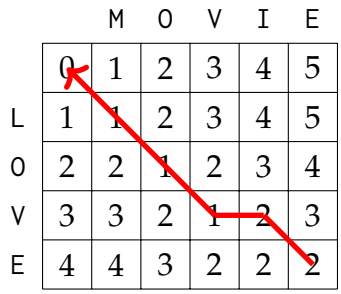
Example. Input: str1 = "geek", str2 = "gesek"
Output: 1
We can convert str1 into str2 by inserting a 's'.

Example. Input: str1 = "cat", str2 = "cut"
Output: 1
We can convert str1 into str2 by replacing 'a' with 'u'.

Example. The edit distance between LOVE and MOVIE is 2, because we can first perform the operation LOVE $\rightarrow$ MOVE (modify) and then the operation MOVE $\rightarrow$ MOVIE (insert). The following table shows the values of `distance` in the example case:

		M	O	V	I	E
	0	1	2	3	4	5
L	1	1	2	3	4	5
O	2	2	1	2	3	4
V	3	3	2	1	2	3
E	4	4	3	2	2	2

The lower-right corner of the table tells us that the edit distance between LOVE and MOVIE is 2. The table also shows how to construct the shortest sequence of editing operations. In this case the path is as follows:



Solution

```
int editDistDP(string str1, string str2, int m, int n)
{
    // Create a table to store results of subproblems
    int dp[m+1][n+1];

    // Fill d[][] in bottom up manner
    for (int i=0; i<=m; i++)
    {
        for (int j=0; j<=n; j++)
        {
            // If first string is empty, only option is to
            // insert all characters of second string
            if (i==0)
                dp[i][j] = j; // Min. operations = j

            // If second string is empty, only option is to
            // remove all characters of second string
            else if (j==0)
                dp[i][j] = i; // Min. operations = i

            // If last characters are same, ignore last char
            // and recur for remaining string
            else if (str1[i-1] == str2[j-1])
                dp[i][j] = dp[i-1][j-1];

            // If the last character is different, consider all
            // possibilities and find the minimum
            else
                dp[i][j] = 1 + min(dp[i][j-1], // Insert
                                   dp[i-1][j], // Remove
                                   dp[i-1][j-1]); // Replace
        }
    }

    return dp[m][n];
}
```

Time Complexity is $\mathcal{O}(mn)$

7.12 Express n as sum of k numbers

Problem: Given an integer n, how many ways can K non-negative integers less than or equal to n add up to n?

Example. $N = 5$, $K = 3$
Output: 6
The possible combinations of integers are:
(1, 1, 3)
(1, 3, 1)
(3, 1, 1)
(1, 2, 2)
(2, 2, 1)
(2, 1, 2)

Example. $N = 10$, $K = 4$
Output: 84

Solution

This is also called as the Binomial Coefficient Problem

```
// Returns value of Binomial Coefficient C(n, k)
int binomial(int n, int k)
{
    int C[n + 1][k + 1];

    // Calculate value of Binomial Coefficient in bottom up manner
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= min(i, k); j++) {
            // Base Cases
            if (j == 0 || j == i)
                C[i][j] = 1;

            // Calculate value using previously stored values
            else
                C[i][j] = C[i - 1][j - 1] + C[i - 1][j];
        }
    }

    return C[n][k];
}
```

Time Complexity is $\mathcal{O}(nk)$

Part III

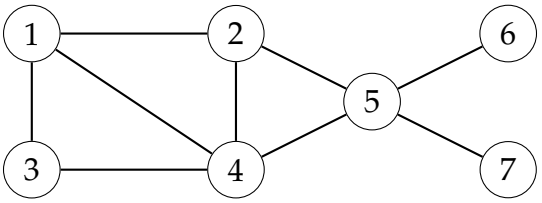
Basic Graph Theory

Chapter 8

Graph Theory

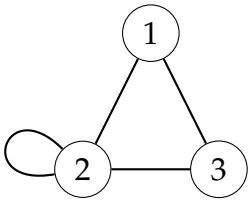
8.1 Formal Definition of Graph

Graph is an abstract mathematical structure, to model pairwise relations between discrete objects. A graph $G = (V, E)$ consists of a finite set V (set of vertices or **nodes**) and a set E (set of **edges**) of 2-subsets of V. Each edge is a relation (adjacency) between two vertices. In general, the number of vertices is denoted by n and the number of edges is denoted by m. The following graph has 6 nodes and 9 edges.



8.2 Degree of Node

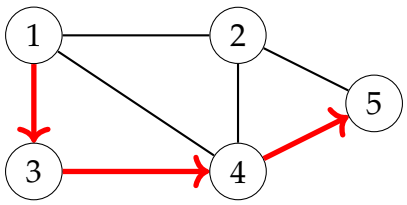
The degree of a node v is the number of edges which are joint to v i.e., the number of edges that are incident on v. As each edge contributes 2 to the degree of its endpoints, the sum of all degrees of vertices equals twice the number of edges. The degree of node 2 in the below graph is 3.



Note: A loop contributes 2 to the degree of the vertex it is incident on.

8.3 Path

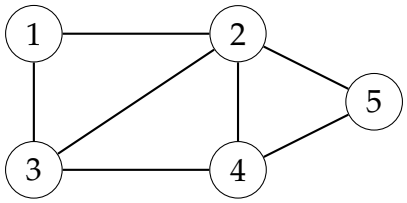
A **path** leads from node a to node b through edges of the graph. The **length** of a path is the number of edges in it. For example, the above graph contains a path $1 \to 3 \to 4 \to 5$ of length 3 from node 1 to node 5:



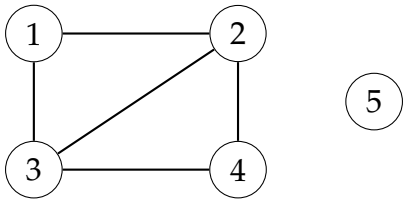
A path is a **cycle** if the first and last node is the same. For example, the above graph contains a cycle $1 \to 3 \to 4 \to 1$. A path is **simple** if each node appears at most once in the path.

8.4 Connectivity

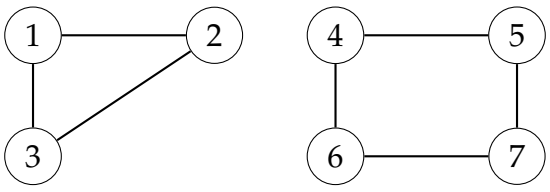
A graph is connected when there is a path(not necessarily a direct edge) between every pair of vertices. In a connected graph, there are no unreachable vertices. A graph that is not connected is disconnected. For example, the following graph is connected:



The following graph is not connected, because it is not possible to get from node 5 to any other node and vice-versa:

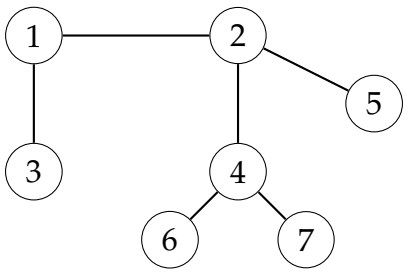


The connected parts of a graph are called its **components**. For example, the following graph contains three components: $\{1, 2, 3\}$ and $\{4, 5, 6, 7\}$

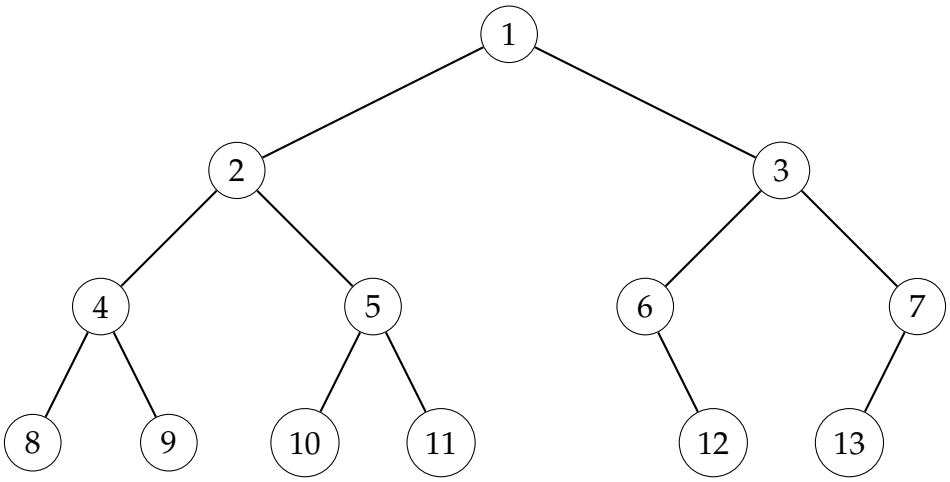


8.5 Trees

A tree is an undirected graph in which any two vertices are connected by exactly one path. It consists of n nodes and $n-1$ edges. A tree cannot contain any cycles or self loops, however, the same does not apply to graphs. For example, the following graph is a tree:



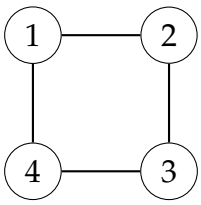
A binary tree is a tree in which each node has at most two children, which are referred to as the left child and the right child. The following tree is a binary tree of size 13 and depth 3, with 1 at the root:



8.6 Types of Graphs

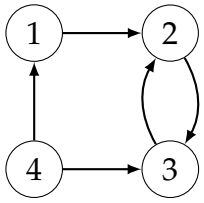
8.6.1 Undirected Graph

An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction. The following graph is undirected:



8.6.2 Directed Graph

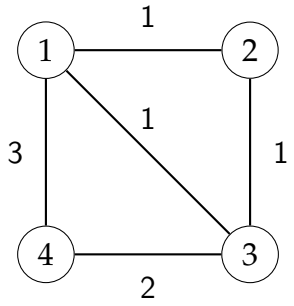
A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction. A typical example would be an airways map. The following graph is directed:



8.6.3 Weighted Graph

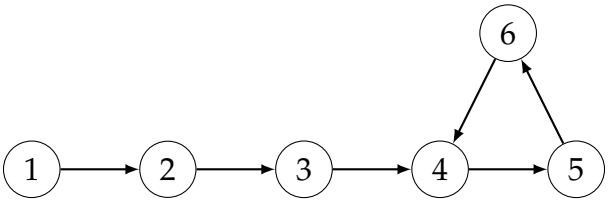
In a weighted graph, each edge is assigned a weight or cost. Consider a graph of 4 nodes as in the diagram below. As you can see each edge has a weight/cost assigned to it. Many problems are based on how to minimize/maximize the cost of a path. In the following graph, if you want to go from vertex 1 to vertex 3, you can take one of the following 3 paths:

1. $1 \rightarrow 2 \rightarrow 3$ **Cost = 3**
2. $1 \rightarrow 3$ **Cost = 1**
3. $1 \rightarrow 4 \rightarrow 3$ **Cost = 5**



8.6.4 Cyclic Graph

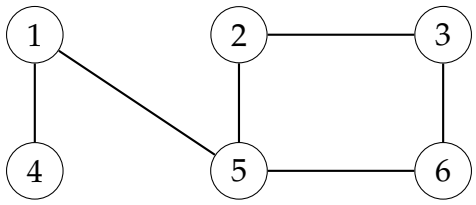
A graph is cyclic if the graph comprises a cycle(a path that starts from a vertex and ends at the same vertex). An acyclic graph is a graph that has no cycle. The following is a cyclic directed graph.



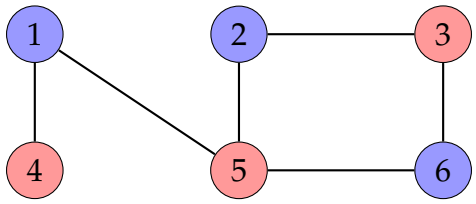
8.7 Graph Colouring

In a **coloring** of a graph, each node is assigned a color so that no adjacent nodes have the same color.

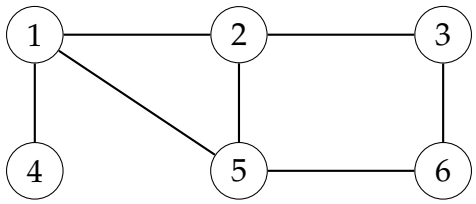
A graph is **bipartite** if it is possible to color it using two colors. It turns out that a graph is bipartite exactly when it does not contain a cycle with an odd number of edges. For example, the graph



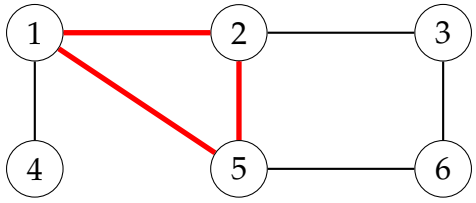
is bipartite, because it can be colored as follows:



However, the graph

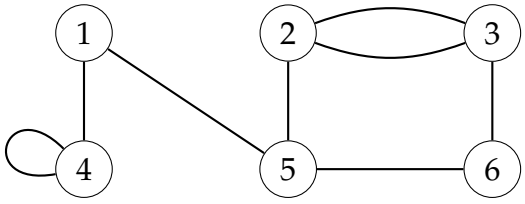


is not bipartite, because it is not possible to color the following cycle of three nodes using two colors:



Simplicity

A graph is **simple** if no edge starts and ends at the same node, and there are no multiple edges between two nodes. Often we assume that graphs are simple. For example, the following graph is **not** simple:



Chapter 9

Graph Representation

You can represent a graph in many ways. The choice of a data structure depends on the size of the graph and the way the algorithm processes it. The two most common ways of representing a graph is as follows:

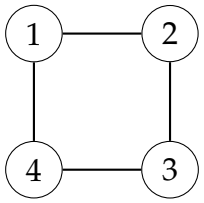
9.1 Adjacency matrix

An adjacency matrix is a $n \times n$ binary matrix A. Element `A[i][j]` is 1 if there is an edge from vertex i to vertex j else is 0. In an undirected graph, if `A[i][j]` $= 1$, then `A[j][i]` $= 1$. In a directed graph, if `A[i][j]` $= 1$, then `A[j][i]` may or may not be 1. The matrix can be stored as an array

```
int adj[n][n];
```

Adjacency matrix provides constant time access $\mathcal{O}(1)$ to determine if there is an edge between two nodes.But, space complexity of the adjacency matrix is $\mathcal{O}(n^2)$.

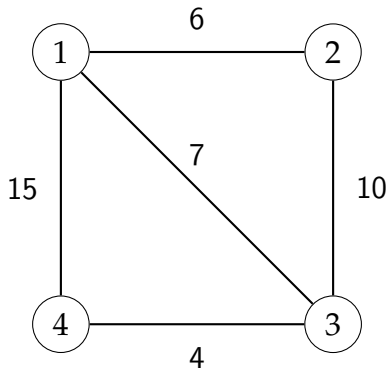
For example the graph



can be represented as follows:

	1	2	3	4
1	0	1	0	1
2	1	0	1	0
3	0	1	0	1
4	1	0	1	0

If the graph is weighted, the adjacency matrix representation can be extended so that the matrix contains the weight of the edge if the edge exists. Using this representation, the graph



corresponds to the following matrix

	1	2	3	4
1	0	6	7	15
2	6	0	10	0
3	7	10	0	4
4	15	0	4	0

The drawback of the adjacency matrix representation is that the matrix contains n^2 elements, and usually most of them are zero. For this reason, the representation cannot be used if the graph is large as a lot of space is wasted.

9.2 Adjacency List

The other way to represent a graph is by using an adjacency list. An adjacency list is an array A of separate lists. Each element of the array A_i is a list, which contains all the vertices that are adjacent to vertex i. In an undirected graph, if vertex j is in list A_i then vertex i will be in list A_j. Adjacency List can be implemented using an array of vectors:

```
vector<int> graph[n];
```

Then edges can be added as follows:

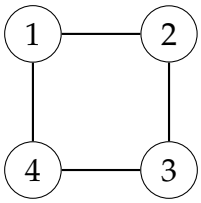
```
graph[u].push_back(v);
```

For a weighted graph, the weight or cost of the edge is stored along with the vertex in the list using pairs:

```
vector<pair<int, int>> graph[n];
```

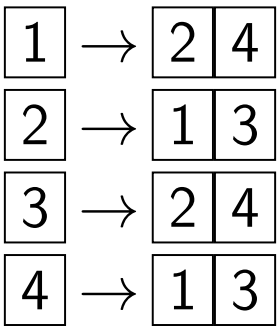
The space complexity of adjacency list is $\mathcal{O}(V + E)$ because in an adjacency list information is stored only for those edges that actually exist in the graph. This is why adjacency list is the most popular form of representation especially in sparse graphs [1].

For example, the graph



can be represented as:

```
graph[1].push_back(2);
graph[1].push_back(4);
graph[2].push_back(1);
graph[2].push_back(3);
graph[3].push_back(2);
graph[3].push_back(4);
graph[4].push_back(3);
graph[4].push_back(1);
```



Another benefit of using adjacency lists is that we can efficiently find the nodes which are directly adjacent to a given node. The following loop goes through all nodes to which we can move from node i through an edge:

```
for(auto u: graph[i])
{
    //we can reach u from i
}
```

P.S.: As a side note, it is important to take note of 0-based indexing or 1-based indexing which is used in the problem statement and adjust all graph algorithms accordingly.

[1] A sparse matrix is a matrix in which most of the elements are zero, whereas a dense matrix is a matrix in which most of the elements are non-zero.

Part IV

Graph Traversals

Chapter 10

Depth First Search

DFS is a simple graph traversal algorithm i.e. it is given a starting node in the graph, and it visits all nodes that can be reached from the starting node. The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

In simple terms, The algorithm begins at a starting node, and proceeds to all other nodes that are reachable from the starting node using the edges of the graph. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

10.1 Implementation

This recursive nature of DFS can be implemented using recursion(duh!). The basic idea is as follows:

The following function `dfs` begins a depth-first search at a given node(usually the root node). The function assumes that the graph is stored as an adjacency list in an array

```
vector<int> graph[n];
```

and also maintains a boolean array

```
bool visited[n];
```

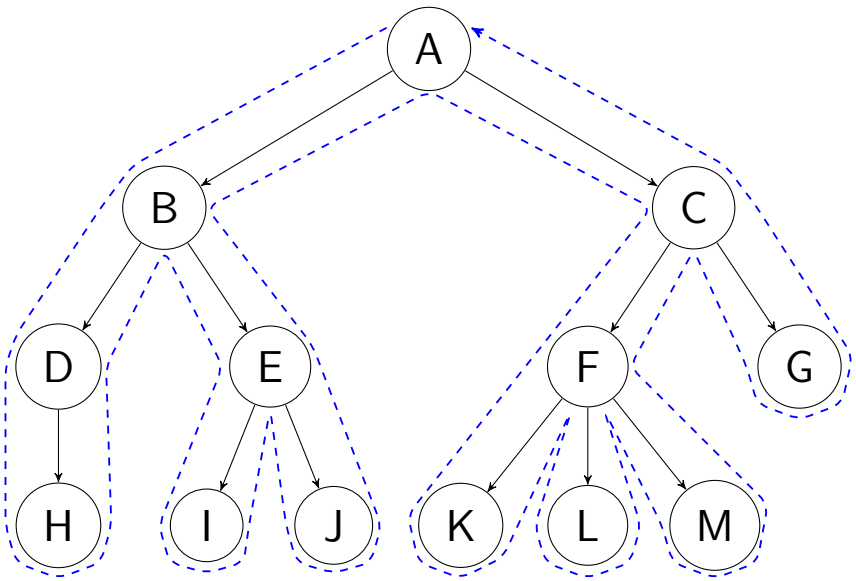
that keeps track of the visited nodes. Initially, each array value is `false`, and when the search arrives at node s, the value of `visited[s]` becomes `true`. The function can be implemented as follows:

```
void dfs(int s)
{
    if (visited[s]) return;
    visited[s] = true;
    // process node s
    for (auto u: adj[s])
    {
        dfs(u); //dfs on all nodes adjacent to s
    }
}
```

The time complexity of depth-first search is $\mathcal{O}(n+m)$ where n is the number of nodes and m is the number of edges,

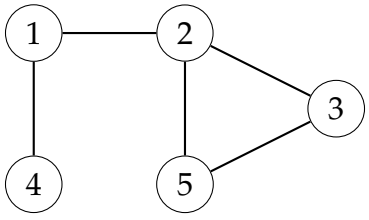
10.2 Visualization

DFS on a tree is shown below. Note that the blue arrow reaches the root node at the end.



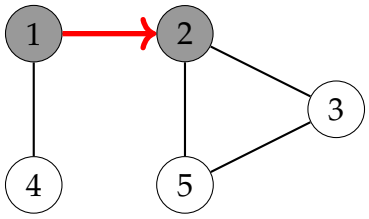
10.2.1 DFS on a normal graph

Let us consider how depth-first search processes the following graph:

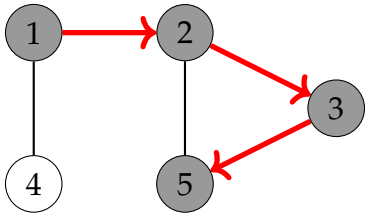


We may begin the search at any node of the graph; now we will begin the search at node 1.

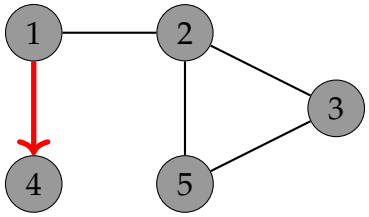
The search first proceeds to node 2:



After this, nodes 3 and 5 will be visited:



The neighbors of node 5 are 2 and 3, but the search has already visited both of them, so it is time to return to the previous nodes. Also the neighbors of nodes 3 and 2 have been visited, so we next move from node 1 to node 4:



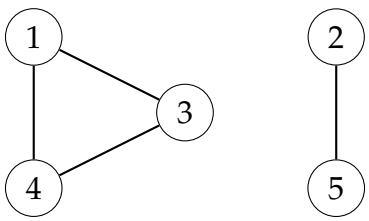
After this, the search terminates because it has visited all nodes.

10.3 Applications of DFS

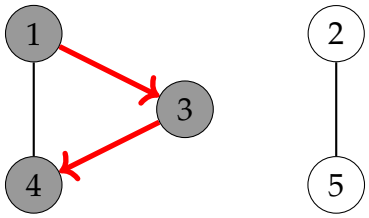
10.3.1 Checking if Graph is Connected

We can check if a graph is connected by starting at an arbitrary node and finding out if we can reach all other nodes.

For example, in the graph



a depth-first search from node 1 visits the following nodes:



Since the search did not visit all the nodes, we can conclude that the graph is not connected. In a similar way, we can also find all connected components of a graph by iterating through the nodes and always starting a new depth-first search if the current node does not belong to any component yet.

10.3.2 Detecting cycle in a graph

A Back Edge is an edge that connects a vertex to a vertex that is discovered before it's parent.

In simple terms, A graph contains a cycle if during a graph traversal, we find a node whose neighbor(other than the previous node in the current path) has already been visited.

A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.

10.3.3 Path Finding

We can specialize the DFS algorithm to find a path between two given vertices u and z. The following are the steps in brief:

1. Call DFS(u) with u as the start vertex.
2. Use a stack S to keep track of the path between the start vertex and the current vertex.
3. As soon as destination vertex z is encountered, return the path as the i.e. the contents of the stack

This path may not be the shortest path

Chapter 11

Breadth First Search

Breadth First Search (BFS) is the most commonly used approach to traverse graph. BFS visits the nodes in increasing order of their distance from the starting node. Thus, we can calculate the distance from the starting node to all other nodes using breadth-first search. However, breadth-first search is more difficult to implement than depth-first search.

Breadth-first search goes through the nodes one level after another. First the search explores the nodes whose distance from the starting node is 1, then the nodes whose distance is 2, and so on. This process continues until all nodes have been visited.

11.1 Implementation

Breadth-first search is more difficult to implement than depth-first search, because the algorithm visits nodes in different parts of the graph. A typical implementation is based on a queue that contains nodes. At each step, the next node in the queue will be processed.

The following code assumes that the graph is stored as adjacency lists and maintains the following data structures:

```
queue<int> q;
bool visited[N];
int distance[N];
```

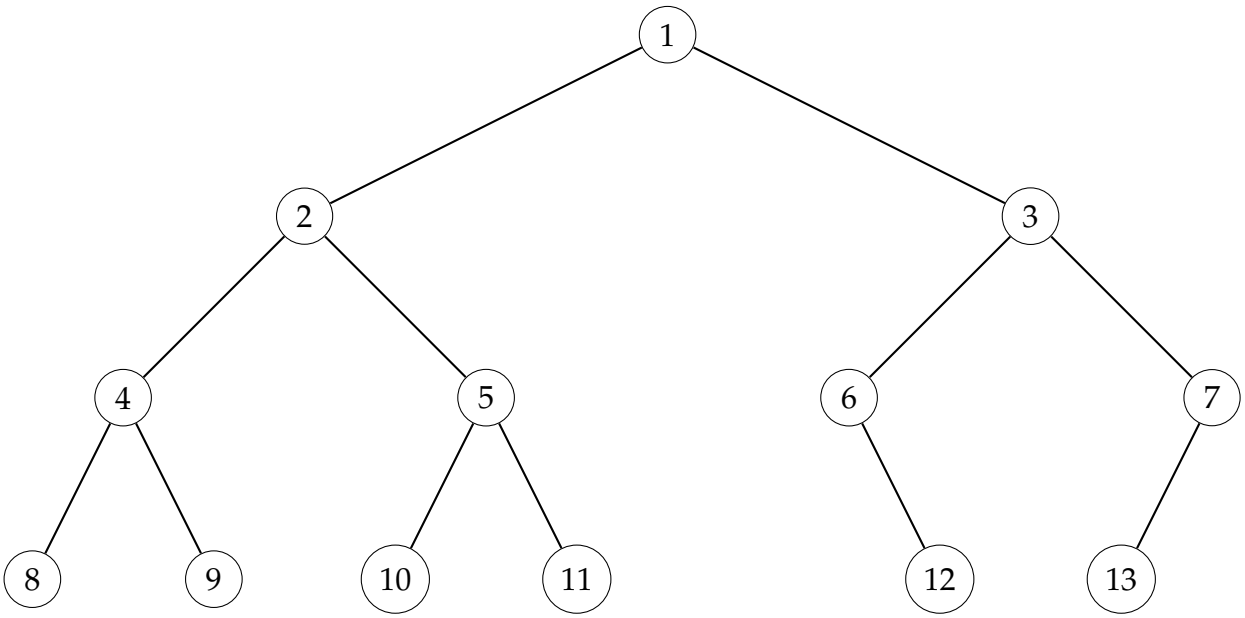
The queue q contains nodes to be processed in increasing order of their distance. New nodes are always added to the end of the queue, and the node at the beginning of the queue is the next node to be processed. The array `visited` indicates which nodes the search has already visited, and the array `distance` will contain the distances from the starting node to all nodes of the graph.

The search can be implemented as follows, starting at node x:

```
visited[x] = true;
distance[x] = 0;
q.push(x);
while (!q.empty()) {
    int s = q.front(); q.pop();
    // process node s
    for (auto u : adj[s]) {
        if (visited[u]) continue;
        visited[u] = true;
        distance[u] = distance[s]+1;
        q.push(u);
    }
}
```

Like in depth-first search, the time complexity of breadth-first search is $\mathcal{O}(n+m)$, where n is the number of nodes and m is the number of edges.

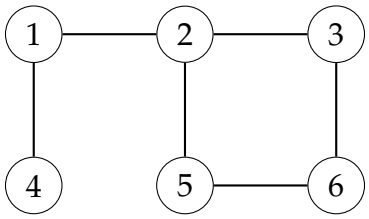
11.2 Visualization



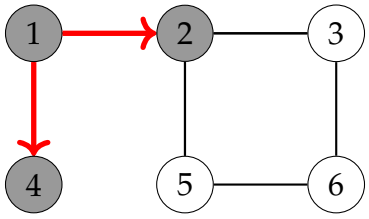
In the above graph BFS traverses nodes in the numeric order(1-2-3-4-5-6...)

11.2.1 BFS on a normal Graph

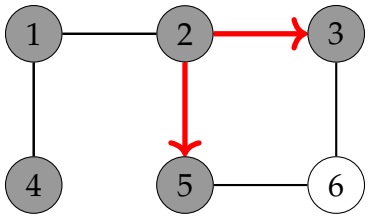
Let us consider how breadth-first search processes the following graph:



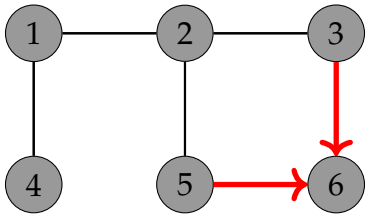
Suppose that the search begins at node 1. First, we process all nodes that can be reached from node 1 using a single edge:



After this, we proceed to nodes 3 and 5:



Finally, we visit node 6:



Now we have calculated the distances from the starting node to all nodes of the graph. The distances are as follows:

node	distance
1	0
2	1
3	2
4	1
5	2
6	3

11.3 Application of BFS

11.3.1 Check whether Graph is Bipartite

Following is a simple algorithm to find out whether a given graph is Bipartite or not using BFS.

1. Assign RED color to the source vertex (putting into set U).
2. Color all the neighbors with BLUE color (putting into set V).
3. Color all neighbor's neighbor with RED color (putting into set U).
4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where $m = 2$.
5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite).

11.3.2 Finding Shortest path in an unweighted graph

BFS can be used to find the shortest path in an **unweighted graph**. The length of shortest path can be accessed from the distance array in our implementation.

11.4 0 - 1 BFS

***Problem*:** Given a graph where every edge has weight as either 0 or k. A source vertex is also given in the graph. Find the shortest path from source vertex to every other vertex.

The basic idea is to take a path if it has edge 0 because that will unlock more paths without increasing the distance. But, using a normal queue data structure , we cannot insert and keep it sorted in $\mathcal{O}(1)$. Using priority queue cost us $\mathcal{O}(\log n)$ to keep it sorted. The problem with the normal queue is the absence of methods which helps us to perform all of these functions :

1. Remove Top Element (To get vertex for BFS)
2. Insert At the beginning (To push a vertex with same level i.e. zero weight)
3. Insert At the end (To push a vertex on next level i.e. one weight)

Fortunately, all of these operations are supported by a double ended queue (or deque in C++ STL).

```
vector<int> d(n, INF); //distance array initialised to infinity
d[s] = 0;
deque<int> q;
q.push_front(s);
while (!q.empty()) {
    int v = q.front();
    q.pop_front();
    for (auto edge : adj[v]) {
        int u = edge.first;
        int w = edge.second;
        if (d[v] + w < d[u]) {
            d[u] = d[v] + w;
            if (w == 1)
                q.push_back(u);
            else
                q.push_front(u);
        }
    }
}
```

Part V

Shortest Path Algorithms

Chapter 12

Theory

- Formal definition of shortest path:

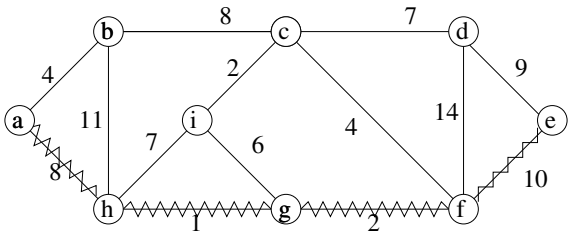
  $G = (V, E)$ weighted graph, directed or undirected.

  Weight of path $P = v_0, v_1, v_2, \cdots, v_k$ is $w(P) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.

  Shortest path $\delta(u, v)$ from u to v has weight

$$\delta(u,v) = \begin{cases} \min\{w(P) : P \text{ is path from } u \text{ to } v\} & \text{If path exists} \\ \infty & \text{Otherwise} \end{cases}$$

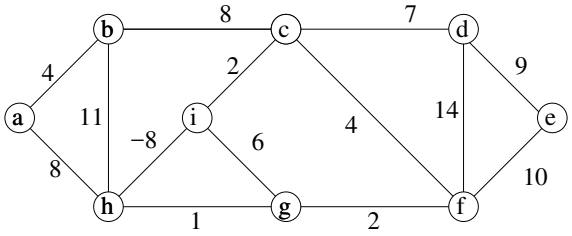
  Example: Shortest path from a to e (of length 21)



- Properties of shortest paths:

  - Subpaths of shortest paths are shortest paths: If $P = u = v_0, v_1, v_2, \cdots, v_k = v$ is shortest path from u to v then for all $i<k$ $P' = u = v_0, v_1, v_2, \cdots, v_i$ is shortest path from u to v_i
  - No (unique) shortest path exists if graph has cycle with negative weight

    Example: If we change weight of edge (h, i) to -8, we have a cycle (i,h,g) with negative weight (-1). Using this we can make the weight of path between a and e arbitrarily low by going through the cycle several times



On the other hand, the problem is well defined if we let edge (h,i) have weight -7 (no negative cycles)

- Different variants of shortest path problem:
  - *Single pair shortest path*: Find shortest path from u to v
  - *Single source shortest path (SSSP)*: Find shortest path from source s to all vertices $v \in V$
  - *All pair shortest path (APSP)*: Find shortest path from u to v for all $u,v \in V$
- Note:
  - No algorithm is known for computing a single pair shortest path better than solving the ("bigger") SSSP problem
  - APSP can be solved by running SSSP $|V|$ times
    $\Downarrow$
    We concentrate only on the SSSP problem to solve APSP.

Chapter 13

Dijkstra's Algorithm

Dijkstra's algorithm for SSSP[1] is a greedy algorithm for directed graphs[2] with only **non-negative weights**.

Dijkstra's algorithm maintains distances to the nodes and reduces them during the search. Dijkstra's algorithm is efficient, because it only processes each edge in the graph once, using the fact that there are no negative edges. It finds a shortest path tree from a single source node, by building a set of nodes that have minimum distance from the source

Dijkstra's algorithm can be efficiently implemented using a `priority queue` in C++ STL.

13.1 Steps of Algorithm

First we initialise three values:

1. `dist[]`: an array of distances from the source node s to each node in the graph, initialised the following way: `dist[s] = 0;` and for all other nodes , `dist[v]` $= \infty$. This is done at the beginning because as the algorithm proceeds, the `dist` from the source to each node v in the graph will be recalculated and finalised when the shortest distance to v is found.

2. Q: a `priority queue` of all nodes in the graph. At the end of the algorithm's progress, Q will be empty. Q contains pairs of the form `(-d, x)`, meaning that the current distance to node x is d. Note that the `priority queue` contains **negative** distances to nodes. The reason for this is that the default version of the C++ `priority queue` finds **maximum** elements, while we want to find **minimum** elements. By using negative distances, we can directly use the default `priority queue`.

3. `visited[]`: a boolean array that indicates whether a node has been processed. Initially all values are set to `false`.

[1] **S**ingle **S**ource **S**hortest **P**ath : Find shortest path from source s to all vertices $v \in V$

[2] Undirected Graph is a special case of directed graph with all edges having same weight

The steps of Dijkstra's Algorithm to find shortest distance of all vertices from vertex s are as follows:

1. Set all values of `dist[]` to ∞ except for the source vertex s, set `dist[s]` to 0. Push `(0,s)` to Q.

2. While Q is not empty, pop the node v, that is not already set to `true` in `visited`, from Q with the smallest `dist[v]`. In the first run, source node s will be chosen because `dist[s]` was initialised to 0. In the next run, the next node with the smallest `dist[]` value is chosen i.e. the first value in Q.

3. Mark `visited[v]` as true to indicate that v has been visited

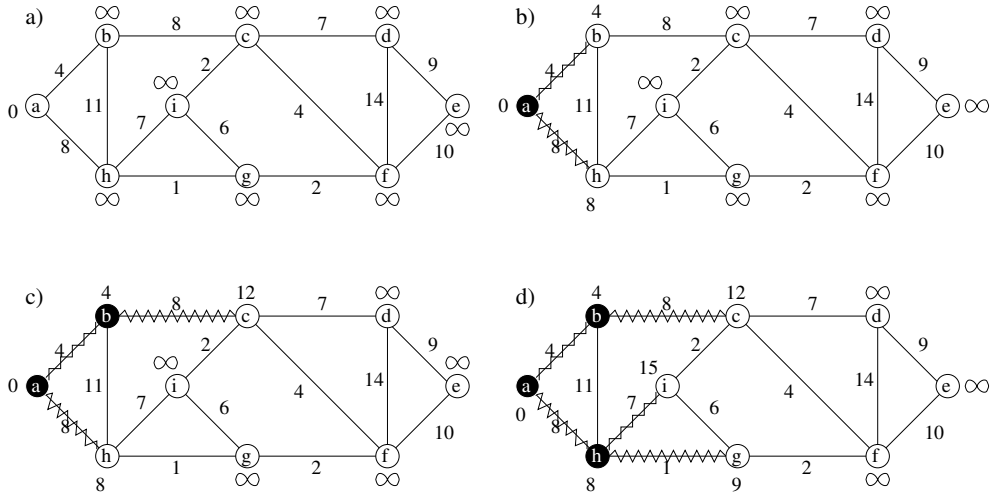
4. Update `dist[]` values of adjacent nodes of the current node v as follows: for each new adjacent node u,

   - if `dist[v]` + `weight(u,v)` $<$ `dist[u]`, there is a new minimal distance found for u, so update `dist[u]` to the new minimal distance value. Then push `(-dist[u],u)` to Q
   - otherwise, no updates are made to `dist[u]`.

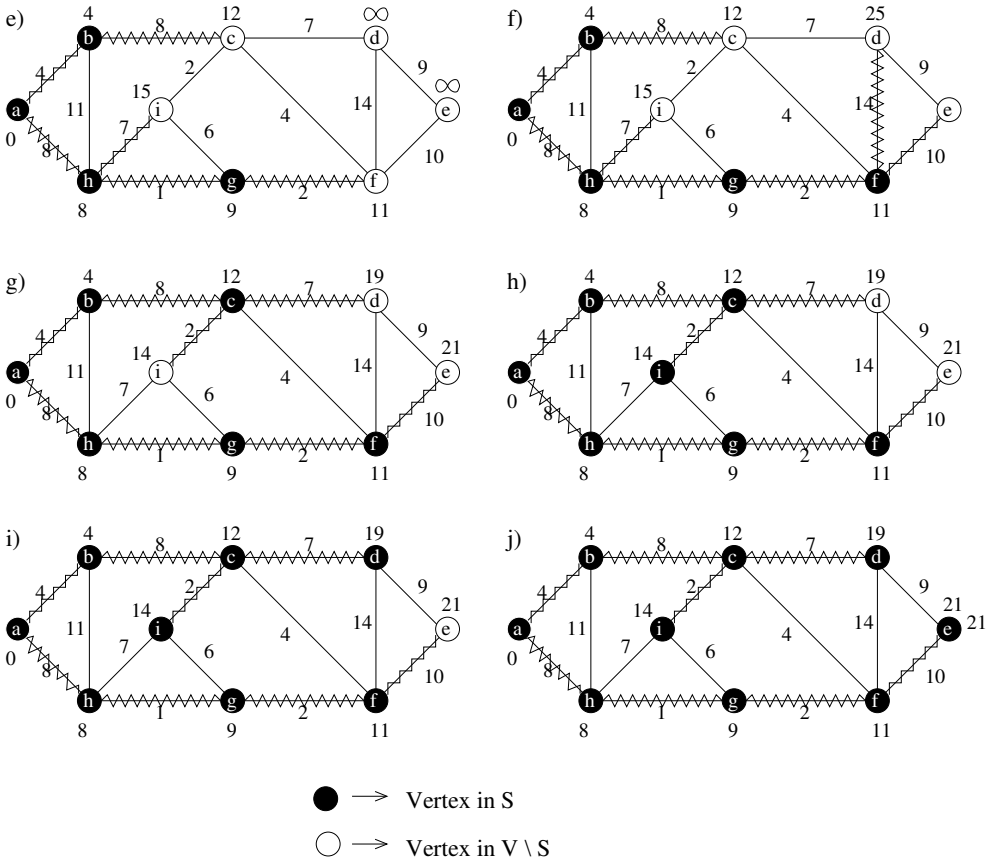
5. Apply the same algorithm again until Q is empty.

6. The algorithm has visited all nodes in the graph when Q **is empty** and found the smallest distance to each node. `dist[]` now contains the shortest path tree from source s.

13.2 Example

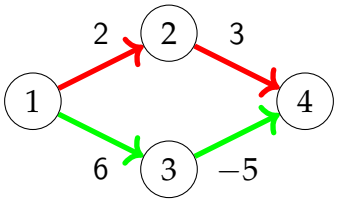


A remarkable property in Dijkstra's algorithm is that whenever a node is selected, its distance is final. For example, at this point of the algorithm, the distances 4, 8 and 12 are the final distances to nodes b, h and c respectively.



13.3 Negative Edges

The efficiency of Dijkstra's algorithm is based on the fact that the graph does not contain negative edges. If there is a negative edge, the algorithm may give incorrect results. As an example, consider the following graph:



The shortest path from node 1 to node 4 is $1 \rightarrow 3 \rightarrow 4$ and its length is 1. However, Dijkstra's algorithm finds the path $1 \rightarrow 2 \rightarrow 4$ by following the minimum weight edges. The algorithm does not take into account that on the other path, the weight -5 compensates the previous large weight 6.

13.4 Implementation

The following implementation of Dijkstra's algorithm calculates the minimum distances from a node x to other nodes of the graph. The graph is stored as adjacency lists so that `adj[a]` contains a pair (b,w) always when there is an edge from node a to node b with weight w.

In the following code, the priority queue `q` contains pairs of the form $(-d,x)$, meaning that the current distance to node x is d. The array `distance` contains the distance to each node, and the array `processed` indicates whether a node has been processed. Initially the distance is 0 to x and ∞ to all other nodes.

Solution

```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
q.push({0,x});
while (!q.empty())
{
    int a = q.top().second;
    q.pop();
    if (processed[a]) continue;
    processed[a] = true;
    for (auto u : adj[a])
    {
        int b = u.first, w = u.second;
        if (distance[a]+w < distance[b])
        {
            distance[b] = distance[a]+w;
            q.push({-distance[b],b});
        }
    }
}
```

Time Complexity of this particular implementation of Dijkstra's Algorithm is $\mathcal{O}(n+m\log m)$ because the algorithm goes through all nodes of the graph and adds for each edge at most one distance to the priority queue.

Chapter 14

Bellman–Ford algorithm

Bellman–Ford algorithm finds shortest paths from a starting node to all nodes of a directed graphs with **any weights**(but if there is a cycle with negative weight, then this problem will be **NP**).

Advantages of using Bellman-Ford over Dijkstra

- Bellman-Ford supports finding shortest path in graphs with negative weight edges.
- Bellman-Ford can find negative weight cycles in graphs. In the n^{th} step, we can reduce any distance $\iff$ we have a negative cycle.
- It is very easy and short to implement.

But, we don't use Bellman-Ford every time because it is **slower** than Dijkstra.

The algorithm keeps track of distances from the starting node to all nodes of the graph. Initially, the distance to the starting node is 0 and the distance to all other nodes is ∞. The algorithm reduces the distances by finding edges that shorten the paths until it is not possible to reduce any distance.

14.1 Pseudo-code

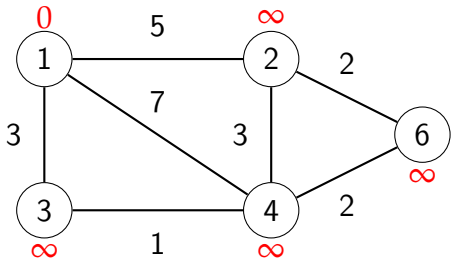
Algorithm: `Bellman-Ford(int v)`

Result: **S**ingle **S**ource **S**hortest **P**ath

```
1  d[i] = ∞ for each vertex i;
2  d[v] = 0;
3  for step = 1 to n do
4      for all edges e do
5          i = e.first ;                                  // first end of e
6          j = e.second ;                                 // second end of e
7          w = e.weight ;                                 // weight of edge of e
8          if d[j] greater than d[i] + w then
9              if step == n then
10                 return "Negative cycle found" ; // for detecting negative cycles
11
12             end
13             d[j] = d[i] + w;
14         end
15     end
16 end
```

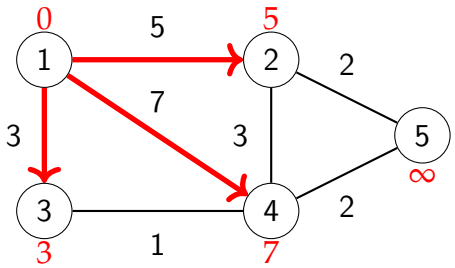
14.2 Example

Let us consider how the Bellman–Ford algorithm works in the following graph:

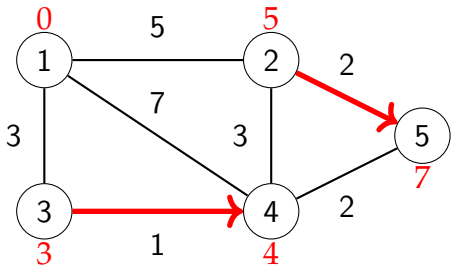


Each node of the graph is assigned a distance. Initially, the distance to the starting node is 0, and the distance to all other nodes is infinite.

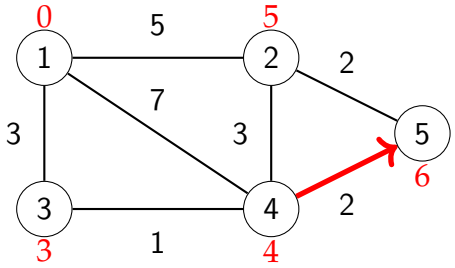
The algorithm searches for edges that reduce distances. First, all edges from node 1 reduce distances:



After this, edges $2 \to 5$ and $3 \to 4$ reduce distances:

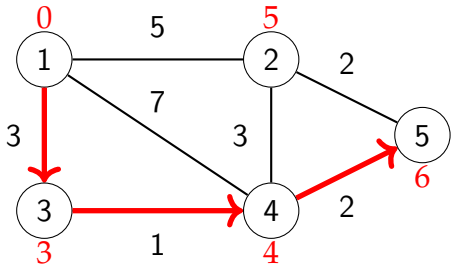


Finally, there is one more change:



After this, no edge can reduce any distance. This means that the distances are final, and we have successfully calculated the shortest distances from the starting node to all nodes of the graph.

For example, the shortest distance 3 from node 1 to node 5 corresponds to the following path:



14.3 Implementation

The following implementation of the Bellman–Ford algorithm determines the shortest distances from a node x to all nodes of the graph. The code assumes that the graph is stored as an edge list `edges` that consists of tuples of the form (a,b,w), meaning that there is an edge from node a to node b with weight w.

The algorithm consists of $n-1$ rounds, and on each round the algorithm goes through all edges of the graph and tries to reduce the distances. The algorithm constructs an array `distance` that will contain the distances from x to all nodes of the graph. The constant `INF` denotes an infinite distance.

Solution

```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a]+w);
    }
}
```

The time complexity of the algorithm is $\mathcal{O}(nm)$, because the algorithm consists of $n-1$ rounds and iterates through all m edges during a round. If there are no negative cycles in the graph, all distances are final after $n-1$ rounds, because each shortest path can contain at most $n-1$ edges.

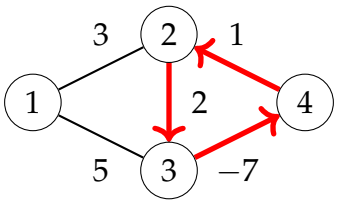
In practice, this algorithm can be somewhat optimised: often we already get the answer in a few phases and no useful work is done in remaining phases till $n-1$, just a waste visiting all edges. So, let's keep the flag, to tell whether something changed in the current phase or not, and if any phase, nothing changed, the algorithm can be stopped.

Solution

```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++)
{
    bool flag = false;//flag
    for (auto e : edges)
    {
        int a, b, w;
        tie(a, b, w) = e;
        if(distance[b] > distance[a]+w)
        {
            distance[b] = distance[a]+w;
            any = true;
        }
    }
    if(!flag)//no change takes place
    {
        break;
    }
}
```

14.3.1 Catching Negative Cycles

The Bellman–Ford algorithm can also be used to check if the graph contains a cycle with negative length. For example, the graph



contains a negative cycle $2 \to 3 \to 4 \to 2$ with length -4.

If the graph contains a negative cycle, we can shorten infinitely many times any path that contains the cycle by repeating the cycle again and again. Thus, the concept of a shortest path is not meaningful in this situation.

A negative cycle can be detected using the Bellman–Ford algorithm by running the algorithm for n rounds. If the last round reduces any distance, the graph contains a negative cycle. Note that this algorithm can be used to search for a negative cycle in the whole graph regardless of the starting node.

Solution

```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++)
{
    bool flag = false;//flag
    for (auto e : edges)
    {
        int a, b, w;
        tie(a, b, w) = e;
        if(distance[b] > distance[a]+w)
        {
            if(i == n-1) cout << "Negative Cycle is found";‘//break
            distance[b] = distance[a]+w;
            any = true;
        }
    }
    if(!flag)//no change takes place
    {
        break;
    }
}
```

14.4 SPFA Algorithm

The **SPFA algorithm**[1] (**S**hortest **P**ath **F**aster **A**lgorithm) is a variant of the Bellman–Ford algorithm, that is often more efficient than the original algorithm. The SPFA algorithm does not go through all the edges on each round, but instead, it chooses the edges to be examined in a more intelligent way.

The algorithm maintains a queue of nodes that might be used for reducing the distances. First, the algorithm adds the starting node x to the queue. Then, the algorithm always processes the first node in the queue, and when an edge $a \to b$ reduces a distance, node b is added to the queue.

The following implementation uses a queue q. In addition, an array inqueue indicates if a node is already in the queue, in which case the algorithm does not add the node to the queue again.

```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
q.push(x);
while (!q.empty())
{
    int a = q.front(); q.pop();
    inqueue[a] = false;
    for (auto b : v[a])
    {
        if (distance[a]+b.second < distance[b.first])
        {
            distance[b.first] = distance[a]+b.second;
            if (!inqueue[b])
            {q.push(b); inqueue[b] = true;}
        }
    }
}
```

The worst-case running time of the algorithm is $\mathcal{O}(mn)$, just like the standard Bellman-Ford algorithm. Experiments suggest that the average running time is $\mathcal{O}(m)$, but this bound on the average run time has not been proved.

[1] The origin of this algorithm is unknown. It's said that at first Chinese coders used it in programming contests.

Chapter 15

Floyd–Warshall algorithm

Floyd-Warshall algorithm solves the APSP[1] problem in a directed graph with any weights but no negative weight cycles (then it becomes NP).

Floyd-Warshall algorithm is a Dynamic Programming based algorithm. The technique used in this algorithm is known as Matrix Chain Multiplication. A shortest path between 2 nodes in a graph would be a chain of nodes.

The algorithm maintains a two-dimensional array that contains distances between the nodes. First, distances are calculated only using direct edges between the nodes, and after this, the algorithm reduces distances by using intermediate nodes in paths.

Why don't we apply Djikstra from each node?

- For simple problems and applications of Floyd-Warshall algorithm, its implementation is very short (5-6 lines of code). Hence, if $\mathcal{O}(n^3)$ is good enough, then Floyd-Warshall is preferable.
- Floyd-Warshall can detect the existence of negative cycles in the graph.

Advantages of Floyd-Warshall over Djikstra

- Find the shortest path between 2 nodes with a restriction that we should traverse exactly L edges (where $L \leq 10^9$). Of course, the question only makes sense when we are allowed to traverse the same edge more than once.
- Find a negative cycle in a weighted graph with minimum number of edges

[1] **A**ll-**P**airs **S**hortest **P**ath: It calculates the value of the shortest path between each pair of nodes in a graph.

forming this cycle (the total weight doesn't matter) it should just be negative and the number of edges is minimum.

- Often, you will notice that Floyd-Warshall technique can be applied when we need to work with the adjacency matrix representation of a graph.

15.1 Pseudo-code

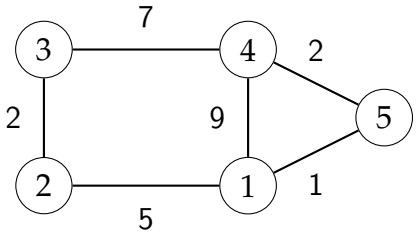
Algorithm: Floyd-Warshall

Result: **A**ll-**P**airs **S**hortest **P**ath

```
d[v][u] = ∞ for each pair (v,u);
d[v][u] = weight(v,u) for each adjacent pair (v,u);
d[v][v] = 0 for each vertex v;
for i = 1 to n do
    for j = 1 to n do
        for k = 1 to n do
            d[j][k] = min(d[j][k], d[j][i] + d[i][k])
        end
    end
end
```

15.2 Example

Let us consider how the Floyd–Warshall algorithm works in the following graph:



Initially, the distance from each node to itself is 0, and the distance between nodes a and b is x if there is an edge between nodes a and b with weight x. All other distances are infinite.

In this graph, the initial array is as follows:

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	∞	∞
3	∞	2	0	7	∞
4	9	∞	7	0	2
5	1	∞	∞	2	0

The algorithm consists of consecutive rounds. On each round, the algorithm selects a new node that can act as an intermediate node in paths from now on, and distances are reduced using this node.

On the first round, node 1 is the new intermediate node. There is a new path between nodes 2 and 4 with length 14, because node 1 connects them. There is also a new path between nodes 2 and 5 with length 6.

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	**14**	**6**
3	∞	2	0	7	∞
4	9	**14**	7	0	2
5	1	**6**	∞	2	0

On the second round, node 2 is the new intermediate node. This creates new paths between nodes 1 and 3 and between nodes 3 and 5:

	1	2	3	4	5
1	0	5	**7**	9	1
2	5	0	2	14	6
3	**7**	2	0	7	**8**
4	9	14	7	0	2
5	1	6	**8**	2	0

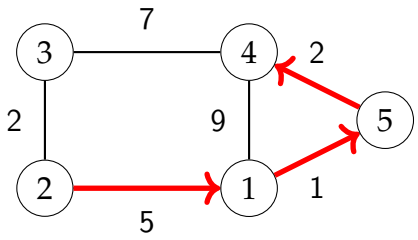
On the third round, node 3 is the new intermediate round. There is a new path between nodes 2 and 4:

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	**9**	6
3	7	2	0	7	8
4	9	**9**	7	0	2
5	1	6	8	2	0

The algorithm continues like this, until all nodes have been appointed intermediate nodes. After the algorithm has finished, the array contains the minimum distances between any two nodes:

	1	2	3	4	5
1	0	5	7	3	1
2	5	0	2	8	6
3	7	2	0	7	8
4	3	8	7	0	2
5	1	6	8	2	0

For example, the array tells us that the shortest distance between nodes 2 and 4 is 8. This corresponds to the following path:



15.3 Implementation

The advantage of the Floyd–Warshall algorithm that it is easy to implement. The following code constructs a distance matrix where `distance`$[a][b]$ is the shortest distance between nodes a and b. First, the algorithm initializes `distance` using the adjacency matrix `adj` of the graph:

```
for (int i = 1; i <= n; i++)
{
    for (int j = 1; j <= n; j++)
    {
        if (i == j) distance[i][j] = 0;//same node
        else if (adj[i][j]) distance[i][j] = adj[i][j];//adjacent nodes
        else distance[i][j] = INF;//currently no known path
    }
}
```

After this, the shortest distances can be found as follows:

```
for (int i = 1; i <= n; i++)
{
    for (int i = j; j <= n; j++)
    {
        for (int k = 1; k <= n; k++)
        {
            distance[j][k] = min(distance[j][k],
                                 distance[j][i] + distance[i][k])
        }
    }
}
```

The time complexity of the algorithm is $\mathcal{O}(n^3)$, because it contains three nested loops that go through the nodes of the graph.

Since the implementation of the Floyd–Warshall algorithm is simple, the algorithm can be a good choice even if it is only needed to find a single shortest path in the graph. However, the algorithm can only be used when the graph is so small that a cubic time complexity is fast enough.

15.3.1 Detecting Negative Cycles

Run Floyd-Warshall algorithm on the graph. Initially $d[v][v] = 0$ for each v. But after running the algorithm $d[v][v]$ will be smaller than 0 if there exists a negative length path from v to v. We can use this to find all pairs of vertices that don't have a shortest path between them.

Part VI

Mathematics

Chapter 16

Basic Maths

16.1 Integer bounds

- `int` : $2^{31}-1 \approx 2 \times 10^9$
- `long long` : $2^{63}-1 \approx 9 \times 10^{18}$
- `unsigned int` : $2^{32}-1 \approx 4 \times 10^9$
- `unsigned long long` : $2^{64}-1 \approx 1.8 \times 10^{19}$

16.2 Method for integers $> 2^{64}-1 \approx 1.8 \times 10^{19}$

Arbitrary precision data type: We can use any precision with the help of cppint data type if we are not sure about how much precision is needed in future. It automatically converts the desired precision at the Run-time.

```
#include <boost/multiprecision/cpp_int.hpp>
using namespace boost::multiprecision;
int main()
{
    cpp_int x; //Can have arbitrary precision
}
```

16.3 Modular arithmetic

An important property of the remainder is that in addition, subtraction and multiplication, the remainder can be taken before the operation:

$$\begin{aligned}(a+b) \pmod m &= (a \pmod m + b \pmod m) \pmod m \\ (a-b) \pmod m &= (a \pmod m - b \pmod m) \pmod m \\ (a \cdot b) \pmod m &= (a \pmod m \cdot b \pmod m) \pmod m\end{aligned}$$

For example, the following code calculates $n!$, the factorial of n, modulo m:

```
long long x = 1;
for (int i = 2; i <= n; i++) {
    x = (x*i)%m;
}
cout << x%m << "\n";
```

16.4 Floating point numbers

Printing floating point numbers up to n digits.

```
float pi = 3.14159;
cout << fixed << setprecision(3) << pi; //3.142
```

16.5 Mathematics

16.5.1 Sum formulas

$$\sum_{x=1}^{n} x = 1+2+3+\ldots+n = \frac{n(n+1)}{2}$$

and

$$\sum_{x=1}^{n} x^2 = 1^2+2^2+3^2+\ldots+n^2 = \frac{n(n+1)(2n+1)}{6}.$$

and

$$\sum_{x=1}^{n} x^3 = 1^3+2^3+3^3+\ldots+n^3 = \frac{n^2(n+1)^2}{4}$$

16.5.2 Sum of Arithmetic Progression

$$\underbrace{a+\cdots+b}_{n \text{ numbers}} = \frac{n(a+b)}{2}$$

where,

a is the first number,
b is the last number and
n is the amount of numbers.

16.5.3 Sum of Geometric Progression

$$a + ak + ak^2 + \cdots + b = \frac{bk - a}{k - 1}$$

where,

a is the first number,

b is the last number and

the ratio between consecutive numbers is k.

A special case of a sum of a geometric progression is the formula

$$1 + 2 + 4 + 8 + \ldots + 2^{n-1} = 2^n - 1.$$

Chapter 17

Number Theory

Number theory is a branch of mathematics that studies integers. Number theory is a fascinating field, because many questions involving integers are very difficult to solve even if they seem simple at first glance.

Mastering as many topics as possible in the field of number theory is important as some mathematics problems become easy (or easier) if you know the theory behind the problems. Otherwise, either a plain brute force attack leads to a TLE response or you simply cannot work with the given input as it is too large without some pre-processing

17.1 Primality check

Problem: Test whether a given natural number N is prime or not.

Solution

We do the following optimisation:

1. Instead of checking till n, we can check till $\sqrt{n}$
2. The algorithm can be improved further by observing that all primes are of the form $6k \pm 1$, with the exception of 2 and 3.

```
bool isPrime(int n)
{
    if (n <= 1) return false;
    if (n <= 3) return true;
    if (n%2 == 0 || n%3 == 0) return false;
    for (int i=5; i*i<=n; i=i+6)
        if (n%i == 0 || n%(i+2) == 0)
           return false;
    return true;
}
```

17.2 Sieve of Eratosthenes

The sieve of Eratosthenes is one of the most efficient ways to find all primes smaller than n when n is smaller than 10 million

Problem: Given a number n, print all primes smaller than or equal to n. It is also given that n is a small number.

Solution

Following is the algorithm to find all the prime numbers less than or equal to a given integer n by Eratosthenes's method:

1. The algorithm builds an array `sieve` whose positions $0,1,2,3,...,n$ are used. The value `sieve[k] = 0` means that k is prime, and the value `sieve[k]` $\neq$ `0` means that k is not a prime and one of its prime factors is `sieve[k]`.

2. The algorithm iterates through the numbers $2...n$ one by one. Always when a new prime x is found, the algorithm records that the multiples of x $(2x,3x,4x,...)$ which are $\geq x^2$ are not primes, because the number x divides them.

For example, if $n = 20$, the array is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	2	0	2	0	2	3	2	0	3	0	2	3	2	0	3	0	2

```
int n = 20; //example
int sieve[n+1];
memset(sieve, 0, sizeof(sieve)); //initialize to zero
for (int x = 2; x*x <= n; x++)
{
    if (sieve[x]) continue;
    for (int u = x*x; u <= n; u += x)
    {
        sieve[u] = x;
    }
}
for(int i = 2; i<=n; i++)
{   //print all prime numbers
    if(!sieve[i])
        cout << i << "\n";
}
```

Time Complexity is $\mathcal{O}(n\log(\log n))$ which is very close to linear $\mathcal{O}(n)$

17.3 Prime Factorization

Problem: Given a number n, print all prime factors of n.

Solution

Following are the steps to find all prime factors.

1. While n is divisible by 2, print 2 and divide n by 2.
2. After step 1, n must be odd. Now start a loop from $i = 3$ to $\sqrt{n}$. While i divides n, print i and divide n by i, increment i by 2 and continue.
3. If n is a prime number and is greater than 2, then n will not become 1 by above two steps. So print n if it is greater than 2.

```
vector<int> factors(int n)
{
    vector<int> f;
    while (n%2 == 0)
    {
        f.push_back(2);
        n = n/2;
    }
    for (int i = 3; i <= sqrt(n); i = i+2)
    {
        while (n%i == 0)
        {
            f.push_back(i);
            n = n/i;
        }
    }
    // This condition is to handle the case when n
    // is a prime number greater than 2
    if (n > 2)
        f.push_back(n);
    return f;
}
```

Time Complexity is $\mathcal{O}(\sqrt{n})$
There is an even efficient solution which uses Sieve of Eratosthenes to pre compute prime numbers. It has time complexity $\mathcal{O}(\log n)$. You can read about it on GeeksforGeeks or CP3 book.

17.4 Common Conjectures

In mathematics, a conjecture is a conclusion or proposition based on incomplete information, for which no proof has been found. But many of them have been verified

for very large numbers. In competitive programming, some problems are based on these common conjectures.

***Conjecture* 1** (Goldbach's conjecture)**:** Each even integer $n > 2$ can be represented as a sum $n = a + b$ so that both a and b are primes.

***Conjecture* 2** (Goldbach's weak conjecture)**:** Every odd number greater than 5 can be expressed as the sum of three primes. (A prime may be used more than once in the same sum.) This is trivial to prove if the above conjecture is proved to be true.

***Conjecture* 3** (Twin prime conjecture)**:** There is an infinite number of pairs of the form $p, p + 2$, where both p and $p + 2$ are primes.

***Conjecture* 4** (Legendre's conjecture)**:** There is always a prime between numbers n^2 and $(n + 1)^2$, where n is any positive integer.

***Conjecture* 5** (Collatz Conjecture)**:** Collatz conjecture states that a number n converges to 1 on repeatedly performing the following operations:

$$n \to n/2 \text{ if } n \text{ is even}$$

$$n \to 3n + 1 \text{ if } n \text{ is odd}$$

This has been verified for numbers up to 5.6×10^{13}. For example is $x = 3$, then:

$$3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

***Conjecture* 6** (Mersenne Prime Conjecture)**:** There are infinitely positive integers n for which $2n - 1$ is a prime number. (There are currently 47 Mersenne primes known)

17.5 Modulo of Big Number

Problem: Given a big number num represented as string and an integer x, find value of $num \mod x$. Output is expected as an integer.

Solution

The idea is to process all digits one by one and use the property that $(xy) \mod a \equiv (x \mod a \times y \mod a) \mod a$. Below is the implementation.

```
int modulo(string num, int x)
{
  // Initialize result
  int res = 0;

  // One by one process all digits of 'num'
  for (int i = 0; i < num.length(); i++)
    res = (res*10 + (int)num[i] - '0') % x;
  return res;
}
```

17.6 Modular Exponentiation

Problem: Given x,m,n, find $x^n \mod m$

Solution

The naive solution would run in $\mathcal{O}(n)$ time. Using modular exponentiation we can bring down the complexity to $\mathcal{O}(\log n)$ by using the following algorithm:

$$x^n = \begin{cases} 1 & n = 0 \\ x^{n/2} \cdot x^{n/2} & n \text{ is even} \\ x^{n-1} \cdot x & n \text{ is odd} \end{cases}$$

```
int power(int x, int n, int m)
{
    if (n == 0) return 1%m;
    long long u = power(x,n/2,m);
    u = (u*u)%m;
    if (n%2 == 1) u = (u*x)%m;
    return u;
}
```

17.7 Euler's Totient Function

Euler's totient function $\varphi(n)$ gives the number of coprime numbers to n between 1 and n. For example, $\varphi(14) = 6$, because 1, 3, 5, 9, 11 and 13 are coprime to 14.

The value of $\varphi(n)$ can be calculated from the prime factorization of n using the formula

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

Where the product is over the distinct prime numbers dividing n.Note that $\varphi(n) = n - 1$ if n is prime. The implementation of totient function is shown below.

```
int totient(int n)
{
    int result = n;
    for (int p = 2; p * p <= n; ++p)
    {
        if (n % p == 0)
        {
            while (n % p == 0) n /= p;
            result -= result / p;
        }
    }

    if (n > 1) result -= result / n;
    return result;
}
```

17.8 Some Common Theorems

Theorem is a mathematical result that has been proved for every input value which lies in it's domain.

17.8.1 Lagrange's Four-Square Theorem

Lagrange's theorem states that every positive integer can be represented as a sum of four squares. i.e.,

$$n = a^2 + b^2 + c^2 + d^2$$

where, $n, a, b, c, d \in \mathbb{N}$

The number of representations of a natural number n as the sum of four squares is denoted by $r_4(n)$. Jacobi's four-square theorem states that this is eight times the sum of the divisors of n if n is odd and 24 times the sum of the odd divisors of n if n is even. In particular, for a prime number p we have the explicit formula $r_4(p) = 8(p + 1)$

17.8.2 Wilson's Theorem

Wilson's theorem states that a number n is prime exactly when

$$(n-1)! \equiv -1 \pmod{n}$$

OR

$$(n-1)! \pmod{n} = n-1$$

However the theorem cannot be applied to large values of n, because it is difficult to calculate values of $(n-1)!$ when n is even as large as 50.

17.8.3 Fibonacci Numbers

The Fibonacci sequence is defined as follows:

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$$

The first elements of the sequence are:

$$0,1,1,2,3,5,8,13,21,34,55,89,...$$

Cassini's identity

$$F_{n-1}F_{n+1} - F_n^2 = (-1)^n$$

Addition Rule

$$F_{n+k} = \begin{cases} F_k F_{n+1} + F_{k-1}F_n \\ F_{2n} = F_n(F_{n+1} + F_{n-1}), & \text{if } k = n \end{cases}$$

GCD identity

$$GCD(F_m, F_n) = F_{GCD(m,n)}$$

Zeckendorf's Theorem

Zeckendorf's theorem states that every positive integer can be represented uniquely as the sum of one or more distinct Fibonacci numbers in such a way that the sum does not include any two consecutive Fibonacci numbers. More precisely, if N is any positive integer, there exist positive integers $c_i \geq 2$, with $c_{i+1} > c_i + 1$, such that

$$N = \sum_{i=0}^{k} F_{c_i}$$

Where F_n is the n^{th} Fibonacci number.

17.8.4 Pythagorean Triples

A **Pythagorean triple** is a triple (a,b,c) that satisfies the Pythagorean theorem $a^2+b^2=c^2$, which means that there is a right triangle with side lengths a, b and c. For example, $(3,4,5)$ is a Pythagorean triple.

If (a,b,c) is a Pythagorean triple, all triples of the form (ka,kb,kc) are also Pythagorean triples where $k>1$. A Pythagorean triple is *primitive* if a, b and c are coprime, and all Pythagorean triples can be constructed from primitive triples using a multiplier k.

Euclid's formula can be used to produce all primitive Pythagorean triples. Each such triple is of the form

$$(n^2-m^2, 2nm, n^2+m^2),$$

where $0<m<n$, n and m are coprime and at least one of n and m is even. For example, when $m=1$ and $n=2$, the formula produces the smallest Pythagorean triple

$$(2^2-1^2, 2\cdot 2\cdot 1, 2^2+1^2) = (3,4,5).$$

17.9 Linear Diophantine Equations

A Diophantine equation is a polynomial equation, usually in two or more unknowns, such that only the integral solutions are required. An Integral solution is a solution such that all the unknown variables take only integer values.

***Problem*:** Given three integers a, b, c representing a linear equation of the form : $ax+by=c$. Determine if the equation has a solution such that x and y are both integral values.

Solution

For linear Diophantine equation equations, integral solutions exist if and only if, the GCD of coefficients of the two variables divides the constant term perfectly. In other words the integral solution exists if, $\gcd(a,b)|c$.
Thus the algorithm to determine if an equation has integral solution is pretty straightforward.

```
bool isPossible(int a, int b, int c)
{
  // __gcd() is an inbuilt function in C++ STL to return
  // GCD of two numbers.
  return (c%__gcd(a,b) == 0);
}
```

17.10 Euclid's Algorithm for GCD

Problem: Given two non-negative integers a and b, we have to find their $\gcd$(greatest common divisor), i.e. the largest number which is a divisor of both a and b. It's commonly denoted by $\gcd(a,b)$. Mathematically it is defined as:

$$\gcd(a,b) = \max_{k=1\ldots\infty \,:\, k|a \,\wedge k \,|b} k.$$

(here the symbol "|" denotes divisibility, i.e. "$k \mid a$" means "k divides a")

Solution

The algorithm is extremely simple:

$$\gcd(a,b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \pmod b), & \text{otherwise.} \end{cases}$$

```
int gcd (int a, int b)
{
    if (b == 0)
        return a;
    else
        return gcd (b, a % b);
}
```

But C++11 has a built-in fuction to calculate $\gcd$

```
__gcd(a,b) //returns gcd(a,b)
```

Time Complexity is $\mathcal{O}(\log\{\min\{a,b\}\})$

17.11 Modular Inverse

The inverse of $x \pmod{m}$ is a number x^{-1} such that

$$xx^{-1} \pmod{m} \equiv 1$$

Using modular inverses, we can divide numbers modulo m, because division by x corresponds to multiplication by x^{-1}. For example, to evaluate the value of $36/6 \pmod{17}$, we can use the formula $2 \times 3 \pmod{17}$, because $36 \pmod{17} = 2$ and $6^{-1} \pmod{17} = 3$.

However, a modular inverse does not always exist. For example, if $x = 2$ and $m = 4$, the equation

$$xx^{-1} \pmod{m} = 1$$

cannot be solved, because all multiples of 2 are even and the remainder can never be 1 when $m = 4$. It turns out that the value of $x^{-1} \bmod m$ can be calculated exactly when x and m are coprime.

A short one-liner to compute modular inverse when x and m are coprime is shown below

```
long long int inv(long long int x, long long int m)
{
    return 1<x ? m - inv(m%x,x)*m/x : 1;
}
```

If m is prime modular inverse can be calculate by: $x^{-1} = x^{m-2}$. This implementation is shown below(using modular exponentiation):

```
int power(int x, int n, int m)
{
    if (n == 0) return 1%m;
    long long u = power(x,n/2,m);
    u = (u*u)%m;
    if (n%2 == 1) u = (u*x)%m;
    return u;
}
int inv(long long int x, long long int m)
{
    return power(x, m-2, m)
    //return x^{m-2} mod m
}
```

Time complexity of both these implementations is $\mathcal{O}(\log m)$

17.12 Chinese Remainder Theorem

Problem: Find x that satisfies the following equations:

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ x &\equiv a_3 \pmod{m_3} \\ &\cdots \\ x &\equiv a_n \pmod{m_n} \end{aligned}$$

where all pairs of $m_1, m_2, \ldots, m_n$ are coprime.

Solution

Let $M = m_1 \times m_2 \times m_3 \times \cdots \times m_n$
Let $M_1, M_2, M_3, \cdots, M_n$ be such that

$$\begin{aligned} M_1 &= M/m_1 \\ M_2 &= M/m_2 \\ M_3 &= M/m_3 \\ &\cdots \\ M_n &= M/m_n \end{aligned}$$

Let $y_1, y_2, y_3, \cdots, y_n$ be such that y_i is the modular inverse of M_i i.e.,

$$\begin{aligned} M_1 \times y_1 &\equiv 1 \pmod{m_1} \\ M_2 \times y_2 &\equiv 1 \pmod{m_2} \\ M_3 \times y_3 &\equiv 1 \pmod{m_3} \\ &\cdots \\ M_n \times y_n &\equiv 1 \pmod{m_n} \end{aligned}$$

Then $x \equiv a_1 M_1 y_1 + a_2 M_2 y_2 + a_2 M_2 y_2 + \cdots + a_n M_n y_n \pmod{M}$

Once we have found a solution x, we can create an infinite number of other solutions, because all numbers of the form

$$x + kM$$

where k is any whole number are solutions.

Example.

$$\begin{aligned} x &\equiv 3 \pmod{8} \\ x &\equiv 1 \pmod{9} \\ x &\equiv 4 \pmod{11} \end{aligned}$$

$\therefore M = 8 \times 9 \times 11 = 792$

$$\begin{aligned} M_1 &= 792/8 = 99 \\ M_2 &= 792/9 = 88 \\ M_3 &= 792/11 = 72 \end{aligned}$$

$$
\begin{aligned}
99 \times y_1 &\equiv 1 \pmod 8 \text{ or } y_1 = 3 \\
88 \times y_2 &\equiv 1 \pmod 9 \text{ or } y_2 = 4 \\
72 \times y_3 &\equiv 1 \pmod{11} \text{ or } y_3 = 2
\end{aligned}
$$

$\therefore x = 3 \times 99 \times 3 + 1 \times 88 \times 3 + 4 \times 72 \times 2 = 1819$
$\therefore x$ can be $1819, 2611, 3403 \cdots$

Part VII

Question Bank

Chapter 18

ZCO Questions

Zonal **C**omputing **O**lympiad is the first stage of the Indian Computing Olympiad.

The Indian Computing Olympiad is used to select the team of four students to represent India at the International Olympiad for Informatics (IOI). IOI is one of the twelve international Science Olympiads held annually. Other prominent Science Olympiads include Mathematics, Physics, Chemistry, Biology and Astronomy.

The problems are quite easy and mostly related to these topics:

- Basic Maths
- Dynamic Programming
- Greedy Algorithms
- Sorting

You can see and submit solutions of the problems on Codechef: `https://www.codechef.com/ZCOPRAC`.

The question bank starts from the next page.

18.1 Smart Phone

URL: https://www.codechef.com/ZCOPRAC/problems/ZCO14003

Problem Code: ZCO14003

Year: ZCO 2014

***Problem*:** You are developing a smartphone app. You have a list of potential customers for your app. Each customer has a budget and will buy the app at your declared price if and only if the price is less than or equal to the customer's budget.

You want to fix a price so that the revenue you earn from the app is maximised. Find this maximum possible revenue.

For instance, suppose you have 4 potential customers and their budgets are 30, 20, 53 and 14. In this case, the maximum revenue you can get is 60.

***Input Format*:** Line 1: N, the total number of potential customers.

Lines 2 to $N+1$: Each line has the budget of a potential customer.

***Output Format*:** The output consists of a single integer, the maximum possible revenue you can earn from selling your app.

Example

```
4
30
20
53
14
```

```
60
```

Example

```
5
40
3
65
33
21
```

```
99
```

Constraints:

Each customers' budget is between 1 and 10^8, inclusive.

$1 \leq N \leq 5 \times 10^5$

Note: The answer might not fit in a variable of type `int`. We recommend that you use variables of type `long long` to read the input and compute the answer. If you use `printf` and `scanf`, you can use `%lld` for `long long`.

Solution

The key observation in this problem is that our optimal budget will be among the customer's budget. In the first example our optimal budget can be 30 or 20.

Naive Solution: Simply try all combinations by putting price equal to every element one by one and check the number of integers/elements bigger than this price. Multiplying price and the bigger numbers will give the revenue for that price. Take the maximum of all such revenues and output it. Since we are trying every element and checking the bigger numbers in whole array takes $\mathcal{O}(N)$, total time complexity comes out to be $\mathcal{O}(N^2)$.

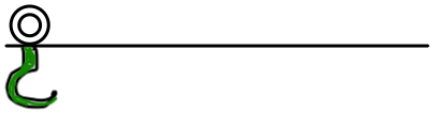
Correct solution: First sort the array. Then take the maximum of $a[i] * (N - i)$ for all i in the range $[0, N)$. This works because all elements after the i^{th} element(including the i^{th} element) are greater than or equal to $a[i]$ and there are $N - i$ elements after i including i itself.

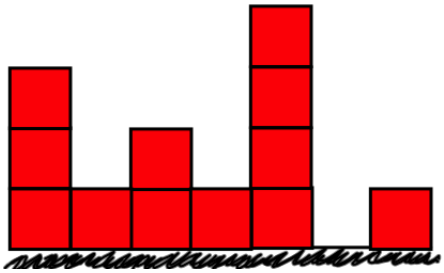
Code: `https://repl.it/@SalilGokhale/Smart-Phone`

18.2 Video Game

URL: `https://www.codechef.com/ZCOPRAC/problems/ZCO14001`
Problem Code: ZCO14001
Year: ZCO 2014

***Problem*:** You are playing a video game in which several stacks of boxes are lined up on the floor, with a crane on top to rearrange the boxes, as shown in the picture below.





The crane supports the following commands:

- Move one position left (does nothing if already at the leftmost position)
- Move one position right (does nothing if already at the rightmost position)
- Pick up a box from the current stack (does nothing if the crane already has a box)
- Drop a box on the current stack (does nothing if the crane doesn't already have a box)

Further, there is a limit H on the number of boxes on each stack. If a 'drop' command would result in a stack having more than H boxes, the crane ignores this drop command. If the current stack has no boxes, a 'pick up' command is ignored.

You are given the initial number of boxes in each stack and the sequence of operations performed by the crane. You have to compute the final number of boxes in each stack.

For example, suppose the initial configuration of the game is as shown in the figure above, with 7 stacks and $H = 4$. Then, after the following sequence of instructions,

1. Pick up box
2. Move right
3. Move right
4. Move right

5. Move right
6. Drop box
7. Move left
8. Pick up box
9. Move left
10. Drop box

the number of boxes in each stack from left to right would be 2,1,3,1,4,0,1.

***Input Format*:** Line 1 : The width of the game (the number of stacks of boxes), N, followed by the max height H of each stack.

Line 2 : N integers, the initial number of boxes in each stack, from left to right. Each number is $\leq H$.

Line 3 : A sequence of integers, each encoding a command to the crane. The commands are encoded as follows

- $1 \rightarrow$ Move left
- $2 \rightarrow$ Move right
- $3 \rightarrow$ Pick up box
- $4 \rightarrow$ Drop box
- $0 \rightarrow$ Quit

The command Quit(0) appears exactly once, and is the last command.

The initial position of the crane is above the leftmost stack, with the crane not holding any box.

***Output Format*:** A single line with N integers, the number of boxes in each stack, from left to right.

Example

```
7 4
3 1 2 1 4 0 1
3 2 2 2 2 4 1 3 1 4 0
```

```
2 1 3 1 4 0 1
```

Example

```
3 5
2 5 2
3 2 4 2 2 2 1 4 1 1 1 1 0
```

```
1 5 2
```

Constraints:

The number of commands is between 1 and 10^5, inclusive.

$$1 \leq N \leq 10^5$$
$$1 \leq H \leq 10^8$$

Solution

The problem statement is long and intimidating but in reality this is a simple implementation problem. There are no techniques or optimisation. You have to do as directed!!! You can use an array or a vector to store the number of boxes in each stack after every operation.

Code: `https://repl.it/@SalilGokhale/Video-Game`

18.3 Tournament

URL: `https://www.codechef.com/ZCOPRAC/problems/ZCO13001`

Problem Code: ZCO13001

Year: ZCO 2013

***Problem*:** N teams participate in a league cricket tournament on Mars, where each pair of distinct teams plays each other exactly once. Thus, there are a total of $\frac{N \times (N-1)}{2}$ matches. An expert has assigned a strength to each team, a positive integer. Strangely, the Martian crowds love one-sided matches and the advertising revenue earned from a match is the absolute value of the difference between the strengths of the two matches. Given the strengths of the N teams, find the total advertising revenue earned from all the matches.

For example, suppose N is 4 and the team strengths for teams 1, 2, 3, and 4 are 3, 10, 3, and 5 respectively. Then the advertising revenues from the 6 matches are as follows:

Match	Team A	Team B	Ad Revenue
1	1	2	7
2	1	3	0
3	1	4	2
4	2	3	7
5	2	4	5
6	3	4	2

***Input Format*:** Line 1 : A single integer, N.

Line 2 : N space-separated integers, the strengths of the N teams.

***Output Format*:** A single integer, the total advertising revenue from the tournament.

Example

```
4
3 10 3 5
```

```
23
```

Constraints:

In all subtasks, the strength of each team is an integer between 1 and 10^3 inclusive

$$2 \leq N \leq 2 \times 10^5$$

Note: The answer might not fit in a variable of type `int`. We recommend that type `long long` be used for computing all advertising revenues. If you use `printf` and `scanf`, you can use `%lld` for `long long`.

Solution

The first impulse would be to calculate all the match revenues and then sum up. but it would be $\mathcal{O}(n^2)$ which will surely give TLE.
The key to the problem lies in an observation. Let's consider an example and then generalise it.
Let us take a different example than the given sample case.

```
4
3 10 7 5
```

Now the revenues for each match are:

$$(10-5)+(10-7)+(10-3)+(7-5)+(7-3)+(5-3)=23$$

but wait that's not what we will do. Look at it in the following manner:

$$10\times(3-0)+7\times(2-1)+5\times(1-2)+3\times(0-3)$$

If we sort the array then we get `3, 5, 7, 10`.
More formally we can write:
If the array is sorted then,

$$\text{Total Revenue}=\sum_{i=0}^{n-1} strength[i]\times(2i-n-1)$$

But make sure to sort the array first. Time Complexity is $\mathcal{O}(n\log n)$
Code: `https://repl.it/@SalilGokhale/Tournament`

18.4 Variation

URL: https://www.codechef.com/ZCOPRAC/problems/ZCO15002

Problem Code: ZCO15002

Year: ZCO 2015

***Problem*:** We say that two integers x and y have a variation of at least K, if $|x-y| \geq K$ (the absolute value of their difference is at least K). Given a sequence of N integers $a_1, a_2, \cdots, a_N$ and K, the total variation count is the number of pairs of elements in the sequence with variation at least K, i.e. it is the size of the set of pairs

$$(i,j) : 1 \leq i \leq j \leq N \text{and} \mid a_i - a_j \mid \geq Kj$$

For example if $K = 1$ and the sequence is 3,2,4 the answer is 3. If $K = 1$ and the sequence is 3, 1, 3 then the answer is 2.

Your task is to write a program that takes a sequence and the value K as input and computes the total variation count.

***Input Format*:** The first line contains two positive integers N and K, separated by a space.

This is followed by a line containing N integers separated by space giving the values of the sequence.

***Output Format*:** A single integer in a single line giving the total variation count.

Example

```
3 1
3 1 3
```

```
2
```

Constraints:

You may assume that all integers in the input are in the range 0 to 10^8 inclusive.

$1 \leq N \leq 65000, 1 \leq K \leq 10^8$

Solution

The first thing to note is that we have to take the absolute value of the difference of the two elements. This simply means that the order in which we discover the elements, (a_i, a_j) satisfying the condition, $(a_i - a_j \geq k)$ doesn't matter. Also, the constraints are small enough to allow sorting. So, we just sort the array.

Now that the array is sorted, we can naively check for the left most element a_1, how many a_j satisfy the given condition. This can be done in $O(n^2)$. But this is not good enough to pass the second sub-task.

Taking it a step further - as the elements are sorted - we can see that a sliding window would do the job in $\mathcal{O}(n)$ time.

In this, we'll iterate over the array maintaining two indices, a and b starting from the leftmost part of the array. The basic aim is to find the first $a[j]$ for a particular i, such that it's the smallest element, where $a[j] - a[i] \geq k$. This can be done by incrementing i, if $a[j] - a[i] \geq k$, else increment j. For a particular i, the number of matching pairs will be $N - j$. The sum of matching elements for all i is the answer.

Time Complexity is $\mathcal{O}(n \log n)$

Code: `https://repl.it/@SalilGokhale/Variation`

18.5 Matched Brackets

URL: https://www.codechef.com/ZCOPRAC/problems/ZCO12001

Problem Code: ZCO12001

Year: ZCO 2012

***Problem*:** A sequence of opening and closing brackets is well-bracketed if we can pair up each opening bracket with a matching closing bracket in the usual sense. For instance, the sequences (), (()) and ()(()) are well-bracketed, while (, ()), (()(), and)(are not well-bracketed.

The nesting depth of a well-bracketed sequence tells us the maximum number of levels of inner matched brackets enclosed within outer matched brackets. For instance, the nesting depth of () and ()()() is 1, the nesting depth of (()) and ()(()) is 2, the nesting depth of ((())) is 3, and so on.

Given a well-bracketed sequence, we are interested in computing the following:

- The nesting depth, and the first position where it occurs-this will be the position of the first opening bracket at this nesting depth, where the positions are numbered starting with 1.

- The maximum number of symbols between any pair of matched brackets, including both the outer brackets, and the first position where this occurs-that is, the position of the first opening bracket of this segment

For instance, the nesting depth of ()(())()(()())(()()) is 2 and the first position where this occurs is 4. The opening bracket at position 10 is also at nesting depth 2 but we have to report the first position where this occurs, which is 4.

In this sequence, the maximum number of symbols between a pair of matched bracket is 6, starting at position 9. There is another such sequence of length 6 starting at position 15, but this is not the first such position.

***Input Format*:** The input consists of two lines. The first line is a single integer N, the length of the bracket sequence. Positions in the sequence are numbered $1,2,\cdots,N$. The second line is a sequence of N space-separated integers that encode the bracket expression as follows: 1 denotes an opening bracket (and 2 denotes a closing bracket). Nothing other than 1 or 2 appears in the second line of input and the corresponding expression is guaranteed to be well-bracketed.

***Output Format*:** Your program should print 4 space-separated integers in a line, denoting the four quantities asked for in the following order: nesting depth, first position that achieves the nesting depth, length of the maximum sequence between matching brackets and the first position where such a maximum length sequence occurs

Example

```
20
1 2 1 1 2 2 1 2 1 1 2 1 2 2 1 1 2 1 2 2
```

```
2 4 6 9
```

Constraints:
You may assume that $2 \leq N \leq 10^5$. In 30% of the test cases, $2 \leq N \leq 10^3$.

Solution

Maximum depth: To find the maximum depth, you can maintain variables `depth` and `maximum_depth`. Increment it if character is 1 and decrement it if character is 2. Each time you increment it, you can check if `depth` exceeds `maximum_depth` and update `maximum_depth` to `depth`. You must store the index when you update `maximum_depth`.

Maximum length: A key observation while solving this problem is that the maximum length will be between 2 matching brackets with depth **0** i.e. the brackets are not nested. You can start counting the length when you encounter a opening bracket with depth 0 and stop counting when you find the corresponding closing bracket. The maximum of all such lengths will be the answer. Finding the index is a trivial problem.
Take care of 0-indexing and edge cases!!
Code: `https://repl.it/@SalilGokhale/Matched-Brackets`

18.6 Chewing

URL: https://www.codechef.com/ZCOPRAC/problems/ZCO13003

Problem Code: ZCO13003

Year: ZCO 2013

***Problem*:** Hobbes has challenged Calvin to display his chewing skills and chew two different types of Chewing Magazine's Diabolic Jawlockers chewing gum at the same time. Being a generous sort of tiger, Hobbes allows Calvin to pick the two types of gum he will chew.

Each type of chewing gum has a hardness quotient, given by a non-negative integer. If Calvin chews two pieces of gum at the same time, the total hardness quotient is the sum of the individual hardness quotients of the two pieces of gum.

Calvin knows that he cannot chew any gum combination whose hardness quotient is K or more. He is given a list with the hardness quotient of each type of gum in the Diabolic Jawlockers collection. How many different pairs of chewing gum can Calvin choose from so that the total hardness quotient remains strictly below his hardness limit K?

For instance, suppose there are 7 types of chewing gum as follows:

Chewing Gum Type	1	2	3	4	5	6	7
Hardness Quotient	10	1	3	1	5	5	0

If Calvin's hardness limit is 4, there are 4 possible pairs he can choose: type 2 and 7 $(1+0<4)$, type 3 and 7 $(3+0<4)$, type 2 and 4 $(1+1<4)$ and type 4 and 7 $(1+0<4)$.

***Input Format*:** Line 1 : Two space separated integers N and K, where N is the number of different types of chewing gum and K is Calvin's hardness limit.

Line 2: N space separated non-negative integers, which are the hardness quotients of each of the N types of chewing gum.

***Output Format*:** The output consists of a single non-negative integer, the number of pairs of chewing gum with total hardness quotient strictly less than K.

Example

```
7 4
10 1 3 1 5 5 0
```

```
4
```

Constraints:

In all subtasks, you may assume that all the hardness quotients as well as the hardness limit K are between 0 and 10^6 inclusive.

$2 \leq N \leq 10^5$

Note: The answer might not fit in a variable of type `int`. We recommend that type `long long` be used for computing the answer. If you use `printf` and `scanf`, you can use `%lld` for `long long`.

Solution

Naive Solution: Checking all pairs and checking if their sum is below k gives a time complexity of $\mathcal{O}(n^2)$ and TLE.

This problem can be solved using the two pointer-technique. We first sort the array. Then we apply the two-pointer technique. Whenever we get two pointer i, j such that `hardness[i] + hardness[j < k`, we add $j - i$ to a variable `sum` because all such i and j can be paired up. We stop when $i == j$
Code: `https://repl.it/@SalilGokhale/Chewing`

18.7 SUPW

URL: https://www.codechef.com/ZCOPRAC/problems/ZCO14002

Problem Code: ZCO14002

Year: ZCO 2014

***Problem*:** In ICO School, all students have to participate regularly in SUPW. There is a different SUPW activity each day, and each activity has its own duration. The SUPW schedule for the next term has been announced, including information about the number of minutes taken by each activity.

Nikhil has been designated SUPW coordinator. His task is to assign SUPW duties to students, including himself. The school's rules say that no student can go three days in a row without any SUPW duty.

Nikhil wants to find an assignment of SUPW duty for himself that minimizes the number of minutes he spends overall on SUPW.

***Input Format*:** Line 1: A single integer N, the number of days in the future for which SUPW data is available.

Line 2: N non-negative integers, where the integer in position i represents the number of minutes required for SUPW work on day i.

***Output Format*:** The output consists of a single non-negative integer, the minimum number of minutes that Nikhil needs to spend on SUPW duties this term

Example

```
10
3 2 1 1 2 3 1 3 2 1
```

```
4
```

Example

```
8
3 2 3 2 3 5 1 3
```

```
5
```

Constraints:

$1 \leq N \leq 2 \times 10^5$

The number of minutes of SUPW each day is between 0 and 10^4, inclusive

Solution

Code: https://repl.it/@SalilGokhale/Smart-Phone