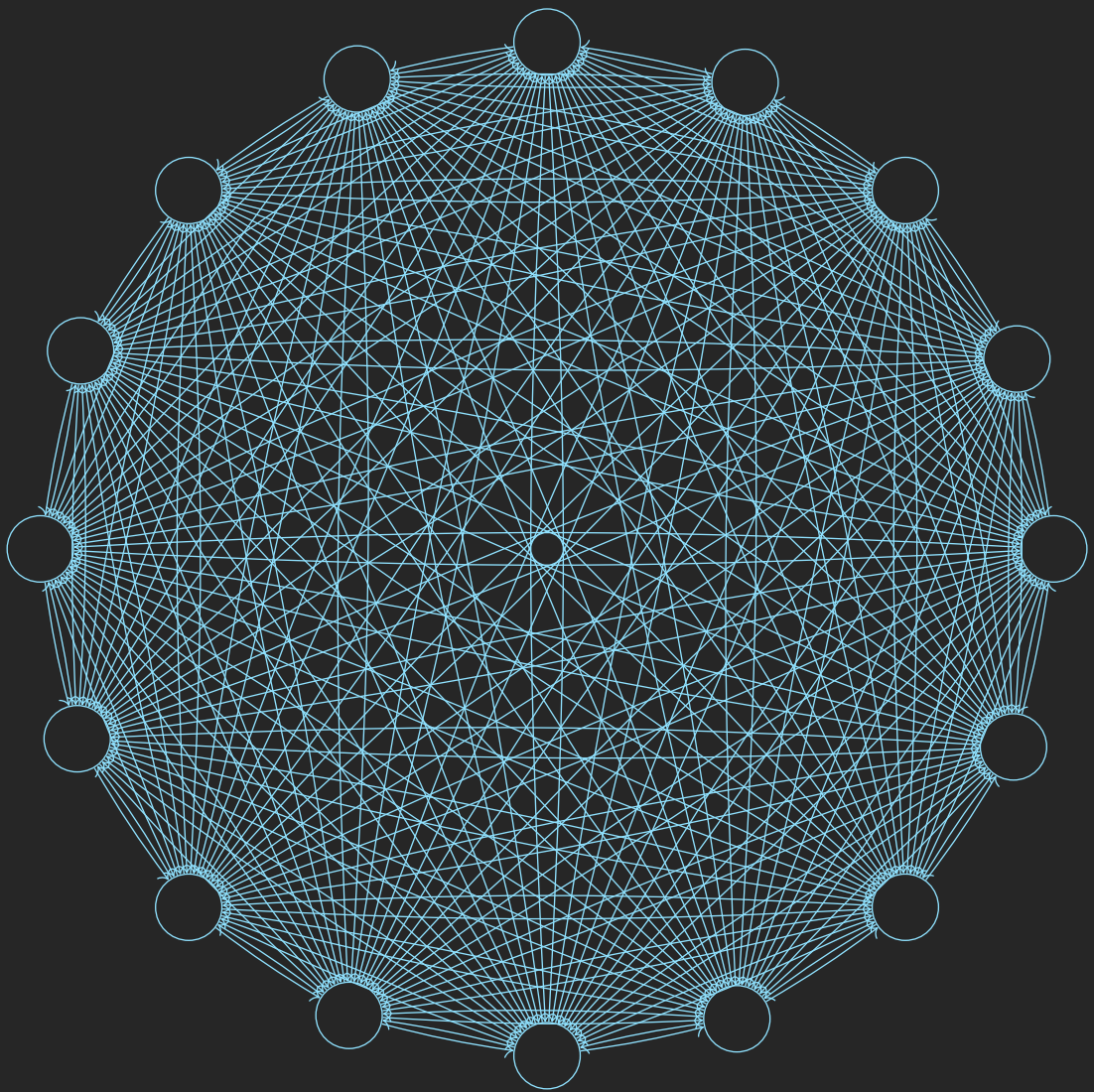
Competitive Programming Notebook

Salil Gokhale

Draft January 24, 2019



Contents

I Basic techniques **2**

1 C++ Tricks **4**

1.1 Special functions . . . 4

1.1.1 Copy Elements . . . 4

1.1.2 Ceiling Division $\lceil \frac{x}{y} \rceil$ . . . 4

1.1.3 Ceiling and Floor round-off . . . 4

1.1.4 Use `emplace back()` . . . 4

1.1.5 Iota . . . 5

1.1.6 Insert value in middle of vector . . . 5

1.1.7 Accurate value of π . . . 5

1.1.8 GCD of Two Numbers . . . 5

1.1.9 Range Queries . . . 5

1.1.10 Initialize number in binary form . . . 5

1.2 Input and output . . . 6

2 Sorting Algorithms **7**

2.1 Problem Statement of Sorting Algorithms . . . 7

2.2 Properties of Sorting Algorithms . . . 7

2.2.1 Types of Sorting Algorithms . . . 7

2.2.2 Stability . . . 8

2.2.3 Adaptability . . . 8

2.2.4 Memory Usage . . . 8

2.2.5 Computational Complexity . . . 8

2.3 Common $\mathcal{O}(n^2)$ Algorithms . . . 8

2.3.1 Bubble Sort . . . 8

2.3.2 Insertion Sort . . . 9

2.4 Common $\mathcal{O}(n \log n)$ Algorithms . . . 9

2.4.1 Merge Sort . . . 9

2.4.2 Quick Sort . . . 10

2.4.3 Heap Sort . . . 12

2.4.4 Counting Sort . . . 12

2.5 Comparison of Sorting Algorithms . . . 12

2.6 Sorting in C++ . . . 12

3 Bit Manipulation **14**
3.1 Tricks . . . 14
3.1.1 Multiply and divide by 2^i . . . 14
3.1.2 Checking if a number is odd or even . . . 14
3.1.3 Swapping of 2 numbers using XOR . . . 14
3.1.4 Compute XOR from 1 to n (direct method) . . . 15
3.1.5 Check if a number is a power of 2 . . . 16
3.1.6 Change case of English alphabet . . . 16
3.1.7 Find $\log_2 x$ of 32-bit integer . . . 16
3.2 Bit Shift . . . 16
3.2.1 Application . . . 16
3.3 C++ Special Functions . . . 18
3.4 Set Representation . . . 18
3.4.1 Set implementation . . . 18
3.4.2 Set Operations . . . 19
3.5 Example Problems . . . 19
3.5.1 Counting Grids with Black Corners . . . 19

II Dynamic Programming **22**

4 Common DP Problems **24**
4.1 Rod Cutting . . . 24
4.2 Longest Increasing Subsequence(LIS) . . . 25
4.3 Longest Common Subsequence(LCS) . . . 26
4.4 Tiling Problem . . . 27
4.5 Coin - 1 . . . 29
4.6 Coin - 2 . . . 30
4.7 Largest Sum Contiguous Subarray . . . 31
4.8 0-1 Knapsack Part 1 . . . 32
4.9 0 - 1 Knapsack Part 2 . . . 33
4.10 Maximum Sum Path in a Grid . . . 34
4.11 Edit Distance . . . 35
4.12 Express n as k partitions . . . 37

III Mathematics **38**

5 Basic Maths **40**
5.1 Integer bounds . . . 40
5.2 Method for integers $> 2^{64} - 1 \approx 1.8 \times 10^{19}$ . . . 40
5.3 Modular arithmetic . . . 40
5.4 Floating point numbers . . . 41
5.5 Mathematics . . . 41
5.5.1 Sum formulas . . . 41
5.5.2 Sum of Arithmetic Progression . . . 41
5.5.3 Sum of Geometric Progression . . . 42

6 Number Theory **43**
6.1 Primality check 43
6.2 Sieve of Eratosthenes 44
6.3 Prime Factorization 45
6.4 Common Conjectures 46
6.5 Modular Exponentiation 47

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

C++ Tricks

1.1 Special functions

1.1.1 Copy Elements

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

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

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

1.1.3 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
```

1.1.4 Use `emplace back()`

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

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

1.1.5 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
```

1.1.6 Insert value in middle of vector

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

1.1.7 Accurate value of π

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

1.1.8 GCD of Two Numbers

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

1.1.9 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());
```

1.1.10 Initialize number in binary form

```
auto number = 0b011; //Initialize number in binary form(3)
```

1.2 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 2

Sorting Algorithms

2.1 Problem Statement of Sorting Algorithms

***Problem* 2.1.1:** Given an array that contains n elements, your task is to sort the elements in increasing order.

Example 2.1.1. 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

2.2 Properties of Sorting Algorithms

2.2.1 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 ther is no need to rearrange lists.

2.2.2 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.

2.2.3 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.

2.2.4 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".

2.2.5 Computational Complexity

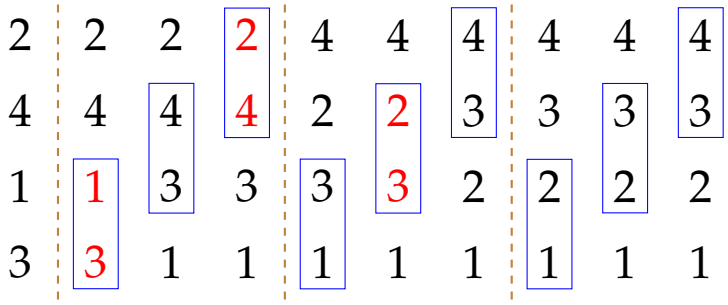
For typical serial sorting algorithms good behaviour is $\mathcal{O}(n \log n)$, with parallel sort in $\mathcal{O}(\log^2 n)$, and bad behaviour is $\mathcal{O}(n^2)$.

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

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

2.3.1 Bubble Sort

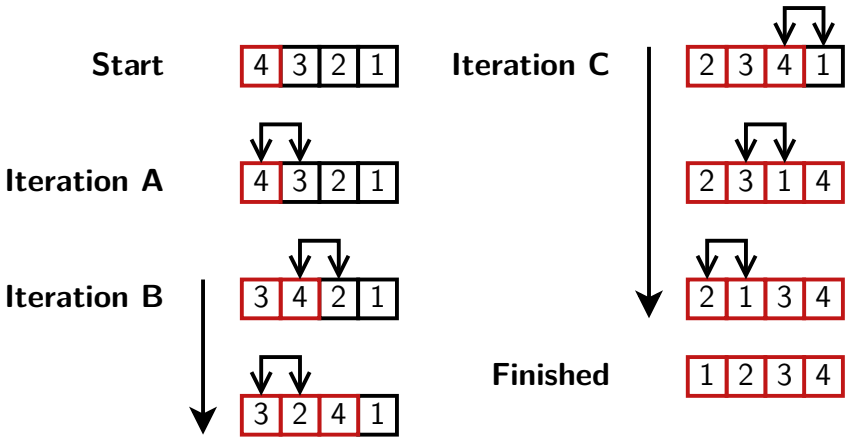
Bubble sort is a simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again 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

2.3.2 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

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

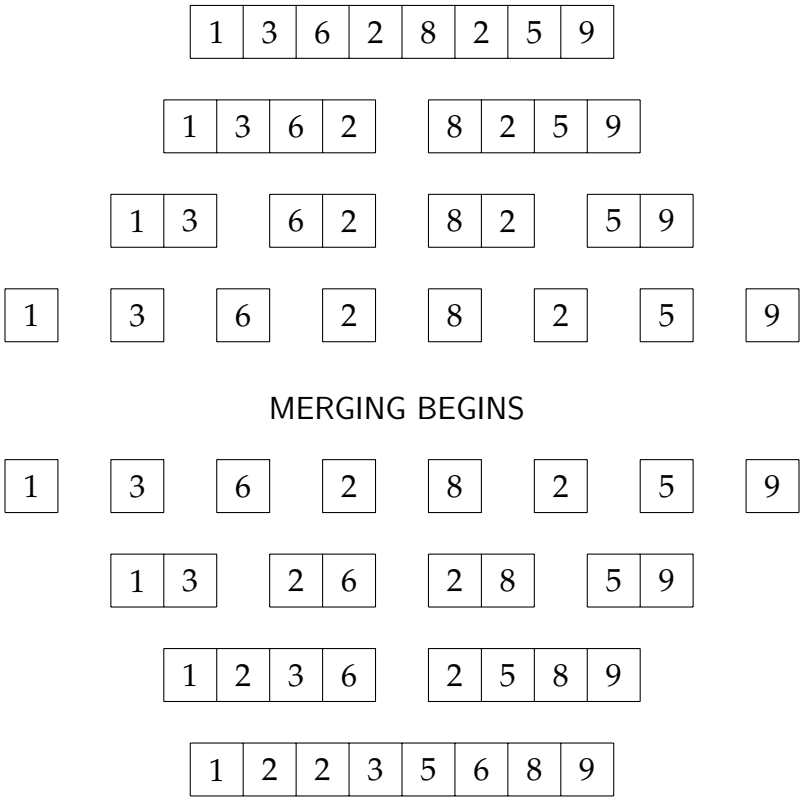
2.4.1 Merge Sort

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:



2.4.2 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.

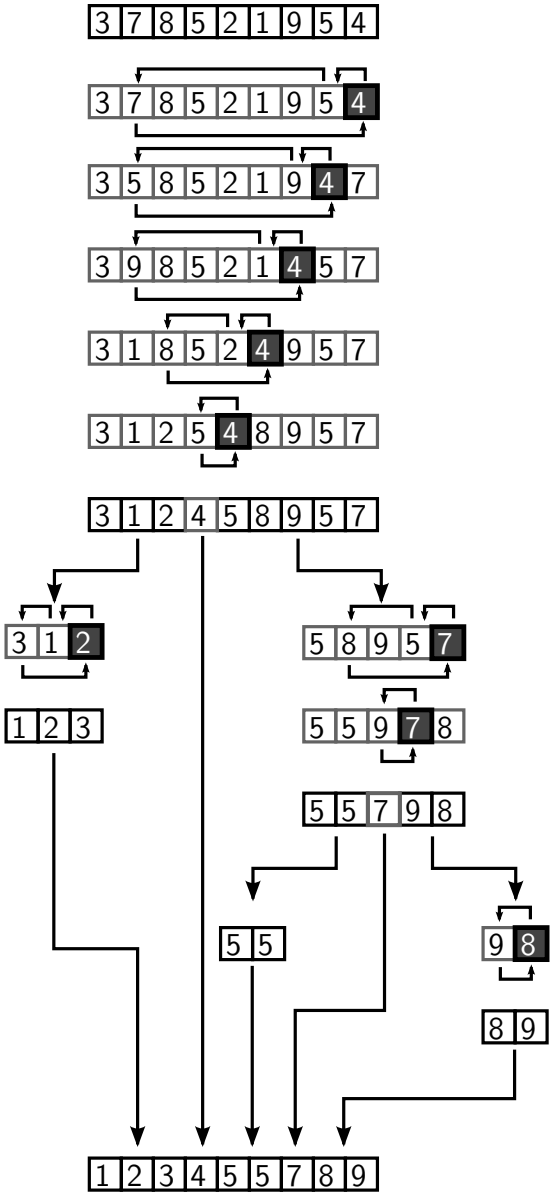
2. Partition, or rearrange, the array into two subarrays: $A[p,...,q-1]$ such that all elements are less than $A[q]$, and $A[q+1,...,r]$ such that all elements are greater than or equal to $A[q]$.

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

2.4.3 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.

2.4.4 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).

2.5 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

2.6 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 3

Bit Manipulation

3.1 Tricks

3.1.1 Multiply and divide by 2^i

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

3.1.2 Checking if a number is odd or even

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

3.1.3 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;
```

3.1.4 Compute XOR from 1 to n (direct method)

Algorithm 1: 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
```

3.1.5 Check if a number is a power of 2

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

3.1.6 Change case of English alphabet

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

3.1.7 Find $\log_2 x$ of 32-bit integer

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

3.2 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

3.2.1 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.

3.3 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).

3.4 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 $2^8+2^4+2^3+2^1=282$.

3.4.1 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
```

3.4.2 Set Operations

Set operations can be implemented as follows as bit operations:

	set syntax	bit syntax
intersection	$a \cap b$	a & b
union	$a \cup b$	$a \mid b$
complement	$\bar{a}$	$\sim a$
difference	$a \setminus b$	a & $(\sim b)$

Iteration Through Subsets

The following code goes through the subsets of $\{0,1,\ldots,n-1\}$:

```
for (int b = 0; b < (1<<n); b++) {
    // process subset b
}
```

The following code goes through the subsets with exactly k elements:

```
for (int b = 0; b < (1<<n); b++) {
    if (__builtin_popcount(b) == k) {
        // process subset b
    }
}
```

The following code goes through the subsets of a set x:

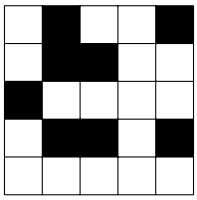
```
int b = 0;
do {
    // process subset b
} while (b=(b-x)&x);
```

3.5 Example Problems

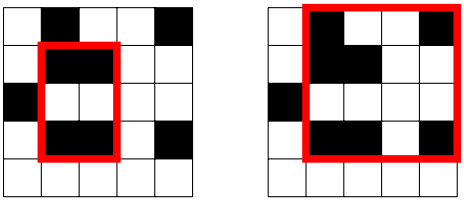
3.5.1 Counting Grids with Black Corners

***Problem* 3.5.1:** 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 3.5.1. 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.

Part II

Dynamic Programming

Chapter 4

Common DP Problems

4.1 Rod Cutting

***Problem* 4.1.1 (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 4.1.1. 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 $\mathcal{O}(n^2)$

4.2 Longest Increasing Subsequence(LIS)

***Problem* 4.2.1 (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 4.2.1. 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 $\mathcal{O}(n^2)$

4.3 Longest Common Subsequence(LCS)

***Problem* 4.3.1:** 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 4.3.1. LCS for input Sequences "ABCDGH" and "AEDFHR" is "ADH" of length 3.

Example 4.3.2. 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 $\mathcal{O}(nm)$

4.4 Tiling Problem

***Problem* 4.4.1:** 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 4.4.1. Input n = 3

Output: 3

Explanation:

We need 3 tiles to tile the board of size 2 x 3.

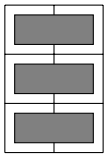
We can tile the board using following ways

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 4.4.2. Input n = 4

Output: 5

Explanation:

For a 2 x 4 board, there are 5 ways

1) All 4 vertical

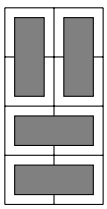
2) All 4 horizontal

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));
}
```

4.5 Coin - 1

***Problem* 4.5.1:** 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 4.5.1. $n = 4$ and `S = [1,2,3]` and answer $= 4$
There are four solutions: `[1,1,1,1];[1,1,2];[2,2];[1,3]`

Example 4.5.2. $n = 10$ 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:

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

```
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 $\mathcal{O}(nm)$

4.6 Coin - 2

***Problem* 4.6.1:** 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 4.6.1. `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 4.6.2. `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 $\mathcal{O}(nm)$

4.7 Largest Sum Contiguous Subarray

***Problem* 4.7.1:** 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 4.7.1. 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 max_sum(int A[], int n)
{
    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
    return ans; //ans is the final answer
}
```

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

4.8 0-1 Knapsack Part 1

***Problem* 4.8.1:** 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 4.8.1.

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.

4.9 0 - 1 Knapsack Part 2

***Problem* 4.9.1:** Given a list of weights $[w_1, w_2, \ldots, w_n]$, determine all sums that can be constructed using the weights

Example 4.9.1. 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 $\mathcal{O}(n \times total)$

4.10 Maximum Sum Path in a Grid

***Problem* 4.10.1:** 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.

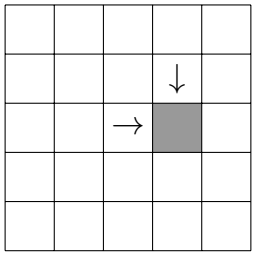
Example 4.10.1.

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 $\mathcal{O}(n^2)$

4.11 Edit Distance

***Problem* 4.11.1:** 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 4.11.1. Input: str1 = "geek", str2 = "gesek"
Output: 1
We can convert str1 into str2 by inserting a 's'.

Example 4.11.2. Input: str1 = "cat", str2 = "cut"
Output: 1
We can convert str1 into str2 by replacing 'a' with 'u'.

Example 4.11.3. 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:

		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

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 $\mathcal{O}(mn)$

4.12 Express n as k partitions

***Problem* 4.12.1:** Given an integer n, how many ways can K non-negative integers less than or equal to n add up to n?

Example 4.12.1. $N = 5$, $K = 3$

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 4.12.2. $N = 10$, $K = 4$

Output: 84

Solution

This is also called as the Binomial Coefficiant Problem

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

    // Caculate 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 previosly 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

Mathematics

Chapter 5

Basic Maths

5.1 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` : $2^{64}-1 \approx 1.8 \times 10^{19}$

5.2 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
}
```

5.3 Modular arithmetic

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

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

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";
```

5.4 Floating point numbers

Printing floating point numbers up to n digits.

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

5.5 Mathematics

5.5.1 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

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

5.5.2 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.

5.5.3 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 6

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

6.1 Primality check

***Problem* 6.1.1:** 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;
}
```

6.2 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* 6.2.1:** 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' 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 $\mathcal{O}(n)$

6.3 Prime Factorization

***Problem* 6.3.1:** 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 precompute prime numbers. It has time complexity $\mathcal{O}(\log n)$. You can read about it on GeeksforGeeks or CP3 book.

6.4 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 upto 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)

6.5 Modular Exponentiation

***Problem* 6.5.1:** 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;
}
```