# Competitive Programming Short Notes

*Based on C++ Language*

Aong Cho Marma

The Publisher $\boxed{\mathcal{PL}}$

# Contents

# Chapter 1

# INTRODUCTION

## 1.1 Basic Competitive Programming

Competitive programming is a sport of coding where participants solve complex algorithmic problems under time constraints. Using programming languages, they devise efficient solutions that navigate data structures, math, and logic. This intellectually stimulating activity hones problem-solving skills and fosters creativity, often practiced through online platforms and contests.

Chapter 2

# The C++ Standard Template Library (STL)

## 2.1 What is STL ?

The C++ Standard Template Library (STL) is a collection of algorithms, data structures, and other components that can be used to simplify the development of C++ programs. The STL provides a range of containers, such as vectors, lists, and maps, as well as algorithms for searching, sorting and manipulating data.

**STL has 4 components:**

1. Algorithms
2. Containers
3. Function Objects or Functors
4. Iterators

## 2.2 Containers in C++ STL

A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows great flexibility in the types supported as elements.
The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

Table 2.1: Containers

| **Sequence Containers** | **Associative Containers** | **Unordered Associative Containers** | **Container Adapters** |
|---|---|---|---|
| Array | Set | Unordered Set | Stack |
| Vector | Map | Unordered Map | Queue |
| Deque | Multiset | Unordered Multiset | Priority Queue |
| Forward List | Multimap | Unordered Multimap | — |
| List | — | — | — |

**Note:** All STL containers are passed by value.

### 2.2.1 Sequence Containers

Sequence containers implement data structures that can be accessed sequentially.

- **array:** Static contiguous array (class template)
- **vector:** Dynamic contiguous array (class template)
- **deque:** Double-ended queue (class template)
- **forward-list:** Singly-linked list (class template)
- **list:** Doubly-linked list (class template)

### 2.2.2 Associative Containers

Associative containers implement sorted data structures that can be quickly searched ($O(\log n)$ complexity).

- **set:** Collection of unique keys, sorted by keys (class template)
- **map:** Collection of key-value pairs, sorted by keys, keys are unique (class template).
- **multiset:** Collection of keys, sorted by keys (class template)
- **multimap:** Collection of key-value pairs, sorted by keys (class template)

### 2.2.3 Unordered associative containers

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched (O(1) amortized, O(n) worst-case complexity).

- **unordered_set:** Collection of unique keys, hashed by keys. (class template)

- **unordered_map:** Collection of key-value pairs, hashed by keys, keys are unique. (class template)
- **unordered_multiset:** Collection of keys, hashed by keys (class template)
- **unordered_multimap:** Collection of key-value pairs, hashed by keys (class template)

### 2.2.4 Container Adapters

Container adapters provide a different interface for sequential containers.

- **atack:** Adapts a container to provide stack (LIFO data structure) (class template).
- **queue:** Adapts a container to provide queue (FIFO data structure) (class template).
- **priority_queue:** Adapts a container to provide priority queue (class template).

## 2.3 rand() in C++

`rand()` function is an inbuilt function in C++ STL, which is defined in header file `<cstdlib>`. rand() is used to generate a series of random numbers. The random number is generated by using an algorithm that gives a series of non-related numbers whenever this function is called. The `rand()` function is used in C++ to generate random numbers in the range `[0, RAND_MAX)`.

- This function does not take any parameters;
- Time Complexity O(1)
- Space Complexity O(1)

## 2.4 Array STL

The `array` is a collection of homogeneous objects and this array container is defined for constant size arrays or (static size). This container wraps around fixed-size arrays and the information of its size are not lost when declared to a pointer. In order to utilize arrays, we need to include the array header: `#include <array>`

**Syntax:** `array<object_type, arr_size> arr_name;`

- **Pass by value:**
  ```
  void updateArray(array<object_type,arr_size) array_name{
       Code;
  }
  ```
- **Pass by reference:**
  ```
  void updateArray(array<object_type,arr_size) &array_name{
       Code;
  }
  ```

### 2.4.1 Operations on Array

- **size() Function:** This method returns the maximum size of the array.
  ```
  arrayname.size()
  Parameters :  No parameters are passed.
  Returns :  Number of elements in the container.
  ```
- **fill(val) Function:** This method is used to fill the array with `val`
  ```
  fill(arr_name.begin()+indx, arr_name.end()-indx,val)
  ```
- **Sorting Array:**
  Include header `#include<algorithm>`
  ```
  sort(arr_name.begin(),arr_name.end())
  ```

## 2.5 Vector STL

Dynamic array with the ability to re-size itself automatically when an element is inserted
Contiguous store locations, reallocation happens when underlying array is full.

### 2.5.1 Vector Constructor

Ways of defining a vector-

```
#include<vector>
```

- Empty container constructor
  ```
  vector<data_type> vector_name
  ```
- Fill constructor (5 integers with 10 value)
  ```
  vector<int> numbers(5,10)
  ```
- Range constructor
  ```
  int num[] = {10, 20, 30, 40}
  vector<int> numbers(num,num+4)
  ```
- Copy constructor
  ```
  vector<int> copy_num(numbers)
  ```

### 2.5.2 Basic Vector Methods

**Iterators:**

- **[ `begin()` ]:** Returns an iterator pointing to the first element of vector. $O(1)$
- **[ `end()` ]:** Returns an iterator pointing to the element behind the last element of vector. $O(1)$

**Accessing the elements:**

- **[ `at(index)` ]:** Returns a reference to element at position 'index'. $O(1)$
- **[ `front()` ]:** Returns a reference to first element. $O(1)$
- **[ `back()` ]:** Returns a reference to last element. $O(1)$

**Capacity:**

- **[ `size()` ]:** Returns the number of elements in the vector. $O(1)$
- **[ `max_size()` ]:** Returns the maximum number of elements that the vector can hold. $O(1)$
- **[ `capacity()` ]:** Returns the size of the storage space currently allocated. $O(1)$

- [ `resize(n)` ]: Re-sizes the container so that it contains ‘n’ elements. $O(n)$
- [ `empty()` ]: Returns whether the container is empty. $O(1)$

**Modifiers:**

- [ `push_back()` ]: Inserts the elements into a vector from the back. $O(1)$
- [ `pop_back()` ]: Removes the last element from the vector. $O(1)$
- [ `insert()` ]: Inserts new elements before the position specified by the iterator. *Linear* $O(n)$

  Syntax: `vectorName.insert(position, val)`
- [ `erase()` ]: Remove elements from a container form the specified position or range. *Linear* $O(n)$

  Syntax: `vectorName.erase(position)`

  Syntax: `vectorName.erase(startPos, endPos)`
- [ `clear()` ]: Remove all the elements of the container. *Linear* $O(n)$
- [ `assign()` ]: It assigns a new value to the vector elements by replacing old ones. *Linear* $O(n)$

  The syntax for assigning constant values:

  `vectorName.assign(int size, int value)`

  The syntax for assigning values from an array:

  `vectorName.assign(arr, arr+size)`

- **Vector reserve() method**

  ```
  arr.reserve(size)
  ```

  The C++ reserve() function helps us reserve a vector capacity. This capacity must be enough to contain n number of elements. This function can help us increase the capacity of any given vector with a value greater than or equal to the new capacity, which we will specify in the reserve function.

### 2.5.3 Sorting Vector

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main() {
    vector <int> nums{6,4,1,3,5};
    // Ascending Order
    sort(nums.begin(),nums.end());
    // Descending Order
    sort(nums.begin(),nums.end(),greater<>());
    return 0;
}
```

## 2.6 Pair

Pair is a container that stores two data elements in it. It is not necessary that the two values or data elements have to be of the same data type.

Syntax: `pair<dataType1,dataType2>pairName`

### 2.6.1 Pair Initialization

- `pairName = make_pair(1,"Name")`
- `pair pairName(1,"Name")`

### 2.6.2 Methods

- **[** `pairName.first` **]:** Returns the first element of the pair. $O(1)$
- **[** `pairName.second` **]:** Returns the second element of the pair. $O(1)$
- **[** `pair1.swap(pair2` **]:** Swaps the contents of one pair object with the contents of the another pair object. The pair must be of the same type.

## 2.7 Vector of Pairs

Vector of Pairs is a dynamic array filled with pairs instead of any primitive data type.

- ◇ **Declaration:**

  `vector<pair<int,string»p1;`

- ◇ **Add an item:**

  `p1.push_back(make_pair(1,"aaa"));`

  or

  `p1.emplace_back(1,"aaa");`

- ◇ **Delete an item:**

  Delete Last Pair: `p1.pop_back();`

  Delete $i^{th}$ Pair: `p1.erase(a.begin()+i);`

### 2.7.1 Sorting the vector of pair

#### Make vector of pairs

```

#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main() {
    vector <int> arr1{5,3,1,6,5};
    vector <int> arr2{10,2,7,3,5};
    vector <pair<int,int>> vp;

    for(int i=0; i<arr1.size(); i++)
        vp.push_back(make_pair(arr1[i],arr2[i]));

    return 0;
}
```

#### Sort on the basis of first element of pairs ascending order

```

sort(vp.begin(), vp.end());
```

#### Sort on the basis of first element of pairs descending order

```

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

### Sort on the basis of second element of pairs ascending order

```cpp
// Create the sortBySec() function outside of the main() function.

bool sortBySec(const pair<int,int> &a, const pair<int,int> &b) {
  return (a.second < b.second);
}

// Now sort the vector of pair inside the main() function.
sort(vp.begin(), vp.end(), sortBySec);
```

### Sort on the basis of second element of pairs descending order

```cpp

sort(vp.rbegin(), vp.rend(), sortBySec);
```

## 2.8 SET STL

Sets are a type of associative container in which each element has to be unique because the value of the element identifies it. The values are stored in a specific sorted order i.e. either ascending or descending.

```cpp

#include <iostream>
#include <algorithm>
#include <set>
using namespace std;

void show(set<char, greater<char>> &S) {
    for(auto i:S) cout << i << endl;
}

int main() {
    string str = "92341778065310";
    set <char> S;
    set <char, greater<char>> T;

    // Insert elements into the set in ascending order
    for(auto i:str) S.insert(i);
    cout << "Ascending Order:\n";
    for(auto i:S) cout << i << endl;

    // Copy & Store elements into a set in descending order
    set <char,greater<char>> S1(S.begin(), S.end());
    cout << "Descending Order:\n";
    show(S1);

    // Insert elements into a set in descending order
    for(auto i: str) T.insert(i);
    cout << "Value of T:\n";
    show(T);

    return 0;
}
```

## 2.9 Double-ended Queue (deque)

Deque is sequence containers with dynamic sizes that can be expanded or contracted on both ends (front or back)
deques are not guaranteed to store all its elements in contiguous storage locations: accessing elements in deque by offsetting a pointer to another element causes undefined behavior.
Elements of a deque can be scattered in different chunks of storage.
Header file:

```
#include<deque>
```

### 2.9.1 Syntax to creating a deque object

```
deque<object_type> deque_name;
```

**Create a deque:**

- Empty deque of ints
  ```
  deque<int> arr1;
  ```
- Four ints with value 100
  ```
  deque<int> arr2(4,100);
  ```
- Copy using iterators
  ```
  deque<int> arr3(arr2.begin(),arr2.end());
  ```
- Full Copy
  ```
  deque<int> arr4(arr3);
  ```

### 2.9.2 Deque Methods

- **`arr.insert[iterator_position,value]`** : Insert value at iterator_position.
- **`arr[index]`** : Access index value.
- **`arr.at(index)`** : Access index value.
- **`arr.size()`** : It returns the number of elements.
- **`push_back()`** : It adds a new element at the end of the container.
- **`push_front()`** : It adds a new element at the beginning of the container.
- **`pop_back()`** : It deletes the last element from the deque.
- **`pop_front()`** : It deletes the first element from the deque.

## 2.10 STACK

The stack data structure follows the *LIFO* (Last In First Out) principle. That is, the element added last will be removed first.

### 2.10.1 Create a Stack

In order to create a stack in C++, we first need to include the `stack` header file.

Header: `#include<stack>`

Syntax: `stack<type>st`

### 2.10.2 Methods of Stack

In C++, the `stack` class provides various methods to perform different operations on a stack.

- **[ `push()` ]:** Adds an element at the top of the stack. $O(1)$
  Syntax: `st.push(element);`
- **[ `pop()` ]:** Removes an element from the top of the stack. $O(1)$
  Syntax: `st.pop();`
- **[ `top()` ]:** Returns the element at the top of the stack. $O(1)$
  Syntax: `st.top();`
- **[ `size()` ]:** Returns the number of the elements in the stack. $O(1)$
  Syntax: `st.size();`
- **[ `empty()` ]:** Returns `true` if the stack is empty. $O(1)$
  Syntax: `st.empty();`

## 2.11 Queue

The Queue data structure follows the **FIFO (First In First Out)** principle where elements that are added first will be removed first.

### 2.11.1 Create Queue

Include header file

```
#include <queue>
```

Syntax to create Queue:

```
queue <dataType> q;
```

### 2.11.2 Queue Methods

- [ `push()` ]: Insert an element at the end of the queue. $O(1)$
  ```
  q.push(element);
  ```
- [ `pop()` ]: Removes an element from the front of the queue. $O(1)$
  ```
  q.pop();
  ```
- [ `front()` ]: Returns the first element of the queue. $O(1)$
  ```
  q.front();
  ```
- [ `back()` ]: Returns the last element of the queue. $O(1)$
  ```
  q.back();
  ```
- [ `size()` ]: Returns the number of elements in the queue. $O(1)$
  ```
  q.size();
  ```
- [ `empty()` ]: Returns `true` if the queue is empty. $O(1)$
  ```
  q.empty();
  ```

## 2.12 Priority Queue

A priority queue is a special type of queue in which each element is associated with a priority value. And, elements are served on the basis of their priority. That is, higher priority elements are served first.
However, if elements with the same priority occur, they are served according to their order in the queue.

### 2.12.1 Create Priority Queue

Include header file

```
#include <queue>
```

Syntax to create Priority Queue:

```
priority_queue <dataType> pq;
```

### 2.12.2 Priority Queue Methods

- ◇ **[ `push()` ]:** Insert an element in to the priority queue. $O(log(N))$

  ```
  pq.push(element);
  ```

- ◇ **[ `pop()` ]:** Removes the element with highest priority. $O(log(N))$

  ```
  pq.pop();
  ```

- ◇ **[ `top()` ]:** Returns the element with highest priority. $O(1)$

  ```
  pq.top();
  ```

- ◇ **[ `size()` ]:** Returns the number of elements. $O(1)$

  ```
  pq.size();
  ```

- ◇ **[ `empty()` ]:** Returns `true` if the priority_queue is empty. $O(1)$

  ```
  pq.empty();
  ```

### 2.12.3 Min-Heap Priority-Queue

We can also create a **min-heap** `priority_queue` that arranges elements in ascending order. Its syntax is:

```
priority_queue<type, vector<type>, greater<type>> pq;
```

# Chapter 3

# Basic Hashing

Hashing is a technique of mapping keys and values into the hash table by using hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.
Hashing is used to solve a lot of problems involving linked lists, arrays, sliding window, graph algorithms etc.

Hash table of a menu of a restaurant.

| **Key** | **Values** |
|---|---|
| Cold Drink | 20 |
| Pizza | 100 |
| Dosa | 50 |

Hash table supports three basic operations.

| **Operation** | **Time complexity** |
|---|---|
| insertion | $O(1)$ on an average |
| searching | $O(1)$ on an average |
| erase | $O(1)$ on an average |

## 3.1 Containers to Implement Hash Table in C++

`Unordered_Maps` & `Unordered_stes` can be used to implement hash table in C++.

- Time Complexity is $O(1)$ on an average.
- `Unordered_Maps` container stores **`key-value`** pairs. (Similar as `dictionary in` **`Python`**)
- `Unordered_sets` container stores **`key`** only.

These containers can be used to solve-

- **Array** problems
- **Linked Lists** problems
- **Sliding Window** problems
- **Graphs** problems

## 3.2 Unordered_map

Internally `unordered_maps` is implemented using `Hash Table`, the key provided to map is hashed into indices of a hash table which is why the performance of data structure depends on the hash function a lot but on average, the cost of **search insert** and **delete** from the hash table is $O(1)$

> **Note:** In the worst case, its complexity can go from $O(1)$ to $O(n)$, especially for big prime numbers. In this situation, it is highly advisable to use a map instead to avoid a **TLE**(Time Limit Exceeded) error.

**Syntax:**

```cpp
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;
int main() {
  //unordered_map<key, value> containerName;
  unordered_map<string, int> umap;
  ump = {
      {"first",1},
      {"second",2},
      {"third",3}
    };
  return 0;
}
```

### 3.2.1 Methods on unordered_map

◇ **insert():** Insert one or more key-value pairs.

```cpp
ump.insert({{"fourth",4},{"fifth",5}});
ump.insert({"sixth",6});

```

◇ **at():** Returns the element at the specified key.

```cpp
cout << "Value of key ump['first'] = " << ump.at("first") << endl;

```

- **count():** Returns 1 if key exists and 0 if not.

```cpp
cout << "Is ump['first'] exist = " << ump.count("first") << endl;

```

- **find():** Returns an iterator if the given key exists otherwise returns the end of the map iterator.

```cpp
if(ump.find("first") == ump.end()) {
  cout << "'first' key doesn't exist." << endl;
else
  cout << "'first' key exist." << endl;

```

- **size():** Returns the number of elements.

```cpp
cout << "Size of the container = " << ump.size() << endl;

```

- **empty():** Returns `true` if the container is empty.

```cpp
cout << "The container is empty = " << ump.empty() << endl;

```

- **erase():** Removes elements with specified key.

```cpp
//Erase by iterator
ump.erase(ump.end());
//Erase by key
ump.erase("fifth");
//Erase by range
ump.erase(ump.begin(),++ump.begin());

```

- **clear():** Removes all elements.

```cpp
ump.clear();

```

### 3.2.2 Implementation of unordered_map

◇ **Search in $O(1)$ time:**

```cpp
string item; cin >> item;
if(ump.count(item)==0) {
  cout << item << " is not available." << endl;
}
else {
  cout << item << " is available, and its value is " << ump[item] << endl;
}

```

◇ **Iterate over all the key-value pairs:**

```cpp
//for(auto item: ump) also valid
for(pair<string,int> item: ump) {
  cout << item.first << "-" << item.second << endl;
}

```

Chapter 4

# Number Theory

Here we will discuss about the basic number theory related CP problems.

## 4.1 Prime Number

### 4.1.1 Primality Test

Check if a given integer is prime or not.

```

bool isPrime(int n) {
  if(n<2)
    return false;
  else if(n<=3)
    return true;
  else if(n%2==0)
    return false;
  else {
    for(int i=3; i<=sqrt(n); i+=2) {
      if(n%i==0)
        return false;
    }
    return true;
  }
}
```

### 4.1.2 Count Prime Number

Count the total prime number within the given integer number.

#### Sieve of Eratosthenes Technique

```

int countPrimes(int n) {
    vector <bool> isPrime;
    for(int i=0; i<n; i++)
      isPrime.push_back(true);

    for(int i=2; i*i<n; i++) {
        if(isPrime[i]) {
            for(int j=i*i; j<n; j+=i)
              isPrime[j] = false;
        }
    }

    int countP=0;
    for(int i=2; i<n; i++){
        if(isPrime[i])
          countP++;
    }
    return countP;
}
```

# Chapter 5

# Bit-Manipulation Basics

## 5.1 Binary AND

**Check whether a number is odd or even**

```cpp

bool isOdd(int n) {
    return (n&1) ? true : false;
}
```

## 5.2 Bit Shifting

### 5.2.1 Binary Left Shift

$$a << b = a * 2^b$$

```cpp

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

int main() {
    int a=5,b=2;
    cout << (a<<b) << endl;
    cout << a*pow(2,b) << endl;
    return 0;
}
```

### 5.2.2 Binary Right Shift

$$a >> b = a/(2^b)$$

```cpp

int main() {
    int a=5,b=2;
    cout << (a>>b) << endl;
    cout << floor(a/pow(2,b)) << endl;
    return 0;
}
```

### 5.2.3 Get $i^{th}$ Bit of an Integer

```
int getIthBit(int n, int i) {
    return (n & (1<<i)) ? 1 : 0;
}
```

### 5.2.4 Clear $i^{th}$ Bit of an Integer

```
void clearIthBit(int &n, int i) {
    int mask = ~(1<<i);
    n = n & mask;
}
```

### 5.2.5 Clear last i Bits of an Integer

```
void clearLastBits(int &n, int i) {
    int mask = (-1 << i);
    n = n & mask;
}
```

### 5.2.6 Set $i^{th}$ Bit of an Integer

```
void setIthBit(int &n, int i) {
    int mask = (1<<i);
    n = (n | mask);
}
```

### 5.2.7 Update $i^{th}$ Bit with value v

```
void updateIthBit(int &n, int i, int v) {
    int mask = ~(1<<i);
    // Clear ith value first
    n = n & mask;
    // Set ith bit with value v
    mask = (v << i);
    n = n | mask;
}
```

### 5.2.8 Clear Bits in a Range

$$
\begin{aligned}
input: &\quad 11111111 \\
output: &\quad 11100011 \\
&\textit{here, we can divide the expected out into 2 components} \\
a: &\quad 111\,\overbrace{00000}^{\text{j+1}} \\
b: &\quad 000000\,\overbrace{11}^{\text{i+1}} \qquad \Rightarrow 2^i - 1 \text{ or } (1 << i) - 1 \\
\hline
performing\ bitwise\ or(|)\ a\,|\,b: &\quad 11100011
\end{aligned}
$$

**Code**

```cpp
void clearBitsInRange(int &n, int i, int j) {
    // make a = 11100000
    int a = (~0) << (j+1);
    // make b = 00000011
    int b = (1<<i)-1;
    // make mask = 11100011
    int mask = a | b;
    // now clear the bits in between range[i,j]
    n = n & mask;
}
```

### 5.2.9 Check whether a number is power of 2

**For power of 2**

$$
\begin{aligned}
2^3 &= 8 \\
(8)_2 &= 00001000 \\
(7)_2 &= 00000111 \\
\hline
\text{bitwise AND } 8\&7 &= 00000000
\end{aligned}
$$

**For non power of 2**

$$
\begin{aligned}
(10)_2 &= 00001010 \\
(9)_2 &= 00001001 \\
\hline
\text{bitwise AND } 10\&9 &= 00001000
\end{aligned}
$$

**Code Implementation:**

```cpp

bool isPowerOfTwo(int n) {
    return (n & (n-1)) ? false : true;
}

int main() {
    int n; cin >> n;
    if(isPowerOfTwo(n))
        cout << "Power of two" << endl;
    else
        cout << "Not a power of 2" << endl;

    return 0;
}
```

### 5.2.10 Count set Bits

**Code:**

```cpp

int countSetBits(int n) {
    int count = 0;
    while(n>0) {
        if(n&1)
            count++;
        n = (n>>1);
    }
    return count;
}
```

Here,

$$
\begin{aligned}
\text{if } n &= 20 \\
(20)_2 &= 00010100 \\
\text{Operation-1} &: 00001010 \qquad n >> 1 \\
\text{Operation-2} &: 00000101 \qquad n >> 2 \\
\text{Operation-3} &: 00000010 \qquad n >> 3 \\
\text{Operation-4} &: 00000001 \qquad n >> 4 \\
\text{Operation-5} &: 00000000 \qquad n >> 5
\end{aligned}
$$

**Count Set Bits Hack:**

```cpp

int countSetBits(int n) {
    int count = 0;
    while(n>0) {
        n = n & (n-1);
        count++;
    }
    return count;
}
```

Here,

$$\begin{aligned}
\text{if } n &= 20 \\
(20)_2 &= 00010100 \\
\text{Operation-1}: &\ 00010000 \qquad n\&(n-1) \\
\text{Operation-2}: &\ 00000000 \qquad n\&(n-1)
\end{aligned}$$

### 5.2.11 Check whether a number is power of 4

➤ First check whether the number is power of 2 or not

➤ Then check for the numbers that are power of 4

$$\begin{aligned}
(-16)_2 &= 1111111111110000 \\
(0x5555)_2 &= 0101010101010101 \\
\hline
AND(\&) &= 0101010101010000 > 0
\end{aligned}
\qquad\Bigg|\qquad
\begin{aligned}
(16)_2 &= 0000000000010000 \\
(0x5555)_2 &= 0101010101010101 \\
\hline
AND(\&) &= 0000000000010000 > 0
\end{aligned}$$

**Code to check if a number is power of 4**

```

bool isPowerOfFour(int n) {
    bool multipleOfTwo = (n&(n-1)); // Check for power of 2
    bool setAtEven = !(n&(0x55555555)); // Check for power of 4

    return (multipleOfTwo or setAtEven) ? false : true;
}
```

## 5.3 Bit-Manipulation Problems

### 5.3.1 Unique Number-1

Given 2n+1 numbers where every number comes twice expect one number. Find out that unique number.

$$\underbrace{2 \oplus 2}_{0} \oplus \underbrace{7}_{7} \oplus \underbrace{3 \oplus 3}_{0} \oplus \underbrace{4 \oplus 4}_{0}$$

**Unique Number-1 Code:**

```

int getUniqueNum(vector <int> arr) {
    int xOR = 0;
    for(auto i:arr) { xOR ^= i; }
    return xOR;
}
```

### 5.3.2 Finding all Subsets

Given a string and find out the all possible subset of that string

$$Sub\ sets\ of\ (abc) = \{\phi, a, b, c, ab, ac, bc, abc\}$$

| $\phi$ | a | b | ab | c | c a | cb | abc |
|---|---|---|---|---|---|---|---|
| 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |

**Code Implementation:**

```
void overLayNumber(string str, int n) {
    int j=0;
    while(n) {
        if(n&1) {
            cout << str[j];
        }
        n = (n>>1);
        j++;
    }
    cout << endl;
}

int main () {
    string str;
    cin >> str;
    for(int i=0; i<(1<<str.length()); i++) {
        overLayNumber(str,i);
    }
}
```

### 5.3.3 Binary Exponentiation

Binary exponentiation (also known as exponentiation by squaring) is a trick which allows to calculate $a^n$ using only $O(\log_2 n)$ multiplications (instead of $O(n)$ multiplications required by the naive approach).

**Algorithm :**

Raising $a$ to the power of $n$ is expressed naively as multiplication by $a$ done $n-1$ times: $a^n = a \cdot a \cdot \ldots \cdot a$. However, this approach is not practical for large $a$ or $n$.

The idea of binary exponentiation is, that we split the work using the binary representation of the exponent.

Let's write, $n$ in base 2, for example:

$$3^{13} = 3^{1101_2} = 3^8 \cdot 3^4 \cdot 3^1$$

Since the number $n$ has exactly $\lfloor \log_2 n \rfloor + 1$ digits in base 2, we only need to perform $O(\log n)$ multiplications, if we know the powers $a^1, a^2, a^4, a^8, \ldots, a^{2^{\lfloor \log n \rfloor}}$.

**Code :**

```
long long binaryExp(int a, int n) {
    long long result = 1;
    while(n) {
        if(n&1) {
            result *= a;
        }
        a *= a;
        n >>= 1;
    }
    return result;
}
```

### 5.3.4 Compute $x^n$ *mod m*

Since, we know that,

$$(a.b.c) \bmod x = \big((a \bmod x).(b \bmod x).(c \bmod x)\big) \bmod x$$

**Code :**

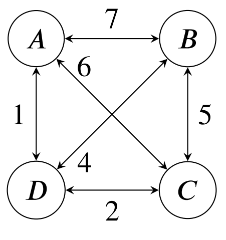
```
int expMod(int x, int n, int m) {
    int exp = 1;
    while(n) {
        if(n&1) exp = (exp*x) % m;
        x = (x*x) % m;
        n >>= 1;
    }
    return exp;
}
```

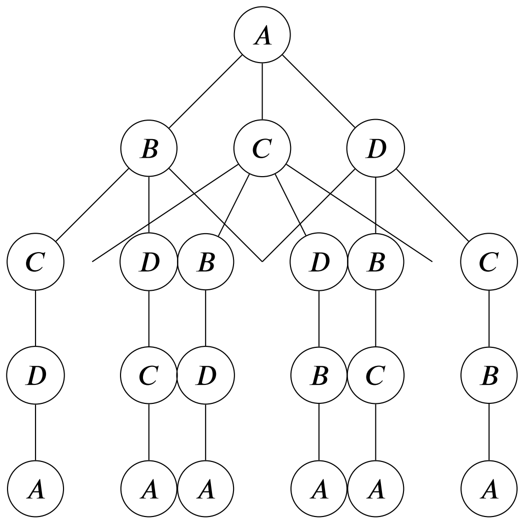
# Chapter 6

# Dynamic Programming

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

## 6.1 Travel at Minimum Coast





$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 7 & 6 & 1 \\ B & 7 & 0 & 5 & 4 \\ C & 6 & 5 & 0 & 2 \\ D & 1 & 4 & 2 & 0 \end{array}$$

### 6.1.1 Recursive Solution

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

int tsp(vector<vector<int>>dist, int setOfCities, int currentCity, int numberOfCities) {
    // Check if all the cities are visited
    if(setOfCities==(1<<numberOfCities)-1) {
        // return the cost to visit from last note to start node
        return dist[currentCity][0];
    }

    int ans = INT_MAX;

    for(int nextCity=1; nextCity<numberOfCities; nextCity++) {
        // Check if the current city is visited or not
        if((setOfCities & (1<<nextCity)) == 0) {
            int cost = dist[currentCity][nextCity];
            // Update the visited cities status
            int update = (setOfCities | (1<<nextCity));
            cost += tsp(dist, update, nextCity, numberOfCities);
            ans = min(ans,cost);
        }
    }

    return ans;
}

int main() {
    vector < vector<int>> dist;
    dist = {
        {0,7,6,1},
        {7,0,5,4},
        {6,5,0,2},
        {1,4,2,0}
    };
    int numberOfCities = 4;
    cout << tsp(dist,1,0,numberOfCities) << endl;

    return 0;
}
```

### 6.1.2 DP Based Solution

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

int tsp(vector<vector<int>>dist, int setOfCities, int crntCT, int numOfCTs, vector<vector<
    int>>dp) {
    // Check if all the cities are visited
    if(setOfCities==(1<<numOfCTs)-1)
        return dist[crntCT][0]; // cost to visit last to start

    if(dp[setOfCities][crntCT] != -1) // check if current is visited
        return dp[setOfCities][crntCT];

    int ans = INT_MAX;
    for(int nxtCT=1; nxtCT<numOfCTs; nxtCT++) {
        if((setOfCities & (1<<nxtCT)) == 0) {
            int update = (setOfCities | (1<<nxtCT));
            int cost = dist[crntCT][nxtCT] + tsp(dist, update, nxtCT, numOfCTs,dp);
            ans = min(ans,cost);
        }
    }
    dp[setOfCities][crntCT] = ans; // Memorise visited city cost
    return ans;
}

int main() {
    vector < vector<int>> dist;
    dist = {
        {0,5,7,3},
        {2,0,4,2},
        {5,2,0,3},
        {4,2,3,0}
    };
    int numOfCTs = 4;
    vector < vector<int>> dp(1<<numOfCTs, vector<int>(numOfCTs,-1));
    cout << tsp(dist,1,0,numOfCTs,dp) << endl;
    return 0;
}
```

Chapter 7

# Big Integer

In C or C++, we have different types of data types like integers, long, float, characters etc. Each data type occupies some amount of memory. There is a range of numbers that can be occupied by that data type.
For example, an integer occupies 4 bytes of memory so that we can have numbers from -2147483648 to +2147463647.
So, if we want to have an integer where the number of digits in decimal format is 22 or more, then we cannot store it using primitive data types.

## 7.1 Addition of two Big Integers

```cpp
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;

// Char to int .........
int charToInt ( char ch) {
    return ch - '0';
}

// Int to char ........
char intToChar (int digit) {
    return digit + '0';
}

// Addition method ....................
string addition(string n1, string n2) {

    // make sure n1 <= n2 ........
    if(n1.length() > n2.length()) {
        swap(n1,n2);
    }

    // Reverse the numbers .....
    reverse(n1.begin(),n1.end());
    reverse(n2.begin(),n2.end());

    int carry = 0;
    string result;

    for(int i=0; i<n1.length(); i++) {
        int d1 = charToInt(n1[i]);
        int d2 = charToInt(n2[i]);
        int sum = d1 + d2 + carry;
        int outputDigit = sum % 10;
        carry = sum/10;
        result.push_back(intToChar(
    outputDigit));
    }

    for(int i=n1.length(); i<n2.length(); i
    ++) {
        int d = charToInt(n2[i]);
        int sum = d + carry;
        int outputDigit = sum % 10;
        carry = sum /10;
        result.push_back(intToChar(
    outputDigit));
    }

    if(carry) {
        result.push_back('1');
    }

    // Reaverse the result ........
    reverse(result.begin(), result.end());

    return result;
}

int main () {
    string s1,s2;
    cin >> s1 >> s2;

    string sum = addition(s1,s2);
    cout << sum << endl;

    return 0;
}
```

## 7.2 Big Factorial

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

// ....... Char to Int ...........
int charToInt (char ch) {
    return ch - '0';
}

// ....... Int to Char ..........
char intToChar (int n) {
    return n + '0';
}

// ......... Multiplication ............
string multiplication(string n1, int n2) {

    reverse(n1.begin(), n1.end());
    string result;
    int carry = 0;

    for (char ch: n1) {
        int mul = charToInt(ch)*n2 + carry;
        result.push_back(intToChar(mul%10))
    ;
        carry = mul/10;
    }

    while(carry) {
        result.push_back(intToChar(carry
    %10));
        carry /= 10;
    }

    reverse(result.begin(), result.end());
    return result;
}

// ......... Big Factorial Method
    ..........
string bigFactorial(int n) {
    string fact = "1";
    for(int i=2; i<=n; i++) {
        fact = multiplication(fact,i);
    }
    return fact;
}
```

```
int main () {
    int n;
    cin >> n;
    cout << bigFactorial(n) << endl;
    return 0;
}
```

# Chapter 8

# Problems and Solutions

## 8.1 Maximum Sub-array Sum

Given an array of $n$ integers. Find the maximum possible sub-array sum.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -1 | 2 | 4 | -3 | 5 | 2 | -5 | 2 |

### 8.1.1 Solution with $O(n^3)$ time complexity

```
long long maxSubArraySum(int n, int arr[]) {
    long long maxSum=0;
    for(int a=0; a<n; a++) {
        for(int b=a; b<n; b++) {
            long long sum=0;
            for(int i=a; i<=b; i++) {
                sum += arr[i];
            }
            maxSum = max(maxSum,sum);
        }
    }
    return maxSum;
}
```

The code will take about 1 second to execute for a maximum value of n=500

### 8.1.2 Solution with $O(n^2)$ time complexity

```
long long maxSubArraySum(int n, int arr[]) {
    long long maxSum=0;
    for(int a=0; a<n; a++) {
        long long sum=0;
        for(int b=a; b<n; b++) {
            sum += arr[b];
            maxSum = max(maxSum,sum);
        }
    }
    return maxSum;
}
```

For this code the execution time is approximately 1 second for n=5000

### 8.1.3 Solution with $O(n)$ time complexity

```
long long maxSubArraySum(int n, long long arr[]) {
    long long maxSum=0, sum=0;
    for(int a=0; a<n; a++) {
        sum = max(arr[a], sum+arr[a]);
        maxSum = max(maxSum, sum);
    }
    return maxSum;
}
```

The time complexity of the above code is linear. It will execute about within 1 second for $n \leq 10^6$.