

# Chapter 1:

## Intro and C++ Syntax:

- The benefits of using C++ are that it is a very efficient language and its standard library contains a large collection of data structures and algorithms.
- `#include <bits/stdc++.h>`: This line at the beginning of the code is a feature of the g++ compiler that allows us to include the entire standard library. Thus, it is not needed to separately include libraries such as `iostream`, `vector` and `algorithm`, but rather they are available automatically.
- `using namespace std;`: The using line declares that the classes and functions of the standard library can be used directly in the code. Without the using line we would have to write, for example, `std::cout`, but now it suffices to write `cout`.
- The usual floating point types in competitive programming are the 64-bit double and, as an extension in the g++ compiler, the 80-bit long double. In most cases, double is enough, but long double is more accurate
- `typedef long long ll;`
  - Shortens long long to ll
  - `typedef vector vi;` `typedef pair pi`
- Another way to shorten code is to define macros. A macro means that certain strings in the code will be changed before the compilation. In C++, macros are defined using the `#define` keywords.
  - Example:

For example, we can define the following macros:

```
#define F first
#define S second
#define PB push_back
#define MP make_pair
```

After this, the code

```
v.push_back(make_pair(y1,x1));
v.push_back(make_pair(y2,x2));
int d = v[i].first+v[i].second;
```

can be shortened as follows:

```
v.PB(MP(y1,x1));
v.PB(MP(y2,x2));
int d = v[i].F+v[i].S;
```

## Mathematics:

Math:

Sum: formulas:

- Formulas exist to quickly compute the sum of sequences where each term follows a specific pattern.

- The sum of natural numbers raised to the power  $k$ :

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

- Ex: to add numbers from range (1, 11):  $\frac{10(10+1)}{2} = 55$

$$\sum_{x=1}^n x^2 = 1^2 + 2^2 + 3^2 + 4^2 \dots n^2 = \frac{n(n+1)(2n+1)}{6}$$

- Ex: to add squares of nums from 1<sup>2</sup> to 10<sup>2</sup> =  $\frac{10(10+1)(2(10)+1)}{6} = 385$

- An arithmetic progression is a sequence of numbers where the diff between consecutive terms is constant.

- Formula:  $\frac{a + \dots + b}{n \text{ terms}} = \frac{n(a+b)}{2}$

$$\text{- Ex: } 3 + 7 + 11 + 15 = \frac{4 \cdot (3 + 15)}{2} = 36$$

- Sum consists of  $n$  nums, value is

$(a+b)/2$  on average.

Prepared by

Index no.

Date

- A geometric progression is a sequence of num's where the ratio between any 2 numbers is constant

- Formula =  $\frac{b \cdot k - a}{k - 1}$ , where  $a$  = first term,  $b$  = last term,  $k$  = ratio

- Assume  $S = a + ak + ak^2 + \dots + b$ . If we multiply each term by  $k$ , we get another sequence which when subtracted, gives the formula.

- Ex: The sum of a GP where each term doubles =  $2^n - 1$

$$1 + 2 + 4 + \dots + 2^{(n-1)} = 2^n - 1$$

- A harmonic progression is a sequence formed by taking the reciprocals of an arithmetic progression.

- Form:  $\sum_{x=1}^n \frac{1}{x} = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$

- Upper bound: An upper bound for HS is  $\log_2(n) + 1$ . We can modify each term  $1/k$  so that  $k$  becomes the nearest power of 2 that does not exceed  $k$ . For ex,  $n=6$ :

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} \leq 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}$$

### Set Theory:

- A Set is a collection of elements. For ex:  $X = \{2, 4, 7\}$  contains elements 2, 4, 7. The symbol  $\emptyset$  denotes empty set.

$|S|$  denotes the size of set, i.e. num of elements in set. For ex,  $\{2, 4, 7\} = |X| = 3$ .

- If a set contains an element  $x$ , we write  $x \in S$ , otherwise we write  $x \notin S$ .

- For ex in above set:  $4 \in X$ ,  $5 \notin X$ .

Prepared by

Index no.

Date



- new sets can be constructed using set operations:
  - Intersection  $A \cap B$  consists of elements in  $A$  &  $B$ . Ex: if  $A = \{3, 7\}$  and  $B = \{2, 3, 8\}$ , then  $A \cap B = \{3\}$ .
  - The union  $A \cup B$  consists of elements in  $A$  or  $B$  or both. ex: if  $A = \{3, 7\}$  and  $B = \{2, 3, 8\}$  then  $A \cup B = \{2, 3, 7, 8\}$
  - The complement  $\bar{A}$  consists of elements not in  $A$ . Interpretation of complement depends on universal set which contains all possible elements. Ex: if  $A = \{1, 2, 5, 7\}$  and universal set is  $\{1, 2, \dots, 10\}$ , then  $\bar{A} = \{3, 4, 6, 8, 9, 10\}$
  - The difference  $A \setminus B = A \cap \bar{B}$  consists of elements that are in  $A$ , but not in  $B$ . Ex: if  $A = \{2, 3, 7, 8\}$  and  $B = \{3, 5, 8\}$  then  $A \setminus B = \{2, 7\}$
- If each element of  $A$  also belongs to  $S$ , we say that  $A$  is a subset of  $S$ , denoted by  $A \subseteq S$ . A set  $S$  always has  $2^{|S|}$  subsets, including empty sets. Ex: The subsets of  $\{2, 4, 7\}$  are:  $\emptyset, \{2\}, \{4\}, \{7\}, \{2, 4\}, \{2, 7\}, \{4, 7\}$  and  $\{2, 4, 7\}$
- Some often used sets are  $N$  (natural num),  $Z$  (integers),  $Q$  (rational num) and  $R$  (real num). The set  $N$  can be defined two ways, depending on the situation. Either  $N = \{0, 1, 2, \dots\}$  or  $N = \{1, 2, 3, \dots\}$ . We can also construct a set using a rule or form:  $\{f(n) : n \in S\}$ , where  $f(n)$  is a function. This set contains all elements of the form  $f(n)$  where  $n$  is an element in  $S$ . For ex: the set  $X = \{2n : n \in Z\}$  contains all even integers.

Prepared by

Index no.

Date

## - Logic

- The value of a logical expression is either True (1) or False (0).

The most important logical operators are:

$\neg$  (negation):  $\neg A$  is opposite value of A

$\wedge$  (conjunction):  $A \wedge B$  = True if both A and B are true

$\vee$  (disjunction):  $A \vee B$  = True if both A or B or both are true

$\Rightarrow$  (implication):  $A \Rightarrow B$  = True if whenever A is true, B is also true

$\Leftrightarrow$  (Equivalence):  $A \Leftrightarrow B$  = True if both A and B are both true or both false.

- Predicate: Expression that is true or false based on parameters.

Ex:  $p(x)$  is true when x is a prime number.

- Quantifiers:

$\forall$  (for all),  $\exists$  (there is)

Ex:  $\forall x (\exists y (y < x))$  = For each element x, there's a y such that y is smaller than x.

Ex:  $\forall x (x > 1 \wedge \neg p(x)) \Rightarrow (\exists a (\exists b (a > 1 \wedge b > 1 \wedge x = ab)))$

means that if a number  $x > 1$  and is not a prime number,

there are numbers a and b that are larger than 1 and whose product is x.

- Functions

- The function  $\lfloor x \rfloor$  rounds the number x down to an integer, and

the function  $\lceil x \rceil$  rounds the number x up to an integer. For ex:

$\lfloor 3/2 \rfloor = 1$  and  $\lceil 3/2 \rceil = 2$ .

- The functions  $\min(x_1, x_2, \dots, x_n)$  and

Prepared by

Index no.

Date

$\min(x_1, x_2, \dots, x_n)$  give the smallest and largest of values  $x_1, x_2, \dots, x_n$ . For ex:

$$\min(1, 2, 3) = 1 \text{ and } \max(1, 2, 3) = 3$$

- The factorial  $n!$  can be defined by:

$$\prod_{x=1}^n x = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n \text{ or recursively: } n! = n \cdot (n-1)! \quad 0! = 1$$

- The Fibonacci numbers arise in many situations. Can be defined

$$\text{recursively: } \begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

- The first Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

- Also a closed form formula for Fibonacci called Binet's formula:

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

- Logarithms

- $\log_k(x)$ : denotes a logarithm of a number  $x$  with base  $k$ .

$$\log_k(x) = a \text{ exactly when } k^a = x.$$

- $\log_k(x)$  equals the number of times we have to divide  $x$  by  $k$  before reaching 1. For ex:  $\log_2(32) = 5$ , because 5 divisions by 2 is needed:  $32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$ .

- Logarithm formulas:

$$\text{Product: } \log_k(ab) = \log_k(a) + \log_k(b).$$

$$\text{Power: } \log_k(x^n) = n \log_k(x)$$

$$\text{Quotient: } \log_k\left(\frac{a}{b}\right) = \log_k(a) - \log_k(b)$$

$$\text{Change of base: } \log_u(x) = \frac{\log_k(x)}{\log_k(u)}$$

- It's possible to calculate logarithms to any base if a fixed bases logs can be

Prepared by

Index no.

Date

calculated.

- $\ln(x)$  is a natural logarithm of a number  $x$  whose base is  $e \approx 2.71828$ . Another property of logarithms is that the number of digits of an integer  $x$  in base  $b$  is  $\lfloor \log_b(x) \rfloor + 1$ . For ex, the representation of 123 in base 2 is 1111011 and  $\lfloor \log_2(123) \rfloor + 1 = 7$ .



## Time Complexity:

- The time complexity of an algorithm estimates how much time the algorithm will use for some input.
- Denoted by  $O(\dots)$  where the three dots represent some function. Usually, the variable  $n$  denotes the input size. For example, if the input is an array of numbers,  $n$  will be the size of the array, and if the input is a string,  $n$  will be the length of the string.
- `for(int i = 0; i < n; ++i)` : time complexity is  $O(n)$ .
  - If there was another loop under this one, then it would be  $O(n^2)$ .
  - The following code's time complexity is  $O(nm)$ : `for (int i = 1; i <= n; i++){for (int j = 1; j <= m; j++){`
- The time complexity of a recursive function depends on the # of times a function is called and the time complexity of a single call. The time complexity is the product of these vals.
  - $O(1)$  The running time of a constant-time algorithm does not depend on the input size. A typical constant-time algorithm is a direct formula that calculates the answer.
  - $O(\log n)$  A logarithmic algorithm often halves the input size at each step. The running time of such an algorithm is logarithmic, because  $\log_2 n$  equals the number of times  $n$  must be divided by 2 to get 1.
  - $O(n)$  A linear algorithm goes through the input a constant number of times. This is often the best possible time complexity, because it is usually necessary to access each input element at least once before reporting the answer.
  - $O(n \log n)$  This time complexity often indicates that the algorithm sorts the input, because the time complexity of efficient sorting algorithms is  $O(n \log n)$ . Another possibility is that the algorithm uses a data structure where each operation takes  $O(\log n)$  time.
  - $O(n^2)$  A quadratic algorithm often contains two nested loops. It is possible to go through all pairs of the input elements in  $O(n^2)$  time.  $O(n^3)$  A cubic algorithm often contains three nested loops. It is possible to go through all triplets of the input elements in  $O(n^3)$  time.
  - $O(2^n)$  This time complexity often indicates that the algorithm iterates through all subsets of the input elements. For example, the subsets of  $\{1,2,3\}$  are  $\{1\}$ ,  $\{2\}$ ,  $\{3\}$ ,  $\{1,2\}$ ,  $\{1,3\}$ ,  $\{2,3\}$  and  $\{1,2,3\}$ .
  - $O(n!)$  This time complexity often indicates that the algorithm iterates through all permutations of the input elements. For example, the permutations of  $\{1,2,3\}$  are  $(1,2,3)$ ,  $(1,3,2)$ ,  $(2,1,3)$ ,  $(2,3,1)$ ,  $(3,1,2)$  and  $(3,2,1)$ .
- An algorithm is polynomial if its time complexity is at most  $O(n^k)$  where  $k$  is a constant.
- Can infer times based on input sizes; For example, if the input size is  $n = 105$ , it is probably expected that the time complexity of the algorithm is  $O(n)$  or  $O(n \log n)$ .

input size	required time complexity
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ or $O(n)$
$n$ is large	$O(1)$ or $O(\log n)$

Ex: Max Subarray:

We can optimize algorithms significantly. For example, with calculating a max subarray, we can go from this, which is an  $O(n^3)$  solution:

```
int main(){
    int best = 0;
    for(int a = 0; a < n; ++a){
        for (int b = a; b < n; ++b){
            int sum = 0;
            for(int k = a; k <= b; ++k){
                sum += array[k]
            }
            best = max(best, sum)
        }
    }
    cout << best << "\n"
}
```

To this, which is an  $O(n)$  solution (Also called Kadane's algorithm):

```
int main(){
    int best = 0;
    int sum = 0;
    for(int k = 0; k < n; ++k){
        sum = max(array[k], sum+array[k]);
        best = max(best, sum);
    }
    cout << best << endl;

}
```

## Sorting:

- An array is completely sorted when there are no inversions
- Bubble sort is an  $O(n^2)$  algorithm. On each round, the algorithm iterates through the array and whenever two consecutive elements are found that are not in correct order, the algorithm swaps them.
  - always swaps consecutive elements in the array.
  -

```
for(int i = 0; i < n; ++i){
    for(int j = 0; j < n - 1; ++j){
        if(array[j] > array[j+1]){
            swap(array[j], array[j+1])
        }
    }
}
```



- Merge sort is an  $O(n \log n)$  algorithm.
  - If  $a = b$ , do not do anything, because the subarray is already sorted.
  - Calculate the position of the middle element:  $k = (a + b) / 2$ .
  - Recursively sort the subarray `array[a...k]`.
  - Recursively sort the subarray `array[k + 1...b]`.
  - Merge the sorted subarrays `array[a...k]` and `array[k + 1...b]` into a sorted subarray `array[a...b]`.
- Merge sort halves the size of the subarray at each step. The recursion consists of  $O(\log n)$  levels, and processing each level takes  $O(n)$  time. Merging the subarrays `array[a...k]` and `array[k + 1...b]` is possible in linear time, because they are already sorted.
- Binary Search is another  $O(n \log n)$  algorithm. At each step, the search checks the middle element of the active region. If the middle element is the target element, the search terminates. Otherwise, the search recursively continues to the left or right half of the region, depending on the value of the middle element:
 

```
while(l <= r){
    int mid = (l + r)/2;
    if(nums[mid] == target){
        return mid;
    } else if(nums[mid] > target){
        r = mid - 1;
    } else if(nums[mid] < target){
        l = mid + 1;
    }
    return -1;
}
```
- The C++ standard library contains the following functions that are based on binary search and work in logarithmic time:
  - `lower_bound` returns a pointer to the first array element whose value is at least  $x$ .
  - `upper_bound` returns a pointer to the first array element whose value is larger than  $x$ .
  - `equal_range` returns both above pointers.
-

## Data Structures:

- A data structure is a way to store data in the memory of a computer
- A dynamic array is an array whose size can be changed during the execution of the program. The most popular dynamic array in C++ is the vector structure, which can be used almost like an ordinary array. Syntax:
  - `vector<int> nums; nums.push_back(1); nums.push_back(2); cout << nums[0];`
  - `for(int i = 0; i < nums.size(); ++i){cout << nums[i] << endl;} //print out all elements.`  
Can also iterate like this: `for(int ch: nums){cout << ch << endl;}`
  - `cout << nums.back()` returns the last element in the vector
  - The internal implementation of a vector uses an ordinary array. If the size of the vector increases and the array becomes too small, a new array is allocated and all the elements are moved to the new array.
  - Another way to create a vector is to give the number of elements and the initial value for each element: `vector<int> v(10);`
- The string structure is also a dynamic array that can be used almost like a vector. Strings can be combined using the + symbol.
  - `string a = hatti; string b = a + a; cout << b << endl; //hattihatti; b[5] = v; cout << b << endl; //hattivatti; string c = b.substr(3,4); cout << c << endl; //tiva`
- A set is a data structure that maintains a collection of elements. The basic operations of sets are element insertion, search and removal. The C++ STL contains two set implementations: The structure set is based on a balanced binary tree and its operations work in  $O(\log n)$  time. The structure unordered\_set uses hashing, and its operations work in  $O(1)$  time on average.
  - Only contains distinct elements
  - The function count always returns either 0 (the element is not in the set) or 1 (the element is in the set), and the function insert never adds an element to the set if it is already there.
  - `std::unordered_set<int> s; s.insert(2); s.insert(3); cout << s.count(3) // 1; cout << s.count(4) //0; s.erase(3);`
  - A set can be used mostly like a vector, but it is not possible to access the elements using the [] notation. Ex: `set s = {2,5,6,8}; cout << s.size() << "\n"; // 4;`  
`for(auto x: s){cout << x << endl;}`
- A map is a generalized array that consists of key-value-pairs. The keys in a map can be of any data type and they do not have to be consecutive values.
  - The structure unordered\_map uses hashing and accessing elements takes  $O(1)$  time on average.
  - `std::unordered_map<string, int> m; m["monkey"] = 4;m["banana"] = 3; cout << m["banana"] << endl; // 3`
  - If the value of a key is requested but the map does not contain it, the key is automatically added to the map with a default value.
  - This function checks if a key exists in a map: `if (m.count("aybabtu")) { // key exists }`
- An iterator is a variable that points to an element in a data structure.

- The iterator `begin` points to the first element in the data structure, and the iterator `end` points to the position after the last element. `s.end()` points outside the data structure. Thus, the range defined by the iterators is half-open.
- Iterators are used in C++ standard library functions that are given a range of elements in a data structure. Usually, we want to process all elements in a data structure, so the iterators `begin` and `end` are given for the function.
  - For example, the following code sorts a vector using the function `sort`, then reverses the order of the elements using the function `reverse`, and finally shuffles the order of the elements using the function `random_shuffle`:
    - `sort(v.begin(), v.end()); reverse(v.begin(), v.end());`  
`random_shuffle(v.begin(), v.end());`
    - These functions can also be used with an ordinary array. In this case, the functions are given pointers to the array instead of iterators: `sort(a, a+n);`  
`reverse(a, a+n); random_shuffle(a, a+n);`
- Iterators are often used to access elements of a set. The following code creates an iterator `it` that points to the smallest element in a set: `set<int>::iterator it = s.begin();` can write shorter by: `auto it = s.begin();` The element to which an iterator points can be accessed using the `*` symbol. For example, the following code prints the first element in the set: `auto it = s.begin(); cout << *it << endl;`
- Iterators can be moved using the operators `++` (forward) and `--` (backward), meaning that the iterator moves to the next or previous element in the set.
  - prints all the elements in increasing order: `for (auto it = s.begin(); it != s.end(); it++) {cout << *it << "\n";}`.
  - Prints the largest elements in a set: `auto it = s.end(); it--; cout << *it << "\n";`
  - The function `find(x)` returns an iterator that points to an element whose value is `x`. if the set does not contain `x`, the iterator will be `end`: `auto it = s.find(x); if(it == s.end()) { // x is not found }`
- The function `lower_bound(x)` returns an iterator to the smallest element in the set whose value is at least `x`, and the function `upper_bound(x)` returns an iterator to the smallest element in the set whose value is larger than `x`. In both functions, if such an element does not exist, the return value is `end`. For example, the following code finds the element nearest to `x`:
- A `bitset` is an array whose each value is either 0 or 1. `bitset<10> s; s[3] = 1; cout << s[3] // 1; cout << s[4] // 0;`
  - The benefit of using `bitsets` is that they require less memory than ordinary arrays, because each element in a `bitset` only uses one bit of memory.
    - For example, if `n` bits are stored in an `int` array, `32n` bits of memory will be used, but a corresponding `bitset` only requires `n` bits of memory.
  - the values of a `bitset` can be efficiently manipulated using bit operators.
    - `bitset<10> s(string("0010011010")); // from right to left; cout << s[4] << "\n"; // 1; cout << s[5] << "\n"; // 0`
    - The function `count` returns the number of ones in the `bitset`: `bitset<10> s(string("0010011010")); cout << s.count() << "\n"; // 4`

- A deque is a dynamic array whose size can be efficiently changed at both ends of the array. Like a vector, a deque provides the functions `push_back` and `pop_back`, but it also includes the functions `push_front` and `pop_front` which are not available in a vector.
  - a deque is slower than a vector but adding and removing elements take  $O(1)$  time on avg at both ends.
- A stack is a data structure that provides two  $O(1)$  time operations: adding an element to the top, and removing an element from the top. It is only possible to access the top element of a stack. Follows LIFO principle.
- A queue also provides two  $O(1)$  time operations: adding an element to the end of the queue, and removing the first element in the queue. It is only possible to access the first and last element of a queue. Follows a FIFO principle.
  - Enqueue means add deque means remove.
- A priority queue maintains a set of elements. The supported operations are insertion and, depending on the type of the queue, retrieval and removal of either the minimum or maximum element. Insertion and removal take  $O(\log n)$  time, and retrieval takes  $O(1)$  time.
  - While an ordered set efficiently supports all the operations of a priority queue, the benefit of using a priority queue is that it has smaller constant factors. A priority queue is usually implemented using a heap structure that is much simpler than a balanced binary tree used in an ordered set.
  - By default, the elements in a C++ priority queue are sorted in decreasing order, and it is possible to find and remove the largest element in the queue.
  - If we want to create a priority queue that supports finding and removing the smallest element, we can do it as follows: `priority_queue<int>`, `vector<int>`, `greater<int>` > q;

```
deque<int> d;
d.push_back(5); // [5]
d.push_back(2); // [5,2]
d.push_front(3); // [3,5,2]
d.pop_back(); // [3,5]
d.pop_front(); // [5]
```

```
stack<int> s;
s.push(3);
s.push(2);
s.push(5);
cout << s.top(); // 5
s.pop();
cout << s.top(); // 2
```

```
queue<int> q;
q.push(3);
q.push(2);
q.push(5);
cout << q.front(); // 3
q.pop();
cout << q.front(); // 2
```

```
priority_queue<int> q;
q.push(3);
q.push(5);
q.push(7);
q.push(2);
cout << q.top() << "\n"; // 7
q.pop();
cout << q.top() << "\n"; // 5
q.pop();
q.push(6);
cout << q.top() << "\n"; // 6
q.pop();
```

## Complete Search:

- Complete search is a general method that can be used to solve almost any algorithm problem. The idea is to generate all possible solutions to the problem using brute force, and then select the best solution or count the number of solutions, depending on the problem.
- subsets of  $\{0,1,2\}$  are  $\{0\}$ ,  $\{1\}$ ,  $\{2\}$ ,  $\{0,1\}$ ,  $\{0,2\}$ ,  $\{1,2\}$  and  $\{0,1,2\}$ . There are two common methods to generate subsets: we can either perform a recursive search or exploit the bit representation of integers.



- An elegant way to go through all subsets of a set is to use recursion. The following function search generates the subsets of the set  $\{0,1,\dots,n-1\}$ . The function maintains a vector subset that will contain the elements of each subset.

```
void search(){
    if(k == n){ //if we've considered all elements in the set...
        //..process subset (print out or store);
    } else{
        search(k+1); //first explore the subsets without the current element k
        subset.push_back(k); //add the current k element to subset
        search(k+1); //explore the subsets with the current element k
        subset.pop_back(); //remove k from the current subset before backtracking and exploring future
                           //subsets so we do not mistakenly include k again
    }
}
```

- The following code shows how we can find the elements of a subset that corresponds to a bit sequence. When processing each subset, the code builds a vector that contains the elements in the subset
  - `for(int b = 0; b < (1<<n), ++b){vector<int> subset; for(int i = 0; i < n; ++i){if (b&(1<<i)) subset.push_back(i);}}`
- Generating Permutations:
  - 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). There are two approaches: we can either use recursion or go through the permutations iteratively. Here is the recursive approach:

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

void search() {
    //check if the size of the current permutation is equal to size of set
    if(permutation.size() == n){
        //permutation is complete, process it (print or store it)
    } else{
        //if permutation is not complete, try to add more elements to it
        for(int i = 0; i < n; ++i){
            if(chosen[i]) continue; //if current element has already been processed, continue
            chosen[i] = true; //mark the element as used
            permutation.push_back(i); //add element to the current permutation
            search(); //recursive, start the process all over again till every element is processed
            chosen[i] = false; //backtrack, after all elements have been recursively processed, the last
                               //element is removed, and chosen[i] is reset to false. lets the for loop try next element
                               //in the set.
            permutation.pop_back(i);
        }
    }
}
```

- Another method for generating permutations is to begin with the permutation  $\{0,1,\dots,n-1\}$  and repeatedly use a function that constructs the next permutation in increasing order. The C++ standard library contains the function `next_permutation` that can be used for this:
  - `vector<int> permutation;`
  - `for (int i = 0; i < n; i++) {`

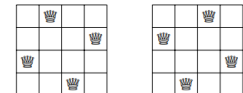
```

        permutation.push_back(i);
    }
    do {
        // process permutation
    } while (next_permutation(permutation.begin(), permutation.end()));

```

- A backtracking algorithm begins with an empty solution and extends the solution step by step. The search recursively goes through all different ways how a solution can be constructed. For example: consider the problem of calculating the number of ways  $n$  queens can be placed on an  $n \times n$  chessboard so that no two queens attack each other.

other: For example, when  $n = 4$ , there are two possible solutions:



- Ex:
- The problem can be solved using backtracking by placing queens to the board row by row. More precisely, exactly one queen will be placed on each row so that no queen attacks any of the queens placed before.

```

void search(){
    if(y == n){ //base case; check if y == n, meaning a queen has been placed in all rows
        count++; //tracking number of solutions found
        return; //exit function as full solution has been found
    }
    for(int x = 0; x < n; ++x){ //
        //check if no queen in the same column, no queen in the same diagonal(top left to bottom right
        //), check if no queen in the other diagonal (top right to bottom left). If true, continue
        if(column[x] || diag1[x+y] || diag2[x-y+n-1]) continue;
        //mark the column and both diagonals as occupied by setting array elements to 1
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 1;
        //move to next row to place next queen
        search(y+1);
        //remove queen (backtrack) and try next position
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 0;
    }
}

```

- We can often optimize backtracking by pruning the search tree. The idea is to add "intelligence" to the algorithm so that it will notice as soon as possible if a partial solution cannot be extended to a complete solution. Such optimizations can have a tremendous effect on the efficiency of the search.
- Meet in the middle is a technique where the search space is divided into two parts of about equal size. A separate search is performed for both of the parts, and finally the results of the searches are combined.
  - The technique can be used if there is an efficient way to combine the results of the searches.
  - Ex: a problem where we are given a list of  $n$  numbers and a number  $x$ , and we want to find out if it is possible to choose some numbers from the list so that their sum is  $x$ . For example, given the list  $[2, 4, 5, 9]$  and  $x = 15$ , we can choose the

numbers [2,4,9] to get  $2+4+9 = 15$ . However, if  $x = 10$  for the same list, it is not possible to form the sum.

- A simple algorithm to the problem is to go through all subsets of the elements and check if the sum of any of the subsets is  $x$ . The running time of such an algorithm is  $O(2^n)$ , because there are  $2^n$  subsets.
- divide the list into two lists A and B such that both lists contain about half of the numbers. The first search generates all subsets of A and stores their sums to a list S of A. Correspondingly, the second search creates a list S of B from B. After this, it suffices to check if it is possible to choose one element from S of A and another element from S of B such that their sum is  $x$ . This is possible exactly when there is a way to form the sum  $x$  using the numbers of the original list.
  - For example, suppose that the list is [2,4,5,9] and  $x = 15$ . First, we divide the list into  $A = [2,4]$  and  $B = [5,9]$
  - After this, we create lists S of A = [0,2,4,6] and S of B = [0,5,9,14]. In this case, the sum  $x = 15$  is possible to form, because SA contains the sum 6, SB contains the sum 9, and  $6+9 = 15$ . This corresponds to the solution [2,4,9].
  - We can implement the algorithm so that its time complexity is  $O(2^{n/2})$ .

### Greedy Algorithms:

- A greedy algorithm constructs a solution to the problem by always making a choice that looks the best at the moment. A greedy algorithm never takes back its choices, but directly constructs the final solution.
- Consider a problem where we are given a set of coins and our task is to form a sum of money  $n$  using the coins. The values of the coins are  $\text{coins} = \{c_1, c_2, \dots, c_k\}$ , and each coin can be used as many times as we want. What is the minimum number of coins needed?
  - For example, if the coins are the euro coins (in cents)  $\{1, 2, 5, 10, 20, 50, 100, 200\}$  and  $n = 520$ , we need at least four coins. The optimal solution is to select coins  $200+200+100+20$  whose sum is 520.
  - A simple greedy algorithm to the problem always selects the largest possible coin, until the required sum of money has been constructed.
  - we can show for each coin  $x$  that it is not possible to optimally construct a sum  $x$  or any larger sum by only using coins that are smaller than  $x$ . For example, if  $x = 100$ , the largest optimal sum using the smaller coins is  $50+20+20+5+2+2 = 99$ . Thus, the greedy algorithm that always selects the largest coin produces the optimal solution
  - We can prove that a greedy algorithm does not work by showing where the algorithm gives a wrong answer. In this problem we can find a counterexample: if the coins are  $\{1, 3, 4\}$  and the target sum is 6, the greedy algorithm produces the solution  $4+1+1$  while the optimal solution is  $3+3$ .
- Many scheduling problems can be solved using greedy algorithms. A classic problem is as follows: Given  $n$  events with their starting and ending times, find a schedule that includes as many events as possible.

- We next consider a problem where we are given  $n$  numbers  $a_1, a_2, \dots, a_n$  and our task is to find a value  $x$  that minimizes the sum  $|a_1 - x|^c + |a_2 - x|^c + \dots + |a_n - x|^c$ .

## Dynamic Programming:

- Dynamic programming is a technique that combines the correctness of complete search and the efficiency of greedy algorithms. Dynamic programming can be applied if the problem can be divided into overlapping subproblems that can be solved independently. There are two uses for dynamic programming:
  - Finding an optimal solution: We want to find a solution that is as large as possible or as small as possible.
  - Counting the number of solutions: We want to calculate the total number of possible solutions.
- Uses Memoization:
  - Memoization ensures that a method doesn't run for the same inputs more than once by keeping a record of the results for the given inputs (usually in a hash map)
- Given a set of coin values  $\text{coins} = \{c_1, c_2, \dots, c_k\}$  and a target sum of money  $n$ , our task is to form the sum  $n$  using as few coins as possible
  -