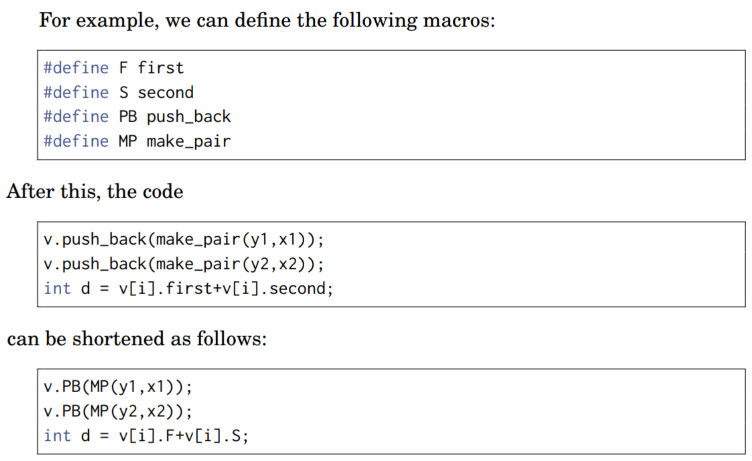
Chapter 1:

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
```

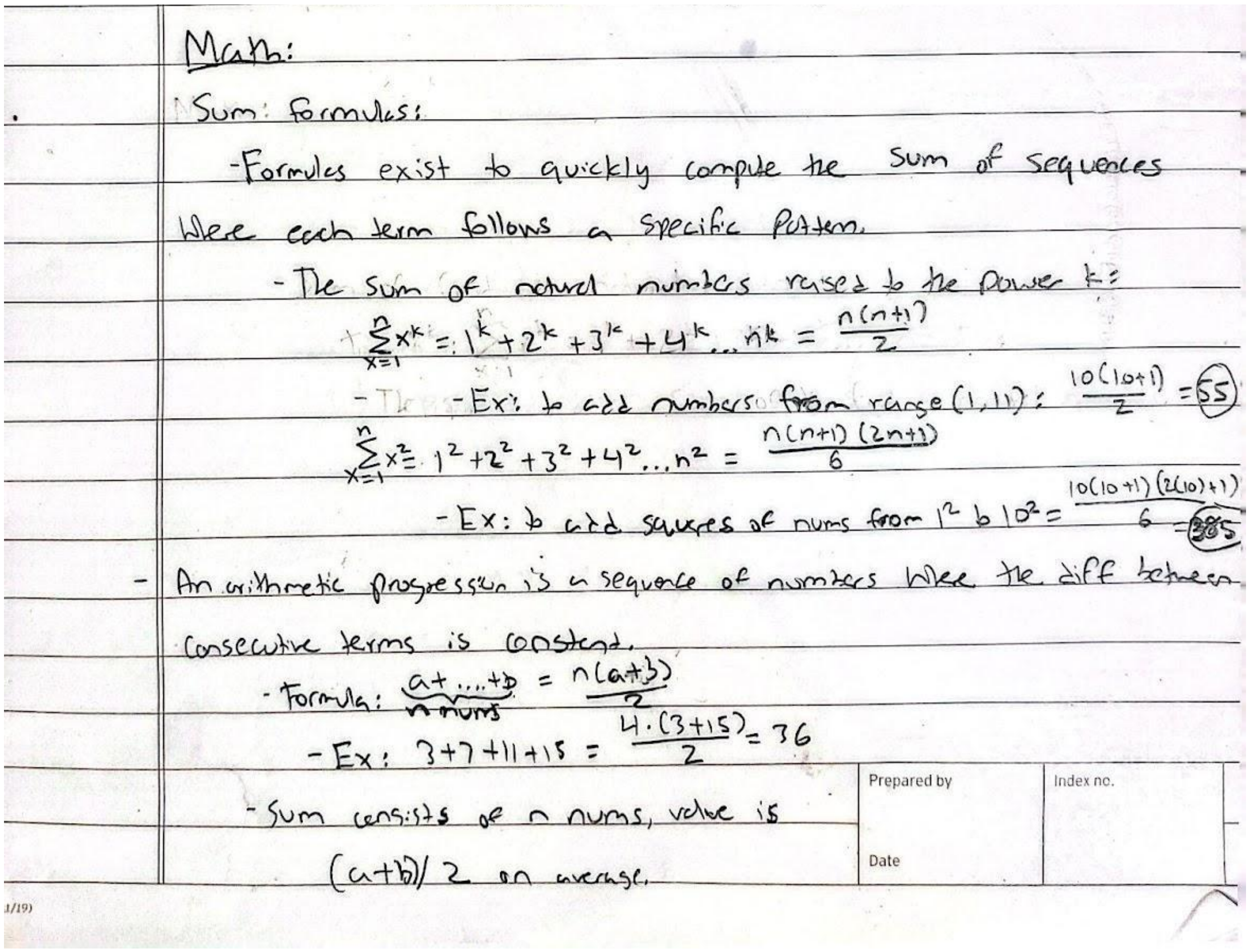
After this, the code

```
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 Were 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): $\frac{10(10+1)}{2} = \boxed{55}$

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

- Ex: to add squares of nums from 1^2 to 10^2 = $\frac{10(10+1)(2(10)+1)}{6} = \boxed{385}$

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

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

- 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	

1/19)

EY
Building a better working world

- A geometric Progression is a sequence of nums where the ratio between any 2 numbers is constant
  - Formula $= \frac{bk-a}{k-1}$, where a = first term, b = last term, k = ratio
  - Assume $S = a + ak + ak^2 + \ldots + b$. 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 $= 2^n - 1$
    $1 + 2 + 4 + \ldots + 2^{(n-1)} = 2^n - 1$
- A harmonic progression is a sequence formed by taking the reciprocals of an arithmetic progression.
  - Form: $\sum_{x=1}^{n} \frac{1}{x} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots \frac{1}{n}$
  - 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, $n=6$:
    $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} \le 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}$
- 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	

EY
Building a better
working world

- 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 $A \cap B = \{2\}$
  - The union $A \cup B$ consists of elements in A or B or Both. ex: if $A = \{3,7\}$ and $B = \{2,3,8\} = A \cup B = \{2,3,7,8\}$
  - 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 $\{1,2,\ldots 10\}$, 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 $A \subset S$. 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 nums), Z (integers), Q (rational nums) and R (real nums). The set N can be defined two ways, depending on the situation. Either $N = \{0,1,2,\ldots\}$ or $N = \{1,2,3,\ldots\}$. 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	

(11/19)

- Logic

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

  The most important logical operators are:

  $\neg$ (negation): $\neg A$ = 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 != to 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 $\min(x_1, x_2, \ldots, x_n)$ and

Prepared by	Index no.
Date	

EY
Building a better working world

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

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

- The factorial n! can be defined by:

$$\prod_{x=1}^{n} x = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \quad \text{or recursively: } \begin{array}{l} 0! = 1 \\ n! = n \cdot (n-1)! \end{array}$$

- The Fibonacci numbers arise in many situations. Can be defined recursively: $\begin{array}{l} f(0)=0 \\ f(1)=1 \\ f(n)=f(n-1)+f(n-2) \end{array}$
  - The first fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...
- Also a closed form formula for Fibonacci's 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.
  - $\log_k(x) = a$ 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: $32 \to 16 \to 8 \to 4 \to 2 \to 1$.
- Logarithm formulas:
- Product: $\log_k(ab) = \log_k(a) + \log_k(b)$.
- Power: $\log_k(x^n) = n \times \log_k(x)$
- Quotient: $\log_k(\frac{a}{b}) = \log_k(a) - \log_k(b)$
- Change of base: $\log_u(x) = \frac{\log_k(x)}{\log_k(u)}$

Prepared by	Index no.
Date	

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

EY
Building a better working world

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~~ the number of digits of an integer x in base b is $\lfloor \log_b(x) + 1 \rfloor$. For ex, the representation of 123 in base 2 is 1111011 and $\lfloor \log_2(123) + 1 \rfloor = 7$.

Time Complexity:

- The time complexity of an algorithm estimates how much time the algorithm will use for some input.
- Denoted by O(⋯) 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++){/code}}
- 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(logn) A logarithmic algorithm often halves the input size at each step. The running time of such an algorithm is logarithmic, because log2 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(nlogn) This time complexity often indicates that the algorithm sorts the input, because the time complexity of efficient sorting algorithms is O(nlogn). Another possibility is that the algorithm uses a data structure where each operation takes O(logn) 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(nlogn).

input size	required time complexity
$n \le 10$	$O(n!)$
$n \le 20$	$O(2^n)$
$n \le 500$	$O(n^3)$
$n \le 5000$	$O(n^2)$
$n \le 10^6$	$O(n \log n)$ 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 = b(a+ b)/2c.
    - 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(logn) 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) algorithms. 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(logn) 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.

```
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]
```

- 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.

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

- 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.

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

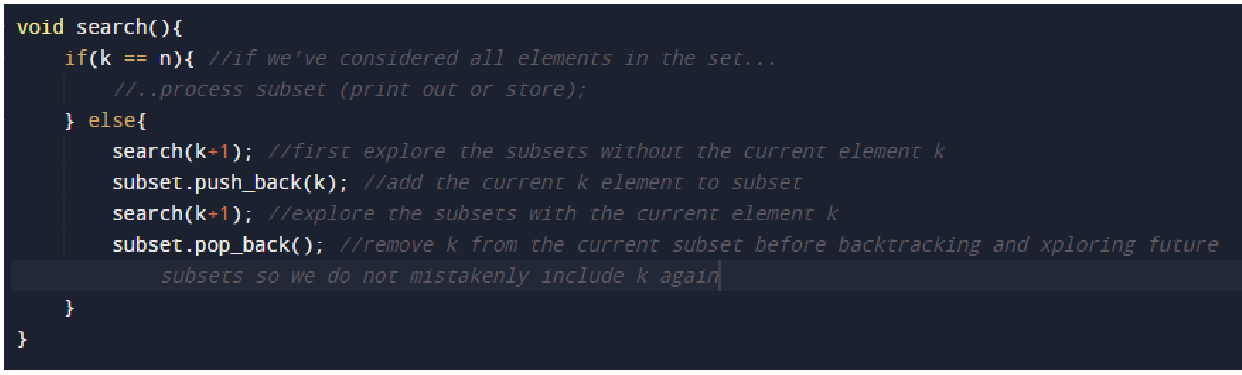
- A priority queue maintains a set of elements. The supported operations are insertion and, depending on the type of the queue, retrieval and removal of either the minimum or maximum element. Insertion and removal take O(logn) time, and retrieval takes O(1) time.
    - While an ordered set efficiently 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;

```
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.

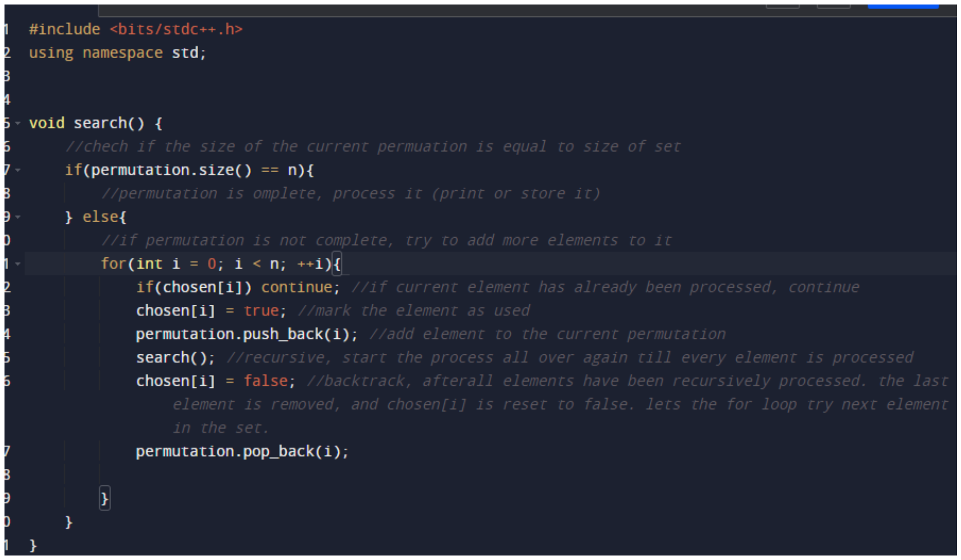
- An elegant way to go through all subsets of a set is to use recursion. The following function search generates the subsets of the set {0,1,...,n − 1}. The function maintains a vector subset that will contain the elements of each subset.



```
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 xploring 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;m++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:

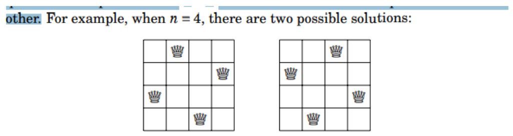


```
#include <bits/stdc++.h>
using namespace std;


void search() {
    //chech if the size of the current permuation is equal to size of set
    if(permutation.size() == n){
        //permutation is omplete, 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, afterall 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);

        }
    }
}
```

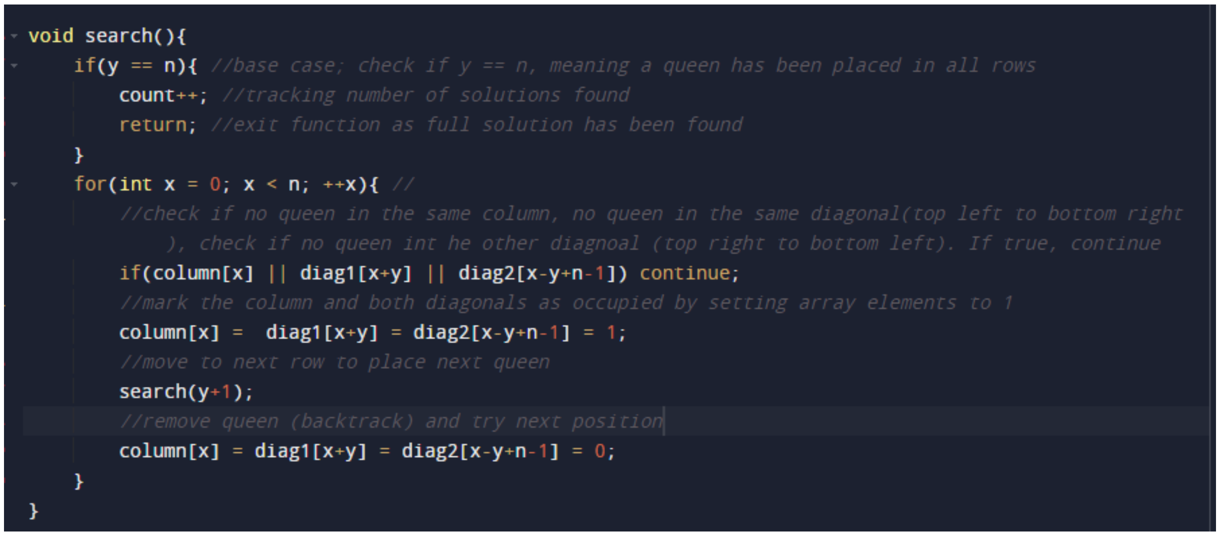
  - Another method for generating permutations is to begin with the permutation {0,1,...,n − 1} and repeatedly use a function that constructs the next permutation in increasing order. The C++ standard library contains the function next_permutation that can be used for this:
    - vector<int> permutation;
    - 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× 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 int he other diagnoal (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(2n), because there are 2n 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(2n/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 coins = {c1, c2,..., ck}, 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 a1,a2,...,an and our task is to find a value x that minimizes the sum |a1 − x|^c +|a2 − x|^c+⋯ +|an − 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.
    - The dynamic programming algorithm is based on a recursive function that goes through all possibilities of how to form the sum, like a brute force algorithm.
- 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 coins = {c1, c2,..., ck} and a target sum of money n, our task is to form the sum n using as few coins as possible
    - In the coin problem, a natural recursive problem is as follows: what is the smallest number of coins required to form a sum x?
    -

Amortized Analysis:

- Amortized analysis can be used to analyze algorithms that contain operations whose time complexity varies. The idea is to estimate the total time used to all such operations during the execution of the algorithm, instead of focusing on individual operations.
- In the **two pointers** method, two pointers are used to iterate through the array values. Both pointers can move to one direction only, which ensures that the algorithm works efficiently.
    - For example, solving a problem where we are given an array of n positive integers and a target sum x, and we want to find a subarray whose sum is x or report that there is no such subarray.
        - This problem can be solved in O(n) time by using the two pointers method. The idea is to maintain pointers that point to the first and last value of a subarray. On each turn, the left pointer moves one step to the right, and the right pointer moves to the right as long as the resulting subarray sum is at most x. If the sum becomes exactly x, a solution has

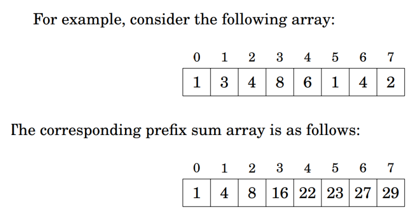
        - been found. Since both the left and right pointer move O(n) steps during the algorithm, the algorithm works in O(n) time.
    - Another example is the two sum problem, where given an array of n numbers and a target sum x, find two array values such that their sum is x, or report that no such values exist.
        - To solve the problem, we first sort the array values in increasing order. After that, we iterate through the array using two pointers. The running time of the algorithm is O(nlogn), because it first sorts the array in O(nlogn) time, and then both pointers move O(n) steps.
    - Amortized analysis is often used to estimate the number of operations performed on a data structure. The operations may be distributed unevenly so that most operations occur during a certain phase of the algorithm, but the total number of the operations is limited.
        - As an example, consider the problem of finding for each array element the nearest smaller element, i.e., the first smaller element that precedes the element in the array
            - We go through the array from left to right and maintain a stack of array elements. At each array position, we remove elements from the stack until the top element is smaller than the current element, or the stack is empty. Then, we report that the top element is the nearest smaller element of the current element, or if the stack is empty, there is no such element. Finally, we add the current element to the stack.
            - each element is added exactly once to the stack and removed at most once from the stack. Thus, each element causes O(1) stack operations, and the algorithm works in O(n) time.
- A sliding window is a constant-size subarray that moves from left to right through the array.
    - The sliding window minimum (smallest value inside each window) can be calculated by maintaining a queue where each element is larger than the previous element, and the first element always corresponds to the minimum element inside the window. After each window move, we remove elements from the end of the queue until the last queue element is smaller than the new window element, or the queue becomes empty. We also remove the first queue element if it is not inside the window anymore. Finally, we add the new window element to the end of the queue.
        - Since each array element is added to the queue exactly once and removed from the queue at most once, the algorithm works in O(n) time.

Range Queries:

- In a range query, our task is to calculate a value based on a subarray of an array. Typical range queries are:
    - sumq(a,b): calculate the sum of values in range [a,b]

- minq(a,b): find the minimum value in range [a,b]
- maxq(a,b): find the maximum value in range [a,b]

- We first focus on a situation where the array is static, i.e., the array values are never updated between the queries. In this case, it suffices to construct a static data structure that tells us the answer for any possible query
- We can easily process sum queries on a static array by constructing a prefix sum array. Each value in the prefix sum array equals the sum of values in the original array up to that position, i.e., the value at position k is sumq(0,k).



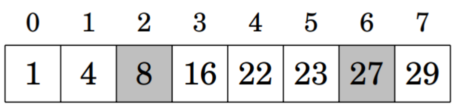
For example, consider the following array:

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

The corresponding prefix sum array is as follows:

0	1	2	3	4	5	6	7
1	4	8	16	22	23	27	29

  - Since the prefix sum array contains all values of sumq(0,k), we can calculate any value of sumq(a,b) in O(1) time as follows: sumq(a,b) = sumq(0,b)−sumq(0,a−1)
  - For example consider the range [3,6]:
    - In this case sumq(3,6) = 8+6+1+4 = 19. This sum can be calculated from two values of the prefix sum array, 6 and 2:
      - Thus, sumq(3,6) = sumq(0,6)−sumq(0,2) = 27−8 = 19



0	1	2	3	4	5	6	7
1	4	8	16	22	23	27	29

- Minimum queries are more difficult to process than sum queries. Still, there is a quite simple O(nlogn) time preprocessing method after which we can answer any minimum query in O(1) time
  - The idea is to precalculate all values of minq(a,b) where b − a +1 (the length of the range) is a power of two. For example, for the array

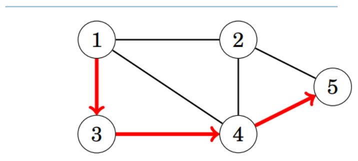
Bit Manipulation:

- All data in computer programs is internally stored as bits, i.e., as numbers 0 and 1.
- In programming, an n bit integer is internally stored as a binary number that consists of n bits. For example, the C++ type int is a 32-bit type, which means that every int number consists of 32 bits.
  - Here is the bit representation of the int number 43: 00000000000000000000000000101011
-

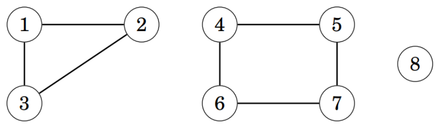
Chapter 2: Graph Algorithms

Basics of Graphs:

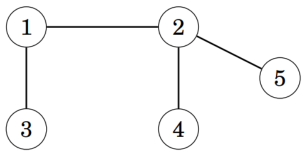
- A graph consists of nodes (or vertex) and edges.
    - The variable n denotes the number of nodes in a graph, and the variable m denotes the number of edges.
- A path leads from node a to node b through edges of the graph. The length of a path is the number of edges in it. For example, this graph contains a path 1 → 3 → 4 → 5 of length 3 from node 1 to node 5:



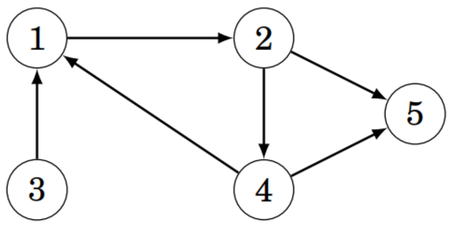
- A path is a cycle if the first and last node is the same. For example, the graph contains a cycle 1 → 3 → 4 → 1. A path is simple if each node appears at most once in the path
- A graph is connected if there is a path between any two nodes.
- The connected parts of a graph are called its components. For example, the following graph contains three components: {1, 2, 3}, {4, 5, 6, 7} and {8} :



- A tree is a connected graph that consists of n nodes and n−1 edges. There is a unique path between any two nodes of a tree. Ex:

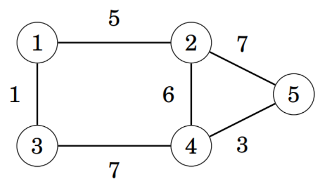


- A graph is directed if the edges can be traversed in one direction only. For example, the following graph is directed:

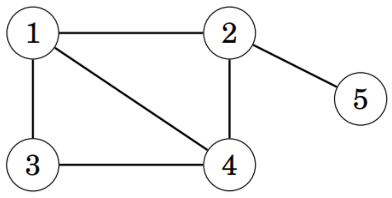


    - The above graph contains a path 3 → 1 → 2 → 5 from node 3 to node 5, but there is no path from node 5 to node 3.

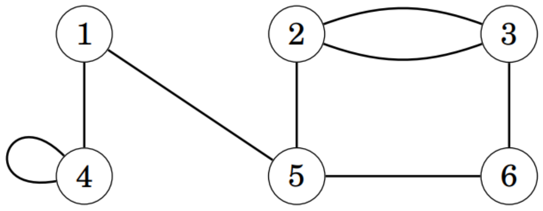
- In a weighted graph, each edge is assigned a weight. The weights are often interpreted as edge lengths. For example, the following graph is weighted:



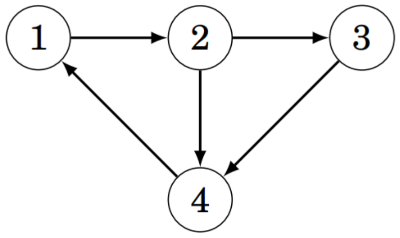
    - The length of a path in a weighted graph is the sum of the edge weights on the path. For example, in the above graph, the length of the path 1 → 2 → 5 is 12, and the length of the path 1 → 3 → 4 → 5 is 11. The latter path is the shortest path from node 1 to node 5.
- Two nodes are neighbors or adjacent if there is an edge between them. The degree of a node is the number of its neighbors. For example, in the following graph, the neighbors of node 2 are 1, 4 and 5, so its degree is 3:



    - The sum of degrees in a graph is always 2m, where m is the number of edges, because each edge increases the degree of exactly two nodes by one. For this reason, the sum of degrees is always even.
    - A graph is regular if the degree of every node is a constant d. A graph is complete if the degree of every node is n−1, i.e., the graph contains all possible edges between the nodes.
    - In a directed graph, the indegree of a node is the number of edges that end at the node, and the outdegree of a node is the number of edges that start at the node.
- In a coloring of a graph, each node is assigned a color so that no adjacent nodes have the same color. A graph is bipartite if it is possible to color it using two colors. It turns out that a graph is bipartite exactly when it does not contain a cycle with an odd number of edges.
- A graph is simple if no edge starts and ends at the same node, and there are no multiple edges between two nodes. Often we assume that graphs are simple. For example, the following graph is not simple:



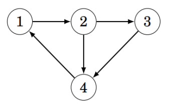
- There are three common representations for representing graphs in DSA.
- In the adjacency list representation, each node x in the graph is assigned an adjacency list that consists of nodes to which there is an edge from x. Most popular.
    - Can store adjacency lists by declaring an array of vectors:
    - vector<int> adj[N]:
        - The constant N is chosen so that all adjacency lists can be stored
        - For example, this graph can be represented as:
        - adj[1].push_back(2); adj[2].push_back(3); adj[2].push_back(4); adj[3].push_back(4); adj[4].push_back(1);



- If the graph is undirected, it can be stored in a similar way, but each edge is added in both directions.
  - adj[1].push_back({2,5}); adj[2].push_back({3,7}); adj[2].push_back({4,6}); adj[3].push_back({4,5}); adj[4].push_back({1,2});
- For example, the following loop goes through all nodes to which we can move from node s:
  - for(auto u: adj[s]){ // process node u}

- An adjacency matrix is a two-dimensional array that indicates which edges the graph contains. We can efficiently check from an adjacency matrix if there is an edge between two nodes. Can be stored as an array:
  - int adj[N][N]
    - where each value adj[a][b] indicates whether the graph contains an edge from node a to node b. If the edge is included in the graph, then adj[a][b] = 1, and otherwise adj[a][b] = 0.

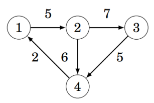
otherwise adj[a][b] = 0. For example, the graph



can be represented as follows:

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

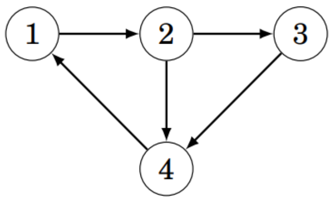
    - Can be extended to use weights. Ex:



corresponds to the following matrix:

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

    - The drawback of the adjacency matrix representation is that the matrix contains n 2 elements, and usually most of them are zero. For this reason, the representation cannot be used if the graph is large.
- An edge list contains all edges of a graph in some order. This is a convenient way to represent a graph if the algorithm processes all edges of the graph and it is not needed to find edges that start at a given node
  - Can be stored in a vector: vector<pair<int, int>> edges;
  - where each pair (a,b) denotes that there is an edge from node a to node b. Thus, this graph can be represented as: edges.push_back({1,2}); edges.push_back({2,3}); edges.push_back({2,4}); edges.push_back({3,4}); edges.push_back({4,1});

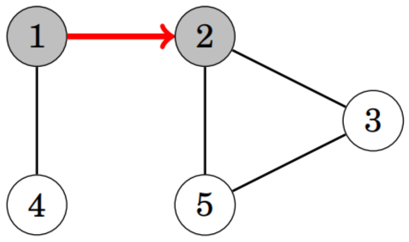


Graph Traversal:

- two fundamental graph algorithms: depth-first search and breadth-first search. Both algorithms are given a starting node in the graph, and they visit all nodes that can be reached from the starting node.

Depth First Search (DFS):

- The algorithm begins at a starting node, and proceeds to all other nodes that are reachable from the starting node using the edges of the graph.
  - always follows a single path in the graph as long as it finds new nodes. After this, it returns to previous nodes and begins to explore other parts of the graph. The algorithm keeps track of visited nodes, so that it processes each node only once.
  - Ex:
    - In the graph to the right, DFS will first begin by searching Node 2. Then it will search Node 3, then Node 5. Finally, it will then move from Node 1 to Node 4.

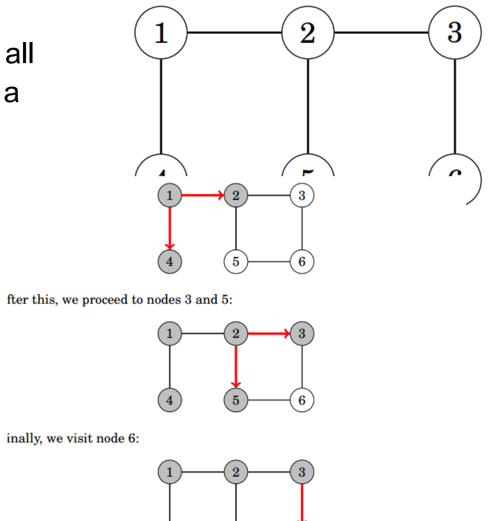


  - The time complexity of depth-first search is O(n+ m) where n is the number of nodes and m is the number of edges, because the algorithm processes each node and edge once.
  - Can be implemented using recursion.The function assumes that the graph is stored as adjacency lists in an array:
    - vector<int> adj[n];
    - Also maintain an array: bool visited[n]
    - void dfs(int s){

```
  if(visited[s]) return;
  visited[s] = true;
  //process node s
  for(auto u: adj[s]){
    dfs(u);
  }
}
```

Breadth First Search(BFS):

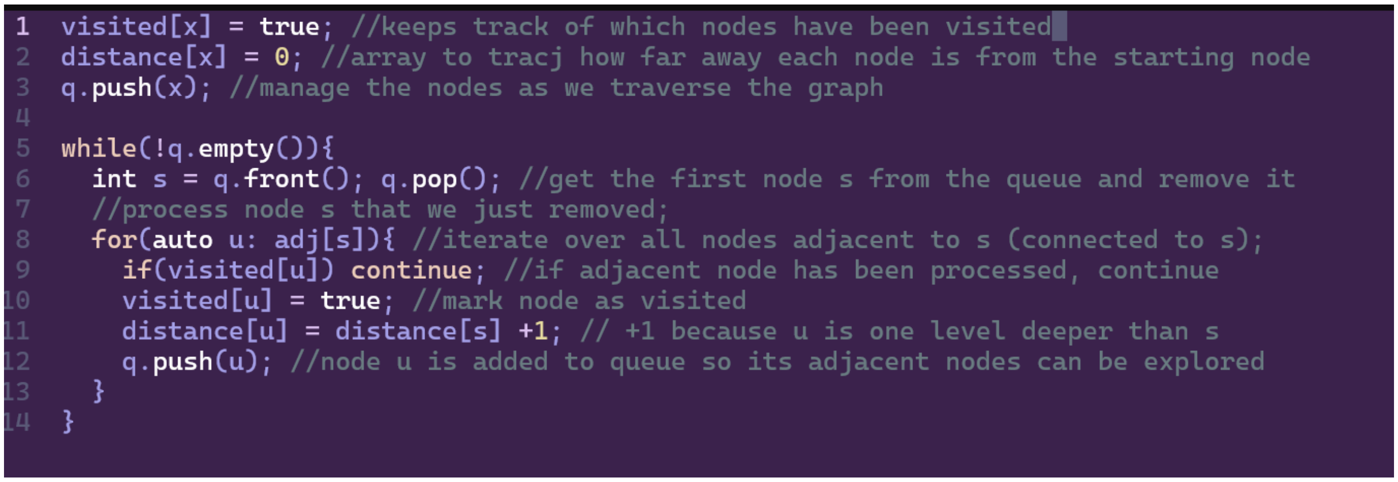
- Visits the nodes in increasing order of their distance from the starting node. Thus, we can calculate the distance from the starting node to all other nodes using breadth-first search. Goes through the nodes one level after another. First the search explores the nodes whose distance from the starting node is 1, then the nodes whose distance is 2, and so on. This process continues until all nodes have been visited.
  - Ex:
    - In the graph to the right, BFS will first process all nodes that can be reached from node 1 using a single edge
    - Then it will proceed to Nodes 3 and 5
    - Then it will proceed to node 6



fter this, we proceed to nodes 3 and 5:

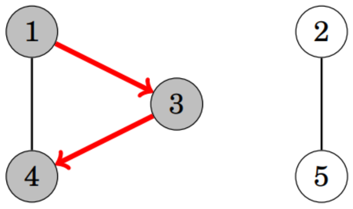
inally, we visit node 6:

- Like in depth-first search, the time complexity of breadth-first search is O(n+ m), where n is the number of nodes and m is the number of edges.
- A typical implementation is based on a queue that contains nodes. At each step, the next node in the queue will be processed.
- The queue q contains nodes to be processed in increasing order of their distance. New nodes are always added to the end of the queue, and the node at the beginning of the queue is the next node to be processed.
- The array visited indicates which nodes the search has already visited, and the array distance will contain the distances from the starting node to all nodes of the graph.
- Code below assumes that the graph is stored as adjacency lists and maintains the following data structures:
    - queue q; bool visited[N]; int distance[N];
    - Code for BFS:

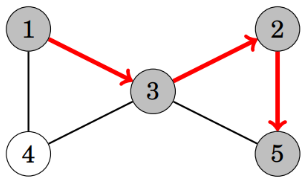


```
visited[x] = true; //keeps track of which nodes have been visited
distance[x] = 0; //array to tracj how far away each node is from the starting node
q.push(x); //manage the nodes as we traverse the graph

while(!q.empty()){
  int s = q.front(); q.pop(); //get the first node s from the queue and remove it
  //process node s that we just removed;
  for(auto u: adj[s]){ //iterate over all nodes adjacent to s (connected to s);
    if(visited[u]) continue; //if adjacent node has been processed, continue
    visited[u] = true; //mark node as visited
    distance[u] = distance[s] +1; // +1 because u is one level deeper than s
    q.push(u); //node u is added to queue so its adjacent nodes can be explored
  }
}
```

- A graph is connected if there is a path between any two nodes of the graph. Thus, we can check if a graph is connected by starting at an arbitrary node and finding out if we can reach all other nodes.
    - Ex:
        - a depth-first search from node 1 visits the following nodes:
        - Since the search did not visit all the nodes, we can conclude that the graph is not connected.



- A graph contains a cycle if during a graph traversal, we find a node whose neighbor (other than the previous node in the current path) has already been visited. For example, the graph
    - Ex:



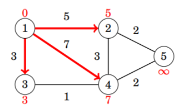
- After moving from node 2 to node 5 we notice that the neighbor 3 of node 5 has already been visited. Thus, the graph contains a cycle that goes through node 3, for example, 3 → 2 → 5 → 3.

Shortest Paths:

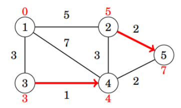
Bellman-Ford Algorithm:

- The Bellman–Ford algorithm finds shortest paths from a starting node to all nodes of the graph. The algorithm can process all kinds of graphs, provided that the graph does not contain a cycle with negative length.
  - The algorithm keeps track of distances from the starting node to all nodes of the graph. Initially, the distance to the starting node is 0 and the distance to all other nodes is infinite. The algorithm reduces the distances by finding edges that shorten the paths until it is not possible to reduce any distance.
  - The following implementation of the Bellman–Ford algorithm determines the shortest distances from a node x to all nodes of the graph. The code assumes that the graph is stored as an edge list of edges that consists of tuples of the form (a,b,w), meaning that there is an edge from node a to node b with weight w.
  - The algorithm consists of n−1 rounds, and on each round the algorithm goes through all edges of the graph and tries to reduce the distances. The algorithm constructs an array distance that will contain the distances from x to all nodes of the graph. The constant INF denotes an infinite distance.
  - The time complexity of the algorithm is O(nm), because the algorithm consists of n−1 rounds and iterates through all m edges during a round. If there are no negative cycles in the graph, all distances are final after n −1 rounds, because each shortest path can contain at most n−1 edges.

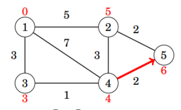
The algorithm searches for edges that reduce distances. First, all edges from node 1 reduce distances:



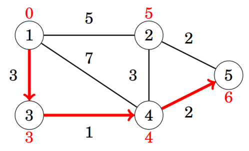
After this, edges 2 → 5 and 3 → 4 reduce distances:



Finally, there is one more change:



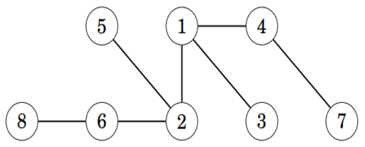
For example, the shortest distance 3 from node 1 to node 5 corresponds to the following path:



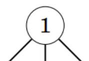
Tree Algorithms:

- A tree is a connected, acyclic graph that consists of n nodes and n − 1 edges. Removing any edge from a tree divides it into two components, and adding any edge to a tree creates a cycle. Moreover, there is always a unique path between any two nodes of a tree.

For example, the following tree consists of 8 nodes and 7 edges:



The **leaves** of a tree are the nodes with degree 1, i.e., with only one neighbor. For example, the leaves of the above tree are nodes 3, 5, 7 and 8.



- In a rooted tree, one of the nodes is appointed the root of the tree, and all other nodes are placed underneath the root. For example, in the tree to the right, node 1 is the root node.
- The structure of a rooted tree is recursive: each node of the tree acts as the root of a subtree that contains the node itself and all nodes that are in the subtrees of its children. For example, in the tree to the right, the subtree of node 2 consists of nodes 2, 5, 6 and 8
- General graph traversal algorithms can be used to traverse the nodes of a tree. However, the traversal of a tree is easier to implement than that of a general graph, because there are no cycles in the tree and it is not possible to reach a node from multiple directions.
    - The typical way to traverse a tree is to start a depth-first search at an arbitrary node. The following recursive function can be used:

        -
          ```
          Void dfs(int s, int e){
            //process node s
            for(auto u: adj[s]){
                if(u != e) dfs(u, e);
            }
          }
          ```
        - The function is given two parameters: the current node s and the previous node e. The purpose of the parameter e is to make sure that the search only moves to nodes that have not been visited yet.
- Dynamic programming can be used to calculate some information during a tree traversal. Using dynamic programming, we can, for example, calculate in O(n) time for each node of a rooted tree the number of nodes in its subtree or the length of the longest path from the node to a leaf.
    - As an example, let us calculate for each node S, a value count[s]: the number of nodes in its subtree. The subtree contains the node itself and all nodes in the subtrees of its children, so we can calculate the number of nodes recursively using the following code:

        -
          ```
          void dfs(int s, int e) {
              count[s] = 1;
              for (auto u : adj[s]) {
                  if (u == e) continue;
                  dfs(u, s);
                  count[s] += count[u];

              }
          }
          ```
-