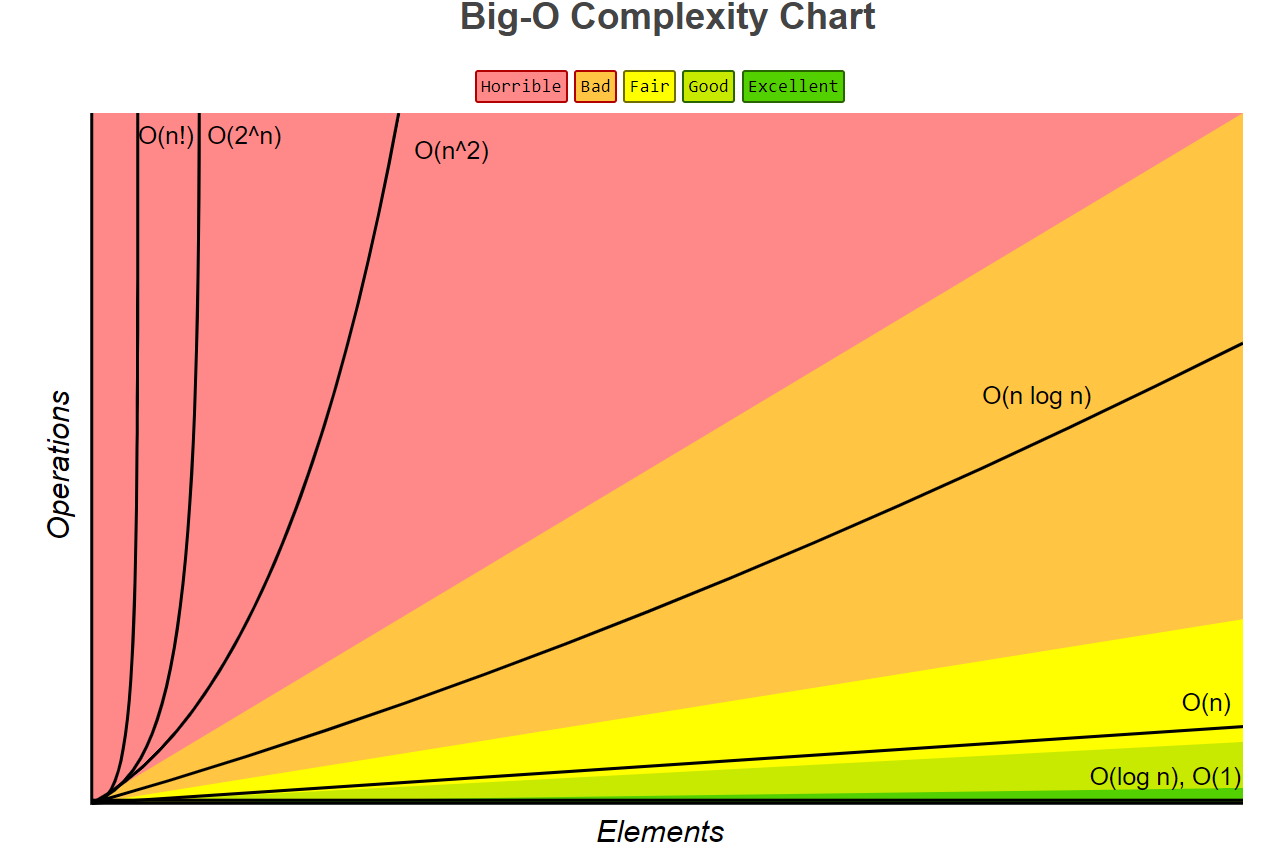
## 1. Algorithm Complexities

Asymptotic notations:

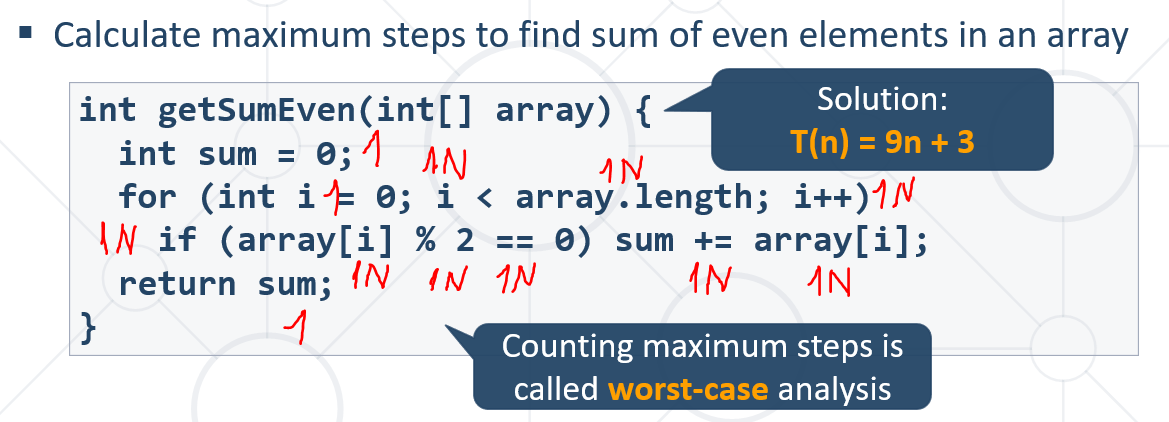
* + Big **O – O(f(n)) – worst case - in our course**
  + Big **Theta – Θ(f(n)) – амортизиран amortized constant time**
  + Big **Omega – Ω(f(n))**



* O(1) – Constant time – time does not depend on **N**
* O(log(N)) – Logarithmic time – grows with rate as **log(N)** *log2 64 = 6 (2^6 = 64)*
* O(N) – Linear time grows at the same rate as **N**
* O(N^2),O(N^3) – Quadratic, Cubic grows as square or cube of **N**
* O(2^N) – Exponential grows as **N** becomes the exponent worst algorithmic complexity

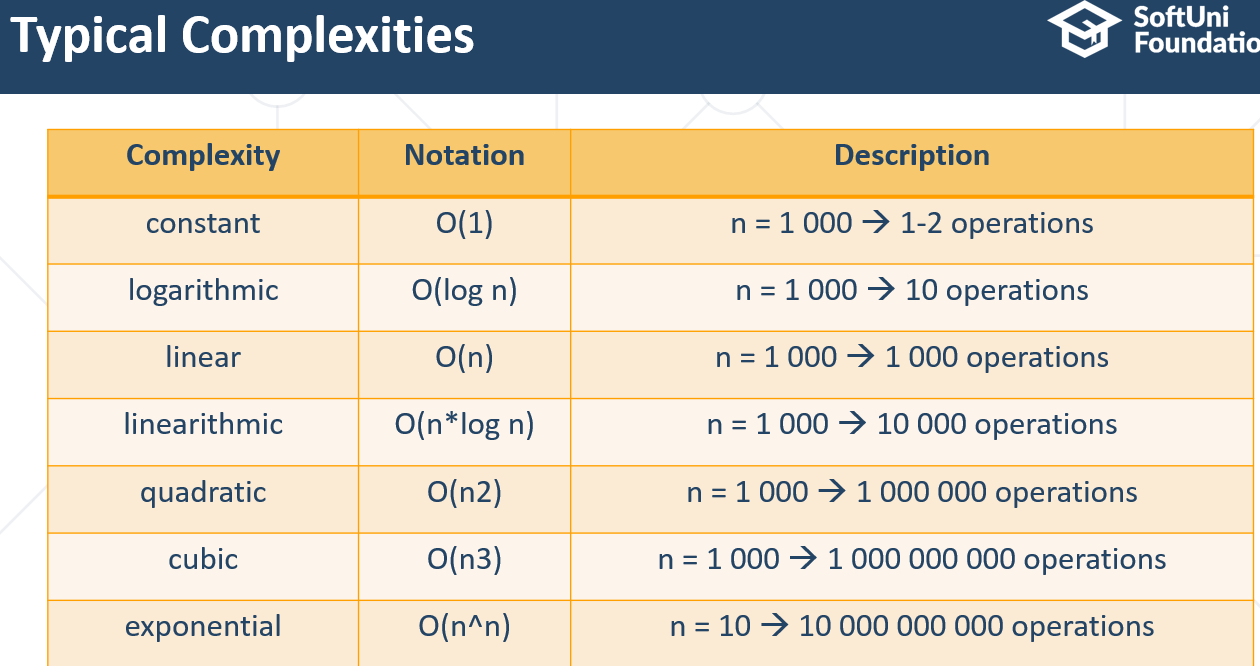
Assume that a **single step** is a single CPU instruction.

Гледаме колко стъпки има алгортъма ни, а не колко памет е заета (което е различно)



Инструкция на процесора можем да кажем, че е код завършващ с точка и запетая накрая.

Реално повече инструкции на процесора има на 1 ред код -аритметични/логически и т.н.



**Brute-Force Algorithms –** преминава през всички случаи/варианти – не е ефективен, но се ползва

## 2. Recursion

За да разберете какво е рекурсия, първо трябва да разберете какво е рекурсия 😊

* **Method** of solving a problem where the solution depends on solutions to smaller instances of the same problem
* A common **computer programing tactic** is to **divide** a problem into **sub-problems** of the same type as the original, **solve** those sub-problems, and **combine** the **results**
* A function or a method that calls itself one or more times until a specified condition is met
* After the recursive call the rest code is processed from the last one called to the first
* Дефиницията за рекурсия, е че може да няма дъно

Recursive methods have 3 parts:

* + **Pre-actions** (before calling the recursion)
  + **Recursive** **calls** (step-in)
  + **Post-actions** (after returning from recursion)

**static void recursion() {**

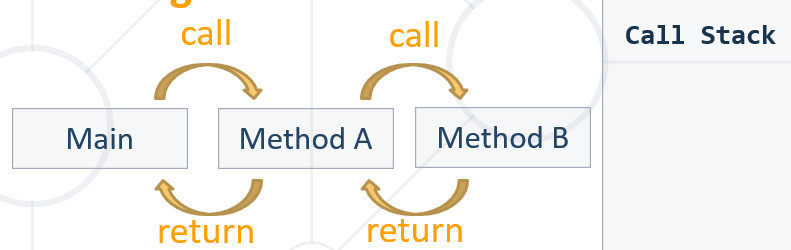
**// *Pre-actions***

**recursion();**

**// *Post-actions***

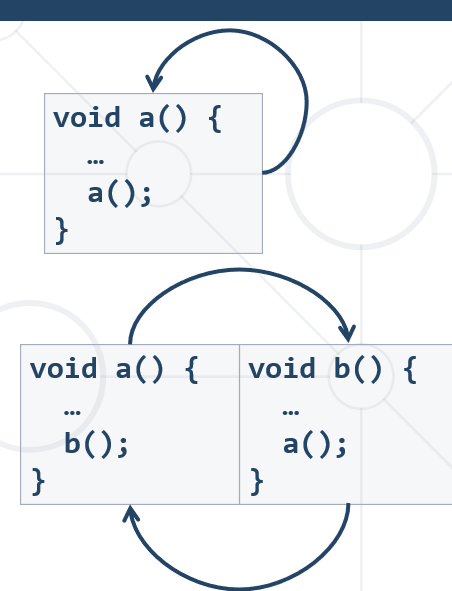
**}**

Работа на Call Stack



**Other Definition of Recursion:**

* Involves a **function calling itself**
* The function should have a **base case /дъно/**
* **Each step** of the recursion should **move towards** the **base case /дъно/**
* **Обаждаме инстанциите до дъното, и след това връщаме от дъното към върха резултата**
* Direct recursion
  + A method directly calls itself
* Indirect recursion
  + Method **A** calls **B**, method **B** calls **A**
  + Or even **A** **🡪** **B** **🡪** **C** **🡪** **A**



**private static void** drawFigure(**int** n) {  
 **if** (n == 0) {  
 **return**;  
 }  
  
 **for** (**int** i = 0; i < n; i++) {  
 System.***out***.print(**"\*"**);  
 }  
 System.***out***.println();  
 *drawFigure*(n-1); **// до тук правим Call до дъното**  
  
 **for** (**int** i = 0; i < n; i++) **{//оттук нататък правим Return от дъното нагоре – post action**  
 System.***out***.print(**"#"**);  
 }  
 System.***out***.println();  
 }  
}

## 3. Backtracking

find all paths from Source to Destination

Рекурсивно извикваме алгоритъма от всяка една точка

**We use recursion and Backtracking for branches problems.**

**Otherwise, we should use linear iterative algorithm**

## **4. Memoization** – not to be confused with memorization

Когато искаме да спестим вече изчислената част от дървото на рекурсията -да не я изчислява за всяко клонче същата част отново и отново.

Записваме в масив или друг тип колекция.

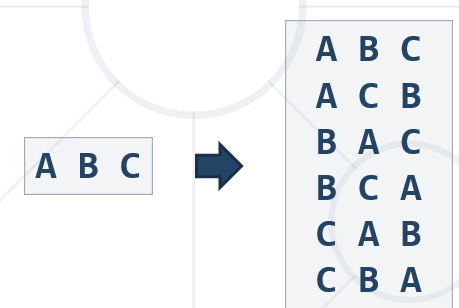
1, 1, 2, 3, 5, 8, 13, 21, 34 – нулевият елемент е 1, първият елемент е също 1, вторият елемент е 2, третият елемент е 3, четвъртият елемент е 5, петият елемент е 8

**public class** RecursiveFibonacci {  
 **public static** Long[] *storedFibbonachiNumbers*;  
  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
 **int** n = Integer.*parseInt*(sc.nextLine());  
 *storedFibbonachiNumbers* = **new** Long[n + 1];  
  
 **for** (**int** i = 0; i < n+1; i++) { *//запълваме си масива с нули  
 storedFibbonachiNumbers*[i] = 0L;  
 }  
  
  
 **if** (n >= 1) { *//нулвеият и първият елемент от масива са 1  
 storedFibbonachiNumbers*[0] = 1L;  
 *storedFibbonachiNumbers*[1] = 1L;  
 } **else** { *//когато n e нула, то дължината на масива е 1. Нулевият елемент на масива е 1  
 storedFibbonachiNumbers*[0] = 1L;  
 }  
  
 Long fib = *fibonacci*(n);  
 System.***out***.println(fib);  
*// System.out.println(fibonacci(50)); // This will hang!* }  
  
 **static long** fibonacci(**int** n) {  
 **if** (n == 0) {  
 **return** *storedFibbonachiNumbers*[0];  
 } **else if** (n == 1) {  
 **return** *storedFibbonachiNumbers*[1];  
 } **else** {  
 **if** (*storedFibbonachiNumbers*[n] > 0L) {  
 **return** *storedFibbonachiNumbers*[n];  
 }  
  
 **return** *storedFibbonachiNumbers*[n] = *fibonacci*(n - 1) + *fibonacci*(n - 2);  
 }  
 }  
}

## 5. Combinatorial Problems

### 5.1. Permutations

Permutation of a set of items is arrangement all the items in the set, linear, in all possible ways



#### Permutations Count - при неповтарящи се елементи

= n! = 3! factorial = 6 possible ways –

#### Нормална пермутация – без повторения

**import** java.util.Scanner;  
  
**public class** PermutationsWithoutRepetitions {  
 **public static** String[] *elements*;  
 **public static** String[] *permutes*;  
 **public static boolean**[] *used*;  
  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
 *elements* = sc.nextLine().split(**"\\s+"**);  
 *permutes* = **new** String[*elements*.**length**];  
 *used* = **new boolean**[*elements*.**length**];  
  
 *permute*(0);  
 }  
  
 **private static void** permute(**int** index) {  
 **if** (index == *elements*.**length**) {  
 *print*();  
 **return**;  
 }  
  
 **for** (**int** i = 0; i < *elements*.**length**; i++) {  
 **if** (!*used*[i]) { *//ако не е използван дадения елемент  
 used*[i] = **true**;  
 *permutes*[index] = *elements*[i];  
 *permute*(index + 1);  
 *used*[i] = **false**; *// the Backtracking* }  
 }  
 }  
  
 **private static void** print() {  
 System.***out***.println(String.*join*(**" "**, *permutes*));  
 }  
}

#### Optimize permutations – Swap algorithm: - ползва по-малко памет

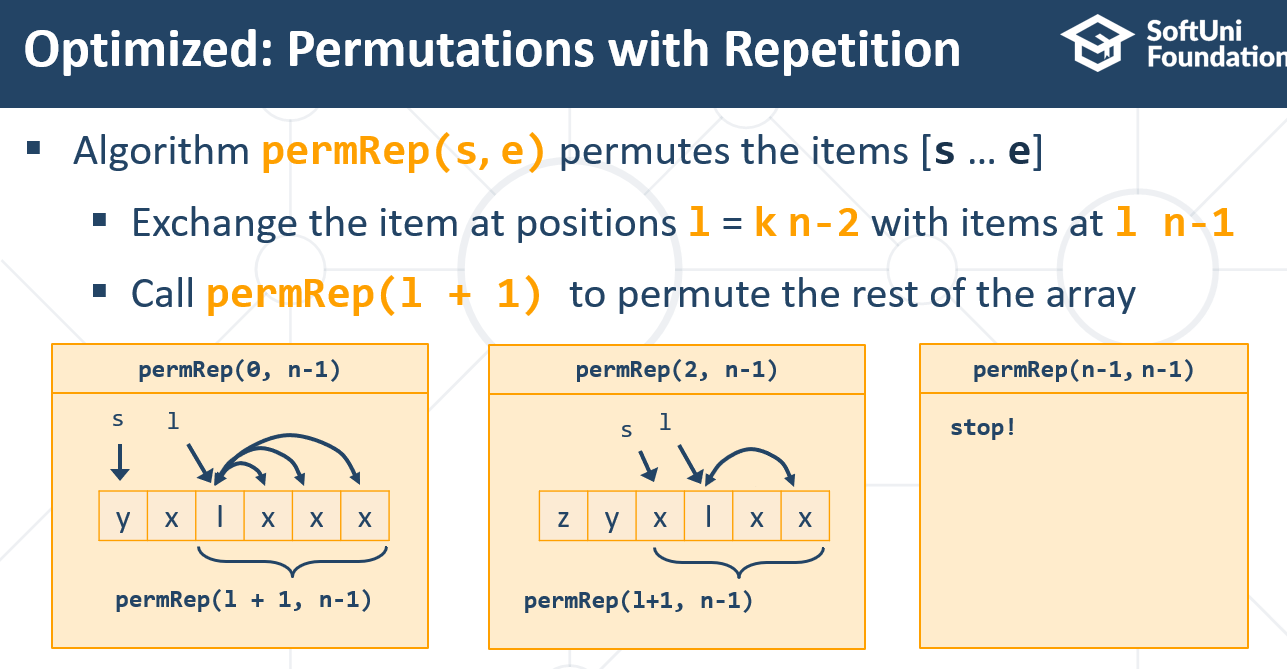
**private static void** permute(**int** index) {  
 **if** (index == *elements*.**length**) {  
 *print*(*elements*);  
 **return**;  
 }  
  
 *permute*(index + 1);  
 **for** (**int** i = index + 1; i < *elements*.**length**; i++) {  
 *swap*(*elements*, index, i);  
 *permute*(index + 1);  
 *swap*(*elements*, index, i); *//backtracking (unswapping)* }  
}  
  
**private static void** swap(String[] arr, **int** first, **int** second) {  
 String temp = arr[first];  
 arr[first] = arr[second];  
 arr[second] = temp;  
  
}

#### Permutations with Repetitions and Swap algorithm

**public class** PermutationsWithRepetitionsSwap {  
 **public static** String[] *elements*;  
  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
 *elements* = sc.nextLine().split(**"\\s+"**);  
  
 *permute*(0);  
 }  
  
 **private static void** permute(**int** index) {  
 **if** (index == *elements*.**length**) {  
 *print*(*elements*);  
 **return**;  
 }  
 *permute*(index + 1);  
 HashSet<String> swapped = **new** HashSet<>();  
 swapped.add(*elements*[index]);  
  
 **for** (**int** i = index + 1; i < *elements*.**length**; i++) {  
 **if** (!swapped.contains(*elements*[i])) {  
 *swap*(*elements*, index, i);  
 *permute*(index + 1);  
 *swap*(*elements*, index, i); *//backtracking (unswapping)* swapped.add(*elements*[i]);  
 }  
  
 }  
 }  
  
 **private static void** swap(String[] arr, **int** first, **int** second) {  
 String temp = arr[first];  
 arr[first] = arr[second];  
 arr[second] = temp;  
  
 }  
  
 **private static void** print(String[] arr) {  
 System.***out***.println(String.*join*(**" "**, arr));  
 }  
}

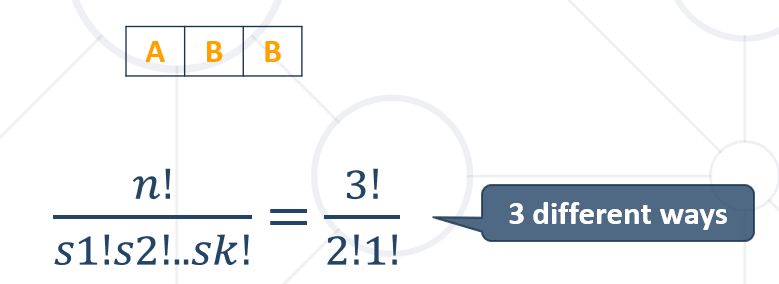
#### Permutations with Repetitions and Swap algorithm – по-комплексен алгоритъм (когато имаме много повтарящи се елементи)

To check it – нещо не работи алгоритъма



**public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
  
 String[] arr = {**"3"**, **"5"**, **"1"**, **"5"**, **"5"**};  
 Arrays.*sort*(arr); *// 1 3 5 5 5  
 permuteRep*(arr, 0, arr.**length** - 1);  
}  
  
**static void** permuteRep(String[] arr, **int** start, **int** end) {  
 *print*(arr);  
 **for** (**int** left = end - 1; left >= start; left--)  
 **for** (**int** right = left + 1; right <= end; right++) {  
 **if** (!arr[left].equals(arr[right])) {  
 *swap*(arr, left, right);  
 *permuteRep*(arr, left + 1, end);  
 }  
 String firstElement = arr[left];  
 **for** (**int** i = left; i <= end - 1; i++) {  
 arr[i] = arr[i + 1];  
 }  
 arr[end] = firstElement;  
 }  
}

#### Permutations with Repetition Count

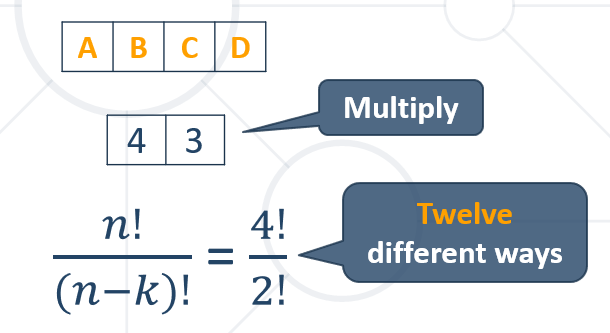


### 5.2. Variations

Given set of elements N and K slots:

order the N elements in all the possible ways inside the K slots

#### Variations Count - при неповтарящи се елементи



#### Нормална вариация – без повторение

**public class** VariationsWithoutRepetition {  
 **public static** String[] *elements*;  
 **public static** String[] *variations*;  
 **public static boolean**[] *used*;  
  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
 *elements* = sc.nextLine().split(**"\\s+"**);  
  
 **int** k = Integer.*parseInt*(sc.nextLine()); *// k Slots  
 variations* = **new** String[k];  
 *used* = **new boolean**[*elements*.**length**];  
  
 *variationsMethod*(0);  
 }  
  
 **private static void** variationsMethod(**int** index) {  
 **if** (index == *variations*.**length**) {  
 *print*(*variations*);  
 **return**;  
 }  
  
 **for** (**int** i = 0; i < *elements*.**length**; i++) {  
 **if** (!*used*[i]) {  
 *used*[i] = **true**;  
 *variations*[index] = *elements*[i];  
 *variationsMethod*(index + 1);  
 *used*[i] = **false**; *// backtracking* }  
 }  
  
 }  
  
 **private static void** print(String[] arr) {  
 System.***out***.println(String.*join*(**" "**, arr));  
 }  
}

#### Вариация с повторение – повтаряме всеки един елемент в слотовете K

**private static void** variationsMethod(**int** index) {  
 **if** (index == *variations*.**length**) {  
 *print*(*variations*);  
 **return**;  
 }  
  
 **for** (**int** i = 0; i < *elements*.**length**; i++) {  
 *variations*[index] = *elements*[i];  
 *variationsMethod*(index + 1);  
  
 }  
}

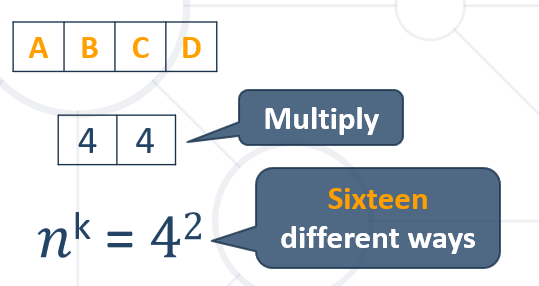
#### Вариация с повторение – с използваме на итерация, т.е. без рекурсия

**int** n = 5;  
**int** k = 3;  
**int**[] arr = **new int**[k];

**while** (**true**) {  
 print(arr);  
 **int** index = k - 1;  
 **while** (index >= 0 && arr[index] == n-1)  
 index--;  
 **if** (index < 0)  
 **break**;  
 arr[index]++;  
 **for** (**int** i = index + 1; i < k; i++)  
 arr[i] = 0;  
}

#### Variations Count - при повтаряне на елементи

nk



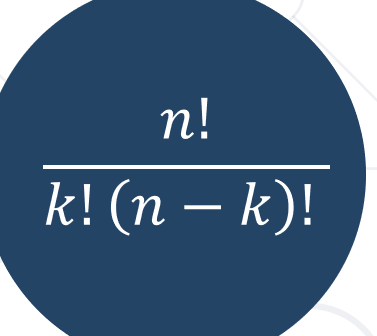
### 5.3. Combinations

Изпринтирай всички k сетове образувани от общо n елемента.

Един сет от K елемента е примерно:

При k ==2, то A B и B A се брои за една комбинация/сет, а не за две.

#### Combinations Count - без повтаряне на елементи



#### Нормална комбинация – без повторение

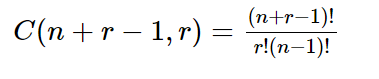
**public class** CombinationsWithoutRepetition {  
 **public static** String[] *elements*;  
 **public static** String[] *variations*;  
  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
 *elements* = sc.nextLine().split(**"\\s+"**);  
  
 **int** k = Integer.*parseInt*(sc.nextLine());  
 *variations* = **new** String[k];  
  
 *combinations*(0, 0);  
 }  
  
 **private static void** combinations(**int** index, **int** start) {  
 **if** (index == *variations*.**length**) { *//дъно  
 print*(*variations*);  
 } **else** {  
 **for** (**int** i = start; i < *elements*.**length**; i++) {  
 *variations*[index] = *elements*[i];  
 *combinations*(index +1, i+1);  
 }  
 }  
 }  
  
 **private static void** print(String[] arr) {  
 System.***out***.println(String.*join*(**" "**, arr));  
 }  
}

#### Комбинация с повторение – един елемент от n-те може да се съдържа в сета/комбинацията с K-слота K пъти

**public class** CombinationsWithRepetition {  
 **public static** String[] elements;  
 **public static** String[] variations;  
  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.in);  
 elements = sc.nextLine().split(**"\\s+"**);  
  
 **int** k = Integer.parseInt(sc.nextLine());  
 variations = **new** String[k];  
  
 combinations(0, 0);  
 }  
  
 **private static void** combinations(**int** index, **int** start) {  
 **if** (index == variations.length) { *//дъно* print(variations);  
 } **else** {  
 **for** (**int** i = start; i < elements.length; i++) {  
 variations[index] = elements[i];  
 combinations(index + 1, i);  
 }  
 }  
 }  
  
 **private static void** print(String[] arr) {  
 System.out.println(String.join(**" "**, arr));  
 }  
}

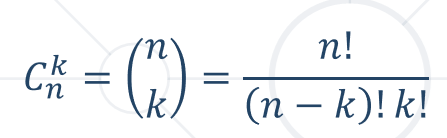
#### Combinations Count - с повтаряне на елементи:

to select *r* things from *n* possibilities

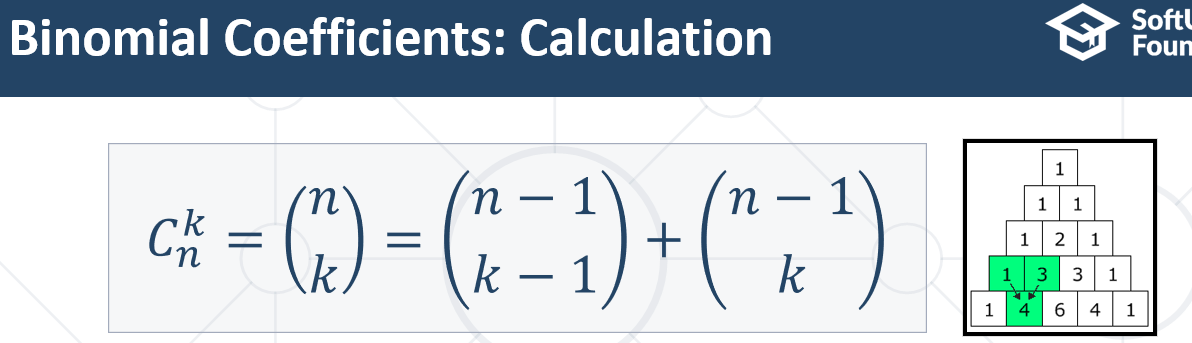


### 5.4. N Choose K Count

Начин на решаване на формулата за комбинации – от гледна точка на спестяване на компютърна памет/операции



Използваме пирамидата на Паскал



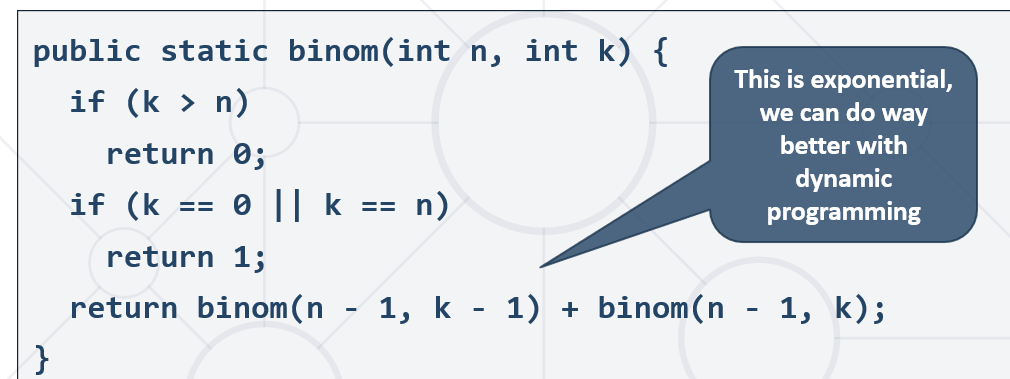
Base cases (дъна):

if k > n 🡪 0

if k == 0 🡪 1

if k == n 🡪 1

**private static int** binom(**int** n, **int** k) {  
 **if** (k > n) {  
 **return** 0;  
 }  
 **if** (k == 0 || k == n) {  
 **return** 1;  
 }  
  
 **return** *binom*(n - 1, k - 1) + *binom*(n - 1, k);  
}



## 6. Searching, Sorting and Greedy Algorithms

### 6.1. Searching

#### 6.1.1 Linear search

Worst & average performance: O(n)

**public class** LinearSearch {  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
 **int**[] arr = {13, 2, 34, 73, 24, 86};  
 System.***out***.println(*indexOf*(arr, 37));  
 }  
  
 **private static int** indexOf(**int**[] arr, **int** key) {  
 **for** (**int** i = 0; i < arr.**length**; i++) {  
 **if** (arr[i] == key) {  
 **return** i;  
 }  
 }   
 **return** -1;  
 }  
   
}

#### 6.1.1 Binary search

finds an item within a ordered data structure – търси половината от половината от половината от ……

Average performance: **O(log(n))**

**public class** BinarySearchIterative {  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
 **int**[] arr = Arrays.*stream*(sc.nextLine().split(**"\\s+"**))  
 .mapToInt(x -> Integer.*parseInt*(x))  
 .toArray();  
 Arrays.*sort*(arr);  
  
 **int** key = Integer.*parseInt*(sc.nextLine());  
 System.***out***.println(*indexOf*(arr, key));  
 }  
  
 **private static int** indexOf(**int**[] arr, **int** key) {  
 **int** start = 0;  
 **int** end = arr.**length** - 1;  
  
 **while** (start <= end) {  
 **int** mid = (start + end) / 2;  
 **int** curr = arr[mid];  
 **if** (key < curr) {  
 end = mid - 1;  
 } **else if** (key > curr) {  
 start = mid + 1;  
 } **else** {  
 **return** mid;  
 }  
 }  
 **return** -1;  
 }  
  
}

### 6.2. Sorting

<https://visualgo.net/> - сайт за визуализация на сортировки и други структури от данни с техните алгоритми

Efficient sorting algorithms

Sorting algorithms are often classified by:

* Computational **complexity** and memory usage
* **Recursive** / non-recursive
* **Stability** – stable / unstable – стабилен например при срещане на еднакви елементи, то няма да ги swap-не
* **Comparison-based** **sort** / non-comparison based (като броим например)
* Sorting **method**: insertion, exchange (bubble sort and quicksort), selection (heapsort), merging, serial / parallel, etc.

#### 6.2.1. Simple algorithms

##### I. Selection sort – не е много ефективен, unstable

Swap the first with the min element on the right, then the second, etc

Като първи елемент отива най-малкия, след това гледаме за всички елементи след 1ият, след това за всички елементи след 2рият

**public class** SelectionSort {  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
 **int**[] arr = {5, 4, 3, 2, 1};  
 *sort*(arr);  
  
 **for** (**int** i : arr) {  
 System.***out***.print(i + **" "**);  
 }  
 }  
  
 **private static void** sort(**int**[] arr) {  
 **for** (**int** index = 0; index < arr.**length**; index++) {  
 **int** min = index;  
 **for** (**int** curr = index + 1; curr < arr.**length**; curr++) {  
 **if** (arr[curr] < arr[min]) {  
 min = curr;  
 }  
 }  
 *swap*(arr, index, min);  
 }  
 }  
  
 **private static void** swap(**int**[] arr, **int** first, **int** second) {  
 **int** temp = arr[first];  
 arr[first] = arr[second];  
 arr[second] = temp;  
 }  
}

##### II. Bubble sort – simple, but inefficient algorithm, but **stable**

Сравнява две съседни и влачи най-тежкия елемент накрая, и не прави swap

**private static void** sort(**int**[] arr) {  
 **for** (**int** i = 0; i < arr.**length**; i++) { *//брой мехурчета изплували / най-тежкия накрая* **for** (**int** j = 1; j < arr.**length** - i; j++) { *//самото сравнение на всички съседни* **if** (arr[j - 1] > arr[j]) { *//възходящо сравнение  
 swap*(arr, j - 1, j);  
 }  
 }  
 }  
}

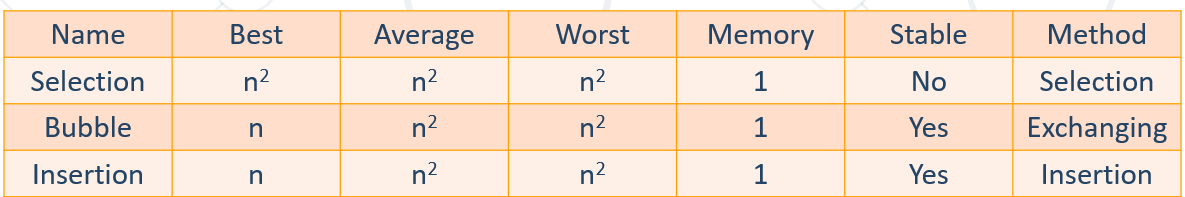
Или

**public static void** sort(**int**[] arr) {  
 **int** n = arr.**length**;  
  
 **for** (**int** k = 0; k < n - 1; k++) { //брой операции

**for** (**int** i = 0; i < n - k - 1; i++) { **//размени два съседни, като всеки път следващи два съседни взема** **if** (arr[i] > arr[i + 1]) {  
 **int** temp = arr[i];  
 arr[i] = arr[i + 1];  
 arr[i + 1] = temp;  
 }  
 }  
 }  
}

##### III. Insertion sort - simple, but inefficient algorithm, but **stable**

Move the first unsorted element left to its place



#### 6.2.2. Shuffling

Randomizing the order of items in a collection - Generate a random permutation

**public class** Shuffling {  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
  
 **int**[] arr = {13, 15, 12, 24, 59};  
 Arrays.*sort*(arr);  
  
 *getAsRand*(arr);  
  
 **for** (**int** i : arr) {  
 System.***out***.print(i + **" "**);  
 }  
 }  
  
 **private static void** getAsRand(**int**[] arr) {  
 Random rand = **new** Random();  
 **for** (**int** i = 0; i < arr.**length**; i++) {  
 *swap*(arr, i, rand.nextInt(arr.**length** - 1)); // може и без минус 1  
 }  
 }  
  
 **private static void** swap(**int**[] arr, **int** first, **int** second) {  
 **int** temp = arr[first];  
 arr[first] = arr[second];  
 arr[second] = temp;  
 }  
}

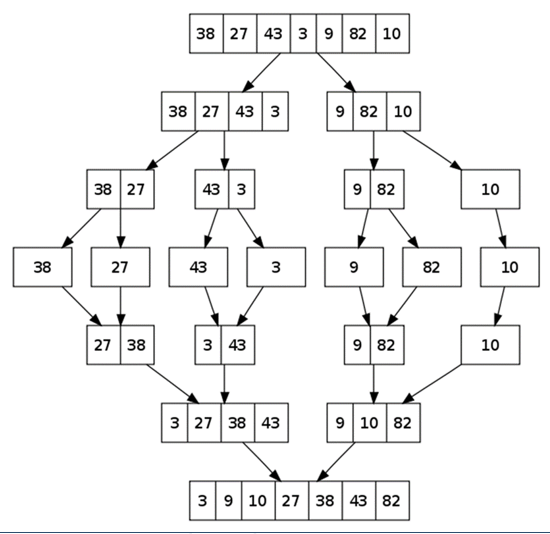
#### 6.2.3. Advanced Sorting Algorithms – recursive, saving iterations and memory/time

##### I. Merge Sort

Efficient- from O(n \* log(n)) up to O(log(n))

Divide the list into sub-lists (typically 2 sub-lists):

* Sort each sub-list (recursively call merge-sort)
* Merge the sorted sub-lists into a single list



**public class** MergeSort {  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
 **int**[] arr = Arrays.*stream*(sc.nextLine().split(**"\\s+"**)).mapToInt(x -> Integer.*parseInt*(x)).toArray();  
 *mergeSort*(arr, 0, arr.**length** - 1);  
  
 StringBuilder builder = **new** StringBuilder();  
 **for** (**int** num : arr) {  
 builder.append(num).append(**" "**);  
 }  
 System.***out***.println(builder.toString());  
 }  
  
 **private static void** mergeSort(**int**[] arr, **int** begin, **int** end) {  
 **if** (begin >= end) {  
 **return**;  
 }  
 **int** mid = (begin + end) / 2;  
  
 *mergeSort*(arr, begin, mid);  
 *mergeSort*(arr, mid + 1, end);  
  
 *merge*(arr, begin, mid, end); *//backtracking* }  
  
 **private static void** merge(**int**[] arr, **int** begin, **int** mid, **int** end) {  
 **if** (mid < 0 || mid >= arr.**length** || arr[mid] < arr[mid + 1]) { *// ако последния елемент на сортирания вече събмасив е по-малък от първият елемент на следващия сортиран събмасив, то пропускаме сортировката* **return**;  
 }  
 **int** left = begin;  
 **int** right = mid + 1;  
  
 **int**[] helper = **new int**[arr.**length**];  
 **for** (**int** i = begin; i <= end; i++) {  
 helper[i] = arr[i];  
 }  
  
 **for** (**int** i = begin; i <= end; i++) {  
 **if** (left > mid) { *//when 1st substring is over* arr[i] = helper[right++]; *//we take next element from the sorted 2nd substring* } **else if** (right > end) { *//when 2nd substring is over* arr[i] = helper[left++]; *//we take next element from the sorted 1st substring* } **else if** (helper[left] < helper[right]) { *//when the element of 1st substring is lower than the element of 2nd substing* arr[i] = helper[left++]; *//arr[i] is with new value helper[left]* } **else** {  
 arr[i] = helper[right++]; *//arr[i] is with new value helper[right]* }  
 }  
 }

##### II. Quick Sort

Best & average case: **O(n\*log(n))**; Worst: **O(n2)**

The algorithm in short:

* Quicksort takes unsorted partitions of an array and sorts them
* We choose the **pivot**
  + We pick the first element from the unsorted partition and move it in such a way, that all smaller elements are on its left and all greater, to its right
* With pivot moved to its correct place, we now have two unsorted partitions – one to the left of it and one to the right
* **Call the procedure recursively** for each partition

In Lomuto partition scheme, the starting **pivot** is the last element of an array / the **last (high)** element

The bottom of the recursion is when a partition has a size of 1, which is by definition sorted

**public class** Qucksort {  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
 **int**[] arr = Arrays.*stream*(sc.nextLine().split(**"\\s+"**)).mapToInt(x -> Integer.*parseInt*(x)).toArray();  
 *quickSort*(arr, 0, arr.**length** - 1);  
  
 StringBuilder builder = **new** StringBuilder();  
 **for** (**int** num : arr) {  
 builder.append(num).append(**" "**);  
 }  
 System.***out***.println(builder.toString());  
 }  
  
 *// low is the start index  
 // high is the end index* **private static void** quickSort(**int**[] arr, **int** low, **int** high) {  
 **if** (low < high) {  
 **int** pi = *partition*(arr, low, high);  
  
 *// Recursively sort elements before partition and after partition  
 quickSort*(arr, low, pi - 1);  
 *quickSort*(arr, pi + 1, high);  
 }  
 }  
  
 */\*This method takes last element as pivot, places the pivot at its correct position in sorted array,  
 and places all smaller (smaller than pivot) to left of pivot, and all greater elements to right of pivot  
 \*/* **private static int** partition(**int**[] arr, **int** low, **int** high) {  
 **int** pivot = arr[high];  
 **int** i = (low - 1); *// index of the smaller element* **for** (**int** j = low; j < high; j++) {  
 *//If current element is smaller or equal to pivot* **if** (arr[j] <= pivot) {  
 i++;  
 *swap*(arr, i, j);  
 }  
 }  
  
 *swap*(arr, i + 1, high);  
  
 **return** i + 1;  
 }  
  
 **private static void** swap(**int**[] arr, **int** first, **int** second) {  
 **int** temp = arr[first];  
 arr[first] = arr[second];  
 arr[second] = temp;  
 }  
  
}

##### III. Counting Sort - sorting without comparison, using counts only - very efficient and stable – iterative algorithm

Sorts small integers by counting their occurrences

Създаваме масив counts с дължина най-големият елемент от arr. И всеки елемент на масива counts е 0 в началото. Реално масива counts e сортиран по подразбиране възходящо(индекси 0, 1, 2….). На индекс 2 има 2 броя, на индекс 5 има 2 броя, и на идекс 13 има 1 брой.

**public class** SortWithoutComparison {  
 **public static int**[] *counts*;  
  
 **public static void** main(String[] args) {  
 **int**[] arr = {13, 5, 2, 2, 5};  
 **int** max = Arrays.*stream*(arr).max().getAsInt();  
  
 *counts* = **new int**[max + 1];  
 *sort*(arr);  
  
 **for** (**int** index = 0; index < *counts*.**length**; index++) {  
 **if** (*counts*[index] != 0) {  
 **for** (**int** i = 0; i < *counts*[index]; i++) {  
 System.***out***.print(index + **" "**);  
 }  
 }  
 }  
 }  
  
 **private static void** sort(**int**[] arr) {  
 **for** (**int** i = 0; i < arr.**length**; i++) {  
 *counts*[arr[i]]++;  
 }  
 }  
  
}

##### IV. Bucket Sort

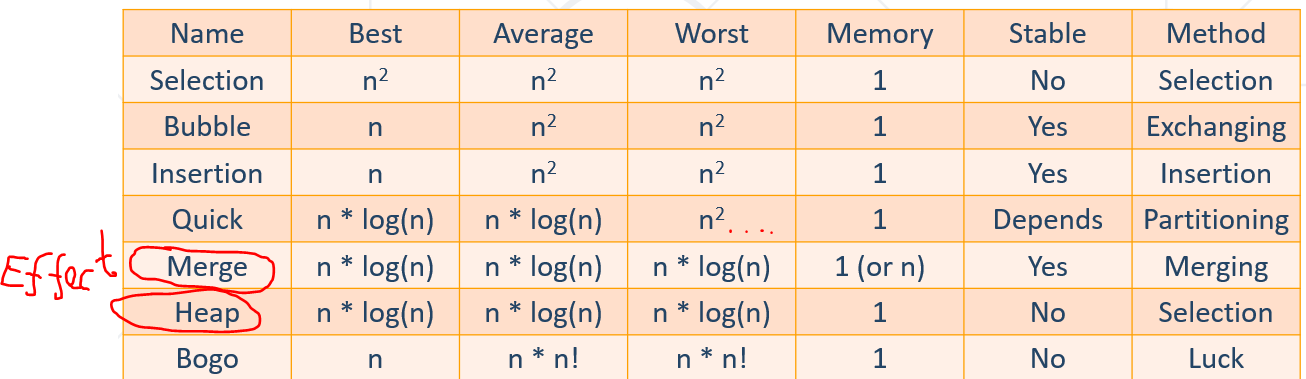
**Bucket sort** partitions an array into a number of buckets:

* Each bucket is then sorted individually with a different algorithm
* Not a comparison-based sort

##### V. Heap Sort

##### VI. Bogo Sort – прави безброй много пермутации, докато видите ли елементите не се окаже, че са се подредили сами

#### 6.2.4. Summary



Counting Sort is also high effective/efficient

### 6.3. Greedy Algorithms

Usually more efficient than the other algorithms

Pick the best local solution

Greedy algorithms assume that always choosing a local optimum leads to the global optimum **– което не винаги е верно**

Examples:

* Find the **shortest** path from Sofia to Varna
* Find the **maximum** **increasing** **subsequence**
* Find the shortest route that visits each city and returns to the origin city

**Greedy Failure Cases -** Greedy Algorithms Often Fail, when all local optimal results do not give the optimal global maximum!

**When we can use Greedy?**

* **Greedy choice property -** A global optimal solution can be obtained by greedily selecting a locally optimal choice

Случаят когато имаме монета със стойност 4, то 18 постигаме с 3 стъпки: 10 +4 +4.

А иначе постигаме 18 с 5 стъпки: 10+5+1+1+1

* **Optimal substructure -** An optimal global solution contains the optimal solutions of all its sub-problems

Any problem having the above properties (**Greedy choice property & Optimal substructure**) is guaranteed to have an optimal greedy solution

Интересен сайт за това кой алгоритъм къде се използва най-добре:

<https://stackoverflow.com/questions/1933759/when-is-each-sorting-algorithm-used/1934004#1934004>

## 7. Graph Theory, Traversal and Shortest Paths

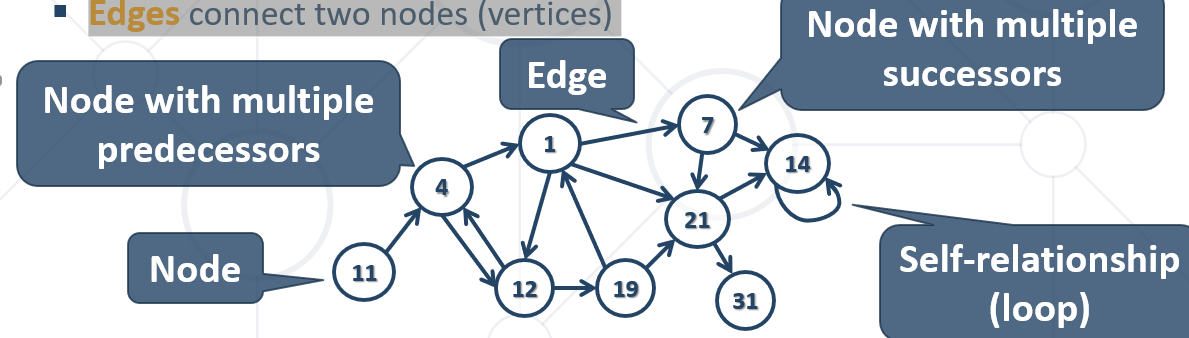
### 7.1. Graphs

**Graph,** denoted as **G(Vertice, Edge):**

* Node / Vertice – върхове / връх
* Edges – ребра/стрелки свързващи върховете

Each **node** (**vertex**) has **multiple** predecessors(предшественици) and **multiple** successors(наследници)

**Edges** connect two nodes (vertices)

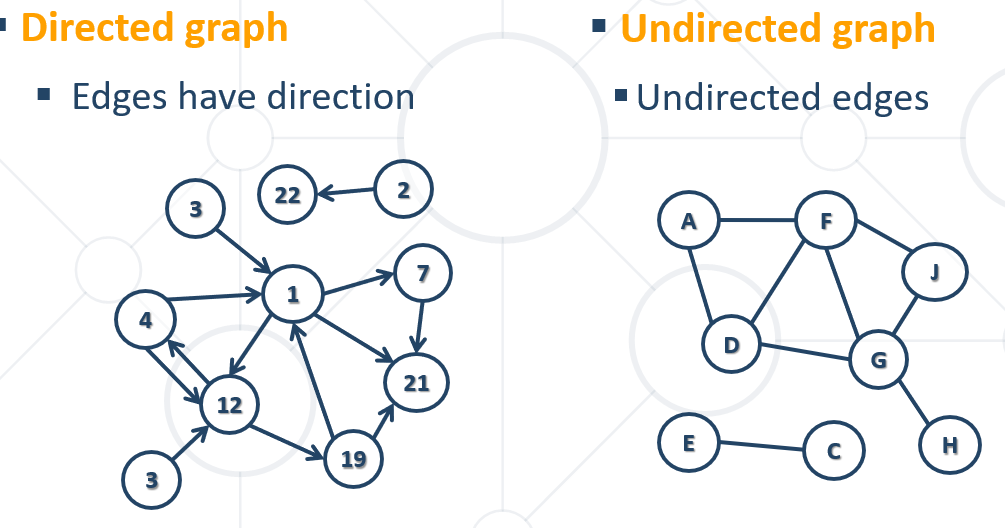


**Node (vertex):**

* + Element of a graph
  + Can have name / value
  + Keeps a list of adjacent nodes

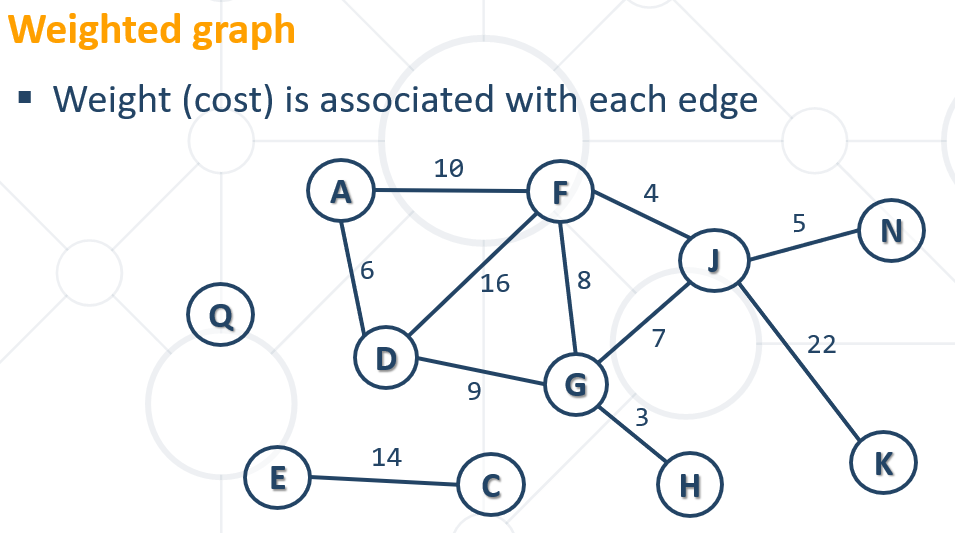
**Edge:**

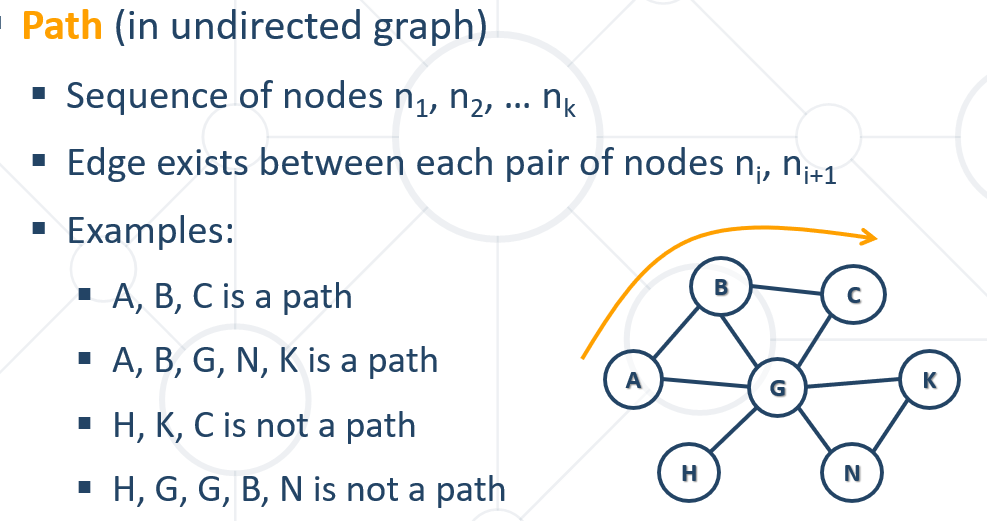
* + Connection between two nodes
  + Can be directed / undirected
  + Can be weighted / unweighted
  + Can have name / value

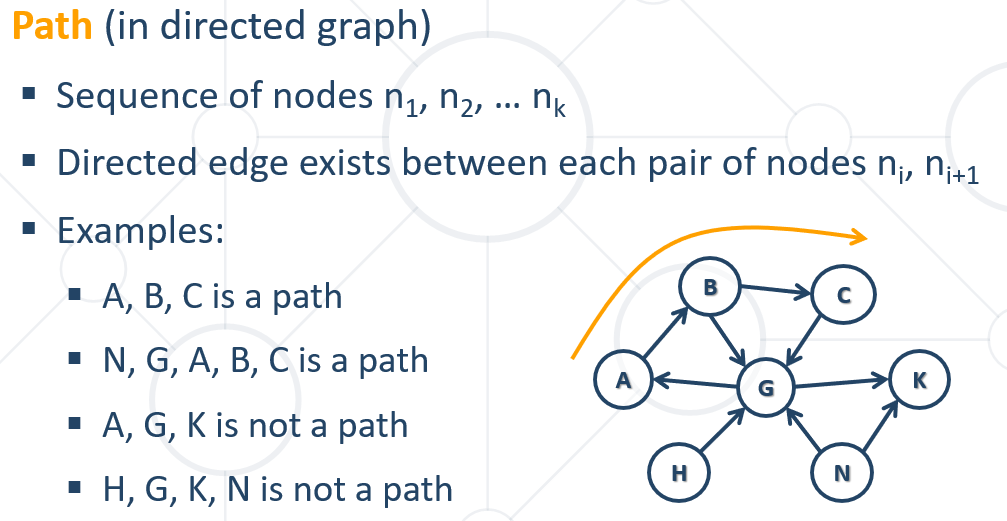


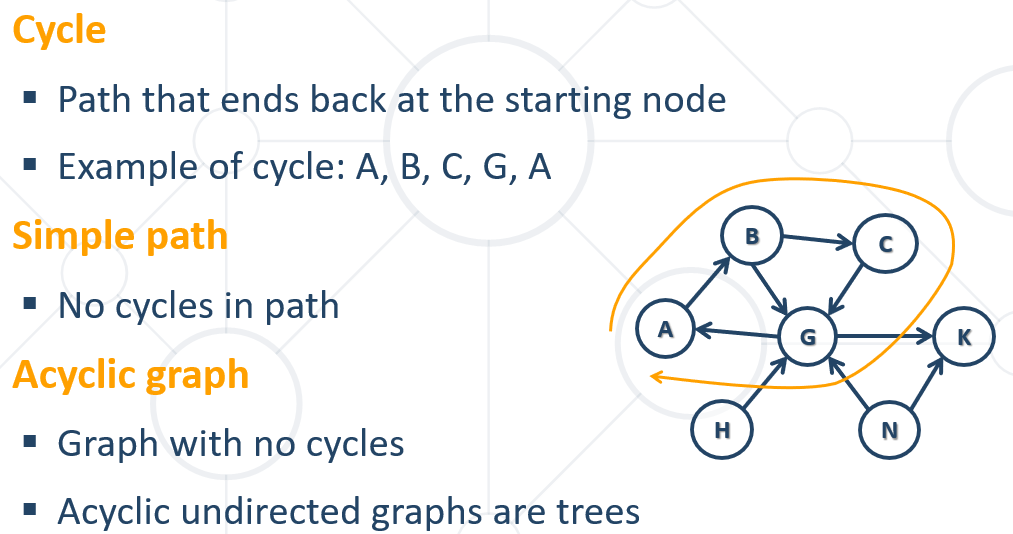
Идеята на графите е да не се връщаме на вече посетено място/Node.

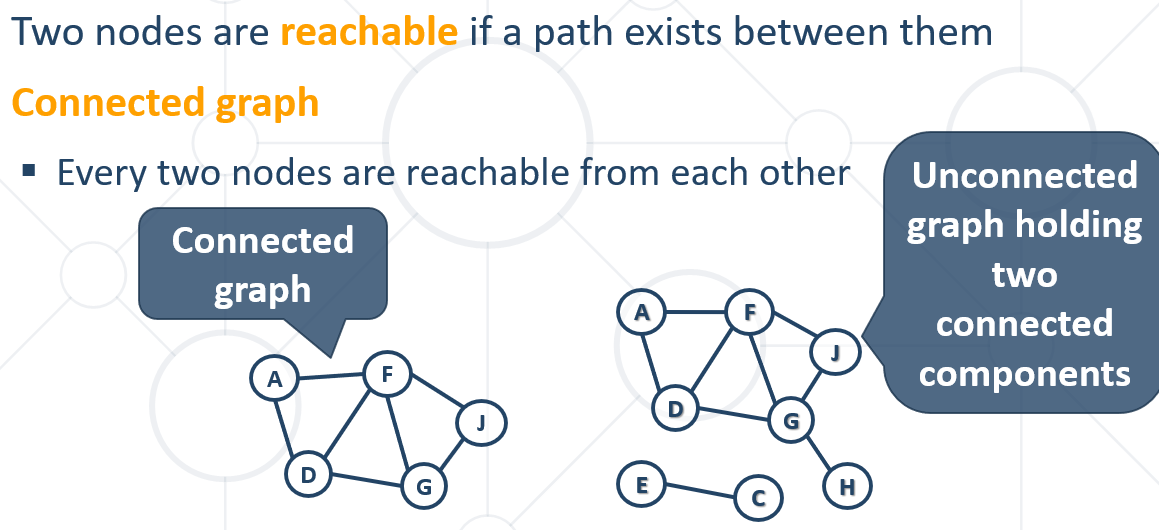
В този курс, ще работим само с непретеглени графи.











### 7.2. Representing graphs

При графи, много често пропускаме нулевият елемент, затова добавяме +1 за броя на елементите

#### Adjacency list- Each node holds a list of its neighbors

List<List<Integer>> graph = **new** ArrayList<>();  
**for** (**int** i = 0; i < 10 + 1; i++) {  
 graph.add(**new** ArrayList<>());  
}  
  
graph.get(1).addAll(Arrays.*asList*(9, 8, 5));  
graph.get(9).add(1); // от 1 ходим към 9 и от 9 ходим към 1 – това е неподреден граф, има ненасочено ребро

#### Adjacency Matrix – матрица на съседство

**int** nodes = 10;  
**int**[][] graph = **new int**[nodes + 1][nodes + 1];  
  
graph[3][6] = 1;

**int[][] graph = new int[][] {  
 // 0 1 2 3 4 5 6  
 { 0, 0, 0, 1, 0, 0, 1 }, // node 0  
 { 0, 0, 1, 1, 1, 1, 1 }, // node 1  
 { 0, 1, 0, 0, 1, 1, 0 }, // node 2  
 { 1, 1, 0, 0, 0, 1, 0 }, // node 3  
 { 0, 1, 1, 0, 0, 0, 1 }, // node 4  
 { 0, 1, 1, 1, 0, 0, 0 }, // node 5  
 { 1, 1, 0, 0, 1, 0, 0 }, // node 6  
};  
// Add an edge { 3 -> 6 }  
 graph[3][6] = 1;  
// List the children of node #1  
int[] childNodes = graph[1];**

#### List of edges

{1,2}, {1,4}, {2,3}, {3,1}, {4,2}

#### Representing with a Class

**public static class** Edge {  
 **public int source**;  
 **public int destination**;  
  
 **public** Edge(**int** source, **int** destination) {  
 **this**.**source** = source;  
 **this**.**destination** = destination;  
 }  
}  
  
**public static void** main(String[] args) {  
 List<Edge> graph = **new** ArrayList<>();  
 graph.add(**new** Edge(1, 2));  
 graph.add(**new** Edge(1, 3));  
 graph.add(**new** Edge(1, 4));  
 graph.add(**new** Edge(1, 5));  
 graph.add(**new** Edge(1, 6));  
}

ИЛИ

**public static class** Graph {  
 **int source**;  
 List<Edge> **edges**;  
  
 **public** Graph(**int** source) {  
 **this**.**source** = source;  
 **this**.**edges** = **new** ArrayList<>();  
 }  
}  
  
**public static class** Edge {  
 **public int source**;  
 **public int destination**;  
  
 **public** Edge(**int** source, **int** destination) {  
 **this**.**source** = source;  
 **this**.**destination** = destination;  
 }  
}  
  
**public static void** main(String[] args) {  
 Graph graph = **new** Graph(1);  
 graph.**edges**.add(**new** Edge(1, 2));  
 graph.**edges**.add(**new** Edge(1, 3));  
 graph.**edges**.add(**new** Edge(1, 4));  
}

*// List the children of node #1*List<Edge> childNodes = graph.stream().filter(e -> e.source == 1).collect(Collectors.*toList*());

#### Матрица на теглото – като има ребро, то не е 1-ца, а има стойност/тежест

**int**[][] graph = **new int**[2 + 1][2 + 1];  
graph[1][2] = 12; //тежест 12

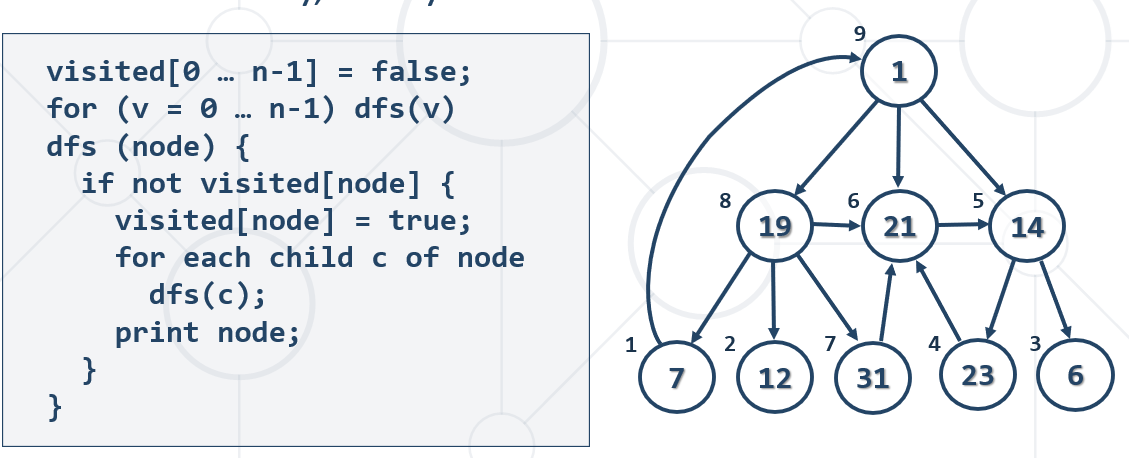
### 7.3. Graphs Traversals

Traversing a graph means to visit each of its nodes exactly once.

The order of visiting nodes may vary on the traversal algorithm

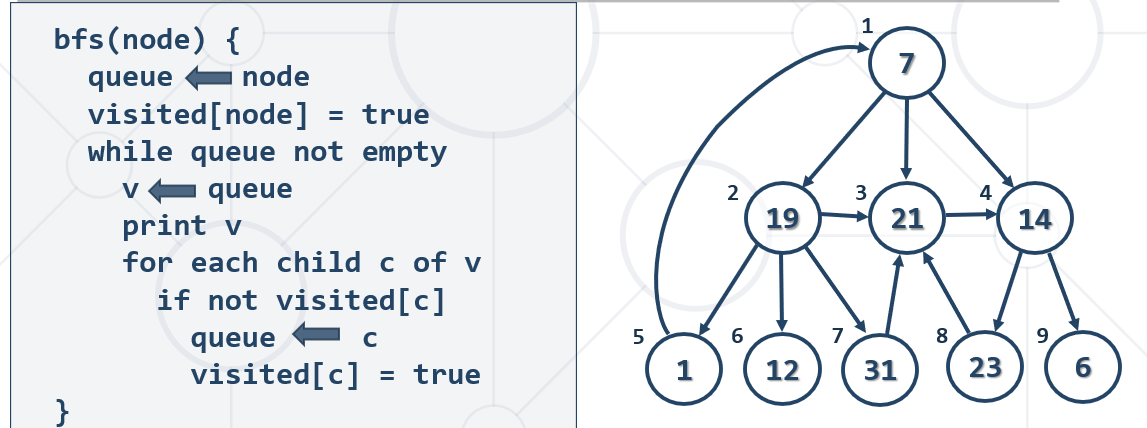
#### Depth-First Search (DFS) – на нива дълбочина

first visits all descendants of given node recursively, finally visits the node itself

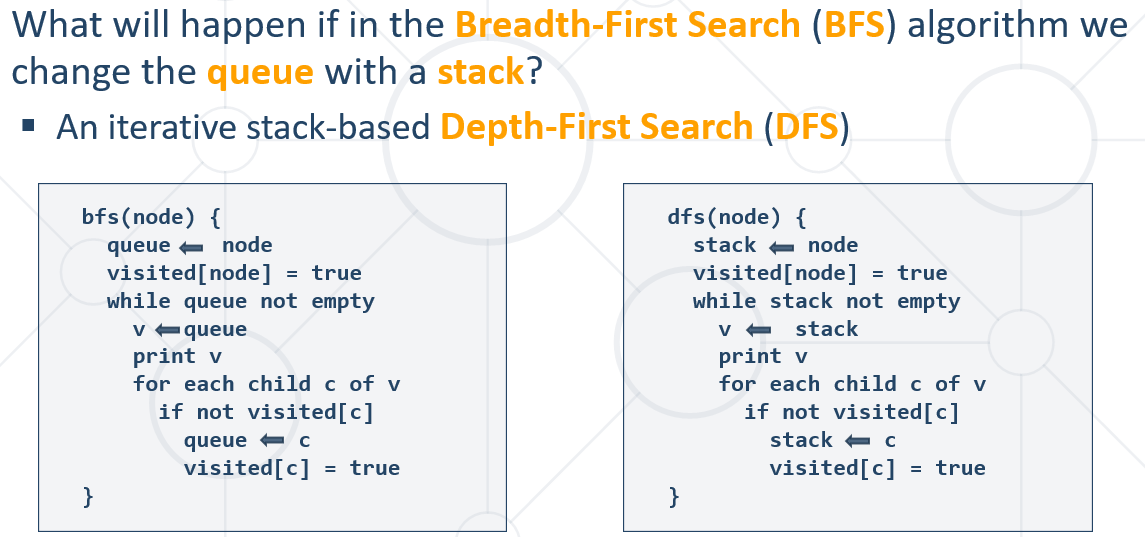


#### Breadth-First Search (BFS) – на вълни

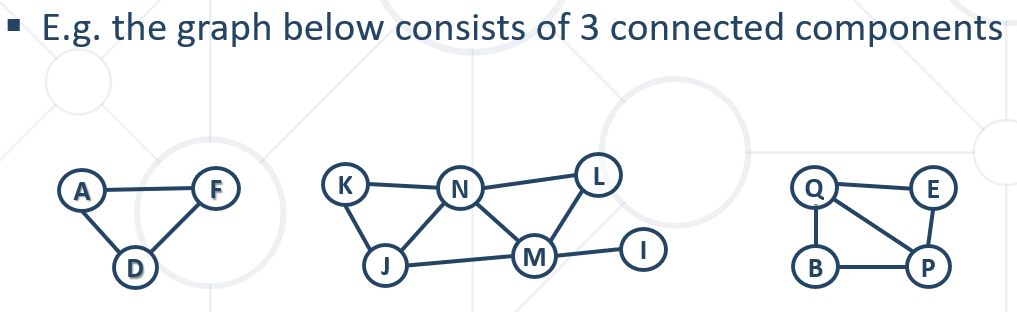
first visits the neighbor nodes, then the neighbors of neighbors, then their neighbors, etc.



#### Iterative DFS and BFS



### 7.4. Graph Connectivity



Finding the connected components - Loop through all nodes and start a **DFS** / **BFS** traversing from any **unvisited** node

Всички свързани компоненти на даден граф – **DFS:**

**public class** Main {  
  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
  
 **int** n = Integer.*parseInt*(sc.nextLine());  
 List<List<Integer>> graph = **new** ArrayList<>();  
  
 **for** (**int** i = 0; i < n; i++) {  
 String nextLine = sc.nextLine();  
 **if** (nextLine.trim().equals(**""**)) {  
 graph.add(**new** ArrayList<>());  
 } **else** {  
 List<Integer> nextNodes = Arrays.*stream*(nextLine.split(**"\\s+"**))  
 .map(Integer::*parseInt*)  
 .collect(Collectors.*toList*());  
  
 graph.add(nextNodes);  
 }  
 }  
  
 List<Deque<Integer>> connectedComponents = *getConnectedComponents*(graph);  
 System.***out***.println();  
 }  
  
 **public static** List<Deque<Integer>> getConnectedComponents(List<List<Integer>> graph) {  
 **boolean**[] visited = **new boolean**[graph.size()];  
 List<Deque<Integer>> components = **new** ArrayList<>();  
  
 **for** (**int** start = 0; start < graph.size(); start++) {  
 **if** (!visited[start]) {  
 *dfs*(start, components, graph, visited);  
 System.***out***.println();  
 }  
 }  
 *dfs*(0, components, graph, visited);  
  
 **return** components;  
 }  
  
 **private static void** dfs(**int** node, List<Deque<Integer>> components, List<List<Integer>> graph, **boolean**[] visited) {  
 **if** (!visited[node]) {  
 visited[node] = **true**;  
 **for** (**int** child : graph.get(node)) {  
 *dfs*(child, components, graph, visited);  
 }  
 System.***out***.print(node + **" "**);  
 }  
 }

}

Всички свързани компоненти на даден граф – **BFS:**  
**public class** ConnectedComponentsBFS {  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
  
 **int** n = Integer.*parseInt*(sc.nextLine());  
 List<List<Integer>> graph = **new** ArrayList<>();  
  
 **for** (**int** i = 0; i < n; i++) {  
 String nextLine = sc.nextLine();  
 **if** (nextLine.trim().equals(**""**)) {  
 graph.add(**new** ArrayList<>());  
 } **else** {  
 List<Integer> nextNodes = Arrays.*stream*(nextLine.split(**"\\s+"**))  
 .map(Integer::*parseInt*)  
 .collect(Collectors.*toList*());  
  
 graph.add(nextNodes);  
 }  
 }  
  
 List<Deque<Integer>> connectedComponents = *getConnectedComponents*(graph);  
 **for** (Deque<Integer> connectedComponent : connectedComponents) {  
 System.***out***.print(**"Connected component: "**);  
 **for** (**int** intNum : connectedComponent) {  
 System.***out***.print(intNum + **" "**);  
 }  
 System.***out***.println();  
 }  
 }  
  
 **public static** List<Deque<Integer>> getConnectedComponents(List<List<Integer>> graph) {  
 **boolean**[] visited = **new boolean**[graph.size()];  
 List<Deque<Integer>> components = **new** ArrayList<>();  
  
 **for** (**int** start = 0; start < graph.size(); start++) {  
 **if** (!visited[start]) {  
 components.add(**new** ArrayDeque<>());  
  
 *bfs*(start, components, graph, visited);  
 }  
 }  
  
 **return** components;  
 }  
  
 **private static void** bfs(**int** start, List<Deque<Integer>> components, List<List<Integer>> graph, **boolean**[] visited) {  
 Deque<Integer> queue = **new** ArrayDeque<>();  
 visited[start] = **true**;  
 queue.offer(start);  
  
 **while** (!queue.isEmpty()) {  
 **int** node = queue.poll();  
  
 components.get(components.size() - 1).offer(node);  
  
 **for** (**int** child : graph.get(node)) {  
 **if** (!visited[child]) {  
 visited[child] = **true**;  
 queue.offer(child);  
 }  
 }  
 }  
 }  
}

### 7.5. Topological Sorting

Ordering a Graph by Set of Dependencies

Rules:

* + Undirected graphs cannot be sorted
  + Graphs with cycles cannot be sorted
  + Sorting is not unique
  + Various sorting algorithms exists and they give different results

#### 7.5.1. Source removal top-sort algorithm – без рекурсия

Source removal top-sort algorithm + cycle detect

Create an empty list

Repeat until the graph is empty:

* + - Find a node without incoming edges
    - Print this node
    - Remove the edge from the graph

**L ← empty list that will hold the sorted elements (output)**

**S ← set of all nodes with no incoming edges**

**while S is non-empty do**

**remove some node n from S**

**append n to L**

**for each node m with an edge e: { n through m }**

**remove edge e from the graph**

**if m has no other incoming edges then**

**insert m into S**

**if graph is empty**

**return L (a topologically sorted order)**

**else**

**return "Error: graph has at least one cycle"**

**public static** Collection<String> topSort(Map<String, List<String>> graph) {  
 Map<String, Integer> dependenciesCount = *getDependenciesCount*(graph);  
  
 List<String> sorted = **new** ArrayList<>();  
  
 **while** (!graph.isEmpty()) {  
 String key = graph.keySet()  
 .stream()  
 .filter(k -> dependenciesCount.get(k) == 0)  
 .findFirst()  
 .orElse(**null**);  
  
 **if** (key == **null**) {  
 **break**;  
 }  
  
 **for** (String child : graph.get(key)) {  
 dependenciesCount.put(child, dependenciesCount.get(child) - 1);  
 }  
 graph.remove(key);  
 sorted.add(key);  
 }  
  
 **if** (!graph.isEmpty()) { // детектване за цикличност  
 **throw new** IllegalArgumentException();  
 }  
  
 **return** sorted;  
}  
  
**private static** Map<String, Integer> getDependenciesCount(Map<String, List<String>> graph) {  
 Map<String, Integer> dependenciesCount = **new** LinkedHashMap<>();  
  
 **for** (Map.Entry<String, List<String>> node : graph.entrySet()) {  
 dependenciesCount.putIfAbsent(node.getKey(), 0);  
 **for** (String child : node.getValue()) {  
 dependenciesCount.putIfAbsent(child, 0);  
 dependenciesCount.put(child, dependenciesCount.get(child) + 1);  
 }  
 }  
  
 **return** dependenciesCount;  
}

#### 7.5.2. DFS Algorithm – с рекурсия

**sortedNodes = { } // linked list to hold the result**

**visitedNodes = { } // set of already visited nodes**

**foreach node in graphNodes**

**topSortDFS(node)**

**topSortDFS(node)**

**if node ∉ visitedNodes**

**visitedNodes ← node**

**for each child c of node**

**TopSortDFS(c)**

**insert node upfront in the sortedNodes**

**public static** Collection<String> topSort(Map<String, List<String>> graph) {  
 List<String> sorted = **new** ArrayList<>();  
  
 Set<String> visited = **new** HashSet<>();  
  
 **for** (Map.Entry<String, List<String>> node : graph.entrySet()) {  
 *dfs*(node.getKey(), visited,graph, sorted);  
 }  
  
 Collections.*reverse*(sorted);  
  
 **return** sorted;  
}  
  
**public static void** dfs(String key, Set<String> visited, Map<String, List<String>> graph, List<String> sorted){  
 **if** (!visited.contains(key)) {  
 visited.add(key);  
 **for** (String child : graph.get(key)) {  
 **if** (!visited.contains(child)) {  
 *dfs*(child, visited, graph, sorted);  
 }  
 }  
 sorted.add(key);  
 }  
}

#### 7.5.3. DFS Algorithm + Cycle Detection

**sortedNodes = { } // linked list to hold the result**

**visitedNodes = { } // set of already visited nodes**

**cycleNodes = { } // set of nodes in the current DFS cycle**

**foreach node in graphNodes**

**topSortDFS(node)**

**topSortDFS(node)**

**if node ϵ cycleNodes**

**return "Error: cycle detected"**

**if node ∉ visitedNodes**

**visitedNodes ← node**

**cycleNodes ← node**

**for each child c of node**

**topSortDFS(c)**

**remove node from cycleNodes**

**insert node upfront in the sortedNodes**

**public static** Collection<String> topSort(Map<String, List<String>> graph) {  
 List<String> sorted = **new** ArrayList<>();  
  
 Set<String> visited = **new** HashSet<>();  
 Set<String> detectCycles = **new** HashSet<>(); // детектване за цикличност  
  
 **for** (Map.Entry<String, List<String>> node : graph.entrySet()) {  
 *dfs*(node.getKey(), visited, graph, sorted, detectCycles);  
 }  
  
 Collections.*reverse*(sorted);  
  
 **return** sorted;  
}  
  
**public static void** dfs(String key, Set<String> visited, Map<String, List<String>> graph, List<String> sorted, Set<String> detectCycles) {  
 **if** (detectCycles.contains(key)) { // детектване за цикличност  
 **throw new** IllegalArgumentException();  
 }  
 **if** (!visited.contains(key)) {  
 visited.add(key);  
 detectCycles.add(key); // детектване за цикличност  
 **for** (String child : graph.get(key)) {  
 *dfs*(child, visited, graph, sorted, detectCycles);  
 }  
 detectCycles.remove(key); // детектване за цикличност  
 sorted.add(key);  
 }  
}

### 7.6. Shortest Path

In **unweighted** graphs finding the **shortest** **path** can be done with **BFS** (all edges ha**v**e the same weight)

**bfs(G, start, end)**

**visited[start] = true**

**queue.offer(start)**

**while (!queue.isEmpty())**

**v = queue.poll()**

**if v is end**

**return v**

**for all edges from v to w in G.adjacentEdges(v) do**

**if w is not labeled as discovered then**

**label w as discovered**

**w.parent = v**

**queue.offer(w)**

## 8. Dynamic Programming == Динамично оптимиране

* "**Controlled**" brute force / exhaustive search – от всички обхождания, по някакъв начин избираме добрите brute force
* **Subproblems**: like original problem, but smaller
* Write solution to one **subproblem** in terms of solutions to smaller **acyclic** subproblems – избягваме чикличност в подпроблема!
* **Memoization**: remember the **solution** to subproblems we’ve already solved, and **re‐use - Avoid exponentials**
* **Guessing**: if you don’t know something, **guess** **it!** (try all possibilities)

### 8.1. Fibonacci Sequence

Recursive mathematical formula:

**F0** = **0**, **F1** = **1**

**Fn** = **Fn-1** + **Fn-2**

**public class** Fibonacci {  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
  
 **int** n = Integer.*parseInt*(sc.nextLine());  
  
 **long** fib = *calcFib*(n);  
  
 System.***out***.println(fib);  
 }  
  
 **private static long** calcFib(**int** n) {  
 **if** (n <= 2) {  
 **return** 1;  
 }  
  
 **return** *calcFib*(n-1) + *calcFib*(n-2);  
 }  
}

**Memoization**

* DP 🡪 sub-problems **overlap**
* In order to **avoid solving** problems **multiple times**, memorize
  + **Memoization** 🡪 **save/cache** sub-problem solutions **for later use**
* Typically using an **array**, **matrix** or a **hash table**
* Рекурсивно решение - **Top down** approach - Solve **recursively** by **breaking down** the problem further and further

**public class** Fibonacci {  
 **public static long**[] *dp*;  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
  
 **int** n = Integer.*parseInt*(sc.nextLine());  
 *dp* = **new long**[n+1];  
  
 **long** fib = *calcFib*(n);  
  
 System.***out***.println(fib);  
 }  
  
 **private static long** calcFib(**int** n) {  
 **if** (n <= 2) {  
 **return** 1;  
 }  
  
 **if** (*dp*[n] != 0) {  
 **return** *dp*[n];  
 }  
  
 **return** *dp*[n] = *calcFib*(n-1) + *calcFib*(n-2);  
 }  
}

* Итеративно решение - **Bottom up** approach

**public class** Fibonacci {  
 **public static long**[] *dp*;  
  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
  
 **int** n = Integer.*parseInt*(sc.nextLine());  
 *dp* = **new long**[n + 1];  
  
 *dp*[1] = 1;  
 *dp*[2] = 1;  
  
 **for** (**int** i = 3; i <= n; i++) {  
 *dp*[i] = *dp*[i - 1] + *dp*[i - 2];  
 }  
  
 System.***out***.println(*dp*[n]);  
 }  
}

### 8.2. Five "Easy" Steps to DP

**D**efine **subproblems**

**Guess** part of the solution

**Relate** subprolems and solutions

**Recurse** and **memoization** or build **DP table** bottom-up

**Check** subproblems **acyclic/topological** order

**Solve** original problem:

* + A **subproblem**
  + Or **combination** of subproblems

**Useful Subproblems** for Sequences – използваме или suffixes или prefixes, но не и двете едновременно

* **Suffixes** x[i… n-1] – от първия до последния - наставка
* **Prefixes** x[n-1… i] – от последния до първия – представка
* Both approaches usually run in **ϴ(x)**

**Substrings** (subsequences) x[i…j] - два вложени цикъла

* + Usually runs in **ϴ(x^2)**

### 8.3. Longest Increasing Subsequence

LIS Optimal Substructure - Recursive Top-Down Approach

…….. – как става с рекурсия

LIS - Iterative Bottom-Up Approach – **left-most or right-most**

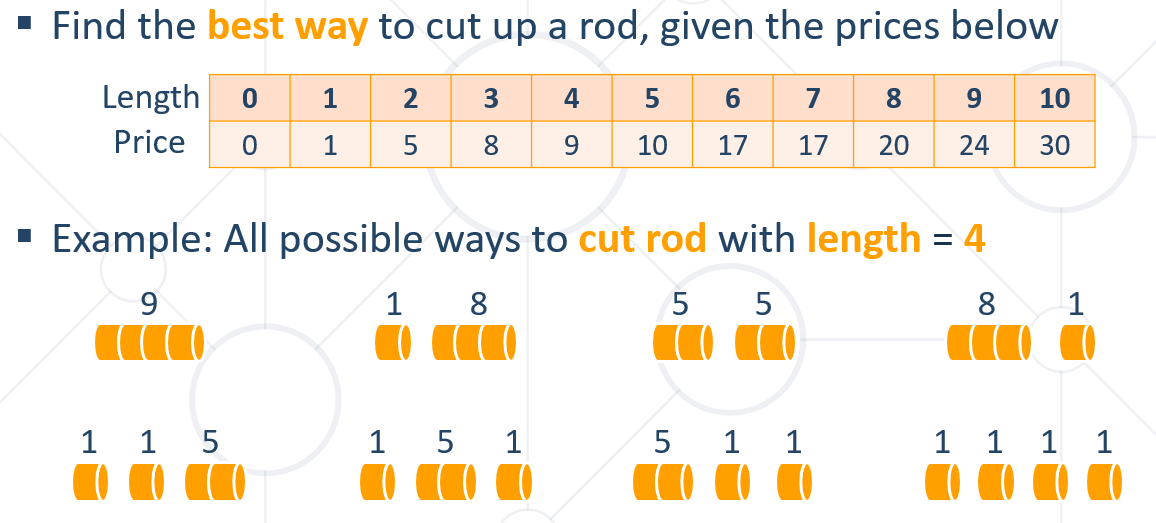
|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| S[] | **3** | **14** | **5** | **12** | **15** | **7** | **8** | **9** | **11** | **10** | **1** |
| len[] | 1 | 2 | 2 | 3 | 4 | 3 | 4 | 5 | 6 | 6 | 1 |
| prevIndex[] | -1 | 0 | 0 | 2 | 3 | 2 | 5 | 6 | 7 | 7 | -1 |
| LIS | {3} | {3,14} | {3,5} | {3,5,12} | {3,5,12,15} | {3,5,7} | {3,5,7,8} | {3,5,7,8,9} | {3,5,7,8,9,11} | {3,5,7,8,9,10} | {1} |

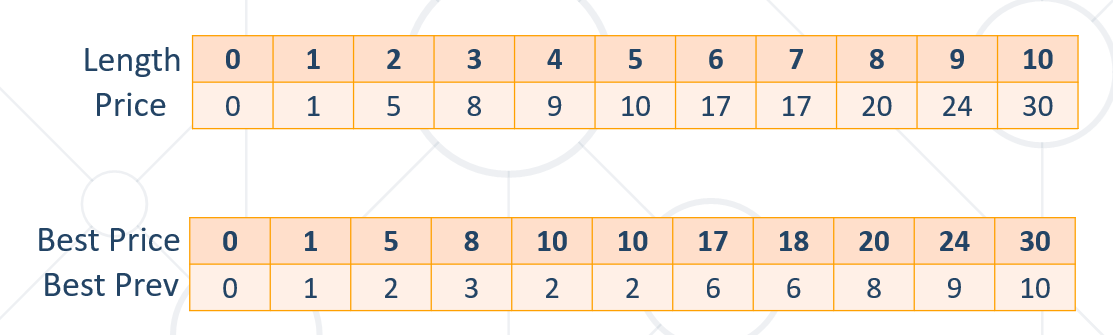
**public class** LIS\_iterative {  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
  
 **int**[] sequence = Arrays.*stream*(sc.nextLine().split(**"\\s+"**)).mapToInt(Integer::*parseInt*).toArray();  
 **int**[] length = **new int**[sequence.**length**];  
 **int**[] prevIndex = **new int**[sequence.**length**];  
 Arrays.*fill*(prevIndex, -1);  
  
 **int** maxLength = 0, maxIndex = -1;  
  
 **for** (**int** i = 0; i < sequence.**length**; i++) {  
 **int** current = sequence[i];  
 **int** bestLength = 1;  
 **int** bestIndex = -1;  
  
 **for** (**int** j = i - 1; j >= 0; j--) {  
 **if** (sequence[j] < current && length[j] + 1 >**=** bestLength) { **//= дали е leftmost или rightmost** bestLength = length[j] + 1;  
 bestIndex = j;  
 }  
 }  
  
 prevIndex[i] = bestIndex;  
 length[i] = bestLength;  
 **if** (maxLength < bestLength) {  
 maxLength = bestLength;  
 maxIndex = i;  
 }  
 }  
  
 List<Integer> LIS = **new** ArrayList<>();  
  
 **int** index = maxIndex;  
 **while** (index != -1) {  
 LIS.add(sequence[index]);  
 index = prevIndex[index];  
 }  
  
 **for** (**int** i = LIS.size() - 1; i >= 0 ; i--) {  
 System.***out***.print(LIS.get(i) + **" "**);  
 }  
 }  
}

### 8.4. Move Down/Right Sum – iterative approach, има опция за разписване и с рекурсия

**public class** MoveDown\_Right {  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
  
 **int** rows = Integer.*parseInt*(sc.nextLine());  
 **int** cols = Integer.*parseInt*(sc.nextLine());  
  
 **int**[][] elements = **new int**[rows][cols];  
  
 **for** (**int** row = 0; row < rows; row++) {  
 elements[row] = Arrays.*stream*(sc.nextLine().split(**"\\s+"**))  
 .mapToInt(Integer::*parseInt*)  
 .toArray();  
 }  
  
 **int**[][] dpTable = **new int**[rows][cols];  
  
 dpTable[0][0] = elements[0][0];  
  
 **for** (**int** col = 1; col < cols; col++) {  
 dpTable[0][col] = dpTable[0][col - 1] + elements[0][col];  
 }  
  
 **for** (**int** row = 1; row < rows; row++) {  
 dpTable[row][0] = dpTable[row - 1][0] + elements[row][0];  
 }  
  
 **for** (**int** row = 1; row < rows; row++) {  
 **for** (**int** col = 1; col < cols; col++) {  
 dpTable[row][col] = Math.*max*(dpTable[row - 1][col] + elements[row][col],  
 dpTable[row][col - 1] + elements[row][col]);  
 }  
 }  
  
 **int** row = rows - 1;  
 **int** col = cols - 1;  
  
 List<String> path = **new** ArrayList<>();  
  
 path.add(*formatOutput*(row, col));  
 **while** (row > 0 || col > 0) {  
 **int** top = -1;  
 **if** (row > 0) {  
 top = dpTable[row - 1][col];  
 }  
  
 **int** left = -1;  
 **if** (col > 0) {  
 left = dpTable[row][col - 1];  
 }  
  
 **if** (top > left) {  
 row--;  
 } **else** {  
 col--;  
 }  
 path.add(*formatOutput*(row, col));  
 }  
  
 Collections.*reverse*(path);  
 System.***out***.println(String.*join*(**" "**, path));  
 }  
  
 **private static** String formatOutput(**int** row, **int** col) {  
 **return "["** + row + **", "** + col + **"]"**;  
 }  
}

### 8.5. Rod Cutting Problem





Recursive solution

**public class** RodCutting {  
 **public static int**[] *bestPrices*;  
 **public static int**[] *prevIndex*;  
 **public static int**[] *prices*;  
  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
  
 *prices* = Arrays.*stream*(sc.nextLine().split(**"\\s+"**))  
 .mapToInt(Integer::*parseInt*)  
 .toArray();  
 **int** length = Integer.*parseInt*(sc.nextLine());  
 *bestPrices* = **new int**[length + 1];  
 *prevIndex* = **new int**[length + 1];  
  
 **int** maxProfit = *cutRope*(length);  
 System.***out***.println(maxProfit);  
 *reconstructSolution*(length);  
 }  
  
 **private static int** cutRope(**int** length) {  
 **if** (length == 0) {  
 **return** 0;  
 }  
 **if** (*bestPrices*[length] != 0) {  
 **return** *bestPrices*[length];  
 }  
 **int** currentBest = *bestPrices*[length];  
  
 **for** (**int** i = 1; i <= length; i++) {  
 currentBest = Math.*max*(currentBest, *prices*[i] + *cutRope*(length - i));  
 **if** (currentBest > *bestPrices*[length]) {  
 *bestPrices*[length] = currentBest;  
 *prevIndex*[length] = i;  
 }  
 }  
  
 **return** *bestPrices*[length];  
 }  
  
 **private static void** reconstructSolution(**int** n) {  
 **while** (n - *prevIndex*[n] != 0) {  
 System.***out***.print(*prevIndex*[n] + **" "**);  
 n = n - *prevIndex*[n];  
 }  
 System.***out***.println(*prevIndex*[n]);  
 }  
}

Iterative solution

**private static int cutRod(int n) {  
 for (int i = 1; i <= n; i++) {  
 int currentBest;  
 for (int j = 1; j <= i; j++) {  
 currentBest =  
 Math.max(bestPrice[i], price[j] + bestPrice[i - j]);  
 if (currentBest > bestPrice[i]) {  
 bestPrice[i] = currentBest;  
 bestCombo[i] = j;  
 }  
 }  
 }  
 return bestPrice[n];  
}**