Podstawa programowa – technik programista klasa 1

Table of Contents

[1. Setup development environment. 3](#_Toc171956930)

[2. Conversion DEC/HEX/OCT/BIN including Horner's scheme. 4](#_Toc171956931)

[2.1 Conversion of a decimal number to a hexadecimal number 4](#_Toc171956932)

[2.2 Conversion of a hexadecimal number to a decimal number 5DE0 4](#_Toc171956933)

[2.3. Horner's scheme used for number conversion 4](#_Toc171956934)

[3. Making Change (Cashier's Problem) 5](#_Toc171956935)

[4. Prime numbers / perfect numbers 5](#_Toc171956936)

[Point 5 Explanation: 5](#_Toc171956937)

[Why is this needed? 5](#_Toc171956938)

[Example: Number 36 6](#_Toc171956939)

[Example: Number 25 (Perfect Square) 6](#_Toc171956940)

[5. Prime factorization 7](#_Toc171956941)

[Definition: 7](#_Toc171956942)

[Steps for Prime Factorization: 7](#_Toc171956943)

[Example: Prime Factorization of 84 8](#_Toc171956944)

[Another Example: Prime Factorization of 120 8](#_Toc171956945)

[General Tips: 8](#_Toc171956946)

[6. Greatest Common Divisor (GCD), Least Common Multiple (LCM) 9](#_Toc171956947)

[7. Solving a Diophantine Equation using extended Euclidean Algorithm 10](#_Toc171956948)

[8. Sieve of Eratosthenes 11](#_Toc171956949)

[Steps of the Sieve of Eratosthenes 11](#_Toc171956950)

[Example: Finding Primes Up to 30 11](#_Toc171956951)

[9. Fast Exponentiation (Exponentiation by Squaring) 12](#_Toc171956952)

[10. Finding Roots - Newton-Raphson Method 13](#_Toc171956953)

[11. Finding the Minimum and Maximum Elements in a Set 13](#_Toc171956954)

[Steps of the Algorithm 13](#_Toc171956955)

[12. Triangle Inequality Theorem 13](#_Toc171956956)

[The Triangle Inequality Theorem 13](#_Toc171956957)

[Why the Triangle Inequality Theorem is Important 14](#_Toc171956958)

[Example 14](#_Toc171956959)

[13. Finding the Leader in a Set Using the Tournament Method 14](#_Toc171956960)

[14. Searching with a Sentinel 14](#_Toc171956961)

[How It Works 14](#_Toc171956962)

[Steps to Implement the Sentinel Search 15](#_Toc171956963)

[Example: Searching for a Value in an Array 15](#_Toc171956964)

[15. Finding the Most Frequent Element in a Set 15](#_Toc171956965)

[Steps to Find the Most Frequent Element 15](#_Toc171956966)

[Example: Finding the Most Frequent Element in a Set 15](#_Toc171956967)

[16. Finding the k-th Largest Element in a Set 16](#_Toc171956968)

[Method 1: Sorting 16](#_Toc171956969)

[Example: Finding the 3rd Largest Element 16](#_Toc171956970)

[Method 2: Min-Heap 16](#_Toc171956971)

[17. Binary Search 16](#_Toc171956972)

[Steps of Binary Search 16](#_Toc171956973)

[Recursive Binary Search in C++ 17](#_Toc171956974)

[18. Pattern searching (or string searching) 17](#_Toc171956975)

[Brute-Force Approach 17](#_Toc171956976)

[Knuth-Morris-Pratt (KMP) Algorithm 17](#_Toc171956977)

[19. Longest Common Subsequence, LCS 18](#_Toc171956978)

[20. Generating anagrams of a string 18](#_Toc171956979)

[21. Power set 18](#_Toc171956980)

[22. Creating k-element combinations (subsets of k elements) without repetition from an n-element set 18](#_Toc171956981)

[23. A minimum spanning tree (MST) (Prim’s, Kruskal's Algorithm) 19](#_Toc171956982)

[24. The Travelling Salesman Problem (TSP) (Heuristic Approaches) 19](#_Toc171956983)

[25. A palindrome 19](#_Toc171956984)

[26. Generating sequences and finding their general terms (including the Fibonacci sequence) 19](#_Toc171956985)

[Fibonacci Sequence 19](#_Toc171956986)

[Recursive Approach 20](#_Toc171956987)

[Iterative Approach 20](#_Toc171956988)

[27. Bubble sort 20](#_Toc171956989)

[Bubble Sort Characteristics 20](#_Toc171956990)

[Bubble Sort Optimization 21](#_Toc171956991)

[28. Selection sort 21](#_Toc171956992)

[Selection Sort Characteristics 21](#_Toc171956993)

[29. Insertion sort 21](#_Toc171956994)

[Insertion Sort Characteristics 22](#_Toc171956995)

[30. Merge sort 22](#_Toc171956996)

[Merge Sort Characteristics 22](#_Toc171956997)

[31. Quick sort 22](#_Toc171956998)

[Complexity Analysis of Quick Sort : 23](#_Toc171956999)

[Advantages of Quick Sort: 23](#_Toc171957000)

[Disadvantages of Quick Sort: 23](#_Toc171957001)

[32. Counting sort 24](#_Toc171957002)

[Counting Sort Characteristics 24](#_Toc171957003)

[33. Odwrotna notacja polska (ONP) Shunting Yard’s algorithm 24](#_Toc171957004)

[34. Binary-coded decimal (BCD) 25](#_Toc171957005)

[35. Gray Code 25](#_Toc171957006)

[36. Transposition cipher 25](#_Toc171957007)

[37. Substitution cipher (Caesar and Vernam cipher) 25](#_Toc171957008)

[38. Public-key cryptography 26](#_Toc171957009)

[39. Dwupodział zbioru 26](#_Toc171957010)

[40. Generating subsets 26](#_Toc171957011)

[41. Johnson’s Rule in Sequencing Problems 26](#_Toc171957012)

[42. Additional content 27](#_Toc171957013)

# Setup development environment.

VS Code or Visual Studio Community

1.1 VS Code

<https://www.youtube.com/watch?v=DMWD7wfhgNY>

download

<https://code.visualstudio.com/docs/?dv=win64user>

follow this instruction:

<https://code.visualstudio.com/docs/cpp/config-mingw>

# Conversion DEC/HEX/OCT/BIN including Horner's scheme.

## 2.1 Conversion of a decimal number to a hexadecimal number

Convert the decimal number 24032 to a hexadecimal number.

24032 / 16 = 1502; remainder = 0

1502 / 16 = 93; remainder = 14 or E

93 / 16 = 5; remainder = 13 or D

5 / 16 = 0; remainder = 5

Reading the remainders from the bottom up, we get the hexadecimal number: 5DE0

## 2.2 Conversion of a hexadecimal number to a decimal number 5DE0

5\**16^3 + 13\**16^2 + 14\*16^1 + 0

Exercise: write a program to convert HEX -> DEC,

consider if the program is universal and if it can be easily modified to work for other bases/number systems (e.g., OCT/BIN). DEC -> HEX HEX -> DEC

## 2.3. Horner's scheme used for number conversion

In Polish

<https://www.youtube.com/watch?v=EY5PnPlWnR0&ab_channel=MaturaInformatyka-Ma%C5%82gorzataPiekarska>

In English

<https://www.youtube.com/watch?v=qDG311jci_0&ab_channel=AhmedAlmansor>

<https://eduinf.waw.pl/inf/alg/006_bin/0003.php>

# Making Change (Cashier's Problem)

In Polish

<https://eduinf.waw.pl/inf/utils/021_2021/1004.php>

in English

<https://www.geeksforgeeks.org/greedy-algorithm-to-find-minimum-number-of-coins/>

# Prime numbers / perfect numbers

Prime numbers

<https://www.geeksforgeeks.org/c-program-to-check-prime-number/>

hint -> sqrt(n)

<https://stackoverflow.com/questions/4424374/determining-if-a-number-is-prime>

Perfect numbers

<https://prepinsta.com/cpp-program/cpp-program-to-check-whether-a-number-is-perfect-number-or-not/>

Badanie pierwszości liczby

<https://home.agh.edu.pl/~zobmat/2021/rzepka_radoslaw/algorytmy.html>

Liczby doskonałe

<https://www.algorytm.edu.pl/algorytmy-maturalne/liczby-doskonale.html>

### Point 5 Explanation:

if(n == p \* p) s -= p;

#### What this does:

If n is a perfect square, the divisor p (which is the square root of n) will be added twice in the loop. To correct this, we subtract p once from the sum s.

### Why is this needed?

When calculating the sum of the divisors of n, each divisor pair (i.e., i and n/i) is added to the sum. For example, for a number like 28:

* Divisors are 1, 2, 4, 7, 14, 28.
* Proper divisors are 1, 2, 4, 7, 14.
* Sum of proper divisors: 1 + 2 + 4 + 7 + 14 = 28 (perfect number).

But if the number is a perfect square, like 36:

* Divisors are 1, 2, 3, 4, 6, 9, 12, 18, 36.
* Proper divisors are 1, 2, 3, 4, 6, 9, 12, 18.
* Pairs of divisors: (1, 36), (2, 18), (3, 12), (4, 9), (6, 6).

Notice here that 6 is a divisor that pairs with itself because 6\*6 = 36. In the loop, 6 would be added twice, so we need to correct this by subtracting it once.

### Example: Number 36

#### Step-by-Step:

1. n = 36
2. Calculate p = sqrt(36) = 6
3. Initialize s = 1 (since 1 is a divisor of all numbers)

#### Loop through possible divisors:

* For i = 2:
  + 36 % 2 == 0 (True), so add 2 + 36/2 = 2 + 18 = 20 to s
  + s = 1 + 20 = 21
* For i = 3:
  + 36 % 3 == 0 (True), so add 3 + 36/3 = 3 + 12 = 15 to s
  + s = 21 + 15 = 36
* For i = 4:
  + 36 % 4 == 0 (True), so add 4 + 36/4 = 4 + 9 = 13 to s
  + s = 36 + 13 = 49
* For i = 5:
  + 36 % 5 == 0 (False), nothing added
* For i = 6:
  + 36 % 6 == 0 (True), so add 6 + 36/6 = 6 + 6 = 12 to s
  + s = 49 + 12 = 61

At this point, we've added the square root divisor 6 twice.

#### Adjust for perfect square:

* Since 36 == 6\*6, subtract one occurrence of 6 from s
  + s = 61 - 6 = 55

### Example: Number 25 (Perfect Square)

1. n = 25
2. Calculate p = sqrt(25) = 5
3. Initialize s = 1

#### Loop through possible divisors:

* For i = 2:
  + 25 % 2 == 0 (False), nothing added
* For i = 3:
  + 25 % 3 == 0 (False), nothing added
* For i = 4:
  + 25 % 4 == 0 (False), nothing added
* For i = 5:
  + 25 % 5 == 0 (True), so add 5 + 25/5 = 5 + 5 = 10 to s
  + s = 1 + 10 = 11

At this point, we've added the square root divisor 5 twice.

#### Adjust for perfect square:

* Since 25 == 5\*5, subtract one occurrence of 5 from s
  + s = 11 - 5 = 6

Now, we correctly sum up the divisors:

* Proper divisors of 25: 1, 5
* Correct sum: 1 + 5 = 6

#### Conclusion:

For perfect squares, the square root is counted twice in the divisor pairs, so we must subtract it once to correct the sum. This adjustment ensures the algorithm accurately determines if n is a perfect number.

# Prime factorization

The prime factorization of a number involves breaking it down into its smallest prime number components. Here’s a detailed explanation and example:

### Definition:

**Prime Factorization**: The process of determining the prime numbers that multiply together to give the original number.

### Steps for Prime Factorization:

1. **Start with the smallest prime number (2)**.
2. **Divide the number by the prime number**.
3. **Continue dividing** until the result is no longer divisible by that prime number.
4. **Move to the next prime number (3, 5, 7, etc.)** and repeat the process.
5. **Stop when the result is 1**.

### Example: Prime Factorization of 84

#### Step-by-Step Process:

1. **Divide by 2 (the smallest prime number)**:
   * 84÷2=4284 \div 2 = 4284÷2=42
   * 84 is divisible by 2, so 2 is a factor.
2. **Continue dividing by 2**:
   * 42÷2=2142 \div 2 = 2142÷2=21
   * 42 is still divisible by 2, so 2 is again a factor.
3. **Move to the next prime number (3)**:
   * 21÷3=721 \div 3 = 721÷3=7
   * 21 is divisible by 3, so 3 is a factor.
4. **Move to the next prime number (7)**:
   * 7÷7=17 \div 7 = 17÷7=1
   * 7 is divisible by 7, so 7 is a factor.

#### Result:

* Prime factors of 84 are: 2×2×3×72 \times 2 \times 3 \times 72×2×3×7
* In exponential form: 22×3×72^2 \times 3 \times 722×3×7

### Another Example: Prime Factorization of 120

1. **Divide by 2**:
   * 120÷2=60
2. **Continue dividing by 2**:
   * 60÷2=3060 \div 2 = 3060÷2=30
3. **Continue dividing by 2**:
   * 30÷2=1530 \div 2 = 1530÷2=15
4. **Move to the next prime number (3)**:
   * 15÷3=515 \div 3 = 515÷3=5
5. **Move to the next prime number (5)**:
   * 5÷5=15 \div 5 = 15÷5=1

#### Result:

* Prime factors of 120 are: 2×2×2×3×52 \times 2 \times 2 \times 3 \times 52×2×2×3×5
* In exponential form: 23×3×52^3 \times 3 \times 523×3×5

### General Tips:

* **Always start with the smallest prime number (2)**.
* **Check divisibility**: If the number is even, start with 2. If the number ends in 0 or 5, it's divisible by 5, etc.
* **Proceed to the next prime number** if the current prime no longer divides the number.
* **Prime factorization ends** when the quotient is 1.

Prime factorization is fundamental in number theory and has applications in cryptography, computer science, and mathematics.

# Greatest Common Divisor (GCD), Least Common Multiple (LCM)

The greatest common divisor of two integers a and b is the largest integer that divides both of them without leaving a remainder.

The Euclidean algorithm for finding GCD is based on iterative division:

* Repeat the division of the larger number by the smaller number until the remainder is zero.
* The GCD of two numbers is the last non-zero remainder.

function GCD(a, b):

while b ≠ 0:

r = a % b

a = b

b = r

return a

For example, GCD(24, 36) follows these steps:

36 is greater than 24, so divide: 36 mod 24 = 12

Now 24 is greater than 12, so divide again: 24 mod 12 = 0

The last non-zero remainder is 12, so GCD (24, 36) = 12.

The least common multiple of two integers a and b is the smallest integer that is divisible by both of them.

LCM can be calculated using the relationship with GCD:

LCM(a,b) = ∣a⋅b∣ / GCD(a,b)

The Euclidean algorithm can also be used to calculate LCM efficiently:

For example, LCM(24, 36):

* + Calculate GCD(24, 36), which is 12.
  + LCM(24, 36) = ∣24⋅36∣ / 12 = 864 / 12 = 72

The Euclidean algorithm is computationally efficient and widely used for quickly determining GCD and LCM of two integers.

In Polish

<https://www.algorytm.edu.pl/algorytmy-maturalne/algorytm-eulkidesa.html>

in English

<https://www.freecodecamp.org/news/euclidian-gcd-algorithm-greatest-common-divisor/>

# Solving a Diophantine Equation using extended Euclidean Algorithm

The Extended Euclidean Algorithm is a method to find the greatest common divisor (GCD) of two integers and also to express the GCD as a linear combination of these integers. This is particularly useful for solving Diophantine equations of the form:

ax + by = c

where a, b and c are given integers, and x and y are unknown integers that we need to solve for.

*Extended Euclidean algorithm also finds integer coefficients x and y such that: ax + by = gcd(a, b)*

**Examples:**

***Input:****a = 30, b = 20****Output:****gcd = 10, x = 1, y = -1  
(Note that 30\*1 + 20\*(-1) = 10)*

***Input:****a = 35, b = 15****Output:****gcd = 5, x = 1, y = -2  
(Note that 35\*1 + 15\*(-2) = 5)*

The extended Euclidean algorithm updates the results of gcd(a, b) using the results calculated by the recursive call gcd(b%a, a). Let values of x and y calculated by the recursive call be x1 and y1. x and y are updated using the below expressions.

*ax + by = gcd(a, b)  
gcd(a, b) = gcd(b%a, a)  
gcd(b%a, a) = (b%a)x1+ ay1  
ax + by = (b%a)x1+ ay1  
ax + by = (b – [b/a] \* a)x1+ ay1  
ax + by = a(y1 – [b/a] \* x1) + bx1*

*Comparing LHS and RHS,  
x = y1 – b/a \* x1  
 y = x1*

<https://www.geeksforgeeks.org/euclidean-algorithms-basic-and-extended/>

# Sieve of Eratosthenes

The Sieve of Eratosthenes is a classic algorithm used to find all prime numbers up to a given limit nnn. It works by iteratively marking the multiples of each prime number starting from 2.

### Steps of the Sieve of Eratosthenes

1. **Create a list of consecutive integers** from 2 to n:

[2,3,4,5,…,n]

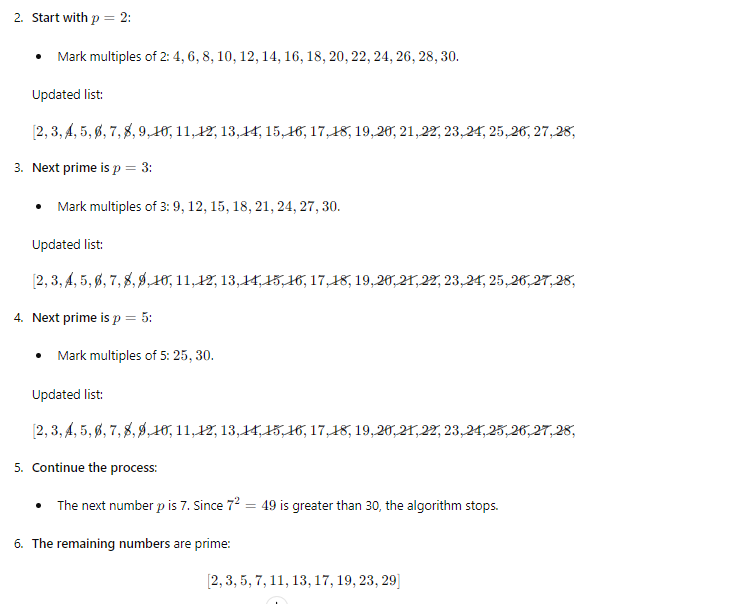
1. **Initialize** the first prime number, p=2.
2. **Mark the multiples of** p as non-prime (starting from p^2), since they are not prime.
3. **Find the next number** in the list that is still prime, set p to this new number, and repeat step 3.
4. **Continue the process** until p^2 is greater than n.
5. **All remaining numbers** in the list that are not marked are prime.

### Example: Finding Primes Up to 30

Let's find all the prime numbers up to n=30

1. **Create the list**:

[2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30]



In English

<https://www.geeksforgeeks.org/sieve-of-eratosthenes/>

In Polish

<https://www.algorytm.edu.pl/algorytmy-maturalne/sito-eratostenesa.html>

# Fast Exponentiation (Exponentiation by Squaring)

In English

<https://discuss.codechef.com/t/a-tutorial-on-fast-modulo-multiplication-exponential-squaring/2899?page=2>

<https://www.geeksforgeeks.org/exponential-squaring-fast-modulo-multiplication/>

<https://www.youtube.com/watch?v=WAzGvZbaAOw&ab_channel=SoftwareEngenius>

in Polish

<https://www.algorytm.edu.pl/algorytmy-maturalne/potegowanie-szybkie.html>

# Finding Roots - Newton-Raphson Method

In Polish

<https://www.algorytm.edu.pl/algorytmy-maturalne/newton-raphson.html>

in English

<https://hackernoon.com/calculating-the-square-root-of-a-number-using-the-newton-raphson-method-a-how-to-guide-yr4e32zo>

# Finding the Minimum and Maximum Elements in a Set

In Polish

<https://eduinf.waw.pl/inf/utils/010_2010/0402.php>

in English

<https://medium.com/enjoy-algorithm/find-maximum-and-minimum-in-an-array-c2049c1411e0>

<https://www.digitalbithub.com/learn/finding-minimum-and-maximum-application-of-divide-and-conquer>

The divide and conquer algorithm for finding the maximum element in an array works by dividing the array into two halves, recursively finding the maximum in each half, and then comparing the two maxima to find the overall maximum. This approach has a time complexity of O(n log n).

### Steps of the Algorithm

1. **Divide**: Split the array into two halves.
2. **Conquer**: Recursively find the maximum element in each half.
3. **Combine**: Compare the maxima of the two halves to determine the overall maximum.

# Triangle Inequality Theorem

The Triangle Inequality Theorem is a fundamental principle in geometry that describes a relationship between the lengths of the sides of a triangle. According to this theorem, for any triangle, the sum of the lengths of any two sides must be greater than or equal to the length of the remaining side.

### The Triangle Inequality Theorem

Given a triangle with sides of lengths a, b, and c:

1. a+b≥c
2. a+c≥b
3. b+c≥a

These inequalities must hold true for the three lengths to form a triangle.

### Why the Triangle Inequality Theorem is Important

* **Validation**: The theorem can be used to determine if three given lengths can form a triangle.
* **Properties**: It ensures that the structure of a triangle is maintained, which is essential in various geometric proofs and constructions.
* **Applications**: It has applications in many areas of mathematics and science, including optimization problems, computer graphics, and the study of distances in metric spaces.

### Example

Let's say we have three lengths: a=3, b=4, and c=5. We can check if these lengths satisfy the Triangle Inequality Theorem.

1. 3+4≥5 (True)
2. 3+5≥4 (True)
3. 4+5≥3 (True)

Since all three conditions are true, the lengths 3, 4, and 5 can form a triangle.

# Finding the Leader in a Set Using the Tournament Method



# Searching with a Sentinel

The sentinel search method is an efficient way to perform a linear search on an array or list. The idea is to add an extra element (sentinel) at the end of the array, which simplifies the search logic by eliminating the need to check for the end of the array within the loop.

### How It Works

1. **Add a Sentinel**: Append the target value as an extra element at the end of the array.
2. **Search Loop**: Traverse the array without checking for the end of the array condition.
3. **Check Result**: If the target value is found before the sentinel, it means the value exists in the original array.

### Steps to Implement the Sentinel Search

1. **Append the Sentinel**: Temporarily add the target value to the end of the array.
2. **Linear Search**: Iterate through the array looking for the target value.
3. **Determine Result**: If the target is found before reaching the appended sentinel, the target is in the array. Otherwise, it is not.

### Example: Searching for a Value in an Array

Consider an array: [3,1,4,1,5,9,2,6] and target value 5.

1. **Initial Array**: [3,1,4,1,5,9,2,6]
2. **Add Sentinel**: [3,1,4,1,5,9,2,6,5] (where 5 is the target)

# Finding the Most Frequent Element in a Set

Finding the most frequent element in a set involves counting the occurrences of each element and identifying the element with the highest count. This can be efficiently done using a hash table or a map to keep track of the frequency of each element.

### Steps to Find the Most Frequent Element

1. **Initialize a Map**: Create a map (or dictionary) to store the frequency of each element.
2. **Count Frequencies**: Iterate through the set and update the frequency of each element in the map.
3. **Identify the Most Frequent Element**: Traverse the map to find the element with the highest frequency.

### Example: Finding the Most Frequent Element in a Set

Consider the set: {3,1,4,1,5,9,2,6,5,3,5}.

1. **Count Frequencies**:
   1. 3: 2 times
   2. 1: 2 times
   3. 4: 1 time
   4. 5: 3 times
   5. 9: 1 time
   6. 2: 1 time
   7. 6: 1 time
2. **Most Frequent Element**: The element with the highest frequency is 5, which appears 3 times.

# Finding the k-th Largest Element in a Set

To find the k-th largest element in a set, there are several approaches, each with different time and space complexities. Here, I'll outline two common methods: sorting and using a min-heap.

### Method 1: Sorting

One of the simplest ways to find the k-th largest element is to sort the set and then index into the sorted list.

1. **Sort the Set**: Sort the elements in descending order.
2. **Indexing**: The k-th largest element will be at index k-1 (0-based index).

### Example: Finding the 3rd Largest Element

Consider the set: {3,1,4,1,5,9,2,6,5,3,5}.

1. **Sort the Set**: {9,6,5,5,5,4,3,3,2,1,1}
2. **3rd Largest Element**: 5 (at index 2 in 0-based indexing)

### Method 2: Min-Heap

Using a min-heap (priority queue) is more efficient for large sets when k is much smaller than the number of elements.

1. **Maintain a Min-Heap**: Keep a min-heap of size k.
2. **Iterate Through Elements**: For each element, if the heap size is less than k, add the element. Otherwise, if the element is larger than the smallest element in the heap, replace the smallest element.
3. **Root of Min-Heap**: After processing all elements, the root of the min-heap is the k-th largest element.

# Binary Search

Binary search is a highly efficient algorithm for finding an element in a sorted array. The basic idea is to repeatedly divide the search interval in half. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

### Steps of Binary Search

1. **Initialize**: Start with two pointers, one pointing to the beginning (low) and the other pointing to the end (high) of the array.
2. **Middle Point**: Calculate the middle point (mid).
3. **Comparison**:
   * If the target value is equal to the middle element, return the middle index.
   * If the target value is less than the middle element, repeat the search on the left half.
   * If the target value is greater than the middle element, repeat the search on the right half.
4. **Repeat**: Continue the process until the target value is found or the search interval is empty.

<https://www.youtube.com/watch?v=MFhxShGxHWc>

### Recursive Binary Search in C++

1. **Function Definition**: Define a recursive function that takes a sorted array, the target value, the low index, and the high index as arguments.
2. **Base Case**: Check if the interval is empty (low > high). If it is, return -1 indicating the target is not found.
3. **Middle Point Calculation**: Calculate the middle index and compare it with the target value.
4. **Recursive Calls**:
   * If the middle element is equal to the target, return the middle index.
   * If the middle element is less than the target, recursively search in the right half.
   * If the middle element is greater than the target, recursively search in the left half.

# Pattern searching (or string searching)

Pattern searching (or string searching) in text is a common problem in computer science where you need to find occurrences of a substring (pattern) within a larger string (text). There are several algorithms for this purpose, ranging from simple brute-force approaches to more efficient algorithms like the Knuth-Morris-Pratt (KMP) algorithm, the Boyer-Moore algorithm, and the Rabin-Karp algorithm.

Below, I'll explain the simple brute-force method and the Knuth-Morris-Pratt (KMP) algorithm.

### Brute-Force Approach

In the brute-force approach, you check every possible position in the text to see if the pattern matches.

### Knuth-Morris-Pratt (KMP) Algorithm

The KMP algorithm improves the efficiency by preprocessing the pattern to determine how far to skip ahead when a mismatch occurs, using a partial match table (also known as the "prefix function" or "failure function").

<https://www.youtube.com/watch?v=V5-7GzOfADQ>

# Longest Common Subsequence, LCS



# Generating anagrams of a string



# Power set



# Creating k-element combinations (subsets of k elements) without repetition from an n-element set



# A minimum spanning tree (MST) (Prim’s, Kruskal's Algorithm)



# The Travelling Salesman Problem (TSP) (Heuristic Approaches)



# A palindrome



# Generating sequences and finding their general terms (including the Fibonacci sequence)

Generating sequences and finding their general terms is a common problem in programming and mathematics. Let's discuss a few examples, including the Fibonacci sequence.

### Fibonacci Sequence

The Fibonacci sequence is defined as:

F(n) = F(n−1) + F(n−2)

with the initial conditions:

F(0)=0,

F(1)=1.

The sequence starts with 0 and 1, and each subsequent term is the sum of the previous two terms.

### Recursive Approach

A straightforward way to generate the Fibonacci sequence is to use recursion. However, this method can be inefficient due to repeated calculations of the same terms.

### Iterative Approach

A more efficient way is to use an iterative approach, which avoids redundant calculations.

<https://www.youtube.com/watch?v=-6O73LQsEmw&ab_channel=MathsGenie>

<https://www.youtube.com/watch?v=WSZN__B9C84&ab_channel=TheMathSorcerer>

The Fibonacci sequence

<https://www.youtube.com/watch?v=ZC-d4dKTyKw&ab_channel=Mathispower4u>

# Bubble sort

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

<https://www.youtube.com/watch?v=xli_FI7CuzA&ab_channel=MichaelSambol>

### Bubble Sort Characteristics

* **Time Complexity**:
  + Best Case: O(n) when the array is already sorted.
  + Average and Worst Case: O(n^2) due to the nested loops.
* **Space Complexity**: O(1) since it is an in-place sorting algorithm.
* **Stability**: Bubble sort is a stable sorting algorithm as it does not change the relative order of equal elements.

### Bubble Sort Optimization

The provided implementation includes an optimization with the swapped flag to terminate early if the array becomes sorted before completing all passes.

Bubble sort is mainly used for educational purposes to introduce the concept of sorting algorithms. For practical purposes, more efficient algorithms like Quick Sort, Merge Sort, or Heap Sort are usually preferred.

# Selection sort

Selection sort is another simple sorting algorithm that divides the input list into two parts: a sorted sublist of items which is built up from left to right at the front (left) of the list and a sublist of the remaining unsorted items that occupy the rest of the list. Initially, the sorted sublist is empty, and the unsorted sublist is the entire input list.

The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element from the unsorted sublist, swapping it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

<https://www.youtube.com/watch?v=g-PGLbMth_g&ab_channel=MichaelSambol>

### Selection Sort Characteristics

* **Time Complexity**:
  + Best, Average, and Worst Case: O(n^2) due to the nested loops.
* **Space Complexity**: O(1) since it is an in-place sorting algorithm.
* **Stability**: Selection sort is not stable as it may change the relative order of equal elements.

Selection sort is simple and easy to understand but is inefficient for large lists compared to more advanced algorithms like Quick Sort or Merge Sort. However, it is still useful for small arrays or for teaching the concept of sorting algorithms.

# Insertion sort

Insertion sort is a simple and intuitive sorting algorithm that builds the final sorted array one element at a time. It works similarly to how you might sort playing cards in your hands: you start with an empty left hand and the cards face down on the table; then you pick up one card at a time from the table and insert it into the correct position in the left hand.

<https://www.youtube.com/watch?v=JU767SDMDvA&ab_channel=MichaelSambol>

### Insertion Sort Characteristics

* **Time Complexity**:
  + Best Case: O(n) when the array is already sorted.
  + Average and Worst Case: O(n^2) due to the nested loops.
* **Space Complexity**: O(1) since it is an in-place sorting algorithm.
* **Stability**: Insertion sort is stable as it does not change the relative order of equal elements.
* **Adaptive**: Efficient for data sets that are already substantially sorted. The complexity is O(n+d) where d is the number of inversions.

Insertion sort is particularly useful for small data sets or for adding a few new elements to an already sorted array. It is easy to implement and understand, making it a good choice for educational purposes and practical applications where simplicity is preferred over efficiency.

# Merge sort

Merge sort is an efficient, stable, and comparison-based sorting algorithm. It works on the divide and conquer principle, dividing the input array into two halves, recursively sorting each half, and then merging the two sorted halves to produce the final sorted array.

<https://www.youtube.com/watch?v=4VqmGXwpLqc&ab_channel=MichaelSambol>

### Merge Sort Characteristics

* **Time Complexity**: O(n log n) for all cases (best, average, and worst) due to the divide and conquer approach.
* **Space Complexity**: O(n) because of the additional temporary arrays used for merging.
* **Stability**: Merge sort is a stable sorting algorithm as it does not change the relative order of equal elements.
* **Divide and Conquer**: Merge sort is based on the divide and conquer paradigm, making it a good choice for parallel processing.

Merge sort is widely used due to its predictable O(n log n) time complexity and stability, making it suitable for sorting linked lists and large datasets.

# Quick sort

Quicksort is a highly efficient and widely used sorting algorithm. It employs a divide-and-conquer strategy to sort elements. The basic idea is to select a "pivot" element from the array and partition the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

<https://www.youtube.com/watch?v=Hoixgm4-P4M&ab_channel=MichaelSambol>

another video is with code at the end, just skip this and try to implement by yourself

<https://www.youtube.com/watch?v=Vtckgz38QHs&ab_channel=BroCode>

## [Complexity Analysis of Quick Sort](https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-quick-sort/):

**Time Complexity:**

* **Best Case**: Ω (N log (N))  
  The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves.  
  In this case, the algorithm will make balanced partitions, leading to efficient Sorting.
* **Average Case: θ ( N log (N))**  
  Quicksort’s average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.
* **Worst Case: O(N2)**  
  The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element. To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort ) to shuffle the element before sorting.
* **Auxiliary Space:**O(1), if we don’t consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make O ( N ).

## **Advantages of Quick Sort:**

* It is a divide-and-conquer algorithm that makes it easier to solve problems.
* It is efficient on large data sets.
* It has a low overhead, as it only requires a small amount of memory to function.

## Disadvantages of Quick Sort:

* It has a worst-case time complexity of O(N 2 ), which occurs when the pivot is chosen poorly.
* It is not a good choice for small data sets.
* It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot’s position (without considering their original positions).

# Counting sort

Counting sort is an integer sorting algorithm that operates by counting the number of occurrences of each distinct element in the input. The count information is then used to place each element in its correct position in the output array. This algorithm is particularly useful when the range of input values is not significantly larger than the number of elements to be sorted.

<https://www.youtube.com/watch?v=0B33As8jPgo&ab_channel=ComExile>

### Counting Sort Characteristics

* **Time Complexity**: O(n+k) where n is the number of elements in the input array and k is the range of the input values.
* **Space Complexity**: O(k) for the count array and O(n) for the output array.
* **Stability**: Counting sort is a stable sorting algorithm as it does not change the relative order of equal elements.
* **Non-comparative**: Counting sort does not compare elements directly but uses counting and indexing.

Counting sort is particularly efficient for sorting integers when the range of values (k) is not significantly larger than the number of elements (n). It is not suitable for large ranges of values or for non-integer data types without modifications.

# Odwrotna notacja polska (ONP) Shunting Yard’s algorithm



# Binary-coded decimal (BCD)



# Gray Code

<https://www.geeksforgeeks.org/what-is-gray-code/>



# Transposition cipher



# Substitution cipher (Caesar and Vernam cipher)



# Public-key cryptography



# Dwupodział zbioru



# Generating subsets

<https://compprog.wordpress.com/2007/10/10/generating-subsets/>



# Johnson’s Rule in Sequencing Problems

In Polish

<https://www.youtube.com/watch?v=HLfxj-e691U&ab_channel=Zrozumto>

in English

<https://www.geeksforgeeks.org/johnsons-rule-in-sequencing-problems/>



# Additional content

<http://www.algorytm.org/>