<https://github.com/seppotk/Datastructures_and_algorithms.git>

<https://www.tellabs.com/>

<https://www.nixu.com/>

<https://github.com/TT00FE39-3001/lecture2>

55d41251

<https://cpp.sh/>

<https://www.youtube.com/watch?v=aGjL7YXI31Q>

# Algorithms Lecture 1 -- Introduction to asymptotic notations

<https://www.youtube.com/watch?v=zWg7U0OEAoE>

# Lecture - 1 Introduction to Data Structures and Algorithms

# Topics

* Review
* Algorithm techniques
  + Brute Force
  + Decrease-and-Conquer
  + Divide-and-Conquer
* Analyses tool(s): Big �

## Part 0: Review

* Course mechanics
  + Activities: Group + Individual (at home),
  + Due before next lecture
  + Approach: We try to present the material in an object-based approach (not OOP), therefore we will not use inheritance etc.s
* Previous lecture:
  + [Abstract Data Types(ADT)](https://en.wikipedia.org/wiki/Abstract_data_type)
  + [Data Structures](https://en.wikipedia.org/wiki/Data_structure)
  + **Array-based** [Stacks](https://www.softwaretestinghelp.com/stack-in-cpp/)
  + **Array-based** [Queues](https://www.softwaretestinghelp.com/queue-in-cpp/)

## Part 1: Brute Force, Decrease-and-Conquer

* Algorithm techniques:
  + [Brute Force](https://www.geeksforgeeks.org/brute-force-approach-and-its-pros-and-cons/): [Linear Search](https://www.softwaretestinghelp.com/searching-algorithms-in-cpp/) of lists
  + [Decrease-and-Conquer](https://www.geeksforgeeks.org/decrease-and-conquer/):
    - [Binary Search](https://www.softwaretestinghelp.com/searching-algorithms-in-cpp/) of sorted lists
    - [Running time of binary search](https://www.khanacademy.org/computing/computer-science/algorithms/binary-search/a/running-time-of-binary-search)
* [Recursion vs Iteration](https://www.geeksforgeeks.org/introduction-to-recursion-data-structure-and-algorithm-tutorials/)
* [Activity 1](https://github.com/TT00FE39-3001/lecture2/blob/main/activity1/README.md)

## Part 2: Analyses tools

* Math review: [Logarithms](https://www.mathsisfun.com/algebra/logarithms.html)
* [Big $O$](https://justin.abrah.ms/computer-science/big-o-notation-explained.html)
* Tools: [Graphing online](https://www.mathway.com/graph)
* [Activity 2](https://github.com/TT00FE39-3001/lecture2/blob/main/activity2/README.md)

## Part 3: Algorithm techniques

* [Divide-and-Conquer](https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm)
* [Activity 3](https://github.com/TT00FE39-3001/lecture2/blob/main/activity3/README.md)

## Misc

* [GitHub workflow](https://github.com/TT00FE39-3001/lecture2/blob/main/github.md)
* [Links](https://github.com/TT00FE39-3001/lecture2/blob/main/links.md)

HOMEWORK

**# Homework**

**## Task 1/3: Reading**

- [Introduction To Searching Algorithms In C++](<https://www.softwaretestinghelp.com/searching-algorithms-in-cpp/>)

> Remember that in many literature, Binary search is referred to use divide-and-conquer technique. However this is not the case, it uses Decrease-and-Conquer.

**## Task 2/3:Videos**

- [Introduction to Recursion (~23min)](<https://youtu.be/B0NtAFf4bvU>)

- [Introduction to Binary Search (~14min)](<https://youtu.be/6ysjqCUv3K4>)

**## Task 3/3: Pre-Lecture**

- [Introduction to Linked Lists (~19min)](<https://youtu.be/WwfhLC16bis>)

**## Recommended**

- <https://www.learncpp.com/>

ANSWERS

**## Task 1/3: Reading**

- [Introduction To Searching Algorithms In C++](<https://www.softwaretestinghelp.com/searching-algorithms-in-cpp/>)

> Remember that in many literature, Binary search is referred to use divide-and-conquer technique. However this is not the case, it uses Decrease-and-Conquer.

#include <iostream>

#include <string>

**using** **namespace** std;

**int** main()

{

**int** myarray[10] = {21,43,23,54,75,13,5,8,25,10};

**int** key,loc;

    cout<<"The input array is"<<endl;

**for**(**int** i=0;i<10;i++){

        cout<<myarray[i]<<" ";

    }

    cout<<endl;

    cout<<"Enter the key to be searched : "; cin>>key;

**for** (**int** i = 0; i< 10; i++)

    {

**if**(myarray[i] == key)

        {

            loc = i+1;

**break**;

        }

**else**

        loc = 0;

    }

**if**(loc != 0)

    {

        cout<<"Key found at position "<<loc<<" in the array";

    }

**else**

    {

        cout<<"Could not find given key in the array";

    }

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\Programs> .\linear\_search

The input array is

21 43 23 54 75 13 5 8 25 10

Enter the key to be searched : 5

Key found at position 7 in the array

#include <iostream>

#include <string>

using namespace std;

int binarySearch(int myarray[], int beg, int end, int key)

{

    int mid;

    if(end >= beg) {

        mid = (beg + end)/2;

        if(myarray[mid] == key)

        {

            return mid+1;

        }

        else if(myarray[mid] < key) {

            return binarySearch(myarray,mid+1,end,key);

        }

        else {

            return binarySearch(myarray,beg,mid-1,key);

        }

    }

    return -1;

}

int main ()

{

    int myarray[10] = {5,8,10,13,21,23,25,43,54,75};

    int key, location=-1;

    cout<<"The input array is"<<endl;

    for(int i=0;i<10;i++){

        cout<<myarray[i]<<" ";

    }

    cout<<endl;

    cout<<"Enter the key that is to be searched:"; cin>>key;

    location = binarySearch(myarray, 0, 9, key);

    if(location != -1)  {

        cout<<"Key found at location "<<location;

    }

    else   {

        cout<<"Requested key not found";

    }

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\Programs> .\binary\_search

The input array is

5 8 10 13 21 23 25 43 54 75

Enter the key that is to be searched:66

Requested key not found

**## Task 2/3:Videos**

- [Introduction to Recursion (~23min)](<https://youtu.be/B0NtAFf4bvU>)

- [Introduction to Binary Search (~14min)](<https://youtu.be/6ysjqCUv3K4>)

Watched these.

**Introduction to Recursion (Data Structures & Algorithms #6)**

579 304 katselukertaa 17.4.2018 [Data Structures and Algorithms](https://www.youtube.com/playlist?list=PLBZBJbE_rGRV8D7XZ08LK6z-4zPoWzu5H)

579 304 katselukertaa • 17.4.2018 • Data Structures and Algorithms

Recursion explained. Java & Python sample code below.

Graphical user interface

Description automatically generated

a problem, by having a function calling itself

0:04 / 22:36

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

return 1; is a base case

return fib(n-1) + fib(n-2); is a recursive case

Graphical user interface, text

Description automatically generated

# Introduction to Binary Search (Data Structures & Algorithms #10)

Graphical user interface, text

Description automatically generated

Graphical user interface, text, chat or text message

Description automatically generated

**## Task 3/3: Pre-Lecture**

- [Introduction to Linked Lists (~19min)](<https://youtu.be/WwfhLC16bis>)

Mainos 2/2 ·

0:04

nibe.eu

Ohita mainos

0:02 / 0:12

**Introduction to Linked Lists (Data Structures & Algorithms #5)**

Graphical user interface, text, application, chat or text message

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text, chat or text message

Description automatically generated

double linked list:

add Node prev

Graphical user interface, text

Description automatically generated

**## Recommended**

- <https://www.learncpp.com/>

ACTIVITY1

**# Activities**

**## Task 1/5**

> Refer to the instruction in [GitHub Workflow](../github.md)

- Clone today's repo

- Create a new branch e.g answers

- Create a repository in GitHub

- Change remote to point to your repo

**## Task 2/5: Brute force**

- What is the maximum number of steps it would take to perform a linear search on an ordered array of size 100,000?

  > Refer to `./src/search-linear.cpp`

**## Task 3/5: Decrease-and-Conquer**

- What is the maximum number of steps it would take to perform a binary search on an ordered array of size 100,000?

  > Refer to `./src/search-binary.cpp`

**## Task 4/5: Quiz**

32 teams qualified for the 2014 World Cup. If the names of the teams were arranged in sorted order (an array), how many items in the array would binary search have to examine to find the location of a particular team in the array, in the worst case?

- [] At most, 32.

- [] At most, 1.

- [] At most, 6.

- [] At most, 16.

**## Task 5/5: Individual, at home**

Refactor the code in `./src/task4.cpp` to use recursion

- Refer to the following [link](<https://www.techiedelight.com/binary-search/>)

- Make sure that you:

  - Replace `printf()` with` std::cout()`

  - Include the right headers e.g. `iostream`

**## Links**

- <https://cpp.sh/>

- <https://www.techiedelight.com/binary-search/>

- <https://www.softwaretestinghelp.com/searching-algorithms-in-cpp/>

- <https://www.khanacademy.org/computing/computer-science/algorithms/binary-search/e/running-time-of-binary-search>

ANSWERS:

**## Task 2/5: Brute force**

- What is the maximum number of steps it would take to perform a linear search on an ordered array of size 100,000?

  > Refer to `./src/search-linear.cpp`

#include <iostream>

#include <string>

using namespace std;

int main()

{

    int myarray[10] = {21, 43, 23, 54, 75, 13, 5, 8, 25, 10};

    int key, loc;

    cout << "The input array is" << endl;

    for (int i = 0; i < 10; i++)

    {

        cout << myarray[i] << " ";

    }

    cout << endl;

    cout << "Enter the key to be searched : ";

    cin >> key;

    for (int i = 0; i < 10; i++)

    {

        if (myarray[i] == key)

        {

            loc = i + 1;

            break;

        }

        else

            loc = 0;

    }

    if (loc != 0)

    {

        cout << "Key found at position " << loc << " in the array";

    }

    else

    {

        cout << "Could not find given key in the array";

    }

}

maximum is 100,000. If linear-search it does not matter if the array is ordered.

**## Task 3/5: Decrease-and-Conquer**

- What is the maximum number of steps it would take to perform a binary search on an ordered array of size 100,000?

  > Refer to `./src/search-binary.cpp`

#include <iostream>

#include <string>

using namespace std;

int binarySearch(int myarray[], int beg, int end, int key)

{

    int mid;

    if (end >= beg)

    {

        mid = (beg + end) / 2;

        if (myarray[mid] == key)

        {

            return mid + 1;

        }

        else if (myarray[mid] < key)

        {

            return binarySearch(myarray, mid + 1, end, key);

        }

        else

        {

            return binarySearch(myarray, beg, mid - 1, key);

        }

    }

    return -1;

}

int main()

{

    int myarray[10] = {5, 8, 10, 13, 21, 23, 25, 43, 54, 75};

    int key, location = -1;

    cout << "The input array is" << endl;

    for (int i = 0; i < 10; i++)

    {

        cout << myarray[i] << " ";

    }

    cout << endl;

    cout << "Enter the key that is to be searched:";

    cin >> key;

    location = binarySearch(myarray, 0, 9, key);

    if (location != -1)

    {

        cout << "Key found at location " << location;

    }

    else

    {

        cout << "Requested key not found";

    }

}

It takes maximum 16 steps. log2100000 = 16.6

**## Task 4/5: Quiz**

32 teams qualified for the 2014 World Cup. If the names of the teams were arranged in sorted order (an array), how many items in the array would binary search have to examine to find the location of a particular team in the array, in the worst case?

- [] At most, 32.

- [] At most, 1.

- [] At most, 6.

- [] At most, 16.

log232 = 5 -> max 6 in the list

**## Task 5/5: Individual, at home**

Refactor the code in `./src/task4.cpp` to use recursion

- Refer to the following [link](<https://www.techiedelight.com/binary-search/>)

- Make sure that you:

  - Replace `printf()` with` std::cout()`

  - Include the right headers e.g. `iostream`

this must be refactored:

#include <stdio.h>

// Iterative implementation of the binary search algorithm to return

// the position of `target` in array `nums` of size `n`

int binarySearch(int nums[], int n, int target)

{

    // search space is nums[low…high]

    int low = 0, high = n - 1;

    // loop till the search space is exhausted

    while (low <= high)

    {

        // find the mid-value in the search space and

        // compares it with the target

        int mid = (low + high) / 2; // overflow can happen

        // int mid = low + (high - low)/2;

        // int mid = high - (high - low)/2;

        // target value is found

        if (target == nums[mid])

        {

            return mid;

        }

        // if the target is less than the middle element, discard all elements

        // in the right search space, including the middle element

        else if (target < nums[mid])

        {

            high = mid - 1;

        }

        // if the target is more than the middle element, discard all elements

        // in the left search space, including the middle element

        else

        {

            low = mid + 1;

        }

    }

    // target doesn't exist in the array

    return -1;

}

int main(void)

{

    int nums[] = {2, 5, 6, 8, 9, 10};

    int target = 5;

    int n = sizeof(nums) / sizeof(nums[0]);

    int index = binarySearch(nums, n, target);

    if (index != -1)

    {

        printf("Element found at index %d", index);

    }

    else

    {

        printf("Element not found in the array");

    }

    return 0;

}

Element found at index 1

after refactoring:

#include <iostream>

#include <string>

using namespace std;

// Recursive implementation of the binary search algorithm to return

// the position of `target` in subarray nums[low…high]

int binarySearch(int nums[], int low, int high, int target)

{

    // Base condition (search space is exhausted)

    if (low > high) {

        return -1;

    }

    // find the mid-value in the search space and

    // compares it with the target

    int mid = (low + high)/2;    // overflow can happen

    // int mid = low + (high - low)/2;

    // Base condition (target value is found)

    if (target == nums[mid]) {

        return mid;

    }

    // discard all elements in the right search space,

    // including the middle element

    else if (target < nums[mid]) {

        return binarySearch(nums, low, mid - 1, target);

    }

    // discard all elements in the left search space,

    // including the middle element

    else {

        return binarySearch(nums, mid + 1, high, target);

    }

}

int main(void)

{

    int nums[] = { 2, 5, 6, 8, 9, 10 };

    int target = 5;

    int n = sizeof(nums)/sizeof(nums[0]);

    int low = 0, high = n - 1;

    int index = binarySearch(nums, low, high, target);

    if (index != -1) {

        cout << "Element found at position " << index;

    }

    else {

        cout << "Element not found in the array";

    }

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture2-main\activity1\src> .\task5

Element found at position 1

**## Links**

- <https://cpp.sh/>

- <https://www.techiedelight.com/binary-search/>

- <https://www.softwaretestinghelp.com/searching-algorithms-in-cpp/>

- <https://www.khanacademy.org/computing/computer-science/algorithms/binary-search/e/running-time-of-binary-search>

Graphical user interface, text, application

Description automatically generated

ACTIVITY 2

**# Activities**

**## Task 1**

Refer to the first [link](#links).

- Why is Algorithm Analysis Important?

- Explain the Big $O$ notation

- Compare `Linear`, `Logarithmic`, `Quadratic` and `Constant` complexities.

**## Task2**

Refer to the first [link](#links).

- Write a simple algorithm in C++ that finds the square of the first item in a list and then prints it on the screen.

- What is the complexity of the algorithm?

**## Task 3**

Refer to the first [link](#links).

- Write a simple program that displays all items in a list to the console.

- What is the complexity of the algorithm?

**## Task 4: : Individual, at home**

Refer to this [pdf](./big\_o.pdf):

- What is the difference between complexity and performance:

- Does complexity affects performance bor is it the other way around?

- Restate the formal definition of big $O$ in plain English

**## Links**

- [Big O Notation and Algorithm Analysis ](<https://stackabuse.com/big-o-notation-and-algorithm-analysis-with-python-examples/>)

- [Visualization](<https://www.cs.usfca.edu/~galles/visualization/Search.html>)

- [Big-O notation explained by a self-taught programmer](<https://justin.abrah.ms/computer-science/big-o-notation-explained.html>)

- [Big-O is easy to calculate, if you know how](<https://justin.abrah.ms/computer-science/how-to-calculate-big-o.html>)

- <https://cpp.sh/>

ANSWERS

**## Task 1**

Refer to the first [link](#links)

.

- Why is Algorithm Analysis Important?

- Explain the Big $O$ notation

- Compare `Linear`, `Logarithmic`, `Quadratic` and `Constant` complexities.

- Why is Algorithm Analysis Important?

You can compare different algorithms

- Explain the Big $O$ notation

Big-O notation is a statistical measure used to describe the complexity of the algorithm.

- Compare `Linear`, `Logarithmic`, `Quadratic` and `Constant` complexities.

best is constant O(1)

next logarithmic O(logN)

next linear O(N)

next quadratic O(N2)

### Introduction

There are usually multiple ways to solve the problem using a computer program. For instance, there are several ways to sort items in an array - you can use [merge sort](https://stackabuse.com/merge-sort-in-python/), [bubble sort](https://stackabuse.com/bubble-sort-in-python/), [insertion sort](https://stackabuse.com/insertion-sort-in-python/), and so on. All of these algorithms have their own pros and cons and the developer's job is to weigh them to be able to choose the best algorithm to use in any use case. In other words, the main question is which algorithm to use to solve a specific problem when there exist multiple solutions to the problem.

Algorithm analysis refers to the analysis of the complexity of different algorithms and finding the most efficient algorithm to solve the problem at hand. Big-O notation is a statistical measure used to describe the complexity of the algorithm.

*In this guide, we'll first take a brief review of algorithm analysis and then take a deeper look at the Big-O notation. We will see how Big-O notation can be used to find algorithm complexity with the help of different Python functions.*

**Note:** Big-O notation is one of the measures used for algorithmic complexity. Some others include Big-Theta and Big-Omega. Big-Omega, Big-Theta and Big-O are intuitively equal to the **best**, **average** and **worst** time complexity an algorithm can achieve. We typically use Big-O as a measure, instead of the other two, because it we can guarantee that an algorithm runs in an acceptable complexity in its worst case, it'll work in the average and best case as well, but not vice versa.

### Why is Algorithm Analysis Important?

To understand why algorithm analysis is important, we will take the help of a simple example. Suppose a manager gives a task to two of his employees to design an algorithm in Python that calculates the factorial of a number entered by the user. The algorithm developed by the first employee looks like this:

def fact(n):  
 product = 1  
 for i in range(n):  
 product = product \* (i+1)  
 return product  
print(fact(5))

Notice that the algorithm simply takes an integer as an argument. Inside the fact() function a variable named product is initialized to 1. A loop executes from 1 to n and during each iteration, the value in the product is multiplied by the number being iterated by the loop and the result is stored in the product variable again. After the loop executes, the product variable will contain the factorial.

Similarly, the second employee also developed an algorithm that calculates the factorial of a number. The second employee used a recursive function to calculate the factorial of the number n:

def fact2(n):  
 if n == 0:  
 return 1  
 else:  
 return n \* fact2(n-1)  
  
print(fact2(5))

The manager has to decide which algorithm to use. To do so, they've decided to choose which algorithm runs faster. One way to do so is by finding the time required to execute the code on the same input.

In the Jupyter notebook, you can use the %timeit literal followed by the function call to find the time taken by the function to execute:

%timeit fact(50)

This will give us:

9 µs ± 405 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

The output says that the algorithm takes 9 microseconds (plus/minus 45 nanoseconds) per loop.

Similarly, we can calculate how much time the second approach takes to execute:

%timeit fact2(50)

This will result in:

15.7 µs ± 427 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

The second algorithm involving recursion takes 15 microseconds (plus/minus 427 nanoseconds).

The execution time shows that the first algorithm is faster compared to the second algorithm involving recursion. When dealing with large inputs, the performance difference can become more significant.

However, execution time is not a good metric to measure the complexity of an algorithm since it depends upon the hardware. A more objective complexity analysis metric for an algorithm is needed. This is where the Big O notation comes to play.

|  |  |
| --- | --- |
| **Name** | **Big O** |
| Constant | O(c) |
| Linear | O(n) |
| Quadratic | O(n²) |
| Cubic | O(n³) |
| Exponential | O(2ⁿ) |
| Logarithmic | O(log(n)) |
| Log Linear | O(nlog(n)) |

You can visualize these functions and compare them:

A picture containing line chart

Description automatically generated

Generally speaking - anything worse than linear is considered a bad complexity (i.e. inefficient) and should be avoided if possible. Linear complexity is okay and usually a necessary evil. Logarithmic is good. Constant is amazing!

**## Task2**

Refer to the first [link](#links).

- Write a simple algorithm in C++ that finds the square of the first item in a list and then prints it on the screen.

- What is the complexity of the algorithm?

#include <iostream>

#include <string>

using namespace std;

#include <cmath>

int main()

{

int nums[] = { 2, 5, 6, 8, 9, 10 };

cout << pow(nums[0],2) << endl;

return 0;

}

complexity is O(1) constant

**## Task 3**

Refer to the first [link](#links).

- Write a simple program that displays all items in a list to the console.

- What is the complexity of the algorithm?

#include <algorithm>

#include <iostream>

#include <list>

int main() {

std::list<int> my\_list = { 12, 5, 10, 9 };

for (int x : my\_list) {

std::cout << x << '\n';

}

}

complexity is O(N)

**## Task 4: : Individual, at home**

Refer to this [pdf](./big\_o.pdf):

- What is the difference between complexity and performance:

- Does complexity affects performance or is it the other way around?

- Restate the formal definition of big $O$ in plain English

answers:

- What is the difference between complexity and performance:

**Big O notation** (with a capital letter O, not a zero), also called **Landau's symbol**, is a

symbolism used in complexity theory, computer science, and mathematics to describe the

asymptotic behavior of functions. Basically, it tells you how fast a function grows or

declines.

How efficient is an algorithm or piece of code? Efficiency covers lots of resources,

including:

· CPU (time) usage

· memory usage

· disk usage

· network usage

All are important but we will mostly talk about time complexity (CPU usage).

Be careful to differentiate between:

1. Performance: how much time/memory/disk/... is actually used when a program is

run. This depends on the machine, compiler, etc. as well as the code.

2. Complexity: how do the resource requirements of a program or algorithm scale,

i.e., what happens as the size of the problem being solved gets larger?

Complexity affects performance but not the other way around.

- Does complexity affects performance or is it the other way around?

Complexity affects performance but not the other way around.

- Restate the formal definition of big $O$ in plain English

**Big *O* notation** is a mathematical notation that describes the [limiting behavior](https://en.wikipedia.org/wiki/Asymptotic_analysis) of a [function](https://en.wikipedia.org/wiki/Function_(mathematics)) when the [argument](https://en.wikipedia.org/wiki/Argument_of_a_function) tends towards a particular value or infinity. Big O is a member of a [family of notations](https://en.wikipedia.org/wiki/Big_O_notation#Related_asymptotic_notations) invented by [Paul Bachmann](https://en.wikipedia.org/wiki/Paul_Gustav_Heinrich_Bachmann),[[1]](https://en.wikipedia.org/wiki/Big_O_notation#cite_note-Bachmann-1) [Edmund Landau](https://en.wikipedia.org/wiki/Edmund_Landau),[[2]](https://en.wikipedia.org/wiki/Big_O_notation#cite_note-Landau-2) and others

We express complexity using **big-O notation**.

For a problem of size N:

\_

\_\_a constant-time algorithm is "order 1": O(1)

\_\_a linear-time algorithm is "order N": O(N)

\_\_a quadratic-time algorithm is "order N squared": O(N2)

Note that the big-O expressions do not have constants or low-order terms. This is

because, when N gets large enough, constants and low-order terms don't matter (a

constant-time algorithm will be faster than a linear-time algorithm, which will be faster

than a quadratic-time algorithm).

Formal definition:

A function T(N) is O(F(N)) if for some constant c and for values of N greater than some

value n0:

T(N) <= c \* F(N)

The idea is that T(N) is the ***exact*** complexity of a procedure/function/algorithm as a

function of the problem size N, and that F(N) is an upper-bound on that complexity (i.e.,

the actual time/space or whatever for a problem of size N will be no worse than F(N)).

**## Links**

- [Big O Notation and Algorithm Analysis ](<https://stackabuse.com/big-o-notation-and-algorithm-analysis-with-python-examples/>)

- [Visualization](<https://www.cs.usfca.edu/~galles/visualization/Search.html>)

- [Big-O notation explained by a self-taught programmer](<https://justin.abrah.ms/computer-science/big-o-notation-explained.html>)

- [Big-O is easy to calculate, if you know how](<https://justin.abrah.ms/computer-science/how-to-calculate-big-o.html>)

- <https://cpp.sh/>

A picture containing table

Description automatically generated

A picture containing calendar

Description automatically generated

ACTIVITY 3

**# Activities**

**## Task 1/3: Divide-and-conquer**

The code in `./src/pow1.cpp` and `./src/pow2.cpp` compute the power `pow(x, n)` using iterative and recursive approaches respectively.

> Refer to the following [link](<https://www.techiedelight.com/power-function-implementation-recursive-iterative/>)

.

- What is the time complexity for both approaches?

**## Task 2/3: Divide-and-conquer vs Decrease and conquer**

- In many literature, Binary search is referred to use divide-and-conquer technique. Discuss in groups whether you agree or disagree and justify your answers. Refer to the the following thread: [Why Binary Search is not a divide and conquer algorithm?](https://stackoverflow.com/questions/8850447/why-is-binary-search-a-divide-and-conquer-algorithm)

**## Task 3/3: Individual, at home**

- Refactor the code in `./src/pow2.cpp` to optimize the time complexity of the recursive approach. Use `./src/pow3.cpp` as a starter.

- Refactor the code in `pow1.cpp` and `pow2.cpp`, as follows:

  - Replace `printf()` with` std::cout()`

  - Include the right headers e.g. `iostream`

**## Links**

- https://www.techiedelight.com/power-function-implementation-recursive-iterative/

- https://cpp.sh/

ANSWERS:

**## Task 1/3: Divide-and-conquer**

The code in `./src/pow1.cpp` and `./src/pow2.cpp` compute the power `pow(x, n)` using iterative and recursive approaches respectively.

> Refer to the following [link](<https://www.techiedelight.com/power-function-implementation-recursive-iterative/>)

.

- What is the time complexity for both approaches?

pow1.cpp

#include <stdio.h>

long power(int x, unsigned n)

{

    // initialize result by 1

    long pow = 1L;

    // multiply `x` exactly `n` times

    for (int i = 0; i < n; i++)

    {

        pow = pow \* x;

    }

    return pow;

}

int main(void)

{

    int x = -2;

    unsigned n = 10;

    printf("pow(%d, %d) = %d", x, n, power(x, n));

    return 0;

}

pow(-2, 10) = 1024

pow 2

#include <stdio.h>

long power(int x, unsigned n)

{

    // base condition

    if (n == 0)

    {

        return 1L;

    }

    if (n & 1)

    { // if `n` is odd

        return x \* power(x, n / 2) \* power(x, n / 2);

    }

    // otherwise, `n` is even

    return power(x, n / 2) \* power(x, n / 2);

}

int main(void)

{

    int x = -2;

    unsigned n = 10;

    printf("pow(%d, %d) = %d", x, n, power(x, n));

    return 0;

}

- What is the time complexity for both approaches?

**Time Complexity:**O(n)

**## Task 2/3: Divide-and-conquer vs Decrease and conquer**

- In many literature, Binary search is referred to use divide-and-conquer technique. Discuss in groups whether you agree or disagree and justify your answers. Refer to the the following thread: [Why Binary Search is not a divide and conquer algorithm?](<https://stackoverflow.com/questions/8850447/why-is-binary-search-a-divide-and-conquer-algorithm>)

Data Structures and Algorithm Analysis in Java (2nd Edition), by Mark Allen Weiss

Says that a D&C algorithm should have **two disjoint recursive calls**, just like QuickSort does.

Binary Search **does not have this**, even though it can be implemented recursively.

To complement [@Kenci's post](https://stackoverflow.com/a/8851907/4594973), DnC algorithms have a few general/common properties; they:

1. divide the original problem instance into a set of smaller sub-instances of itself;
2. independently solve each sub-instance;
3. combine smaller/independent sub-instance solutions to build a single solution for the larger/original instance

The problem with Binary Search is that it does not really even generate a set of independent sub-instances to be solved, as per step 1; it only simplifies the original problem by permanently discarding sections it's not interested in. In other words, it only reduces the problem's size and that's as far as it ever goes.

In short, binary search divides the size of the problem (on which it has to work) into halves but doesn't find the solution in bits and pieces and hence no need of merging the solution occurs!

**## Task 3/3: Individual, at home**

- Refactor the code in `./src/pow2.cpp` to optimize the time complexity of the recursive approach. Use `./src/pow3.cpp` as a starter.

- Refactor the code in `pow1.cpp` and `pow2.cpp`, as follows:

  - Replace `printf()` with` std::cout()`

  - Include the right headers e.g. `iostream`

**## Links**

- <https://www.techiedelight.com/power-function-implementation-recursive-iterative/>

- <https://cpp.sh/>

ANSWER:

#include <iostream>

#include <string>

using namespace std;

// #include <stdio.h>

// Optimized recursive solution to calculate `pow(x, n)`

// using divide-and-conquer

long power(int x, unsigned n)

{

    // base condition

    if (n == 0) {

        return 1L;

    }

    // calculate subproblem recursively

    int pow = power(x, n / 2);

    if (n & 1) { // if `y` is odd

        return x \* pow \* pow;

    }

    // otherwise, `y` is even

    return pow \* pow;

}

int main(void)

{

    int x = -2;

    unsigned n = 10;

    // printf("pow(%d, %d) = %d", x, n, power(x, n));

    cout<<"pow("<<x << ", " <<n<<")= " <<power(x, n);

    return 0;

}

The time complexity of the above solution is O(log(n))