<https://github.com/TT00FE39-3001/lecture5>

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

**Outline**

**Topics**

* Review
* Recursion
* [Dynamic Programming](https://www.geeksforgeeks.org/dynamic-programming/):
  + Memoization
  + Tabulation

**This Week in Points**

* Group Activities (Max 9 points)
* Homework (Max 9 points)
* Peer reviews (Max 7 points)

**Part 1: Recursion**

* The Top-Down Thought Process
* [Recursion In C++](https://www.softwaretestinghelp.com/recursion-in-cpp/)
* Fibonacci:
  + Visualization: [Link 1](https://www.cs.usfca.edu/~galles/visualization/DPFib.html), [Link 2](https://www.educative.io/courses/grokking-dynamic-programming-a-deep-dive-using-cpp/m2JgzWPw9RR)
  + [Fibonacci number](https://en.wikipedia.org/wiki/Fibonacci_number) & [Fibonacci sequence](https://www.mathsisfun.com/numbers/fibonacci-sequence.html)
* [The Staircase Problem](https://www.geeksforgeeks.org/count-ways-reach-nth-stair/)
* [Activity 1](https://github.com/TT00FE39-3001/lecture5/blob/main/activity1)

**Part 2: Dynamic Programming (Memoization)**

* [Overlapping Sub-problems](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)
* Fibonacci Revisited
  + Visualization: [Link 1](https://www.cs.usfca.edu/~galles/visualization/DPFib.html), [Link 2](https://www.educative.io/courses/grokking-dynamic-programming-a-deep-dive-using-cpp/m2JgzWPw9RR)
  + [Fibonacci Revisited](https://www.geeksforgeeks.org/introduction-to-dynamic-programming-data-structures-and-algorithm-tutorials/)
* [The Staircase Problem](https://www.geeksforgeeks.org/count-ways-reach-nth-stair/)
* [The Knapsack Problem](https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/)
* [What is memoization?](https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/)
* [Activity 2](https://github.com/TT00FE39-3001/lecture5/blob/main/activity2)

**Part 3: Dynamic Programming (Tabulation)**

* [Tabulation vs Memoization](https://www.geeksforgeeks.org/tabulation-vs-memoization/)
* Fibonacci Revisited
  + Visualization: [Link 1](https://www.cs.usfca.edu/~galles/visualization/DPFib.html), [Link 2](https://www.educative.io/courses/grokking-dynamic-programming-a-deep-dive-using-cpp/m2JgzWPw9RR)
* [The Staircase Problem](https://www.geeksforgeeks.org/count-ways-reach-nth-stair/)
* [The Knapsack Problem](https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/)
* [Dynamic Programming in the Real-world](https://www.educative.io/courses/grokking-dynamic-programming-a-deep-dive-using-cpp/m2JgzWPw9RR#Real-world-problems)
* [Activity 3](https://github.com/TT00FE39-3001/lecture5/blob/main/activity3)

**Misc**

* [Notes](https://github.com/TT00FE39-3001/lecture5/blob/main/notes.md)

README

**# Outline**

**## Topics**

- Review

- Recursion

- [Dynamic Programming](<https://www.geeksforgeeks.org/dynamic-programming/>):

  - Memoization

  - Tabulation

**## This Week in Points**

- Group Activities (Max 9 points)

- Homework (Max 9 points)

- Peer reviews (Max 7 points)

**## Part 1: Recursion**

- The Top-Down Thought Process

- [Recursion In C++](<https://www.softwaretestinghelp.com/recursion-in-cpp/>)

- Fibonacci:

  - Visualization: [Link 1](<https://www.cs.usfca.edu/~galles/visualization/DPFib.html>)

, [Link 2](<https://www.educative.io/courses/grokking-dynamic-programming-a-deep-dive-using-cpp/m2JgzWPw9RR>)

  - [Fibonacci number](<https://en.wikipedia.org/wiki/Fibonacci_number>)

& [Fibonacci sequence](<https://www.mathsisfun.com/numbers/fibonacci-sequence.html>)

- [The Staircase Problem](<https://www.geeksforgeeks.org/count-ways-reach-nth-stair/>)

- [Activity 1](./activity1)

**## Part 2: Dynamic Programming (Memoization)**

- [Overlapping Sub-problems](<https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/>)

- Fibonacci Revisited

  - Visualization: [Link 1](<https://www.cs.usfca.edu/~galles/visualization/DPFib.html>)

, [Link 2](<https://www.educative.io/courses/grokking-dynamic-programming-a-deep-dive-using-cpp/m2JgzWPw9RR>)

  - [Fibonacci Revisited](<https://www.geeksforgeeks.org/introduction-to-dynamic-programming-data-structures-and-algorithm-tutorials/>)

- [The Staircase Problem](<https://www.geeksforgeeks.org/count-ways-reach-nth-stair/>)

- [The Knapsack Problem](<https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/>)

- [What is memoization?](<https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/>)

- [Activity 2](./activity2)

**## Part 3: Dynamic Programming (Tabulation)**

- [Tabulation vs Memoization](<https://www.geeksforgeeks.org/tabulation-vs-memoization/>)

- Fibonacci Revisited

  - Visualization: [Link 1](<https://www.cs.usfca.edu/~galles/visualization/DPFib.html>)

, [Link 2](<https://www.educative.io/courses/grokking-dynamic-programming-a-deep-dive-using-cpp/m2JgzWPw9RR>)

- [The Staircase Problem](<https://www.geeksforgeeks.org/count-ways-reach-nth-stair/>)

- [The Knapsack Problem](<https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/>)

- [Dynamic Programming in the Real-world](<https://www.educative.io/courses/grokking-dynamic-programming-a-deep-dive-using-cpp/m2JgzWPw9RR#Real-world-problems>)

- [Activity 3](./activity3)

**## Misc**

- [Notes](./notes.md)

Notes

**# Notes**

**## Recursion: The Top-Down Thought Process**

When tackling a top-down problem, it helps to think the following three thoughts:

1. Imagine the function you’re writing has already been implemented by someone else.

2. Identify the subproblem of the problem.

3. See what happens when you call the function on the subproblem and go from there.

**## Dynamic programming**

Dynamic programming is an algorithm design technique with a rather interesting history. It was invented by a prominent U.S. mathematician, Richard Bellman, in the 1950s as a general method for optimizing multistage decision processes. Thus, the word “programming” in the name of this technique stands for **\*\*planning\*\*** and does not refer to computer programming. After proving its worth

as an important tool of applied mathematics, dynamic programming has eventually come to be considered, at least in computer science circles, as a general

algorithm design technique that does not have to be limited to special types of optimization problems. It is from this point of view that we will consider this technique here.

Dynamic programming is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a given problem’s solution to solutions of its smaller subproblems. Rather than solving

overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.

Links

**# Links**

- [Data Structures and Algorithms Interview Course](<https://www.enjoyalgorithms.com/data-structures-and-algorithms-course/>)

- [Dynamic Programming](<https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/DynamicProgramming.html>)

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

- [Dynamic Programming: lecture notes](<https://courses.csail.mit.edu/6.006/fall09/lecture_notes/lecture18.pdf>)

- [Recommended Playlist](<https://www.youtube.com/playlist?list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O>)

- <https://cpp.sh/>

HOMEWORK

**# Homework**

**## Task 1/3:Videos**

- [What Is Dynamic Programming and How To Use It](<https://youtu.be/vYquumk4nWw>)

- [0/1 Knapsack Problem](<https://youtu.be/nLmhmB6NzcM>)

**## Task 2/3: Reading**

- [The Staircase Problem](<https://www.geeksforgeeks.org/count-ways-reach-nth-stair/>)

- [0/1 Knapsack Problem](<https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/>)

- [Dynamic Programming](<https://www.geeksforgeeks.org/dynamic-programming/>)

- [Recursion](<https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/RecIntro.html>)

**## Task 3/3: Pre-Lecture (Videos)**

- [Trees and heaps](<https://youtube.com/watch?v=lhTCSGRAlXI&si=EnSIkaIECMiOmarE>)

- [Heaps 1](<https://youtube.com/watch?v=BzQGPA_v-vc&si=EnSIkaIECMiOmarE>)

ACTIVITY1

**# Activities**

**## Task 1**

- Refer to the following link. Discuss how the

  Recursive Factorial works:

<https://www.cs.usfca.edu/~galles/visualization/RecFact.html>

- Refer to the following link. Discuss how the Recursive Fibonacci works:

<https://www.cs.usfca.edu/~galles/visualization/DPFib.html>

**## Task 2**

There are `n` stairs, a person standing at the bottom wants to reach the top. The person can climb either 1 stair or 2 stairs at a time. There is a simple implementations in `./src/` folder. Discuss how the code works.

**## Task 3**

- There are `n` stairs, a person standing at the bottom wants to reach the top. The person can climb either 1 stair or 2 stairs or **\*\*3 stairs\*\*** at a time. Write a program that counts the number of ways, the person can reach the top. You can use the following program as a starter `./src/staircase1.cpp`. Also the link below might useful:

<https://www.includehelp.com/cpp-programs/stair-case-program-to-solve-the-staircase-problem.aspx>

**## Task 4: Individual (at home)**

- What are the pros/cons of recursive over iterative Programming?

- Difference between recursion and induction.

> Refer to the [links](#links) section below.

**## Links**

- <https://cpp.sh/>

- [Difference Between Recursion and Induction](<https://www.geeksforgeeks.org/difference-between-recursion-and-induction/>)

- [Recursion vs Iterative Programming](<https://www.softwaretestinghelp.com/recursion-in-cpp/>)

ANSWERS:

**## Task 1**

- Refer to the following link. Discuss how the

  Recursive Factorial works:

<https://www.cs.usfca.edu/~galles/visualization/RecFact.html>

- Refer to the following link. Discuss how the Recursive Fibonacci works:

<https://www.cs.usfca.edu/~galles/visualization/DPFib.html>

**## Task 2**

There are `n` stairs, a person standing at the bottom wants to reach the top. The person can climb either 1 stair or 2 stairs at a time. There is a simple implementations in `./src/` folder. Discuss how the code works.

code:

#include <iostream>

using namespace std;

int number\_of\_paths(int n)

{

    if (n <= 0)

        return 0;

    if (n == 1)

        return 1;

    if (n == 2)

        return 2;

    return number\_of\_paths(n - 1) + number\_of\_paths(n - 2);

}

int main()

{

    cout << "number of paths =  " << number\_of\_paths(4);

    return 0;

}

**## Task 3**

- There are `n` stairs, a person standing at the bottom wants to reach the top. The person can climb either 1 stair or 2 stairs or **\*\*3 stairs\*\*** at a time. Write a program that counts the number of ways, the person can reach the top. You can use the following program as a starter `./src/staircase1.cpp`. Also the link below might useful:

<https://www.includehelp.com/cpp-programs/stair-case-program-to-solve-the-staircase-problem.aspx>

starter `./src/staircase1.cpp for 2 steps

#include <iostream>

using namespace std;

int number\_of\_paths(int n)

{

    if (n <= 0)

        return 0;

    if (n == 1)

        return 1;

    if (n == 2)

        return 2;

    return number\_of\_paths(n - 1) + number\_of\_paths(n - 2);

}

int main()

{

    cout << "number of paths =  " << number\_of\_paths(4);

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture5-main\activity1\src> .\staircase1

number of paths = 5

This is wrong, if you can climb 3 stairs

Program after changes: (5 stairs) and up to 3 climbs

#include <iostream>

using namespace std;

int number\_of\_paths(int n)

{

    if(n<0){            //Base Case 1

        return 0;

    }

    if(n==0){           //Base Case 2

        return 1;

    }

    return number\_of\_paths(n - 1) + number\_of\_paths(n - 2) + number\_of\_paths(n - 3);

}

int main()

{

    cout << "number of paths =  " << number\_of\_paths(5);

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture5-main\activity1\src> .\staircase1

number of paths = 13

**## Task 4: Individual (at home)**

- What are the pros/cons of recursive over iterative Programming?

- Difference between recursion and induction.

> Refer to the [links](#links) section below.

**## Links**

- <https://cpp.sh/>

- [Difference Between Recursion and Induction](<https://www.geeksforgeeks.org/difference-between-recursion-and-induction/>)

- [Recursion vs Iterative Programming](<https://www.softwaretestinghelp.com/recursion-in-cpp/>)

- What are the pros/cons of recursive over iterative Programming?

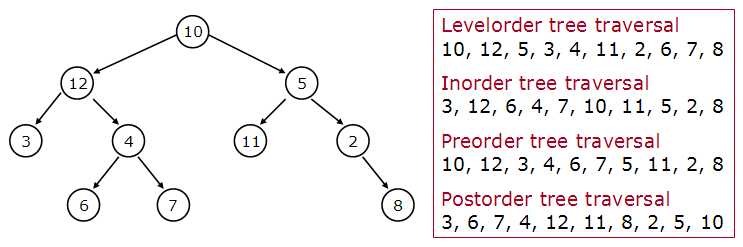
here are several pros and cons to recursion.

PROS:

**Recursion can reduce time complexity**. This was somewhat counter-intuitive to me since in my experience, recursion sometimes increased the time it took for a function to complete the task. An example of this is calculating fibonacci numbers. If you calculate the fibonacci sequence up to a number n using recursion rather than iteration, the time to complete the task when compared to that of the iterative approach was much greater. However, if you **memoize** the result (aka save the value of each calculation for further use in the recursive call) you can in fact reduce the time complexity (read a great answer response for more information about memoization [here](https://cs.stackexchange.com/questions/13055/time-complexity-and-space-complexity-in-recursive-algorithm)).

**Recursion adds clarity and reduces the time needed to write and debug code.** This one is valid to a point. If you know your input into a function is going to be small, then recursion is certainly a good choice if you want to de-clutter your code. If your input is sufficiently large however, the sacrifice of speed and memory for the sake of clarity becomes much less attractive and functional.

**Recursion is better at tree traversal.** This one is a little more advanced. An extremely simplified version of what this means is as follows: A tree is a collection objects that are linked to one another (imagine leaves on a tree connected by branches that are in turn connected to other branches all the way to the roots). One of the more efficient ways to traverse these trees when looking for a specific leaf (or node) is by recursively following a single branch until the end of that branch until you find the value you are looking for. Again, this is extremely abstracted and simplified for what is actually happening and I urge you to look further into what is actually happening in tree traversal.



Example of tree traversal

Recursion in the above tree diagram would be beneficial when used on preorder tree traversal.

CONS:

**Recursion uses more memory.** Because the function has to add to the stack with each recursive call and keep the values there until the call is finished, the memory allocation is greater than that of an iterative function.

**Recursion can be slow.**If not implemented correctly (as stated above with memoization) it can be much slower than iteration. It is actually pretty difficult to write a recursive function where the speed and memory will be less than that of an iterative function completing the same task. The reason that recursion is slow is that it requires the allocation of a new stack frame.

Both iteration and recursion are repetitive processes that repeat a certain process until a certain condition is met. They are both used in programming to complete tasks where a task has to be repeated in order to solve the problem.

**Iteration:** A function repeats a defined process until a condition fails. This is usually done through a loop, such as a for or while loop with a counter and comparative statement making up the condition that will fail. An infinite loop for iteration occurs when the condition never fails.

**Recursion:** Instead of executing a specific process within the function, the function calls itself repeatedly until a certain condition is met (this condition being the base case). The base case is explicitly stated to return a specific value when a certain condition is met. An infinite recursive loop occurs when the function does not reduce its input in a way that will converge on the base case.

### Pros/Cons Of Recursion Over Iterative Programming

Recursive programs provide compact and clean code. A recursive program is a simple way of writing programs. There are some inherent problems like factorial, Fibonacci sequence, towers of Hanoi, tree traversals, etc. which require recursion for solving.

In other words, they are solved efficiently with recursion. They can also be solved with iterative programming using stacks or other data structures but there are chances to become more complex to implement.

Problem-solving powers of recursive as well as iterative programming are the same. However, recursive programs take more memory space as all the function calls need to be stored on the stack until the base case is matched.

Recursive functions also have a time overhead because of too many function calls and return values

- Difference between recursion and induction.

Recursion and induction belong to the branch of Mathematics, these terms are used interchangeably. But there are some differences between these terms.

[Recursion](https://www.geeksforgeeks.org/recursion/)is a process in which a function gets repeated again and again until some base function is satisfied. It repeats and uses its previous values to form a sequence. The procedure applies a certain relation to the given function again and again until some base condition is met. It consists of two components:

**1) Base condition**: In order to stop a recursive function, a condition is needed. This is known as a base condition. Base condition is very important. If the base condition is missing from the code then the function can enter into an infinite loop.

**2)** **Recursive step**: It divides a big problem into small instances that are solved by the recursive function and later on recombined in the results.

### **Induction**

[Induction](https://www.geeksforgeeks.org/principle-of-mathematical-induction/) is the branch of mathematics that is used to prove a result, or a formula, or a statement, or a theorem. It is used to establish the validity of a theorem or result. It has two working rules:

**1)** **Base Step**: It helps us to prove that the given statement is true for some initial value.

**2)** **Inductive Step**: It states that if the theorem is true for the nth term, then the statement is true for (n+1)th term.

***Example*:** The assertion is that the nth Fibonacci number is at most 2n.

### How to Prove a statement using induction?

**Step 1**: Prove or verify that the statement is true for n=1

**Step 2**: Assume that the statement is true for n=k

**Step 3**: Verify that the statement is true for n=k+1, then it can be concluded that the statement is true for n.

## ****Difference between Recursion and Induction****:

| **S.No.** | **Recursion** | **Induction** |
| --- | --- | --- |
| **1.** | Recursion is the process in which a function is called again and again until some base condition is met. | Induction is the way of proving a mathematical statement. |
| **2.** | It is the way of defining in a repetitive manner. | It is the way of proving. |
| **3.** | It starts from nth term till the base case. | It starts from the initial till (n+1)th term. |
| **4.** | It has two components:   * Base condition * Recursive step. | It has two steps:   * Base step * Inductive step |
| **5.** | We backtrack at each step to replace the previous values with the answers using the function. | We just prove that the statement is true for n=1. Then we assume that n = k is true. Then we prove for n=k+1. |
| **6.** | No assumptions are made. | The assumption is made for n= k |
| **7.** | Recursive function is always called to find successive terms. | Here statements or theorems are proved and no terms are found. |
| **8.** | It can lead to infinity if no base condition is given. | There is no concept of infinity. |

### What Is Recursion?

Recursion is a process in which a function calls itself. The function that implements recursion or calls itself is called a Recursive function. In recursion, the recursive function calls itself over and over again and keeps on going until an end condition is met.

ACTIVITY2

**# Activities**

**## Task 1**

Refer to the following link. Discuss how the Recursive Fibonacci with Memoization works:

<https://www.cs.usfca.edu/~galles/visualization/DPFib.html>

**## Task 2**

The stair case problem can be solved based on the Fibonacci series. There is a simple implementations in `./src/staircase2.cpp`.

- Explain how the code works. The following link might be useful:

<https://dev.to/alisabaj/the-climbing-staircase-problem-how-to-solve-it-and-why-the-fibonacci-numbers-are-relevant-3c4o>

- Modify the code to use Dynamic Programming (Memoization)

**## Task 3**

Explain how the code in `./src/staircase3.cpp` works.

**## Task 4: Individual (at home)**

- There are `n` stairs, a person standing at the bottom wants to reach the top. Write a program that counts the number of ways someone can climb up to m stairs for a given value m. For example, if m is 4, it is possible to climb 1 stair or 2 stairs or 3 stairs or 4 stairs at a time. Make sure you use. Refer to the link below:

<https://www.geeksforgeeks.org/count-ways-reach-nth-stair/>

**## Links**

- <https://cpp.sh/>

- [leetcode.com](<https://leetcode.com/problems/climbing-stairs/>)

ANSWERS:

**# Task 1**

Refer to the following link. Discuss how the Recursive Fibonacci with Memoization works:

<https://www.cs.usfca.edu/~galles/visualization/DPFib.html>

**## Task 2**

The stair case problem can be solved based on the Fibonacci series. There is a simple implementations in `./src/staircase2.cpp`.

- Explain how the code works. The following link might be useful:

<https://dev.to/alisabaj/the-climbing-staircase-problem-how-to-solve-it-and-why-the-fibonacci-numbers-are-relevant-3c4o>

- Modify the code to use Dynamic Programming (Memoization)

staircase2.cpp

#include <iostream>

using namespace std;

// A simple recursive program to

// find N'th fibonacci number

int fib(int n)

{

    if (n <= 1)

        return n;

    return fib(n - 1) + fib(n - 2);

}

// Returns number of ways to reach s'th stair

int countWays(int s)

{

    return fib(s + 1);

}

// Driver C

int main()

{

    int s = 4;

    cout << "Number of ways = " << countWays(s);

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture5-main\activity2\src> .\staircase2

Number of ways = 5

0 1 1 2 3 5

**## Task 3**

Explain how the code in `./src/staircase3.cpp` works.

code for staircase3.cpp:

#include <iostream>

#include <cstring>

using namespace std;

// A simple recursive program to find N'th fibonacci number

int fib(int n, int dp[])

{

    if (n <= 1)

        return dp[n] = 1;

    if (dp[n] != -1)

    {

        return dp[n];

    }

    dp[n] = fib(n - 1, dp) + fib(n - 2, dp);

    return dp[n];

}

// Returns number of ways to reach s'th stair

int countWays(int n)

{

    int dp[n + 1];

    memset(dp, -1, sizeof dp);

    fib(n, dp);

    return dp[n];

}

// Driver C

int main()

{

    int n = 4;

    cout << "Number of ways = " << countWays(n);

    return 0;

}

**## Task 4: Individual (at home)**

- There are `n` stairs, a person standing at the bottom wants to reach the top. Write a program that counts the number of ways someone can climb up to m stairs for a given value m. For example, if m is 4, it is possible to climb 1 stair or 2 stairs or 3 stairs or 4 stairs at a time. Make sure you use. Refer to the link below:

<https://www.geeksforgeeks.org/count-ways-reach-nth-stair/>

**## Links**

- <https://cpp.sh/>

- [leetcode.com](<https://leetcode.com/problems/climbing-stairs/>)

ACTIVITY3

**# Activities**

**## Task 1**

Refer to the following link. Explain how the Knapsack Algorithm works:

<https://monicagranbois.com/knapsack-algorithm-visualization/>

**## Task 2**

Refer to the following link. What are the difference between the brute force and the optimized solutions to the Knapsack problem.

<https://www.educative.io/blog/0-1-knapsack-problem-dynamic-solution>

**## Task 3**

There are different implementations of the stair case problem in the following link:

<https://www.enjoyalgorithms.com/blog/climbing-stairs-problem>

Compare the time and space complexity of the different approaches

**## Task 4: Individual (at home)**

- Difference between divide and conquer and dynamic programming

- State some application of dynamic programming

- Difference between recursion vs dynamic programming

- Difference between Top down and bottom up approaches to dynamic programming

- How to solve a Dynamic Programming Problem?

> Refer to the [links](#links) section below

**## Links**

- https://cpp.sh/

- [Recursion vs dynamic programming](<https://www.geeksforgeeks.org/introduction-to-dynamic-programming-data-structures-and-algorithm-tutorials/>)

- [How to solve a Dynamic Programming Problem ?](<https://www.geeksforgeeks.org/solve-dynamic-programming-problem/>)

ANSWERS:

**## Task 1**

Refer to the following link. Explain how the Knapsack Algorithm works:

<https://monicagranbois.com/knapsack-algorithm-visualization/>

**## Task 2**

Refer to the following link. What are the difference between the brute force and the optimized solutions to the Knapsack problem.

<https://www.educative.io/blog/0-1-knapsack-problem-dynamic-solution>

**## Task 3**

There are different implementations of the stair case problem in the following link:

<https://www.enjoyalgorithms.com/blog/climbing-stairs-problem>

Compare the time and space complexity of the different approaches

**## Task 4: Individual (at home)**

- Difference between divide and conquer and dynamic programming

- State some application of dynamic programming

- Difference between recursion vs dynamic programming

- Difference between Top down and bottom up approaches to dynamic programming

- How to solve a Dynamic Programming Problem?

> Refer to the [links](#links) section below

**## Links**

- <https://cpp.sh/>

- [Recursion vs dynamic programming](<https://www.geeksforgeeks.org/introduction-to-dynamic-programming-data-structures-and-algorithm-tutorials/>)

- [How to solve a Dynamic Programming Problem ?](<https://www.geeksforgeeks.org/solve-dynamic-programming-problem/>)

Answers:

- Difference between divide and conquer and dynamic programming

The difference between divide and conquer and dynamic programming is that the former is a method of dividing a problem into smaller parts and then solving each one separately, while the latter is a method of solving larger problems by breaking them down into smaller pieces.

- State some application of dynamic programming

## Some commonly asked problems in Dynamic programming:

| **S. No.** | **Problem** | **Practice link** |
| --- | --- | --- |
| 1 | [Min Cost Path](https://www.geeksforgeeks.org/min-cost-path-dp-6/) | [solve](https://practice.geeksforgeeks.org/problems/path-in-matrix3805/1) |
| 2 | [Subset Sum Problem](https://www.geeksforgeeks.org/subset-sum-problem-dp-25/) | [solve](https://practice.geeksforgeeks.org/problems/subset-sum-problem-1611555638/1) |
| 3 | [Knapsack problem](https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/) | [solve](https://practice.geeksforgeeks.org/problems/0-1-knapsack-problem0945/1) |
| 4 | [Coin Change](https://www.geeksforgeeks.org/coin-change-dp-7/) | [solve](https://practice.geeksforgeeks.org/problems/coin-change2448/1) |
| 5 | [Edit Distance](https://www.geeksforgeeks.org/edit-distance-dp-5/) | [solve](https://practice.geeksforgeeks.org/problems/edit-distance3702/1) |
| 6 | [Cutting a Rod](https://www.geeksforgeeks.org/cutting-a-rod-dp-13/) | [solve](https://practice.geeksforgeeks.org/problems/rod-cutting0840/1) |
| 7 | [Subset Sum Problem](https://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/) | [solve](https://practice.geeksforgeeks.org/problems/subset-sum-problem-1611555638/1) |
| 8 | [Longest Common Subsequence](https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/) | [solve](https://practice.geeksforgeeks.org/problems/longest-common-subsequence-1587115620/1) |
| 9 | [Matrix chain multiplication](https://www.geeksforgeeks.org/matrix-chain-multiplication-dp-8/) | [solve](https://practice.geeksforgeeks.org/problems/matrix-chain-multiplication0303/1) |
| 10 | [Count Distinct Subsequences](https://www.geeksforgeeks.org/count-distinct-subsequences/) | [solve](https://practice.geeksforgeeks.org/problems/number-of-distinct-subsequences0909/1) |
| 11 | [Prefix Sum of Matrix (Or 2D Array)](https://www.geeksforgeeks.org/prefix-sum-2d-array/) | [solve](https://practice.geeksforgeeks.org/problems/prefix-sum-of-matrix-or-2d-array/1) |
| 12 | [Check if it is possible to transform one string into another](https://www.geeksforgeeks.org/check-possible-transform-one-string-another/) | [solve](https://practice.geeksforgeeks.org/problems/string-conversion4603/1) |

- Difference between recursion vs dynamic programming

Recursion: repeated application of the same procedure on subproblems of the same type of a problem. Dynamic programming: caching the results of the subproblems of a problem, so that every subproblem is solved only once.

- Difference between Top down and bottom up approaches to dynamic programming

Top-down is a recursive problem-solving approach, while bottom-up is an iterative approach. In other words, the top-down approach assumes the subproblems will be solved using the smaller sub-problem only once using the recursion.

- How to solve a Dynamic Programming Problem?

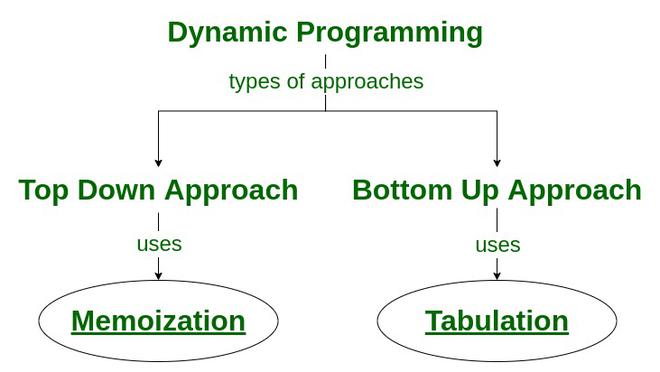
MORE:

## How Memoization technique is used in Dynamic Programming?

Dynamic programming helps to efficiently solve problems that have overlapping subproblems and optimal substructure properties. The idea behind dynamic programming is to break the problem into smaller sub-problems and save the result for future use, thus eliminating the need to compute the result repeatedly.

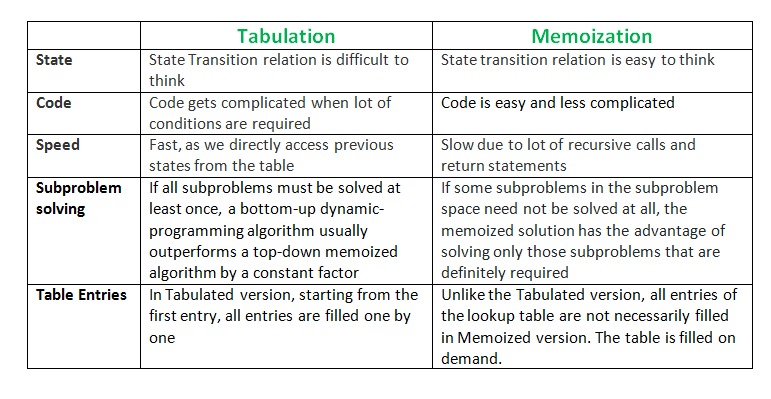
There are two approaches to formulate a dynamic programming solution:

1. **Top-Down Approach:**  *This approach follows the****memoization****technique*. It consists of **recursion** and **caching**. In computation, recursion represents the process of calling functions repeatedly, whereas cache refers to the process of storing intermediate results.
2. **Bottom-Up Approach:***This approach uses the****tabulation****technique* to implement the dynamic programming solution. It addresses the same problems as before, but without recursion. In this approach, iteration replaces recursion. Hence, there is no stack overflow error or overhead of recursive procedures.



*How Memoization technique is used in Dynamic Programming*

## How Memoization is different from Tabulation?

[](https://www.geeksforgeeks.org/tabulation-vs-memoization/)

*Tabulation vs Memoization*

For more details please refer:[Tabulation vs. Memoization](https://www.geeksforgeeks.org/tabulation-vs-memoization/)

## Coding Practice Problems on Memoization

| **Question** | **Article** | **Practice** | **Video** |
| --- | --- | --- | --- |
| Count ways to reach the n’th stair | [View](https://write.geeksforgeeks.org/post/Count%20ways%20to%20reach%20the%20n%E2%80%99th%20stair) | [Solve](https://practice.geeksforgeeks.org/problems/node-at-a-given-index-in-linked-list/1) | Watch |
| Word Break Problem | DP-32 | [View](https://write.geeksforgeeks.org/post/Word%20Break%20Problem%20%7C%20DP-32) | [Solve](https://practice.geeksforgeeks.org/problems/inorder-traversal/1) | [Watch](https://youtu.be/IpyCqRmaKW4) |
| Program for Fibonacci numbers | [View](https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/) | [Solve](https://practice.geeksforgeeks.org/problems/insertion-sort/1) | [Watch](https://youtu.be/OGzPmgsI-pQ) |
| nth Catalan Number | [View](https://www.geeksforgeeks.org/program-nth-catalan-number/) | [Solve](https://practice.geeksforgeeks.org/problems/nth-catalan-number0817/1) | Watch |
| Gold Mine Problem | [View](https://www.geeksforgeeks.org/gold-mine-problem/) | [Solve](https://practice.geeksforgeeks.org/problems/gold-mine-problem2608/1/) | Watch |
| Subset Sum Problem | [View](https://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/) | [Solve](https://practice.geeksforgeeks.org/problems/subset-sum-problem-1611555638/1/#:~:text=Given%20an%20array%20of%20non,%2B3%2B2%20%3D%209.) | Watch |
| Cutting a Rod | [View](https://www.geeksforgeeks.org/dynamic-programming-set-13-cutting-a-rod/) | [Solve](https://practice.geeksforgeeks.org/problems/rod-cutting0840/1/) | Watch |
| Min Cost Path | [View](https://www.geeksforgeeks.org/dynamic-programming-set-6-min-cost-path/) | [Solve](https://practice.geeksforgeeks.org/problems/path-in-matrix/0) | Watch |
| Minimum number of jumps to reach end | [View](https://www.geeksforgeeks.org/minimum-number-of-jumps-to-reach-end-of-a-given-array/) | [Solve](https://practice.geeksforgeeks.org/problems/jump-game/1/) | Watch |
| Longest Palindromic Substring | Set 1 | [View](https://www.geeksforgeeks.org/longest-palindrome-substring-set-1/) | [Solve](https://practice.geeksforgeeks.org/problems/longest-palindrome-in-a-string1956/1/) | Watch |
| Longest Repeating Subsequence | [View](https://www.geeksforgeeks.org/longest-repeating-subsequence/) | [Solve](https://practice.geeksforgeeks.org/problems/longest-repeating-subsequence2004/1/) | Watch |
| Count ways to reach the nth stair using step 1, 2 or 3 | [View](https://www.geeksforgeeks.org/count-ways-reach-nth-stair-using-step-1-2-3/) | [Solve](https://practice.geeksforgeeks.org/problems/topological-sort/1) | Watch |
| Count of different ways to express N as the sum of 1, 3 and 4 | [View](https://www.geeksforgeeks.org/count-ofdifferent-ways-express-n-sum-1-3-4/) | [Solve](https://practice.geeksforgeeks.org/problems/count-ways-to-express-n-as-the-sum-of-13-and-44024/1/) | Watch |
| Count number of ways to cover a distance | [View](https://www.geeksforgeeks.org/count-number-of-ways-to-cover-a-distance/) | [Solve](https://practice.geeksforgeeks.org/problems/count-number-of-hops-1587115620/1/) | Watch |
| Count of arrays having consecutive element with different values | [View](https://www.geeksforgeeks.org/count-arrays-consecutive-element-different-values/) | [Solve](https://practice.geeksforgeeks.org/problems/search-a-node-in-bst/1) | Watch |
| Largest Sum Contiguous Subarray | [View](https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/) | [Solve](https://practice.geeksforgeeks.org/problems/kadanes-algorithm-1587115620/1/) | Watch |
| Smallest sum contiguous subarray | [View](https://www.geeksforgeeks.org/smallest-sum-contiguous-subarray/) | [Solve](https://practice.geeksforgeeks.org/problems/smallest-sum-contiguous-subarray/1/) | Watch |
| Unique paths in a Grid with Obstacles | [View](https://www.geeksforgeeks.org/unique-paths-in-a-grid-with-obstacles/) | [Solve](https://ide.geeksforgeeks.org/) | Watch |
| Different ways to sum n using numbers greater than or equal to m | [View](https://www.geeksforgeeks.org/different-ways-sum-n-using-numbers-greater-equal-m/) | [Solve](https://ide.geeksforgeeks.org/) | Watch |

## Frequently asked questions (FAQs) about Memoization

### **1: Is memoization better than DP?**

Memoization is the top-down approach to solving a problem with dynamic programming. It’s called memoization because we will create a memo for the values returned from solving each problem.

### **2: Is memoization the same as caching?**

Memoization is actually a specific type of caching. The term caching can generally refer to any storing technique (like HTTP caching) for future use, but memoizing refers more specifically to caching function that returns the value.

### **3: Why memoization is top-down?**

The top-Down approach breaks the large problem into multiple subproblems. if the subproblem is solved already then reuse the answer. Otherwise, Solve the subproblem and store the result in some memory.

### **4: Does memoization use recursion?**

Memoization follows top-down approach to solving the problem. It consists of recursion and caching. In computation, recursion represents the process of calling functions repeatedly, whereas cache refers to the process of storing intermediate results.

### **5: Should I use tabulation or memoization?**

For problems requiring all subproblems to be solved, tabulation typically outperforms memoization by a constant factor. This is because the tabulation has no overhead of recursion which reduces the time for resolving the recursion call stack from the stack memory.  
Whenever a subproblem needs to be solved for the original problem to be solved, memoization is preferable since a subproblem is solved lazily, i.e. only the computations that are required are carried out.

### 6: Where is memoization used?

Memoization is an optimization technique used to speed up computer programs by caching the results of expensive function calls and returning them when the same inputs are encountered again.

### 7: Why is it called memoization?

The term “memoization” comes from the Latin word “memorandum” (“to remember”), which is commonly shortened to “memo” in American English, and which means “to transform the results of a function into something to remember.”.

### 8: How does memoization reduce time complexity?

Solving the same problem again and again takes time and increases the run-time complexity of the overall program. This problem can be resolved by maintaining some cache or memory where we will store the already calculated result of the problem for some particular input. So that if we don’t want to recalculate the same problem, we can simply use the result that is stored in the memory and reduce the time complexity.

### 9: What is the difference between memoization and caching?

Memoization is actually a specific type of caching that involves caching the return value of a function based on input. Caching is a more general term. For example, HTTP caching is caching but it is not memoization.

### 10: Why tabulation is faster than memoization?

Tabulation is usually faster than memoization, because it is iterative and solving subproblems requires no overhead of recursive calls.

## Conclusion

Memoization is a programming concept and can be applied to any programming language. Its absolute goal is to optimize the program. Usually, this problem is seen when programs perform heavy computations. This technique cache all the previous result that is computed so that it will not have to recalculate for the same problem.

**Related Articles:**

* [Memoization using decorators in Python](https://www.geeksforgeeks.org/memoization-using-decorators-in-python/#:~:text=Memoization%20is%20a%20technique%20of,the%20help%20of%20function%20decorators.)
* [JavaScript Memoization](https://www.geeksforgeeks.org/javascript-memoization/#:~:text=Importance%20of%20Memoization%3A%20When%20a,cached%20answer%20from%20the%20memory.)

APPENDIX:

## Types of Algorithms:

There are several types of algorithms available. Some important algorithms are:

**1.**[**Brute Force Algorithm:**](https://www.geeksforgeeks.org/brute-force-approach-and-its-pros-and-cons/) It is the simplest approach for a problem. A brute force algorithm is the first approach that comes to finding when we see a problem.

**2.**[**Recursive Algorithm**](https://www.geeksforgeeks.org/recursion/)**:** A recursive algorithm is based on [recursion](http://www.geeksforgeeks.org/recursion/). In this case, a problem is broken into several sub-parts and called the same function again and again.

**3.**[**Backtracking Algorithm**](https://www.geeksforgeeks.org/backtracking-algorithms/)**:** The backtracking algorithm basically builds the solution by searching among all possible solutions. Using this algorithm, we keep on building the solution following criteria. Whenever a solution fails we trace back to the failure point and build on the next solution and continue this process till we find the solution or all possible solutions are looked after.

**4.**[**Searching Algorithm**](https://www.geeksforgeeks.org/searching-algorithms/)**:** Searching algorithms are the ones that are used for searching elements or groups of elements from a particular data structure. They can be of different types based on their approach or the data structure in which the element should be found.

**5.**[**Sorting Algorithm**](https://www.geeksforgeeks.org/sorting-algorithms/)**:** Sorting is arranging a group of data in a particular manner according to the requirement. The algorithms which help in performing this function are called sorting algorithms. Generally sorting algorithms are used to sort groups of data in an increasing or decreasing manner.

**6.**[**Hashing Algorithm**](https://www.geeksforgeeks.org/hashing-set-1-introduction/)**:** Hashing algorithms work similarly to the searching algorithm. But they contain an index with a key ID. In hashing, a key is assigned to specific data.

**7.**[**Divide and Conquer Algorithm**](http://www.geeksforgeeks.org/divide-and-conquer-introduction/)**:** This algorithm breaks a problem into sub-problems, solves a single sub-problem and merges the solutions together to get the final solution. It consists of the following three steps:

* Divide
* Solve
* Combine

**8.**[**Greedy Algorithm**](http://www.geeksforgeeks.org/greedy-algorithms/)**:** In this type of algorithm the solution is built part by part. The solution of the next part is built based on the immediate benefit of the next part. The one solution giving the most benefit will be chosen as the solution for the next part.

**9.**[**Dynamic Programming Algorithm**](https://www.geeksforgeeks.org/dynamic-programming/)**:** This algorithm uses the concept of using the already found solution to avoid repetitive calculation of the same part of the problem. It divides the problem into smaller overlapping subproblems and solves them.

**10.**[**Randomized Algorithm**](https://www.geeksforgeeks.org/randomized-algorithms/)**:** In the randomized algorithm we use a random number so it gives immediate benefit. The random number helps in deciding the expected outcome.

### **Topics:**

* [Analysis of Algorithms](https://www.geeksforgeeks.org/fundamentals-of-algorithms/#analysis)
* [Searching and Sorting](https://www.geeksforgeeks.org/fundamentals-of-algorithms/#searchandsort)
* [Greedy Algorithms](https://www.geeksforgeeks.org/fundamentals-of-algorithms/#greedy)
* [Dynamic Programming](https://www.geeksforgeeks.org/fundamentals-of-algorithms/#dp)
* [Pattern Searching](https://www.geeksforgeeks.org/fundamentals-of-algorithms/#pattern)
* [Backtracking](https://www.geeksforgeeks.org/fundamentals-of-algorithms/#backtracking)
* [Divide and Conquer](https://www.geeksforgeeks.org/fundamentals-of-algorithms/#divideandconquer)
* [Geometric Algorithms](https://www.geeksforgeeks.org/fundamentals-of-algorithms/#geometric)
* [Mathematical Algorithms](https://www.geeksforgeeks.org/fundamentals-of-algorithms/#mathematical)
* [Bit Algorithms](https://www.geeksforgeeks.org/fundamentals-of-algorithms/#bitalgo)
* [Graph Algorithms](https://www.geeksforgeeks.org/fundamentals-of-algorithms/#graph)
* [Randomized Algorithms](https://www.geeksforgeeks.org/fundamentals-of-algorithms/#randomized)
* [Branch and Bound](https://www.geeksforgeeks.org/fundamentals-of-algorithms/#branchandbound)
* [Quizzes](https://www.geeksforgeeks.org/fundamentals-of-algorithms/#quiz)

**Analysis of Algorithms:**

1. [Asymptotic Analysis](https://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/)
2. [Worst, Average and Best Cases](https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/)
3. [Asymptotic Notations](https://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/)
4. [Lower and Upper Bound Theory](https://www.geeksforgeeks.org/lower-and-upper-bound-theory/)
5. [Introduction to Amortized Analysis](https://www.geeksforgeeks.org/analysis-algorithm-set-5-amortized-analysis-introduction/)
6. [What does ‘Space Complexity’ mean?](https://www.geeksforgeeks.org/g-fact-86/)
7. [Polynomial Time Approximation Scheme](https://www.geeksforgeeks.org/polynomial-time-approximation-scheme/)
8. [Accounting Method | Amortized Analysis](https://www.geeksforgeeks.org/accounting-method-amortized-analysis/)
9. [Potential Method in Amortized Analysis](https://www.geeksforgeeks.org/potential-method-in-amortized-analysis/)

**Searching and Sorting:**

1. [Introduction to Searching Algorithms](https://www.geeksforgeeks.org/searching-algorithms/)
2. [Introduction to Sorting Algorithm](https://www.geeksforgeeks.org/sorting-algorithms/)
3. [Stable and Unstable Sorting Algorithms](https://www.geeksforgeeks.org/stable-and-unstable-sorting-algorithms/)
4. [Lower bound for comparison based sorting algorithms](https://www.geeksforgeeks.org/lower-bound-on-comparison-based-sorting-algorithms/)
5. [Can Run Time Complexity of a comparison-based sorting algorithm be less than N logN?](https://www.geeksforgeeks.org/can-run-time-complexity-of-a-comparison-based-sorting-algorithm-be-less-than-n-logn/)
6. [Which sorting algorithm makes minimum number of memory writes?](https://www.geeksforgeeks.org/which-sorting-algorithm-makes-minimum-number-of-writes/)

**Greedy Algorithms:**

1. [Introduction to Greedy Algorithms](https://www.geeksforgeeks.org/greedy-algorithms/)
2. [Activity Selection Problem](https://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/)
3. [Huffman Coding](https://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding/)
4. [Job Sequencing Problem](https://www.geeksforgeeks.org/job-sequencing-problem-set-1-greedy-algorithm/)
5. [Quiz on Greedy Algorithms](http://geeksquiz.com/algorithms/greedy-algorithms/)
6. [Minimum Number of Platforms Required for a Railway/Bus Station](https://www.geeksforgeeks.org/minimum-number-platforms-required-railwaybus-station/)

**Dynamic Programming:**

1. [Introduction to Dynamic Programming](https://www.geeksforgeeks.org/dynamic-programming/)
2. [Overlapping Subproblems Property](https://www.geeksforgeeks.org/dynamic-programming-set-1/)
3. [Optimal Substructure Property](https://www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/)
4. [Longest Increasing Subsequence](https://www.geeksforgeeks.org/dynamic-programming-set-3-longest-increasing-subsequence/)
5. [Longest Common Subsequence](https://www.geeksforgeeks.org/dynamic-programming-set-4-longest-common-subsequence/)
6. [Min Cost Path](https://www.geeksforgeeks.org/dynamic-programming-set-6-min-cost-path/)
7. [Coin Change](https://www.geeksforgeeks.org/dynamic-programming-set-7-coin-change/)
8. [Matrix Chain Multiplication](https://www.geeksforgeeks.org/dynamic-programming-set-8-matrix-chain-multiplication/)
9. [0-1 Knapsack Problem](https://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-knapsack-problem/)
10. [Longest Palindromic Subsequence](https://www.geeksforgeeks.org/dynamic-programming-set-12-longest-palindromic-subsequence/)
11. [Palindrome Partitioning](https://www.geeksforgeeks.org/dynamic-programming-set-17-palindrome-partitioning/)

**Pattern Searching:**

1. [Introduction to Pattern Searching](https://www.geeksforgeeks.org/algorithms-gq/pattern-searching/)
2. [Naive Pattern Searching](https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/)
3. [KMP Algorithm](https://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/)
4. [Rabin-Karp Algorithm](https://www.geeksforgeeks.org/searching-for-patterns-set-3-rabin-karp-algorithm/)
5. [Pattern Searching using a Trie of all Suffixes](https://www.geeksforgeeks.org/pattern-searching-using-trie-suffixes/)
6. [Aho-Corasick Algorithm for Pattern Searching](https://www.geeksforgeeks.org/aho-corasick-algorithm-pattern-searching/)
7. [Z algorithm (Linear time pattern searching Algorithm)](https://www.geeksforgeeks.org/z-algorithm-linear-time-pattern-searching-algorithm/)

**Backtracking:**

1. [Introduction to Backtracking](https://www.geeksforgeeks.org/backtracking-algorithms/)
2. [Print all permutations of a given string](https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/)
3. [The Knight’s tour problem](https://www.geeksforgeeks.org/backtracking-set-1-the-knights-tour-problem/)
4. [Rat in a Maze](https://www.geeksforgeeks.org/backttracking-set-2-rat-in-a-maze/)
5. [N Queen Problem](https://www.geeksforgeeks.org/backtracking-set-3-n-queen-problem/)
6. [Subset Sum](https://www.geeksforgeeks.org/backttracking-set-4-subset-sum/)
7. [m Coloring Problem](https://www.geeksforgeeks.org/backttracking-set-5-m-coloring-problem/)
8. [Hamiltonian Cycle](https://www.geeksforgeeks.org/backtracking-set-7-hamiltonian-cycle/)
9. [Sudoku](https://www.geeksforgeeks.org/backtracking-set-7-suduku/)

**Divide and Conquer:**

1. [Introduction to Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer/)
2. [Merge Sort](https://www.geeksforgeeks.org/merge-sort/)
3. [Write your own pow(x, n) to calculate x\*n](https://www.geeksforgeeks.org/write-a-c-program-to-calculate-powxn/)
4. [Count Inversions](https://www.geeksforgeeks.org/counting-inversions/)
5. [Closest Pair of Points](https://www.geeksforgeeks.org/closest-pair-of-points/)
6. [Strassen’s Matrix Multiplication](https://www.geeksforgeeks.org/strassens-matrix-multiplication/)

**Geometric Algorithm:**

1. [Introduction to Geometric Algorithms](https://www.geeksforgeeks.org/geometric-algorithms/)
2. [Closest Pair of Points | O(nlogn) Implementation](https://www.geeksforgeeks.org/closest-pair-of-points-onlogn-implementation/)
3. [How to check if a given point lies inside or outside a polygon?](https://www.geeksforgeeks.org/how-to-check-if-a-given-point-lies-inside-a-polygon/)
4. [How to check if two given line segments intersect?](https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/)
5. [Given n line segments, find if any two segments intersect](https://www.geeksforgeeks.org/given-a-set-of-line-segments-find-if-any-two-segments-intersect/)
6. [How to check if given four points form a square](https://www.geeksforgeeks.org/check-given-four-points-form-square/)
7. [Convex Hull using Jarvis’ Algorithm or Wrapping](https://www.geeksforgeeks.org/convex-hull-using-jarvis-algorithm-or-wrapping/)

**Mathematical Algorithms:**

1. [Introduction to Mathematical Algorithms](https://www.geeksforgeeks.org/mathematical-algorithms/)
2. [Write an Efficient Method to Check if a Number is Multiple of 3](https://www.geeksforgeeks.org/write-an-efficient-method-to-check-if-a-number-is-multiple-of-3/)
3. [Write a program to add two numbers in base 14](https://www.geeksforgeeks.org/write-a-program-to-add-two-numbers-in-base-14/)
4. [Program for Fibonacci numbers](https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/)
5. [Average of a stream of numbers](https://www.geeksforgeeks.org/average-of-a-stream-of-numbers/)
6. [Multiply two integers without using multiplication, division and bitwise operators, and no loops](https://www.geeksforgeeks.org/multiply-two-numbers-without-using-multiply-division-bitwise-operators-and-no-loops/)
7. [Babylonian method for square root](https://www.geeksforgeeks.org/square-root-of-a-perfect-square/)
8. [Sieve of Eratosthenes](https://www.geeksforgeeks.org/sieve-of-eratosthenes/)
9. [Pascal’s Triangle](https://www.geeksforgeeks.org/pascal-triangle/)
10. [Given a number, find the next smallest palindrome](https://www.geeksforgeeks.org/given-a-number-find-next-smallest-palindrome-larger-than-this-number/)
11. [Program to add two polynomials](https://www.geeksforgeeks.org/program-add-two-polynomials/)
12. [Multiply two polynomials](https://www.geeksforgeeks.org/multiply-two-polynomials-2/)
13. [Count trailing zeroes in factorial of a number](https://www.geeksforgeeks.org/count-trailing-zeroes-factorial-number/)

**Bitwise Algorithms:**

1. [Introduction to Bitwise Algorithms](https://www.geeksforgeeks.org/bitwise-algorithms/)
2. [Little and Big Endian](https://www.geeksforgeeks.org/little-and-big-endian-mystery/)
3. [Detect opposite signs](https://www.geeksforgeeks.org/detect-if-two-integers-have-opposite-signs/)
4. [Swap bits](https://www.geeksforgeeks.org/swap-bits-in-a-given-number/)
5. [Turn off the rightmost set bit](https://www.geeksforgeeks.org/turn-off-the-rightmost-set-bit/)
6. [Rotate bits](https://www.geeksforgeeks.org/rotate-bits-of-an-integer/)
7. [Next higher number with same number of set bits](https://www.geeksforgeeks.org/next-higher-number-with-same-number-of-set-bits/)
8. [Swap two nibbles in a byte](https://www.geeksforgeeks.org/swap-two-nibbles-byte/)

**Graph Algorithms:**

1. [Introduction to Graph Algorithms](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/)
2. [BFS, DFS](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#bfsndfs)
3. [Cycles in Graph](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#cycle)
4. [Shortest paths](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#shortest)
5. [MST](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#MST)
6. [Topological Sorting](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#topo)
7. [Connectivity](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#connectivity)
8. [Max Flow](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#maxflow)

**Randomized Algorithms:**

1. [Introduction to Randomized Algorithms](https://www.geeksforgeeks.org/randomized-algorithms/)
2. [Linearity of Expectation](https://www.geeksforgeeks.org/linearity-of-expectation/)
3. [Expected Number of Trials until Success](https://www.geeksforgeeks.org/expected-number-of-trials-before-success/)
4. [Randomized Algorithms | Set 0 (Mathematical Background)](https://www.geeksforgeeks.org/randomized-algorithms-set-0-mathematical-background/)
5. [Randomized Algorithms | Set 1 (Introduction and Analysis)](https://www.geeksforgeeks.org/randomized-algorithms-set-1-introduction-and-analysis/)
6. [Randomized Algorithms | Set 2 (Classification and Applications)](https://www.geeksforgeeks.org/randomized-algorithms-set-2-classification-and-applications/)
7. [Randomized Algorithms | Set 3 (1/2 Approximate Median)](https://www.geeksforgeeks.org/randomized-algorithms-set-3-12-approximate-median/)
8. [Reservoir Sampling](https://www.geeksforgeeks.org/reservoir-sampling/)

**Branch and Bound:**

1. [Branch and Bound | Set 1 (Introduction with 0/1 Knapsack)](https://www.geeksforgeeks.org/branch-and-bound-set-1-introduction-with-01-knapsack/)
2. [Branch and Bound | Set 2 (Implementation of 0/1 Knapsack)](https://www.geeksforgeeks.org/branch-and-bound-set-2-implementation-of-01-knapsack/)
3. [Branch and Bound | Set 3 (8 puzzle Problem)](https://www.geeksforgeeks.org/branch-bound-set-3-8-puzzle-problem/)
4. [Branch And Bound | Set 4 (Job Assignment Problem)](https://www.geeksforgeeks.org/branch-bound-set-4-job-assignment-problem/)
5. [Branch and Bound | Set 5 (N Queen Problem)](https://www.geeksforgeeks.org/branch-and-bound-set-4-n-queen-problem/)
6. [Branch And Bound | Set 6 (Traveling Salesman Problem)](https://www.geeksforgeeks.org/branch-bound-set-5-traveling-salesman-problem/)

**Quizzes:**

* [Analysis of Algorithms](http://geeksquiz.com/algorithms/analysis-of-algorithms/)
* [Sorting](http://geeksquiz.com/algorithms/searching-and-sorting/)
* [Divide and Conquer](http://geeksquiz.com/algorithms/divide-and-conquer/)
* [Greedy Algorithms](http://geeksquiz.com/algorithms/greedy-algorithms/)
* [Dynamic Programming](http://geeksquiz.com/algorithms/dynamic-programming/)
* [Backtracking](http://geeksquiz.com/algorithms/backtracking/)
* [Misc](http://geeksquiz.com/algorithms/misc-2/)
* [NP Complete](http://geeksquiz.com/algorithms/np-complete/)
* [Searching](http://geeksquiz.com/algorithms/searching/)
* [Analysis of Algorithms (Recurrences)](http://geeksquiz.com/algorithms/analysis-of-algorithms-recurrences/)
* [Recursion](http://geeksquiz.com/algorithms/recursion/)
* [Bit Algorithms](http://geeksquiz.com/algorithms/bit-algorithms/)
* [Graph Traversals](http://geeksquiz.com/algorithms/graph-traversals/)
* [Graph Shortest Paths](http://geeksquiz.com/algorithms/graph-shortest-paths/)
* [Graph Minimum Spanning Tree](http://geeksquiz.com/algorithms/graph-minimum-spanning-tree/)

Please see [Data Structures and Advanced Data Structures](https://www.geeksforgeeks.org/data-structures/) for Graph, Binary Tree, BST and Linked List based algorithms.