<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?

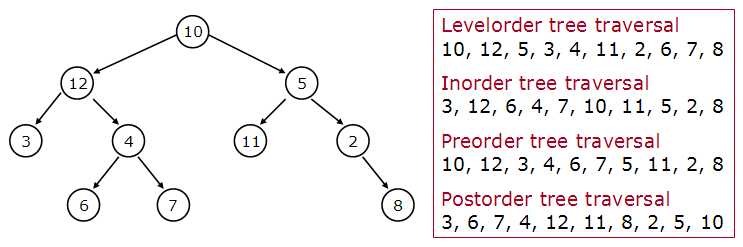
here are several pros and cons to recursion.

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/>)