Data Structures and Algorithms

Lesson 13. Design Techniques and Strategies

Alexandra Ciobotaru

26th of May 2022

Syllabus

- **Lesson 1. Introduction: Algorithms, Data Structures and Cognitive Science.**
- **Lesson 2. Python Data Types and Structures.**
- **Lesson 3. Principles of Algorithm Design.**
- **Lesson 4. Lists and Pointer Structures.**
- **Lesson 5. Stacks and Queues.**
- **Lesson 6. Trees.**
- **Lesson 7. Hashing and Symbol Tables.**
- **Lesson 8. Graphs.**
- **Lesson 9. Searching.**
- **Lesson 10. Sorting.** Mid-term evaluation quiz (30%)
- **Lesson 11. Selection Algorithms.**
- **Lesson 12. String Algorithms and Techniques.**
- **Lesson 13. Design Techniques and Strategies.**
- Lab (20%) + Project (50%).

What you'll learn

1. Classification of algorithms
    1. By implementation
    2. By complexity
    3. By design
2. Technical implementation of some theoretical programming techniques
    1. Using dynamic programming – Fibonacci series
    2. Using divide and conquer – Merge sort
    3. Using greedy algorithms – Travelling salesman
3. Complexity classes

1. Classification of algorithms

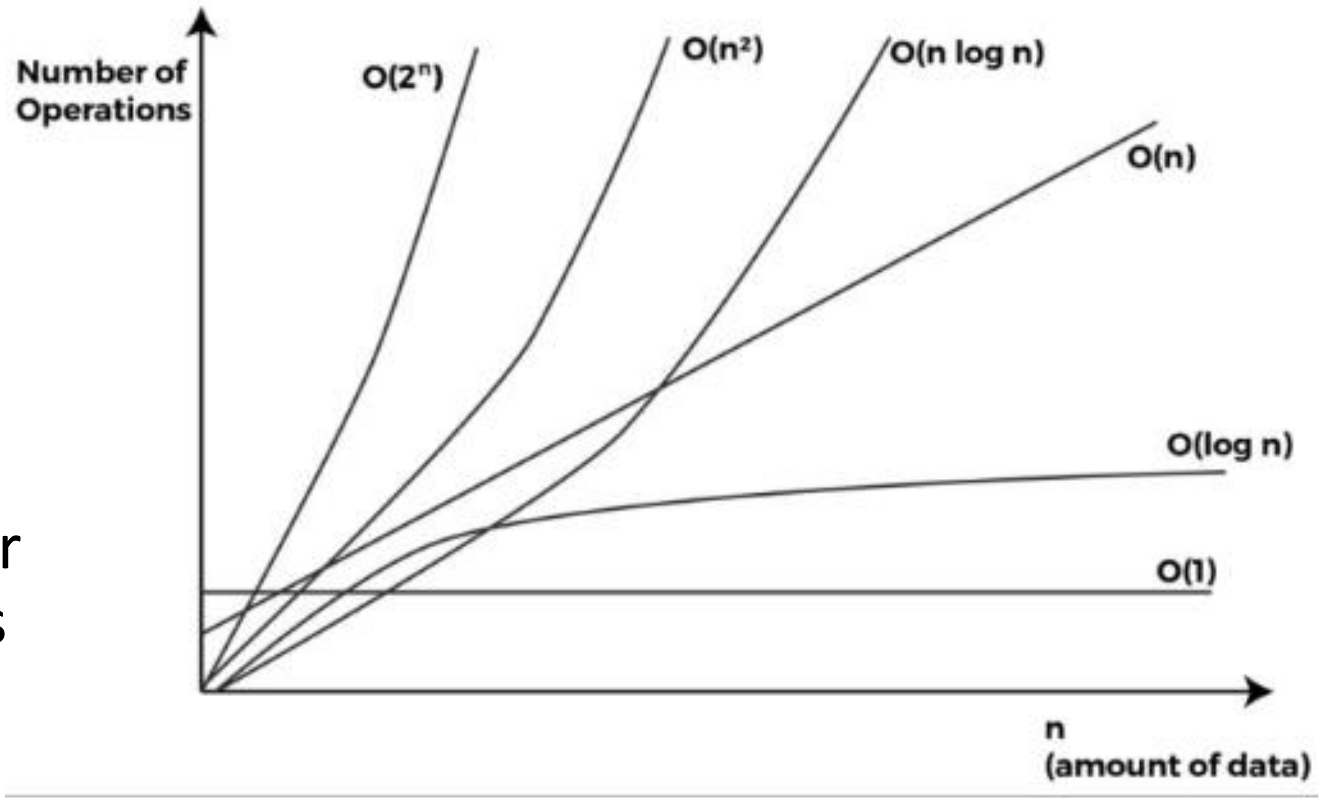
- An algorithm is a step-by-step procedure for solving a problem. It is defined to have these features:
  - Must have some data to operate on it
  - It must produce at least one result
  - It must terminate after a finite number of steps
- The algorithms can be grouped into classes.

1.1. Classification by implementation

- Recursion
  - Recursive algorithms are the ones that call themselves to repeatedly execute code until a certain condition is satisfied.
- Logic
  - An algorithm can be expressed as a controlled logical deduction
- Serial or Parallel
  - Serial algorithms, also known as **sequential algorithms**, are algorithms that are executed sequentially (RAM - there is one processor with memory locations that can each be individually accessed)
  - **Parallel algorithms** perform more than one operation at a time and they divide a problem into sub problems among its processors (PRAM - there are many processors with a common global memory)
- Deterministic vs. nondeterministic
  - **Deterministic algorithms** produce the same output without fail every time the algorithm is run with the same input.
  - **Nondeterministic algorithms** can change the order of execution or some internal sub-process, leading to a change in the final result each time the algorithm is run.

1.2. Classification by complexity

- How do we measure the efficiency of one algorithm against the other?
- The notation O(n) captures the growth rate of an algorithm.
- If an algorithm has a time complexity of **O(1)**, and another algorithm for the same task has the complexity **O(log n),** the first algorithm should be preferred.



Algorithm runtime

1.3. Classification by design

A given problem may have a number of solutions. When these solutions are analyzed, it is observed that each one follows a certain pattern or technique:

- Iterative
  - Certain steps are repeated in loops, until the goal is achieved
- Divide and conquer
  - A given problem is fragmented into sub-problems, which are solved separately
  - The algorithm is terminated when further sub-division cannot be performed
- Dynamic programming
  - Is similar to divide and conquer, does not compute the solution to an already encountered subproblem, but it uses a *remembering* technique to avoid the recomputation.

1.3. Classification by design

- Backtracking - all possible options are explored
  - Backtracking is an algorithmic technique where the goal is to get all solutions to a problem using the brute force approach.
  - It consists of building a set of all the solutions incrementally.
- Greedy algorithms
  - The guiding rule is to always select the option that yields the most beneficial results and to continue doing that, hoping to reach a perfect solution.
  - This technique aims to find a global optimal final solution by making a series of local optimal choices.
  - The local optimal choice seems to lead to the solution.

2. Technical implementation

2.1. Using dynamic programming

- Making the algorithm more efficient by storing intermediary results
- One property that makes a problem an ideal candidate for being solved with dynamic programming is that it has an overlapping set of sub-problems.
- Once we realize that the form of sub-problems has repeated itself during computation, we need not compute it again. Instead, we return a pre-computed result for that previously encountered sub-problem

- Steps:
  - Find the recursive sequence
  - Store results, so that you don’t have to repeat those computations
    - top-down approach (**memoization**)
    - Bottom-up approach (**tabulation**)

Example: The Fibonacci series

- Let's consider an example to better understand how dynamic programming works, in terms of memoization and tabulation.

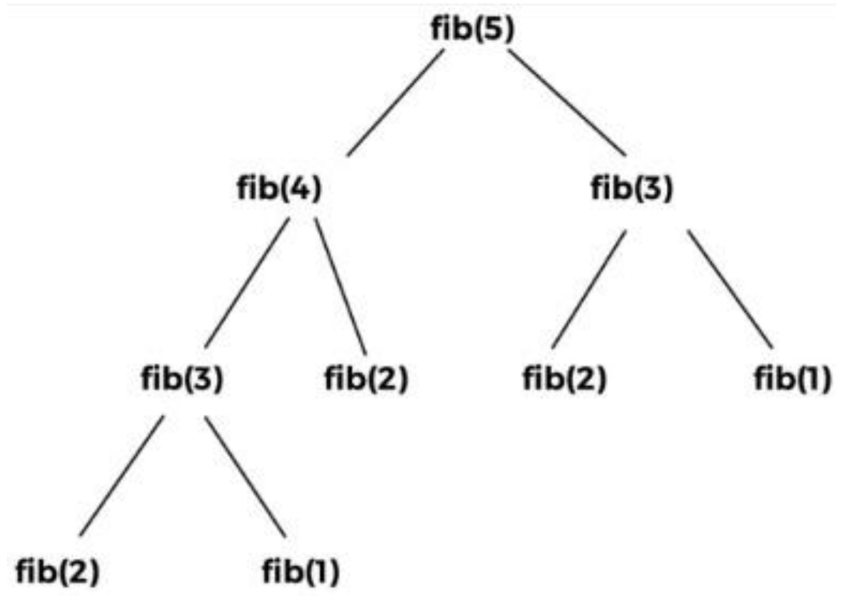
```
func(1) = 1
func(0) = 1
func(n) = func(n-1) + func(n-2)

        def fib(n):
            if n <= 2: # the base case
                return 1
            else:
                return fib(n-1) + fib(n-2)
```

using Memoization

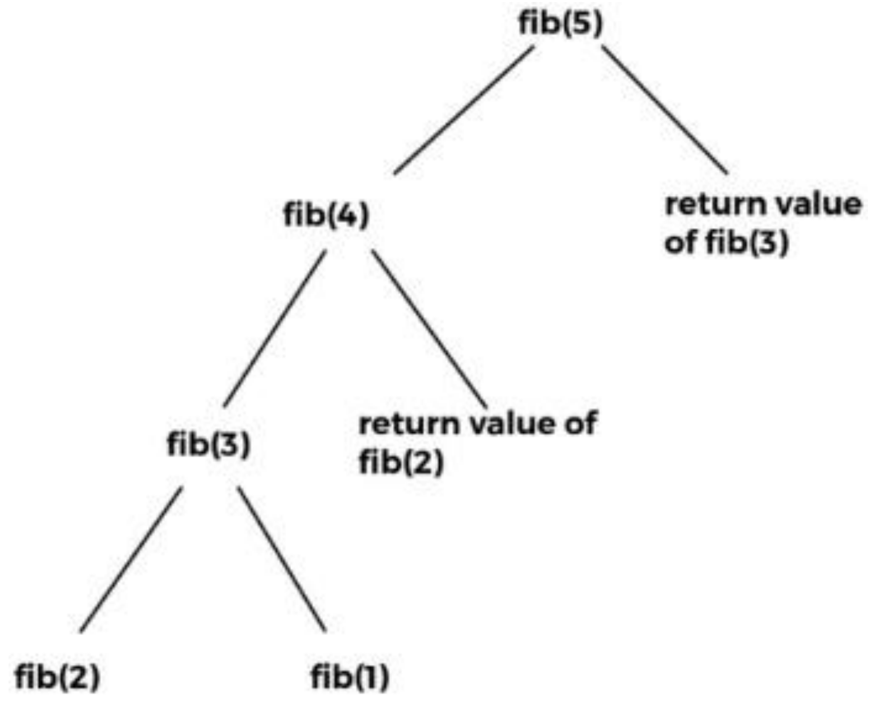
```
def fib(n):
    if n <= 2: # the base case
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

- fib(1) happens twice
- fib(2) happens 3 times
- fib(3) happens twice
- Computational time will be wasted if we compute again whenever we encounter the same function, since the same result is returned.



using Memoization

- A better approach is to store the results of the computation of fib(1) the first time it is encountered.
- Similarly, we should store return values for fib(2) and fib(3).
- Later, anytime we encounter a call to fib(1), fib(2), or fib(3), we simply return their respective results.



```
def dyna_fib(n, lookup):
    if n <= 2:
        lookup[n] = 1
    if lookup[n] is None:
        lookup[n] = dyna_fib(n-1, lookup) + dyna_fib(n-2, lookup)
    return lookup[n]
map_set = [None]*(1000)
print(dyna_fib(10, map_set))
```

using Tabulation

- We can use a table of results, or matrix in some cases, to store the results of computations for later use.
- This approach solves the bigger problem by first working out a route to the final solution.
- In the case of the fib() function, we develop a table with the values of fib(1) and fib(2) predetermined. Based on these two values, we will work our way up to fib(n):

```
def fib(n):
    results = [1, 1]
    for i in range(2, n):
        results.append(results[i-1] + results[i-2])
    return results[-1]
```

2.2. Implementation using divide and conquer

- We break down a problem into smaller sub-problems of the same type or form of the original problem.
- These sub-problems are solved and combined to obtain the solution of the original problem.
- Steps:
  - Divide – we break down the problem into subproblems
  - Conquer – at some point, the smallest indivisible problem will return a solution
  - Merge – and from that point we start reversing the thought process and combine the solutions to the smaller problems in order to solve the bigger problem

Example – Merge sort

- Given a list of unsorted elements, we split the list into two approximate halves.
- We continue to divide the list into two halves recursively.
- After a while, the sub-lists created as a result of the recursive call will contain only one element, and at that point, we begin to merge the solutions.

(see Lesson 10)

2.3. Implementation using greedy algorithms

- Greedy algorithms make decisions to yield the best possible local solution, which in turn provides the optimal solution.
- By making the best possible choices at each step, the total path will lead to an overall optimal solution.

Example – shortest path (Dijkstra's algorithm)

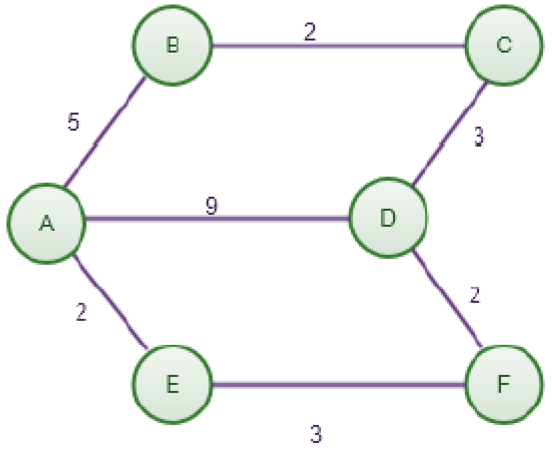
- The shortest path problem requires us to find out the shortest possible route between a pair of nodes in a weighted graph (directed or undirected).
- Applications: mapping and route planning, when plotting the most efficient way to get from point **A** to point **B**.

Shortest path

- Steps:
  1. Mark all nodes as unvisited, and set their distance from the given node to infinity
  2. Set the source node as current
  3. For the current node, look for all the unvisited adjacent nodes; compute the distance to that node from the source node through the current node. Compare the newly computed distance to the currently assigned distance, and if it is smaller, set this as the new value (instead of infinity)
  4. Once we have considered all the unvisited adjacent nodes of the current node, we mark it as visited.
  5. We next consider the next unvisited node which has the shortest distance from the source node. Repeat steps 2 to 4.
  6. We stop when the list of unvisited nodes is empty, meaning we have considered all the unvisited nodes.
- Algorithm explanation: https://youtu.be/CL1byLngb5Q

Shortest path

- At first glance the shortest path between A and D is 9, but the shortest route means the lowest total distance, even if this comprises several parts.
- By comparison, traveling from node **A** to node **E** to node **F** and finally to node **D** will incur a total distance of 7, making it a shorter route.
- We will use a table to keep track of the shortest distance:



Node	Shortest distance from source	Previous node
A	0	None
B	∞	None
C	∞	None
D	∞	None
E	∞	None
F	∞	None

No prior nodes have been visited when the algorithm begins.

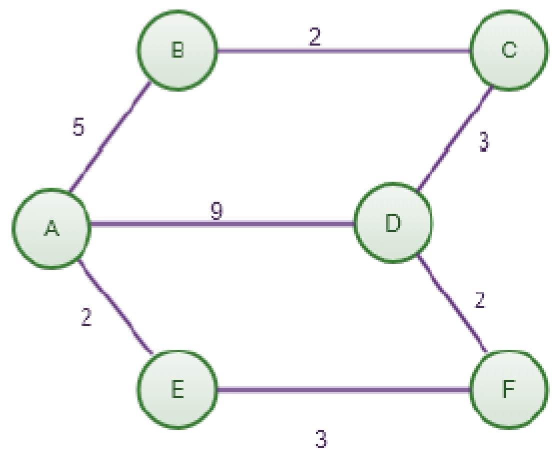
We initially set the distance to all other nodes to infinity, with the exception of starting node.

Shortest path

- The adjacency list will be:

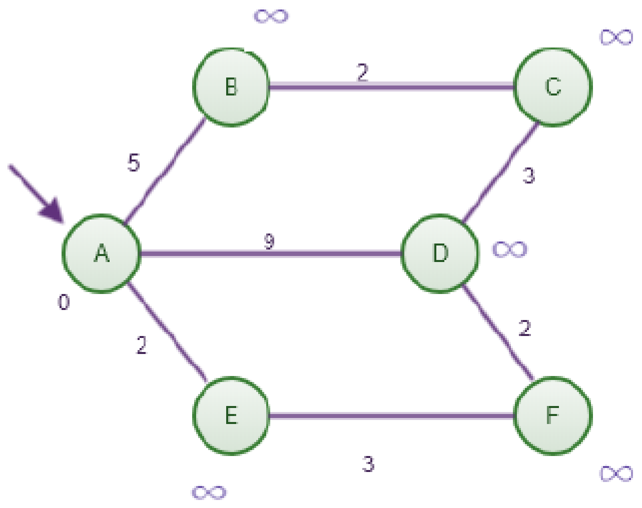
```
graph = dict()
graph['A'] = {'B': 5, 'D': 9, 'E': 2}
graph['B'] = {'A': 5, 'C': 2}
graph['C'] = {'B': 2, 'D': 3}
graph['D'] = {'A': 9, 'F': 2, 'C': 3}
graph['E'] = {'A': 2, 'F': 3}
graph['F'] = {'E': 3, 'D': 2}
```

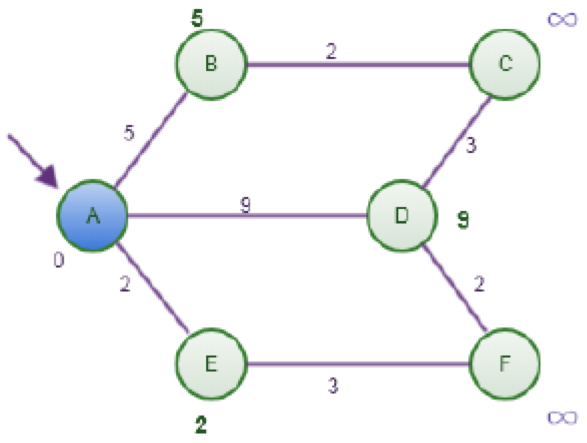
- To find the shortest distance from node A to node B, we need to find the distance from the start node to the previous node of node B, which happens to be node A, and add it to the distance from node A to node B.
- We do this for all adjacent nodes of **A**, which are **B**, **E**, and **D**.



Shortest path

- We take the adjacent node **B** as next node to analyze;
- The distance from the start node (**A**) to the previous node (None) is 0, and the distance from the previous node to the current node (**B**) is **5**.
- This sum is compared with the data in the shortest distance column of node **B**. Since **5** < **∞**, we replace **∞** with the smaller of the two, which is **5**.
- Any time the shortest distance of a node is replaced by a smaller value, we need to update the previous node column too for all the adjacent nodes of the current node.
- After this, we mark node **A** as visited.





Shortest path

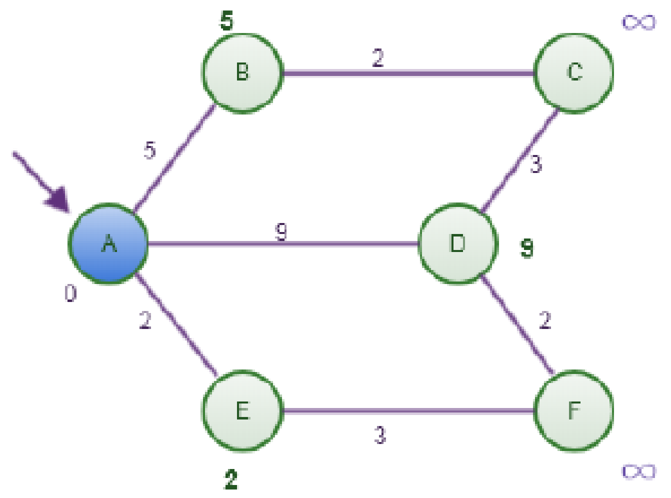
- At the end of first step:

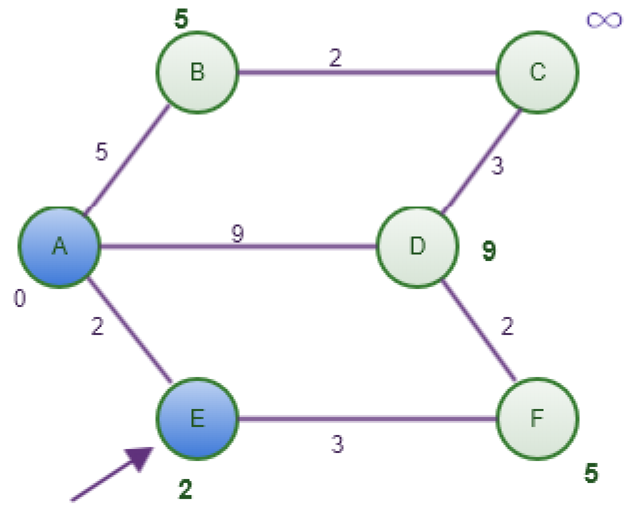
Node	Shortest distance from source	Previous node
A*	0	None
B	5	A
C	∞	None
D	9	A
E	2	A
F	∞	None

- In the second step, we chose the next node as the one with the shortest distance, using our table as a guide. Node **E**, with its value of 2, has the shortest distance.
- To get to node E, we must first visit node A and cover a distance of 2.

Shortest path

- The adjacent nodes to node E are A and F, but A is visited, so we consider node F.
- To find the shortest route to node F we must find the distance from the starting node to node E, and add it to the distance between node E and F.
- The distance from the starting node to node E is 2; the distance from node E to F is 3 -> 2+3=5, which is less than infinity (we mark on the graph 5 instead of infinity)
- Since there are no more adjacent nodes to node E, we mark E as visited, and update the table accordingly

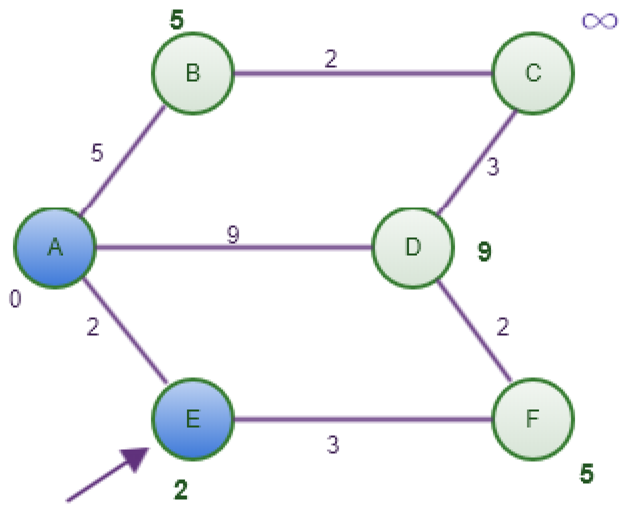




Shortest path

- Next, the shortest values in table belong to B and F

Node	Shortest distance from source	Previous node
A*	0	None
B	5	A
C	∞	None
D	9	A
E*	2	A
F	5	E



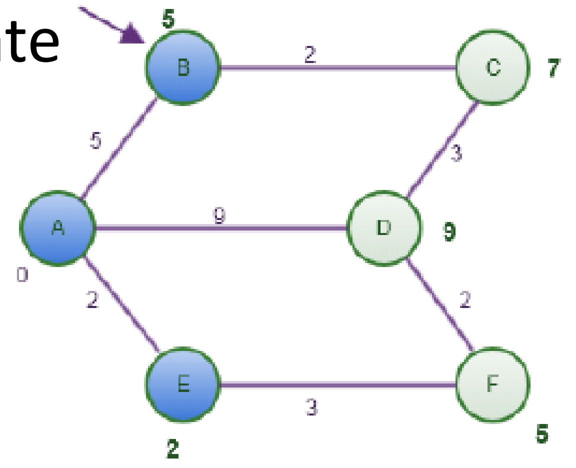
- We chose B for exploration. The adjacent nodes to B are A and C, but A has already been visited.
- Using the same rule, the shortest distance from A to C is 7 (5 + 2).

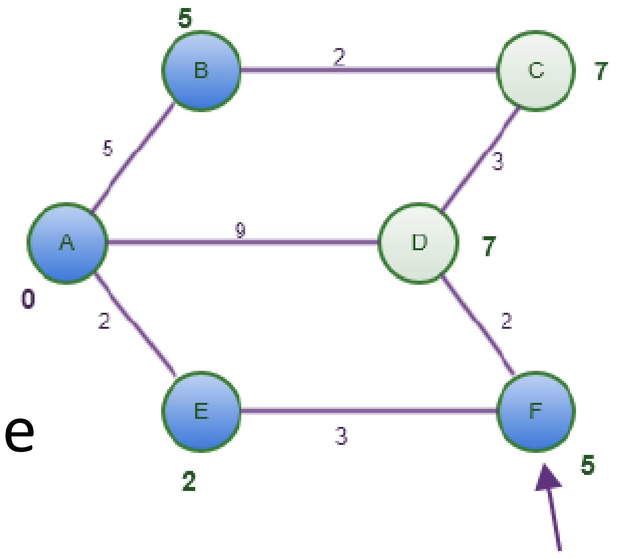
Shortest path

- Since 7 < ∞, we update the shortest distance to 7 and update the previous node column with node **B**.

Node	Shortest distance from source	Previous node
A*	0	None
B*	5	A
C	7	B
D	9	A
E*	2	A
F	5	E

- The next unvisited node with shortest distance is F – its adjacent nodes are D and E, but E was visited.
- We calculate this distance by adding the distance from node **A** to **F** to the distance from node **F** to **D**. This sums up to 7, which is less than **9**. Thus, we update the **9** with **7** and replace **A** with **F** in node **D**'s previous node column.



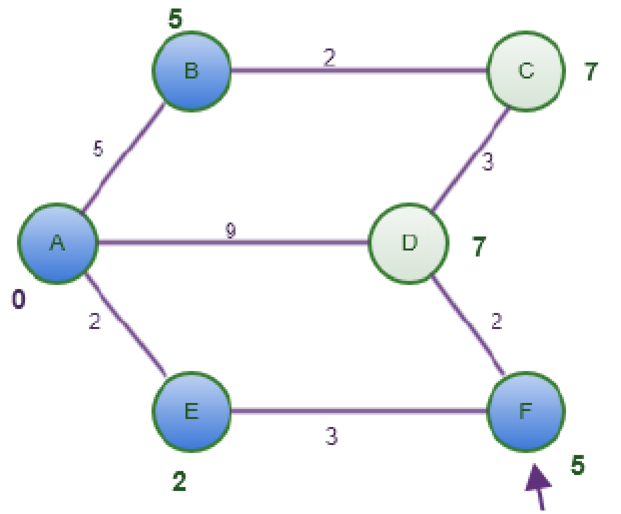


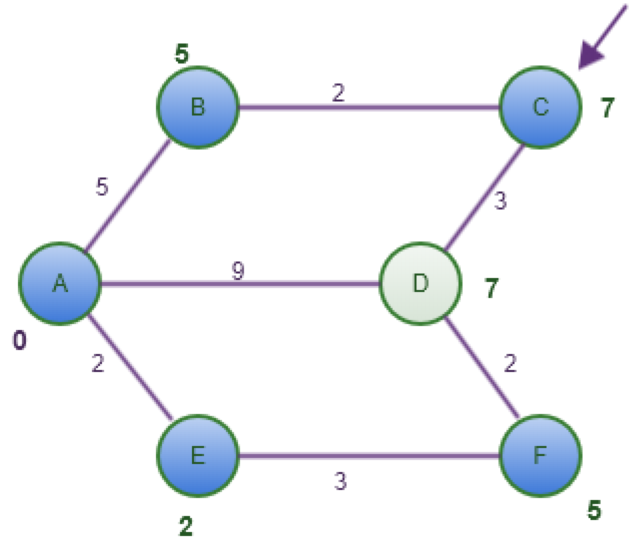
Shortest path

- Node F is now marked as visited

Node	Shortest distance from source	Previous node
A*	0	None
B*	5	A
C	7	B
D	7	F
E*	2	A
F*	5	E

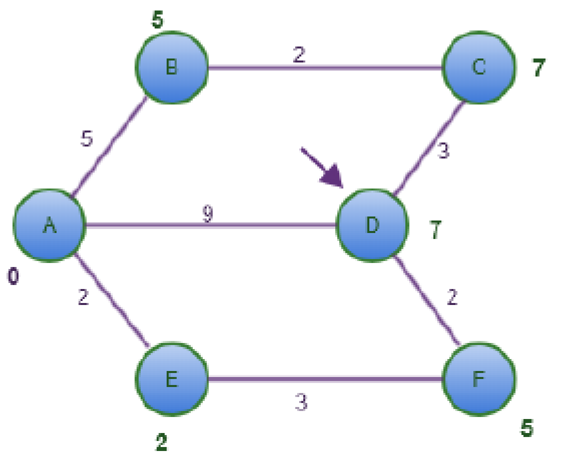
- Only C and D are left unvisited, both with distance cost of 7. We chose C. We mark C as visited.
- Lastly, we take node D and mark it as visited.





Shortest path

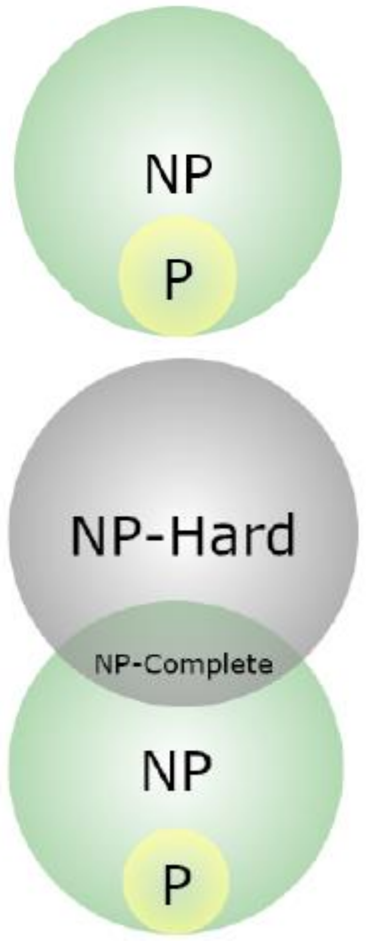
Node	Shortest distance from source	Previous node
A*	0	None
B*	5	A
C*	7	B
D*	7	F
E*	2	A
F*	5	E



- According to the table, the shortest distance from the source column for node **F** is 5. This is true.
- It also tells us that to get to node **F**, we need to visit node **E**, and from **E** to node **A**, which is our starting node. This is actually the shortest path.
- Complexity: O(E(logV)), where E is the edges connected to each node, and V is the number of nodes.

3. Complexity classes

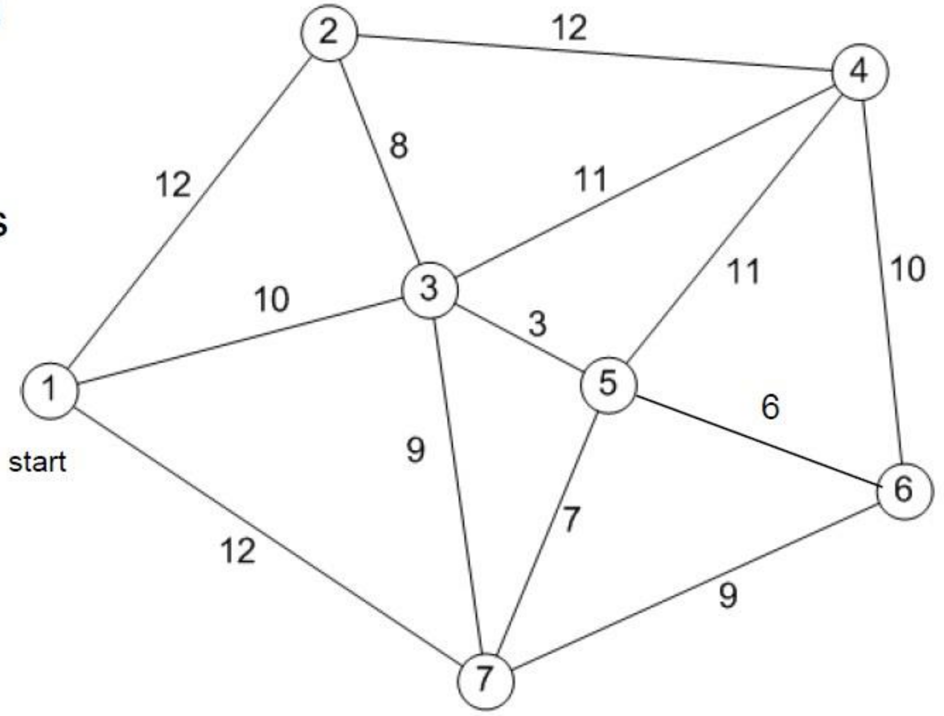
- Complexity classes group problems on the basis of their difficulty level, and the resources required in terms of time and space to solve them.
- P – polynomial time (linear search, bubble sort, etc.)
- NP – nondeterministic polynomial time (contain nondeterministic statements)
- NP-hard – a problem is NP-Hard if all other problems in NP can be polynomial-time-reducible, or mapped to it.
- NP-complete – a problem both NP and NP-hard
- More on this topic: https://www.youtube.com/watch?v=e2cF8a5aAhE



NP problem: Travelling salesman

The Traveling Salesman Problem

- The salesman must travel to all cities once before returning home
- The distance between each city is given, and is assumed to be the same in both directions
- Objective - Minimize the total distance to be travelled



Thank you!