University of Technology
الجامعة التكنولوجية

Computer Science Department
قسم علوم الحاسوب

Algorithms Analysis and Design
تحليل وتصميم الخوارزميات

Lect. Iman Shakir
م. إيمان شاكر

cs.uotechnology.edu.iq

1. Concepts and Properties of Algorithms

The concept of an algorithm is fundamental to computer science. Algorithms exist for many common problems, and designing efficient algorithms plays a crucial role in developing large-scale computer systems. Therefore, before we proceed further we need to discuss this concept more fully. We begin with a definition.

1.2 What Is an Algorithm?

Definition: An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.

The reference to "instructions" in the definition implies that there is something or someone capable of understanding and following the instructions given.

In addition, all algorithms must satisfy the following criteria:

1. **Input**. There are zero or more quantities that are externally supplied.
2. **Output**. At least one quantity is produced.
3. **Definiteness**. Each instruction is clear and unambiguous.
4. **Finiteness**. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness**. Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper.

It is not enough that each operation be definite as in (3); it also must be feasible.

Algorithm definition can be illustrated by a simple diagram (Figure 1).

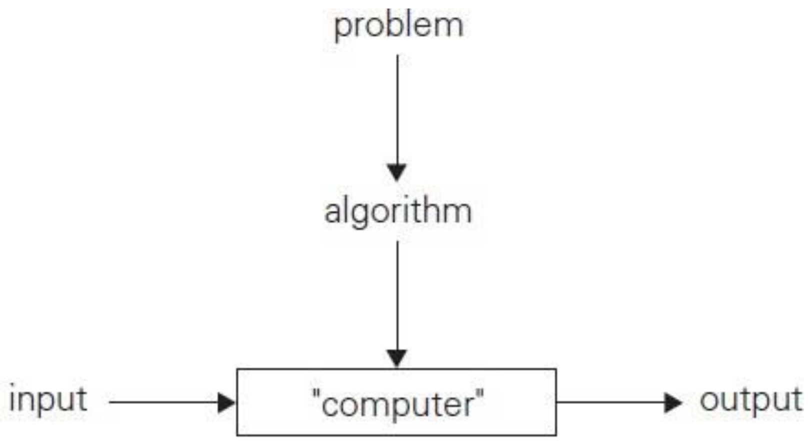

$$\begin{tikzcd} & \text{problem} \arrow[d] & \\ & \text{algorithm} \arrow[d] & \\ \text{input} \arrow[r] & \boxed{\text{"computer"}} \arrow[r] & \text{output} \end{tikzcd}$$

FIGURE 1.1 The notion of the algorithm.

For example, one might need to sort a sequence of numbers into increasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools. Here is how we formally define the ***sorting problem***:

- **Input:** A sequence of *n* numbers (*a*1, *a*2, ..., *an*).

- **Output:** A permutation (reordering) (*a*1', *a*2', ..., *an'*) of the input sequence such that a1'≤a2'≤…≤an'.

For example, given the input sequence (31, 41, 59, 26, 41, 58), a sorting algorithm returns as output the sequence (26, 31, 41, 41, 58, 59). Such an input sequence is called an ***instance*** of the sorting problem. In general, an ***instance of a problem*** consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

2. Fundamentals of Algorithmic Problem Solving

We can consider algorithms to be procedural solutions to problems. These solutions are not answers but specific instructions for getting answers. It is this emphasis on precisely defined constructive procedures that makes computer science distinct from other disciplines.

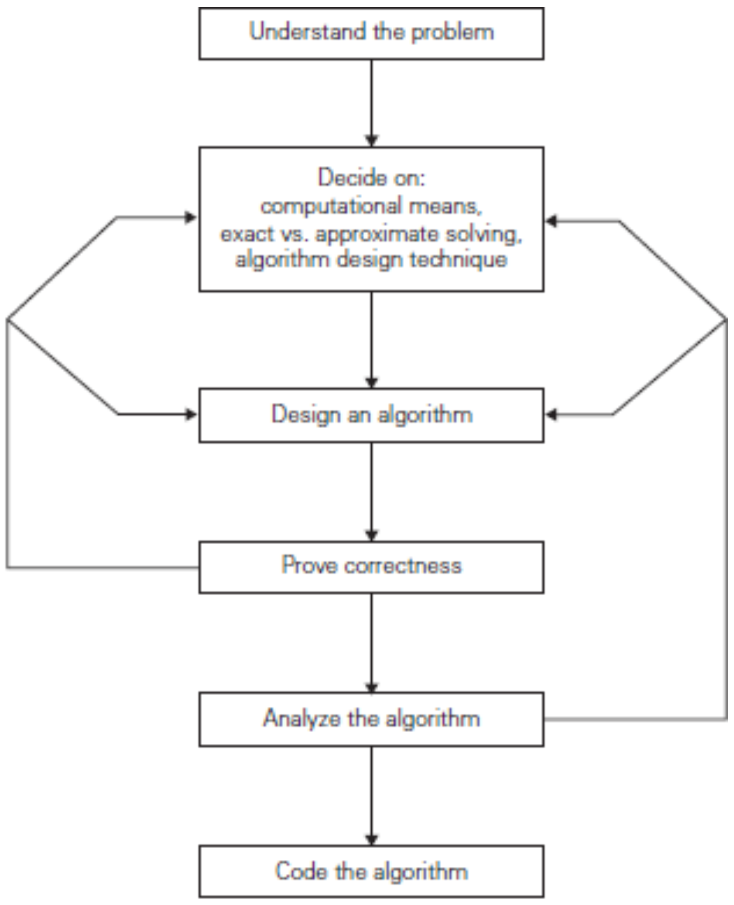


FIGURE 1.2 Algorithm design and analysis process.

2.1 Understanding the Problem

From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given. Read the problem's description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.

An input to an algorithm specifies an **instance** of the problem the algorithm solves. It is very important to specify exactly the set of instances the algorithm needs to handle.

2.2 Algorithm Design

The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space. To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient. However, one has to keep in mind that both time consumption and memory usage cannot be optimized simultaneously. If we require an algorithm to run in lesser time, we have to invest in more memory and if we require an algorithm to run with lesser memory, we need to have more time.

2.2.1 Methods of Specifying an Algorithm

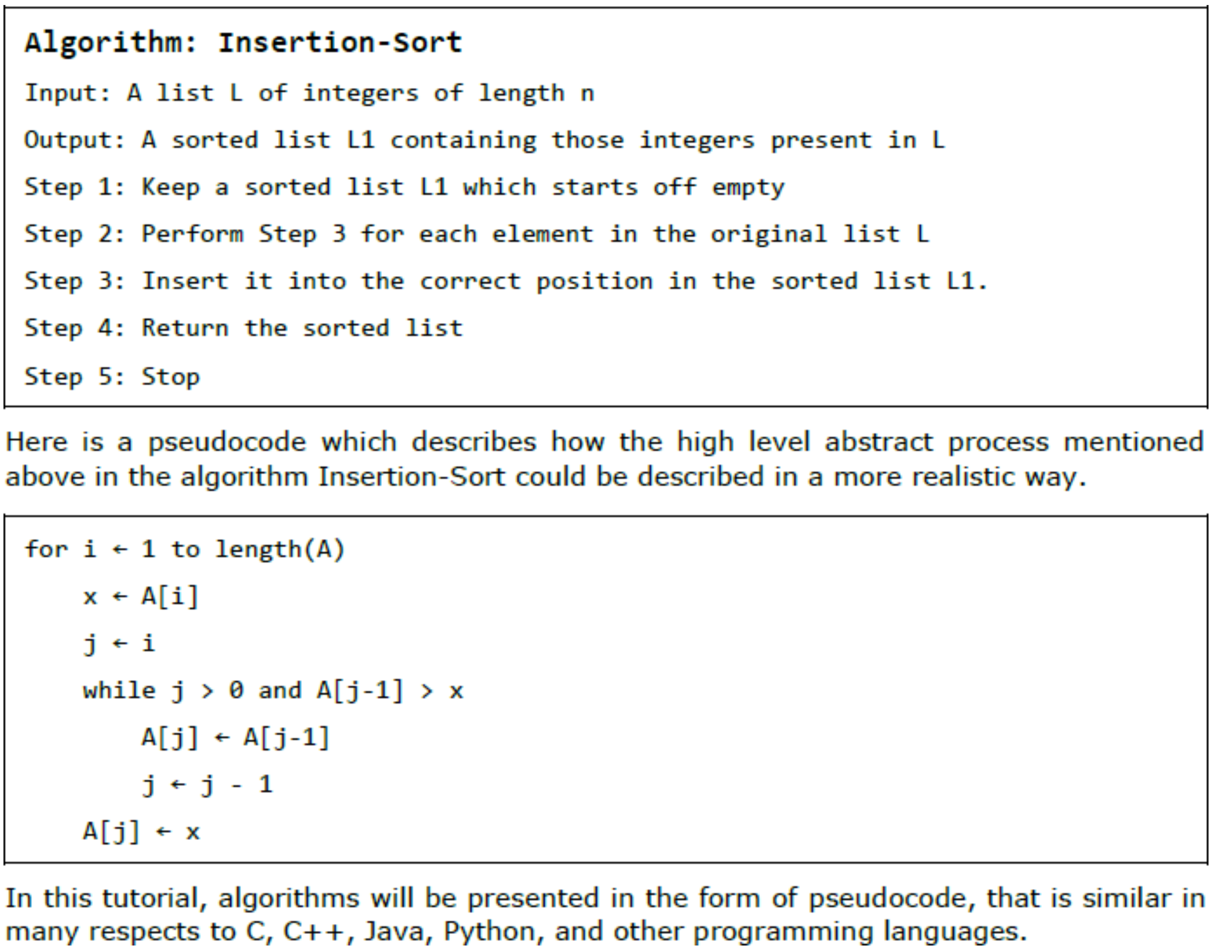
Pseudocode is a mixture of a natural language and programming language like constructs. Pseudocode is usually more precise than natural language, and its usage often yields more succinct algorithm descriptions.

In the earlier days of computing, the dominant vehicle for specifying algorithms was a ***flowchart***, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.

2.2.2 Difference between Algorithm and Pseudocode

An algorithm is a formal definition with some specific characteristics that describes a process, which could be executed to perform a specific task. Generally, the word "algorithm" can be used to describe any high level task in computer science. On the other hand, pseudocode is an informal and human readable description of an algorithm leaving many granular details of it. Writing a pseudocode has no restriction of styles and its only objective is to describe the high level steps of algorithm in a much realistic manner in natural language.

For example, following is an algorithm for Insertion Sort.



```
Algorithm: Insertion-Sort
Input: A list L of integers of length n
Output: A sorted list L1 containing those integers present in L
Step 1: Keep a sorted list L1 which starts off empty
Step 2: Perform Step 3 for each element in the original list L
Step 3: Insert it into the correct position in the sorted list L1.
Step 4: Return the sorted list
Step 5: Stop
```

Here is a pseudocode which describes how the high level abstract process mentioned above in the algorithm Insertion-Sort could be described in a more realistic way.

```
for i ← 1 to length(A)
    x ← A[i]
    j ← i
    while j > 0 and A[j-1] > x
        A[j] ← A[j-1]
        j ← j - 1
    A[j] ← x
```

In this tutorial, algorithms will be presented in the form of pseudocode, that is similar in many respects to C, C++, Java, Python, and other programming languages.

2.3 Proving an Algorithm's Correctness

Once an algorithm has been specified, you have to prove its correctness. That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.

Whenever we have an algorithm, there are three questions we always ask about it:

1. **Is it correct?**
   Will the algorithm stop? – Halting Problem.
   Given input and output specifications, will the algorithm function correctly with respect to input and output specifications?
2. **How much time does it take, as a function of n?**

   $T(n) \infty$ □(□). Worse case, best case and average case.

   Upper bound and lower bound?
3. **And can we do better?**
   In term of time efficiency.

2.4 Analysis of an Algorithm

We usually want our algorithms to possess several qualities. After correctness, by far the most important is efficiency. In fact, there are two kinds of algorithm efficiency: time efficiency, indicating how fast the algorithm runs, and space efficiency, indicating how much extra memory it uses.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time

or performance. Generally, we perform the following types of analysis:

- **Worst-case:** The maximum number of steps taken on any instance of size **n.**
- **Best-case:** The minimum number of steps taken on any instance of size **n.**
- **Average case:** An average number of steps taken on any instance of size **n.**

```
ALGORITHM  SequentialSearch($A[0..n-1]$, $K$)
    //Searches for a given value in a given array by sequential search
    //Input: An array $A[0..n-1]$ and a search key $K$
    //Output: The index of the first element in $A$ that matches $K$
    //         or $-1$ if there are no matching elements
    $i \leftarrow 0$
    while $i < n$ and $A[i] \neq K$ do
        $i \leftarrow i + 1$
    if $i < n$ return $i$
    else return $-1$
```

2.5 Coding an Algorithm

Most algorithms are destined to be ultimately implemented as computer programs. Programming an algorithm presents both a peril and an opportunity. The peril lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently. Some influential computer scientists strongly believe that unless the correctness of a computer program is proven with full mathematical rigor, the program cannot be considered correct.

3. Algorithm Complexity

3.1 Running Time Functions

Most algorithms have a primary parameter N, usually the number of data items to be processed, which affects the running time most significantly. The parameter N might be the degree of a polynomial, the size of a file to be sorted or searched, the number of nodes in a graph, etc. proportional to one of the following functions:

1. **(1)** Most instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is constant. This is obviously the situation to strive for algorithm design.

2. **(log N)** When the running time of a program is logarithmic, the program gets slightly slower as N grows. This running time commonly occurs in programs which solve a big problem by transforming it into a smaller problem by cutting the size by some constant fraction. For our range of interest, the running time can be considered to be less than a **large** constant. The base of the logarithm changes the constant, but not by much: when N is a thousand, log N is 3 if the base is 10, 10 if the base is 2; when N is a million, 1og N is twice as great. Whenever N doubles, log N increases by a constant, but log N doesn't double until N increases to N^2.

3. (**N)** When the running time of a program is linear, it generally is the case that a small amount of processing is done on each input

   element. When N is a million, then so is the running time. Whenever N doubles, then so does the running time. This is the optimal situation for an algorithm that must process N inputs (or produce N outputs).

4. (N **log N)** This running time arises in algorithms which solve a

   problem by breaking it up into smaller sub problems, solving them independently, and then combining the solutions. For lack of a better adjective (linearithmic), we'll say that the running time of such an algorithm is "N log N." When N is a million, N log N is perhaps twenty million. When N doubles, the running times more than doubles (but not much more).

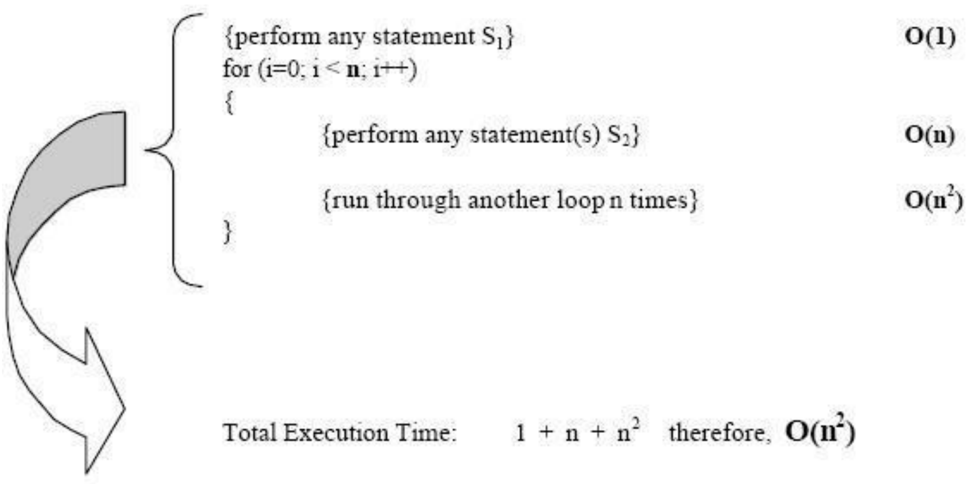
5. **(N^2)** When the running time of an algorithm is *quadratic,* it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms which process all pairs of data items (perhaps in a double nested loop). When N is a thousand, the running time is a million. Whenever N doubles, the running time increases fourfold.

6. **(N^3)** Similarly, an algorithm which processes triples of data items (perhaps in a triple-nested loop) has a *cubic* running time and is practical for use only on small problems. When N is a hundred, the running time is a million. Whenever N doubles, the running time increases eightfold.

7. **(2^N)** Few algorithms with *exponential* running time are likely to be appropriate for practical use, though such algorithms arise naturally as "brute-force" solutions to problems. When N is twenty, the running time is a million. Whenever N doubles, the running time squares.

Example:

.8



```
{perform any statement S1}                        O(1)
for (i=0; i < n; i++)
{
        {perform any statement(s) S2}             O(n)

        {run through another loop n times}        O(n^2)
}
```

Total Execution Time: $1 + n + n^2$ therefore, $O(n^2)$

3.2 Space complexity

The (space) complexity of a program (for a given input) is the number of elementary objects that this program needs to store during its execution. This number is computed with respect to the size n of the input data

Space complexity is measured by using polynomial amounts of memory, with an infinite amount of time.

The difference between space complexity and time complexity is that space can be reused. Space complexity is not affected by determinism or non determinism. Amount of computer memory required during the program execution, as a function of the input size

How to Determine Complexities

In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.

1. Sequence of statements

```
statement 1;

statement 2;

  ...

statement k;
```

(Note: this is code that really is exactly k statements; this is **not** an unrolled loop like the N calls to *add* shown above.) The total time is found by adding the times for all statements:

Total time = time (statement 1) + time (statement 2) + ... + time (statement k)

If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: O(1). In the following examples, assume the statements are simple unless noted otherwise.

2- if-then-else statements

```
if (condition) {

 Sequence of statements 1

    }

  else {

 sequence of statements 2

}
```

Here, either sequence 1 will execute, or sequence 2 will execute. Therefore, the worst-case time is the slowest of the two possibilities: max (time (sequence 1), time (sequence 2)). For example, if sequence 1 is O(N) and sequence 2 is O(1) the worst-case time for the whole if-then-else statement would be O(N).

3- for loops

```
for (i = 0; i < N; i++) {

 sequence of statements

}
```

The loop executes N times, so the sequence of statements also executes N times. Since we assume the statements are O(1), the total time for the for loop is N * O(1), which is O(N) overall.

4- Nested loops

First we'll consider loops where the number of iterations of the inner loop is independent of the value of the outer loop's index. For example:

```
for (i = 0; i < N; i++) {

   for (j = 0; j < M; j++) {

      sequence of statements

   }

}
```

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of N * M times. Thus, the complexity is O(N * M). In a common special case where the stopping condition of the inner loop is j < N instead of j < M (i.e., the inner loop also executes N times), the total complexity for the two loops is O(N^2).

Now let's consider nested loops where the number of iterations of the inner loop depends on the value of the outer loop's index. For example:

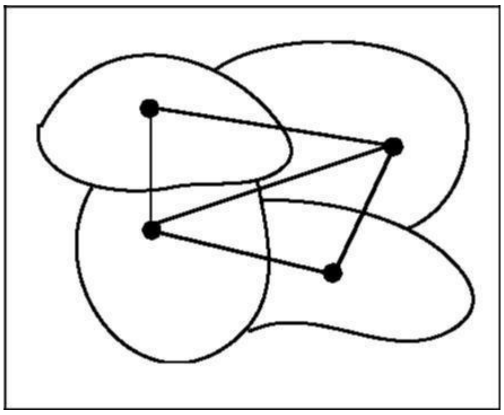
```
for (i = 0; i < N; i++) {

for (j = i+1; j < N; j++) {
        sequence of statements
    }
}
```

4. Easy and Hard Problems

We argue that the class of problems that can be solved in polynomial time (denoted by P) corresponds well with what we can feasibly compute. But sometimes it is difficult to tell when a particular problem is in P or not.

Theoreticians spend a good deal of time trying to determine whether particular problems are in P. To demonstrate how difficult it can be. To make this determination, we will survey a number of problems, some of which are known to be in P, and some of which we think are (probably) not in P. The difference between the two types of problem can be surprisingly small. Throughout the following, an ***"easy"*** problem is one that is solvable in polynomial time; while a ***"hard"*** problem is one that we think cannot be solved in polynomial time. When we say that a problem is hard, it means that **some** instances of the problem **are hard**. It does **not** mean that **all** problem instances **are hard**.

4.1 Color Map

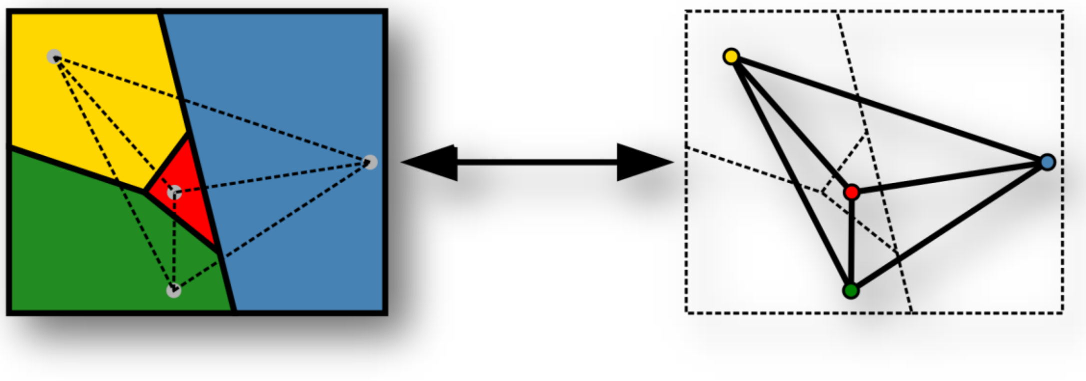


- **2 Color Map (Easy)**
  - INPUT: A graph G=(V, E).
  - DECIDE: Can this map be Colored with 2 colors so that no two adjacent countries have the same color?
- **3 Color Map 3 (Hard)**
  - INPUT: A graph G=(V, E).
  - DECIDE: Can this map be colored with 3 colors so that no two adjacent countries have the same color?

- **4 Color Map 4- (Easy)**
  - INPUT: A graph G=(V, E).
  - DECIDE: Can this map be colored with 4 colors so that no two adjacent countries have the same color?

Some Facts

- **Map 2-colorability**
  - To solve this problem, we simply color the first country arbitrarily. This forces the colors of neighboring countries to be the other color, which in turn forces the color of the countries neighboring those countries, and so on. If we reach a country which borders two countries of different color, we will know that the map cannot be two-colored; otherwise, we will produce a two coloring. So this problem is easily solvable in polynomial time.
- **Map 3-colorability**
  - This problem seems very similar to the problem above, however, it turns out to be much harder. No one knows how this problem can be solved in polynomial time. (In fact this problem is NP-complete.)
- **Map 4-colorability**.
  - Here we have an easy problem again. By a famous theorem, any map can be four-colored. It turns out that finding such a coloring is not that difficult either.



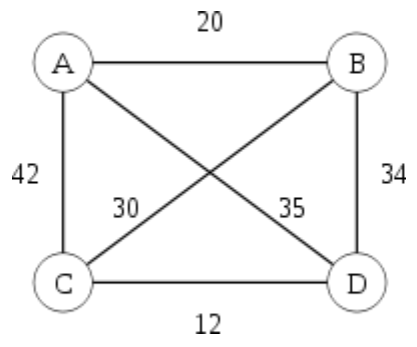
4.2 Hard problems

Our usual measure of efficiency is speed, i.e., how long an algorithm takes to produce its result. There are some problems, however, for which no efficient solution is known. Studies an interesting subset of these problems, which are known as NP-complete.

Why are NP-complete problems interesting? First, although no efficient algorithm for an NP complete problem has ever been found, nobody has ever proven that an efficient algorithm for one cannot exist. In other words, it is unknown whether or not efficient algorithms exist for NP-complete problems. Second, the set of NP-complete problems has the remarkable property that if an efficient algorithm exists for any one of them, then efficient algorithms exists for all of them. This relationship among the NP-complete problems makes the lack of efficient solutions all the more tantalizing. Third, several NP-complete problems are similar, but not identical, to problems for which we do know of efficient algorithms. A small change to the problem statement can cause a big change to the efficiency of the best known algorithm.

5. Travelling salesman problem

The travelling salesman problem (also called the travelling salesperson problem or TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in theoretical computer science and operations research.



TSP can be modelled as an undirected weighted graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's weight. It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once. Often, the model is a complete graph (*i.e.* each pair of vertices is connected by an edge). If no path exists between two cities, adding an arbitrarily long edge will complete the graph without affecting the optimal tour.

Asymmetric and symmetric

In the ***symmetric*** *TSP*, the distance between two cities is the same in each opposite direction, forming an **undirected** graph. This symmetry halves the number of possible solutions. In the *asymmetric TSP*, paths may not exist in both directions or the distances might be different, forming a directed graph. Traffic collisions, one-way streets, and airfares for cities with different departure and arrival fees are examples of how this symmetry could break down.

6. Strategies in Algorithm Design

One very important aspect of problem-solving is devising good strategies. Indeed, there are many strategies for algorithm design.

1. Iteration
2. Recursion
3. 4- Color mapping
4. Traveling Salesman
5. Shortest Path
6. Brute force algorithm
7. Greedy algorithm
8. Divide and conquer
9. Dynamic programming
10. Network flow
11. Branch and bound
12. Heuristics

Algorithm: An algorithm is a sequence of computational steps that transform the input to an output. It is a tool for ***solving*** a well-specified computational ***problem***.

Strategy: A strategy is an approach (or a series of approaches) devised to ***solve*** a computational ***problem***.

Since both are intended to solve computational problems, how are they related? Simply put:
An algorithm is a strategy that always **guarantees** *the correct answer.*
How are they different?

1. A strategy might yield incorrect results, but a correct algorithm will always produce correct results.
2. Strategies are invented, algorithms are more or less tested and trusted standards

3. Strategies are flexible, but algorithms are rigid i.e. they follow only one set of procedures

6.1 Iteration

Iteration involves repeating a block of code until a condition is false. During iteration, the program makes multiple passes through a block of code.
Iteration can be achieved using loops or recursion (more on this later). The basic loop constructs are:

- The for loop
- The for-each loop
- The while loop
- The do-while loop

6.2 Recursion

Recursion is repetition achieved through method calls. A recursive method makes repeated calls to itself before returning a result. A result is returned if and only if a base case exists.
This base case ensures that the solution converges otherwise an infinite recursion occurs which in turn leads to a ***Stack Overflow.***
Recursion is intuitive because each new method call works on clones of the original problem leading to a final result (if it converges).

Example:
Find the factorial of any positive integer n.
Analysis
Mathematically, the factorial of a positive number, n, is the product of all the consecutive numbers from 1 to n. Thus, the formula is:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$

For example, if n = 3, 4, and 5, then the result in Figure bellow is expected.

$$3! = 3 \cdot 2 \cdot 1 = 6$$
$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$
$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

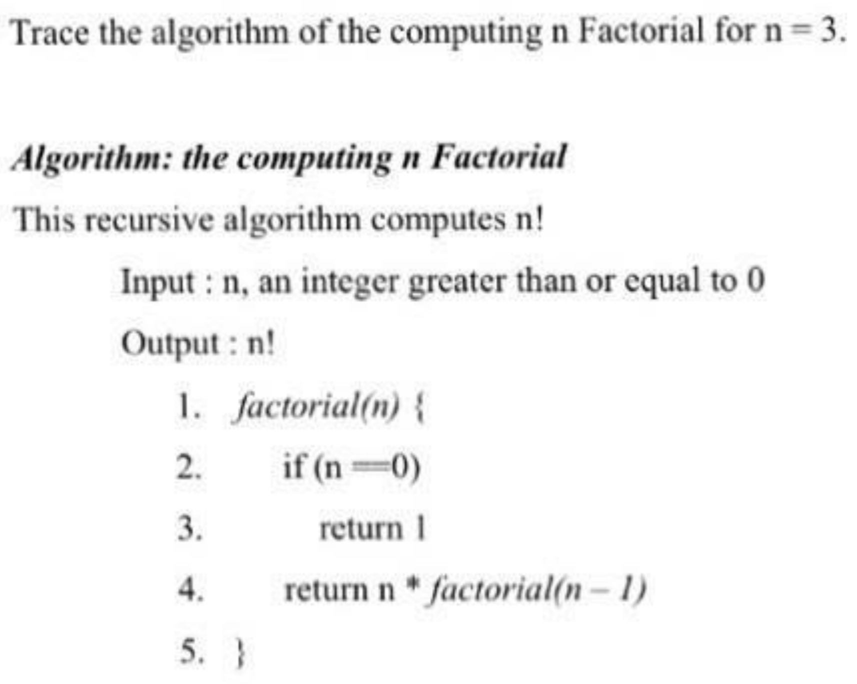
Using Iteration

This solution can be achieved in a variety of ways with iteration. pseudocode makes use while loop and the reduce function. The loop is repeated n times. Thus, the time complexity for the factorial function is O(n).

```
Factorial(n)
{
     i ← 1
     factorial ← 1
     while ( i ≤ n )
     {
          factorial ← factorial * i
          i ← i + 1
     }
     return factorial
}
```

Using Recursion

To better understand how recursion would work for this problem, insight is needed. Notice that factorial function in above is simply calling itself with smaller values of n. Thus, when the result of smaller subproblems is known, we can easily compute the result of other higher problems. This is highlighted in algorithm below. Again, since at most n method calls are made during recursion, the time complexity is O(n).



Trace the algorithm of the computing n Factorial for n = 3.

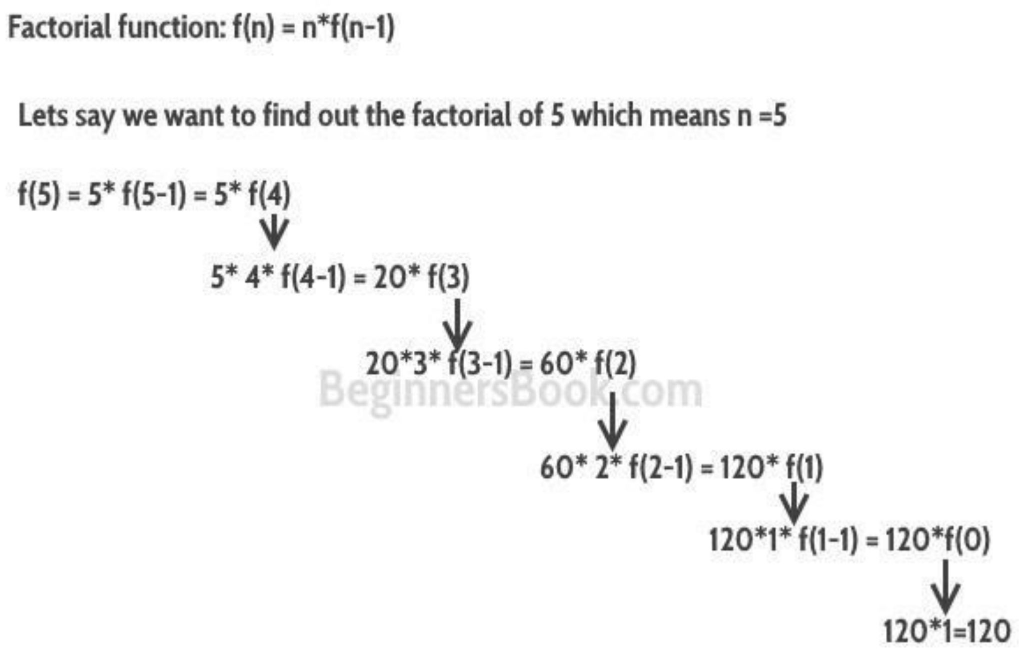
Algorithm: the computing n Factorial

This recursive algorithm computes n!

Input : n, an integer greater than or equal to 0

Output : n!

```
1. factorial(n) {
2.     if (n ==0)
3.         return 1
4.     return n * factorial(n – 1)
5. }
```



Factorial function: f(n) = n*f(n-1)

Lets say we want to find out the factorial of 5 which means n =5

f(5) = 5* f(5-1) = 5* f(4)

5* 4* f(4-1) = 20* f(3)

20*3* f(3-1) = 60* f(2)

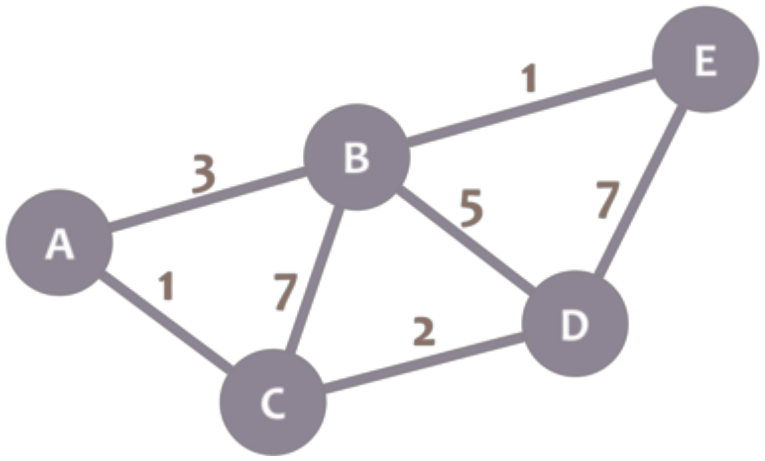
60* 2* f(2-1) = 120* f(1)

120*1* f(1-1) = 120*f(0)

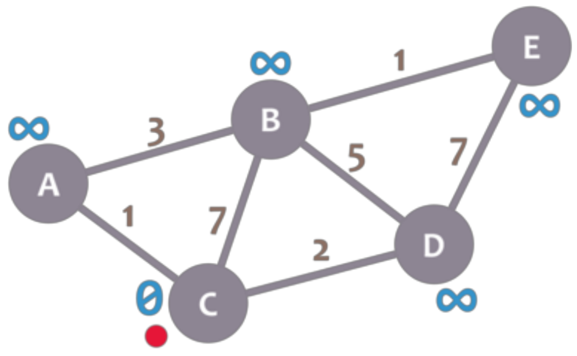
120*1=120

7. Shortest Path (Dijkstra's Algorithm):

Dijkstra's Algorithm allows you to calculate the shortest path between one node (you pick which one) and *every other node in the graph*. You'll find a description of the algorithm at the end of this page, but, let's study the algorithm with an explained example! Let's calculate the shortest path between node C and the other nodes in our graph:



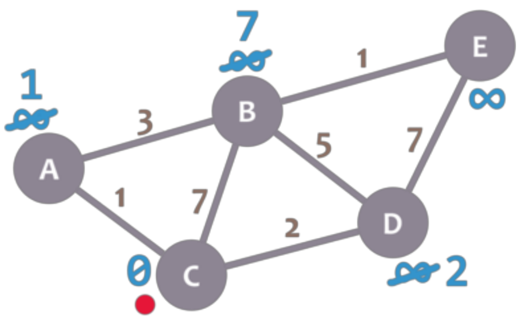
During the algorithm execution, we'll mark every node with its *minimum distance* to node C (our selected node). For node C, this distance is 0. For the rest of nodes, as we still don't know that minimum distance, it starts being infinity (∞):



vertex	Shorter distance from C	Previous vertex
A	∞	
B	∞	
C	**0**	
D	∞	
E	∞	

Visited =[ ] **Unvisited=[ABCDE]**

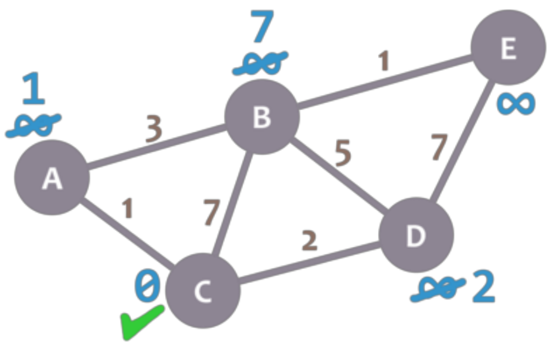
We'll also have a *current node*. Initially, we set it to C (our selected node). In the image, we mark the current node with a red dot. Now, we check the neighbors of our current node (A, B and D) in no specific order. Let's begin with B. We add the minimum distance of the current node (in this case, 0) with the weight of the edge that connects our current node with B (in this case, 7), and we obtain 0 + 7 = 7. We compare that value with the minimum distance of B (infinity); the lowest value is the one that remains as the minimum distance of B (in this case, 7 is less than infinity). Repeat the same procedure for A, D:



vertex	Shorter distance from C	Previous vertex
A	1	C
B	7	C
C	0	
D	2	C
E	∞	

Visited =[] **Unvisited=[ABDE]**

We have checked all the neighbors of C. Because of that, we mark it as visited. Let's represent visited nodes with a green check mark:



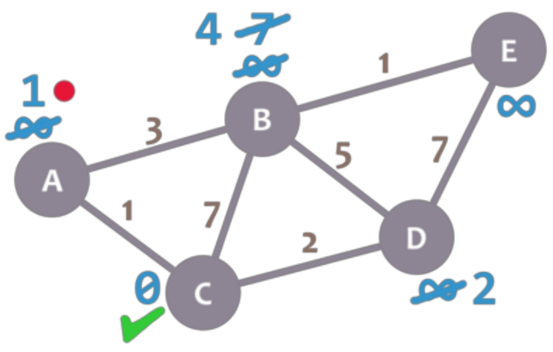
vertex	Shorter distance from C	Previous vertex
A	1	C
B	7	C
C	0	
D	2	C
E	∞	

Visited =[C] **Unvisited=[ABDE]**

We now need to pick a new *current node*. That node must be the unvisited node with the smallest minimum distance (so, the node with the smallest number and no check mark). That's A. Let's mark it with the red dot:

And now we repeat the algorithm. We check the neighbors of our current node, ignoring the visited nodes. This means we only check B.

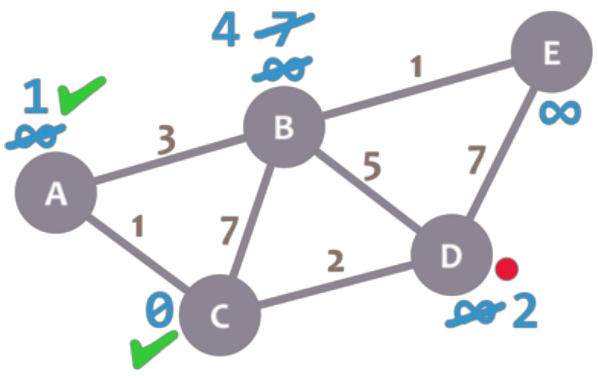
For B, we add 1 (the minimum distance of A, our current node) with 3 (the weight of the edge connecting A and B) to obtain 4. We compare that 4 with the minimum distance of B (7) and leave the smallest value: 4.



vertex	Shorter distance from C	Previous vertex
A	1	C
B	4	A
C	0	
D	2	C
E	∞	

Visited =[C] **Unvisited=[ABDE]**

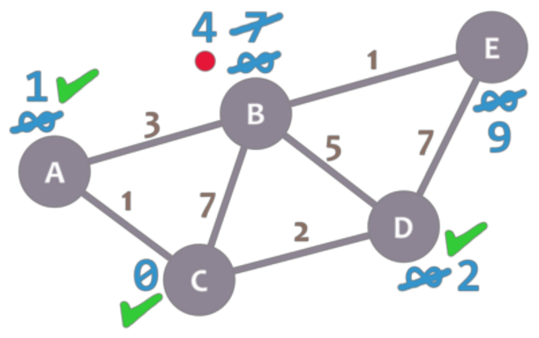
Afterwards, we mark A as visited and pick a new current node: D, which is the non-visited node with the smallest current distance. We repeat the algorithm again. This time, we check B and E.



vertex	Shorter distance from C	Previous vertex
A	1	C
B	4	A
C	0	
D	2	C
E	∞	

Visited =[CA] **Unvisited=[BDE]**

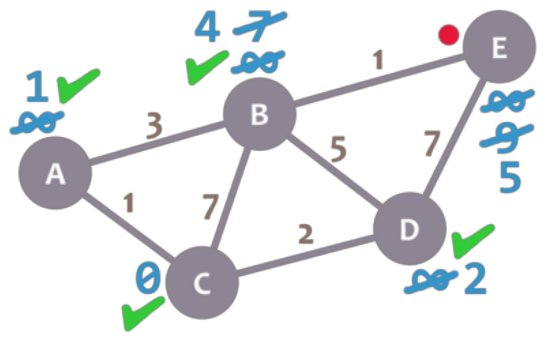
For B, we obtain 2 + 5 = 7. We compare that value with B's minimum distance (4) and leave the smallest value (4). For E, we obtain 2 + 7 = 9, compare it with the minimum distance of E (infinity) and leave the smallest one.)9(We mark D as visited and set our current node to B.



vertex	Shorter distance from C	Previous vertex
A	1	C
B	4	A
C	0	
D	2	C
E	9	D

Visited =[CAD] **Unvisited=[BE]**

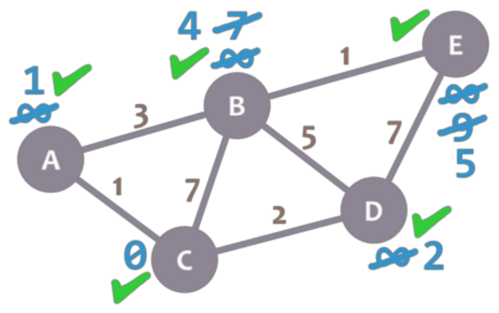
Almost there. We only need to check E. 4 + 1 = 5, which is less than E's minimum distance (9), so we leave the 5. Then, we mark B as visited and set E as the current node.



vertex	Shorter distance from C	Previous vertex
A	1	C
B	4	A
C	0	
D	2	C
E	5	B

Visited =[CADB] **Unvisited=[E]**

E doesn't have any non-visited neighbors, so we don't need to check anything. We mark it as visited.



vertex	Shorter distance from C	Previous vertex
A	1	C
B	4	A
C	0	
D	2	C
E	5	B

Visited =[CADBE] **Unvisited=[]**

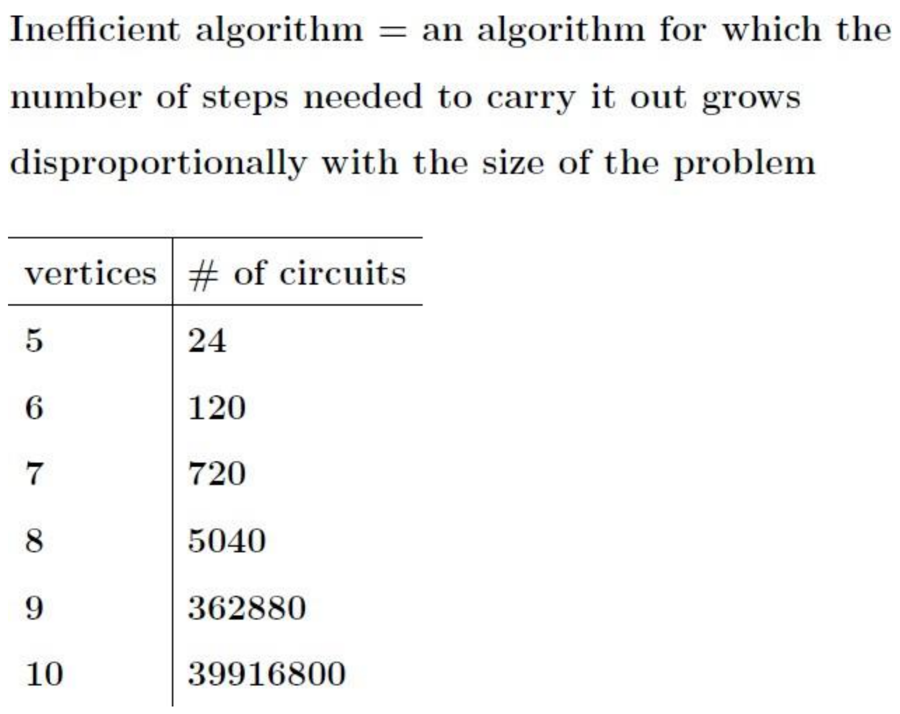
8- Brute Force Algorithm

The brute force solution is simply to calculate the total distance for every possible route and then select the shortest one. This is not particularly efficient because it is possible to eliminate many possible routes through clever algorithms.

The time complexity of brute force is O(mn), which is sometimes written as O(n*m) . So, if we were to search for a string of "n" characters in a string of "m" characters using brute force, it would take us n * m tries.

Advantages: Guaranteed to find the most efficient circuit.

Disadvantages: Can be a lot of work to carry out the algorithm.



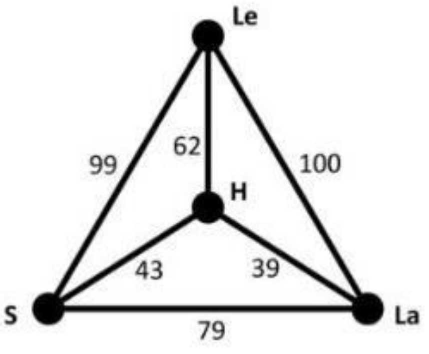
Inefficient algorithm = an algorithm for which the number of steps needed to carry it out grows disproportionally with the size of the problem

vertices	# of circuits
5	24
6	120
7	720
8	5040
9	362880
10	39916800

The Brute Force Algorithm finds the weight of every Hamilton Circuit and chooses the cheapest one.

Example:

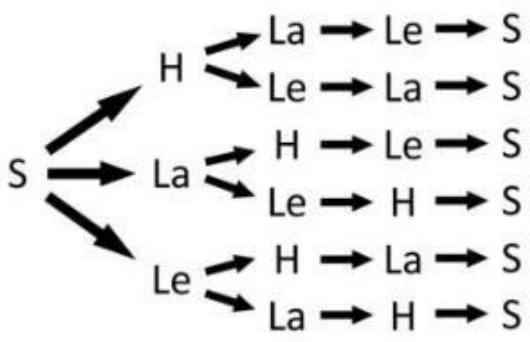
- Suppose you want to take a road trip for Spring Break
- You want to start from Shippensburg (S), and visit Harrisburg (H), Lancaster (La), and Lewisburg (Le) in some order before returning to Ship



Solution:

1. **Find all possible circuit**

- And finally, we must return to S



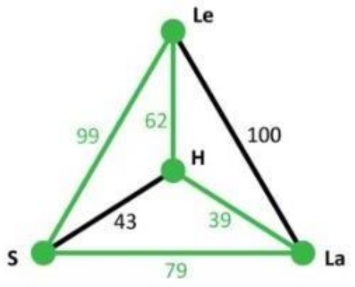
2. **Find the cost for each circuit**

Circuit	Cost
S – H – La – Le – S	43 + 39 + 100 + 99 = 281
S – H – Le – La – S	43 + 62 + 100 + 79 = 284
S – La – H – Le – S	79 + 39 + 62 + 99 = 279
S – La – Le – H – S	79 + 100 + 62 + 43 = 284
S – Le – H – La – S	99 + 62 + 39 + 79 = 279
S – Le – La – H – S	99 + 100 + 39 + 43 = 281

3- Choose the lowest cost circuit.

Circuit	Cost
S – H – La – Le – S	43 + 39 + 100 + 99 = 281
S – H – Le – La – S	43 + 62 + 100 + 79 = 284
S – La – H – Le – S	79 + 39 + 62 + 99 = 279
S – La – Le – H – S	79 + 100 + 62 + 43 = 284
S – Le – H – La – S	99 + 62 + 39 + 79 = 279
S – Le – La – H – S	99 + 100 + 39 + 43 = 281

- If we draw these two circuits, we find that in fact they are the same
- One circuit is the reverse of the other, so the total costs are the same
- In fact, while it looked like there were 6 total circuits, really there were only 3

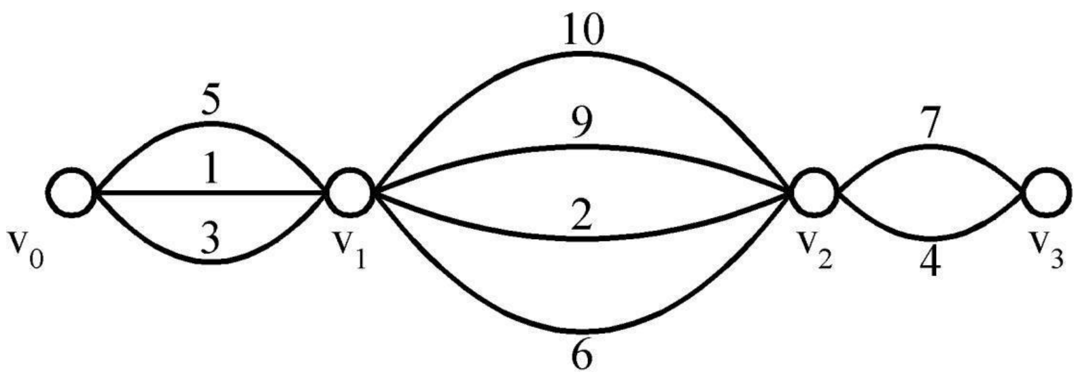


9- Greedy Algorithm

Suppose that a problem can be solved by a sequence of decisions. The greedy method has that <u>each decision is locally optimal. These locally optimal solutions</u> <u>will finally add up to a globally optimal solution.</u> Only a few optimization problems can be solved by the greedy method.

- **Shortest paths on a special graph**

<u>Problem</u>: Find a shortest path from v_0 to v_3.

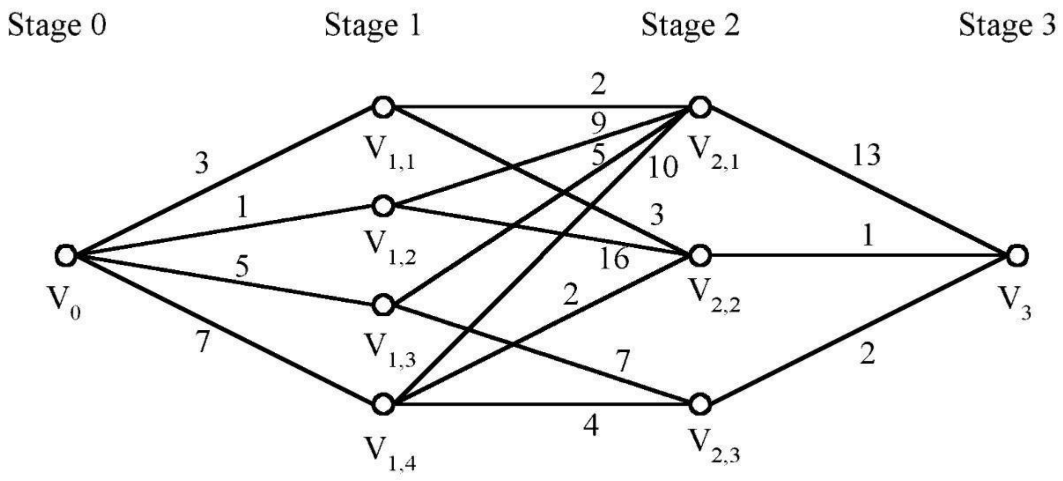


The greedy method can solve this problem.

The shortest path: 1 + 2 + 4 = 7.

Shortest paths on a multi-stage graph

Problem: Find a shortest path from v_0 to v_3 in the multi-stage graph.



Greedy method: $v_0v_{1,2}v_{2,1}v_3 = 23$

Optimal: $v_0v_{1,1}v_{2,2}v_3 = 7$

The greedy method does not work.

10- Divide and Conquer

The ***divide and conquer*** strategy solves a problem by:

1. Breaking into ***sub problems*** that are themselves smaller instances of the same type of problem.

2. Recursively solving these sub problems.

3. Appropriately combining their answers.

Two types of sorting algorithms which are based on this divide and conquer algorithm:

1. **Quick sort:** Quick sort also uses few comparisons. Like heap sort it can sort "in place" by moving data in an array.

2. **Merge sort:** Merge sort is good for data that's too big to have in memory at once, because its pattern of storage access is very regular. It also uses even fewer comparisons than heap sort, and is especially suited for data stored as linked lists.

Quick sort

Pivot element can be any element from the array, it can be the first element, the last element or any random element. In this example, we will take the rightmost element or the last element as pivot.

Quick Sort Algorithm: Steps on how it works:

1. Find a "pivot" item in the array. This item is the basis for comparison for a single round.
2. Start a pointer (the left pointer) at the first item in the array.
3. Start a pointer (the right pointer) at the last item in the array.

4. While the value at the left pointer in the array is less than the pivot value, move the left pointer to the right (add 1). Continue until the value at the left pointer is greater than or equal to the pivot value.
5. While the value at the right pointer in the array is greater than the pivot value, move the right pointer to the left (subtract 1). Continue until the value at the right pointer is less than or equal to the pivot value.
6. If the left pointer is greater than or equal to the right pointer, then swap the values at these locations in the array.
7. Move the left pointer to the right by one and the right pointer to the left by one.
8. If the left pointer and right pointer don't meet, go to step 1.

Complexity of Quicksort

Best case:

Set up a recurrence relation for T(n), the time needed to sort a list of size n. Because a single quicksort call involves O(n) work plus two recursive calls on lists of size n/2 in the best case, the relation would be:

$T(n) = O(n) + 2T(n/2)$

The master theorem tells us that $T(n) = O(n \log n)$.

Average case:

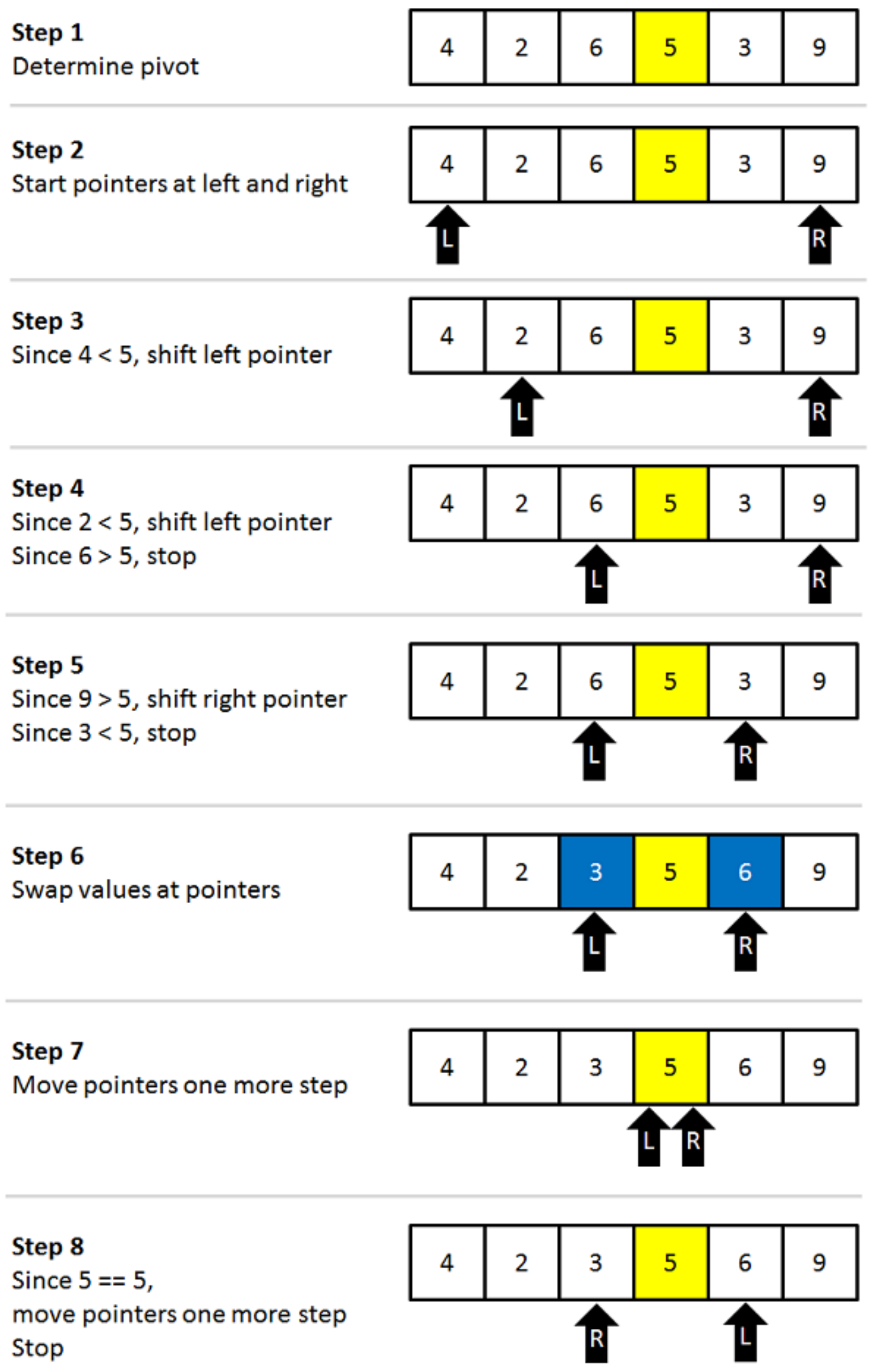
To sort an array of n distinct elements, quicksort takes O(n log n) time in expectation, averaged over all n! permutations of n elements with equal probability. We list here three common proofs to this claim providing different insights into quicksort's workings.

Worst case:

In the worst case, however, the two sublists have size 1 and n-1, and the call tree becomes a linear chain of n nested calls. The recurrence relation is:

T(n) = O(n) + T(1) + T(n - 1) = O(n) + T(n - 1)

This is the same relation as for insertion sort and selection sort, and it solves to T(n) = O(n²).


Step 1
Determine pivot

Step 2
Start pointers at left and right

Step 3
Since 4 < 5, shift left pointer

Step 4
Since 2 < 5, shift left pointer
Since 6 > 5, stop

Step 5
Since 9 > 5, shift right pointer
Since 3 < 5, stop

Step 6
Swap values at pointers

Step 7
Move pointers one more step

Step 8
Since 5 == 5,
move pointers one more step
Stop

Quick sort algorithm:

```
Algorithm quicksort(q)
var list less, pivotList, greater
if length(q) $\leq 1$
return q
else
select a pivot value from q
for each x in q except the pivot element
if $x <$ pivot then add x to less
if $x \geq$ pivot then add x to greater
add pivot to pivotList
return concatenate(quicksort(less), pivotList, quicksort(greater))
```

11- Dynamic programming

The key idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution. Often, many of these subproblems are really the same. The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations: once the solution to a given subproblem has been computed, it is stored, the next time the same solution is needed, it is simply looked up. This approach is especially useful when the number of repeating subproblems **grows exponentially** as a function of the size of the input.

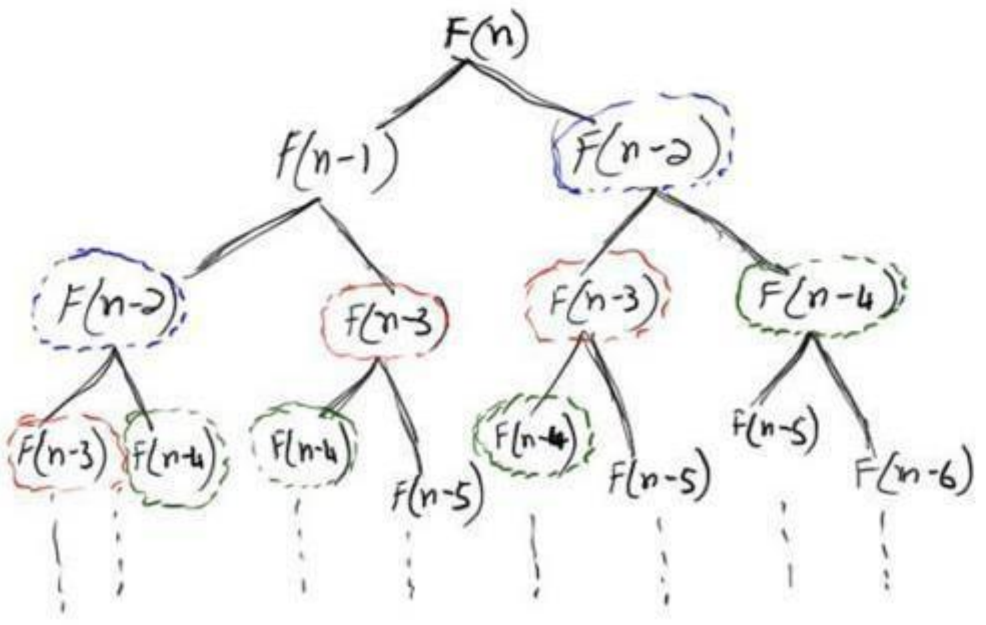
There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping sub problems. If a problem can be solved by combining optimal solutions to *non-overlapping* sub problems, the strategy is called "divide and conquer". This is why merge sort and quick sort are not classified as dynamic programming problems. Such optimal substructures are usually described by means of recursion.

A good example for a problem that has overlapping sub-problem is the relation for Nth Fibonacci number.

It is defined as **F(n)= F(n-1) + F (n-2) .**

Note that the Nth Fibonacci number depends on previous two Fibonacci number.

If we compute F(n) in conventional way, we have to calculate in following manner



The similar colored values are those that will be calculated again and again. Note that F(n-2) is computed 2 times, F(n-3) 3 times and so on … Hence, we are wasting a lot of time. In fact this recursion will perform 2^N operations for a given N.

Hence, dynamic programming is a very important technique to speed up the problems that have overlapping sub problems.

Example One of the most popular examples used to introduce recursion and induction is the problem of computing the Fibonacci sequence:

$$f_1 = 1, f_2 = 1, f_3 = 2, f_4 = 3, f_5 = 5, f_6 = 8, f_7 = 13, \ldots.$$

Each number in the sequence $2, 3, 5, 8, 13, \ldots$ is the the sum of the two preceding

numbers. Consider the inductive definition of this sequence:

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ f(n-1) + f(n-2) & \text{if } n \geq 3. \end{cases}$$

This definition suggests a recursive procedure that looks like the following (assuming that the input is always positive).

```
1. procedure f(n)
2.   if (n = 1) or (n = 2) then return 1
3.   else return f(n − 1) + f(n − 2)
```

12- Network Flow Problem

Definition. *A network is an edge-capacitated directed graph, with two distinguished vertices called the source and the sink.*

To repeat that, this time a little more slowly, suppose first that we are given a directed graph (*digraph*) *G*. That is, we are given a set of vertices, and a set of *ordered* pairs of these vertices, these pairs being the *edges* of the digraph. It is perfectly OK to have both an edge from *u* to *v* and an edge from *v* to *u*, or both, or neither, for all $u \neq v$. No edge (u, u) is permitted. If an edge *e* is directed *from* vertex *v to* vertex *w*, then *v* is the *initial* vertex of *e* and *w* is the *terminal* vertex of *e*. We may then write $v = Init(e)$ and $w = Term(e)$.

Next, in a network there is associated with each directed edge *e* of the digraph a positive real number called its *capacity*, and denoted by $c(e)$. Finally, two of the vertices of the digraph are distinguished. One, *s*, is the source, and the other, *t*, is the sink of the network.

Definition. *A flow in a network* **X** *is a function f that assigns to each edge e of the network a real number*

f(*e*)*, in such a way that*

(1) For each edge e we have $0 \leq f(e) \leq cap(e)$ *and*

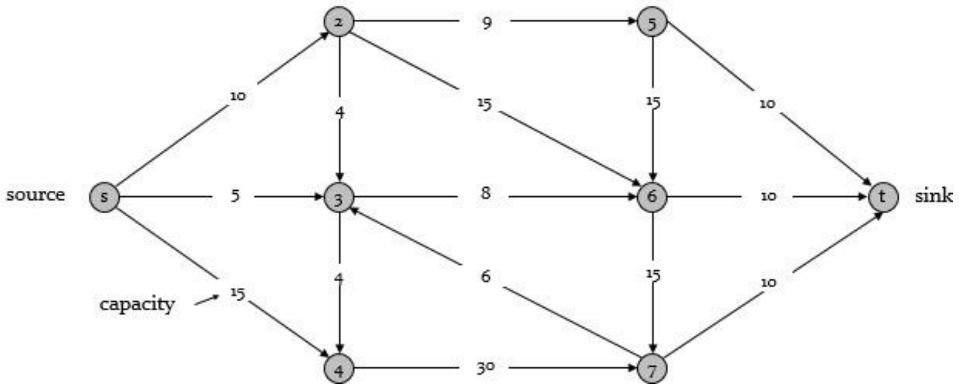
(2) For each vertex v other than the source and the sink, it is true that

$$\sum_{Init(e)=v} f(e) = \sum_{Term(e)=v} f(e).$$

Minimum Cut Problem

Flow network.

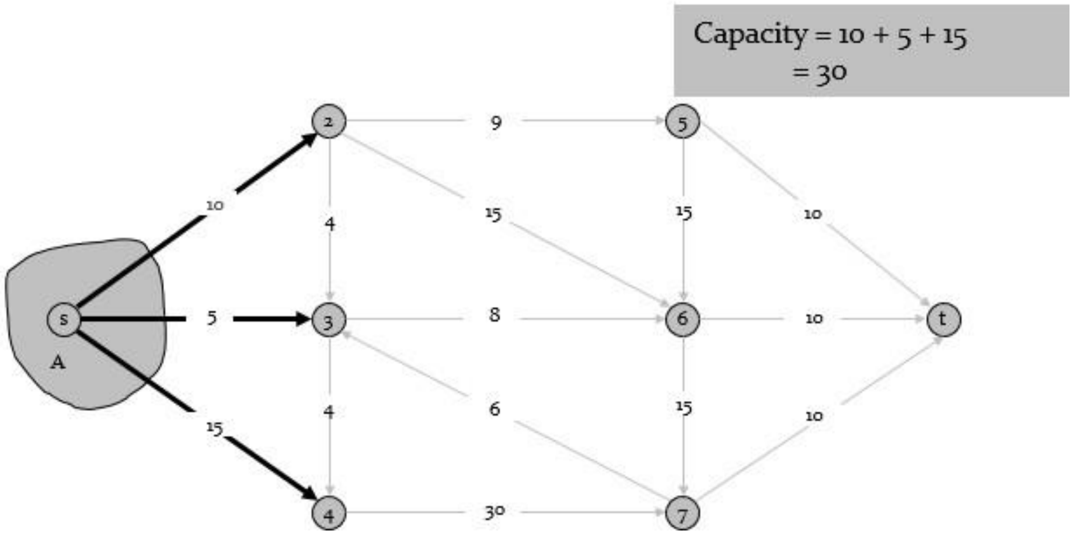
- Abstraction for material flowing through the edges.
- G = (V, E) = directed graph, no parallel edges.
- Two distinguished nodes: s = source, t = sink.
- c(e) = capacity of edge e.



Cuts

Def. An s-t cut is a partition (A, B) of V with $s \in A$ and $t \in B$.

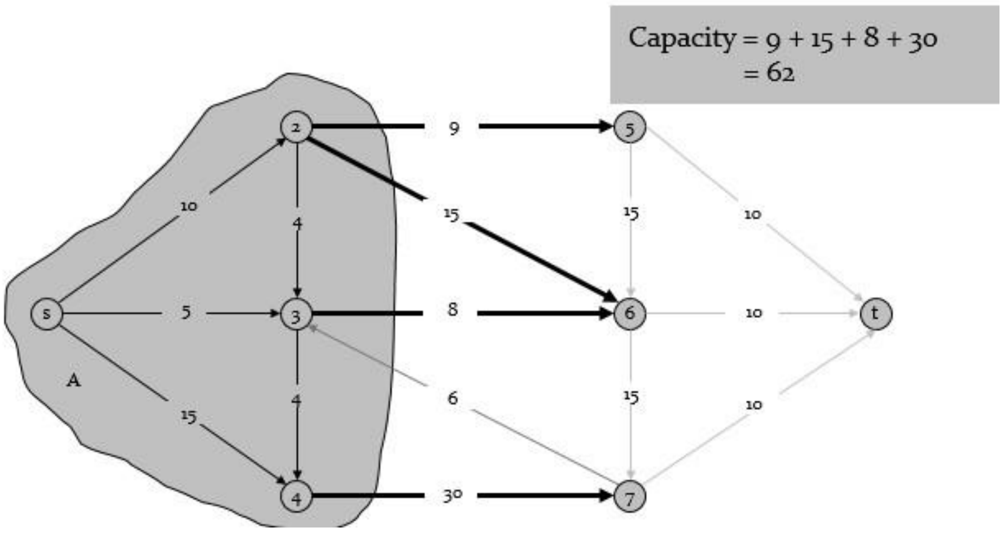
Def. The capacity of a cut (A, B) is: $cap(A, B) = \sum_{e \text{ out of } A} c(e)$



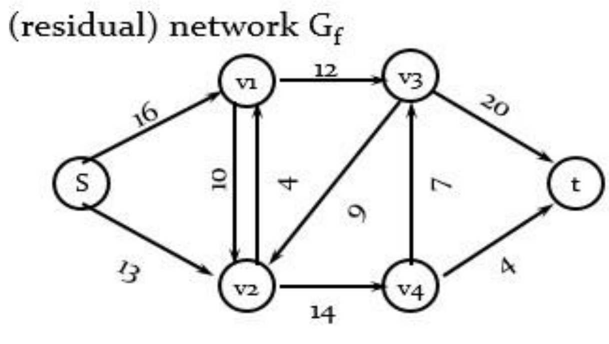
Cuts

Def. An s-t cut is a partition (A, B) of V with $s \in A$ and $t \in B$.

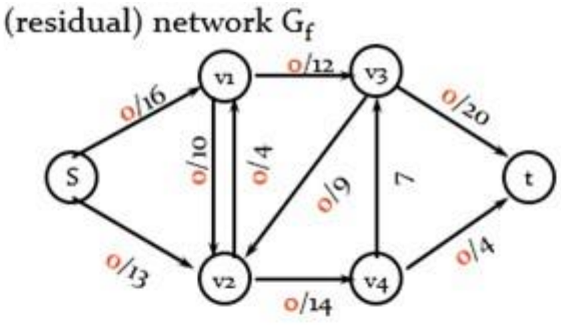
Def. The capacity of a cut (A, B) is: $cap(A, B) = \sum_{e \text{ out of } A} c(e)$



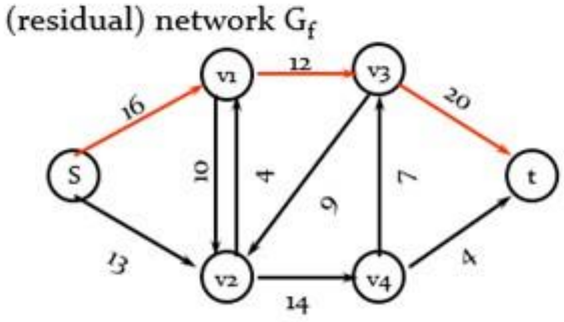
The basic Ford Fulkerson algorithm
example of an execution



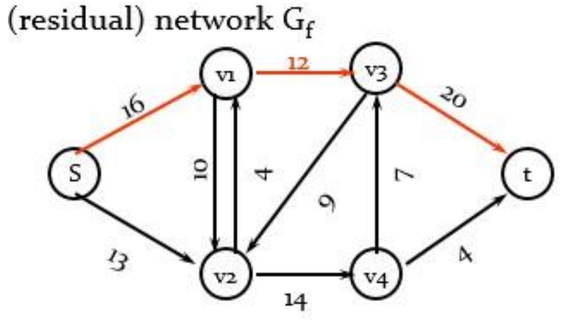
```
1  for each edge $(u, v) \in E[G]$
2     do $f[u, v] = 0$
3        $f[v, u] = 0$
4  while there exists a path p from s to t
   in the residual network $G_f$
5     do $c_f(p) = \min\{c_f(u, v) \mid (u, v) \in p\}$
6        for each edge (u, v) in p
7           do $f[u, v] = f[u, v] + c_f(p)$
8              $f[v, u] = - f[u, v]$
```



```
1 for each edge $(u, v) \in E[G]$
2    do $f[u, v] = 0$
3       $f[v, u] = 0$
4 while there exists a path p from s to t
  in the residual network $G_f$
5    do $c_f(p) = \min\{c_f(u, v) \mid (u, v) \in p\}$
6       for each edge (u, v) in p
7          do $f[u, v] = f[u, v] + c_f(p)$
8             $f[v, u] = - f[u, v]$
```



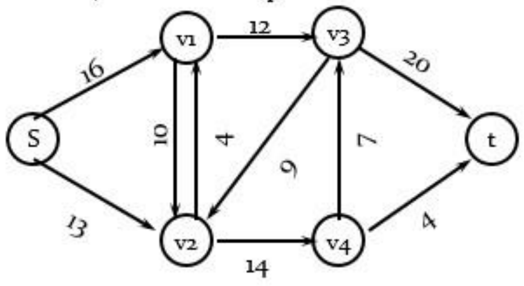
```
1 for each edge $(u, v) \in E[G]$
2    do $f[u, v] = 0$
3       $f[v, u] = 0$
4 while there exists a path p from s to t
  in the residual network $G_f$
5    do $c_f(p) = \min\{c_f(u, v) \mid (u, v) \in p\}$
6       for each edge (u, v) in p
7          do $f[u, v] = f[u, v] + c_f(p)$
8             $f[v, u] = - f[u, v]$
```



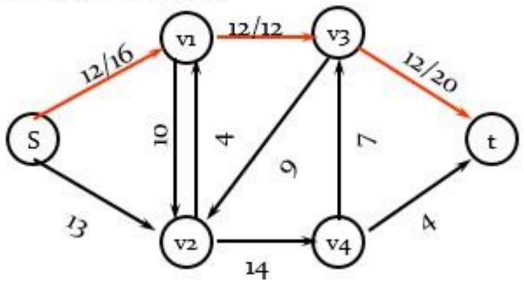
```
1 for each edge $(u, v) \in E[G]$
2    do $f[u, v] = 0$
3       $f[v, u] = 0$
4 while there exists a path p from s to t
  in the residual network $G_f$
5    do $c_f(p) = \min\{c_f(u, v) \mid (u, v) \in p\}$
6       for each edge (u, v) in p
7          do $f[u, v] = f[u, v] + c_f(p)$
8             $f[v, u] = - f[u, v]$
```

temporary variable:
$c_f(p) = 12$

(residual) network G_f



new flow network
G

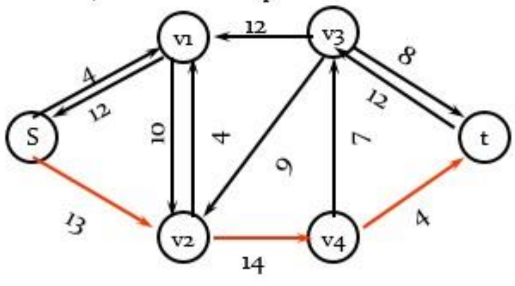


```
1   for each edge (u, v) ∈ E [G]
2       do f [u, v] = 0
3          f [v, u] = 0
4   while there exists a path p from s to t
    in the residual network G_f
5       do c_f(p) = min{c_f(u, v) | (u, v) ∈ p}
6           for each edge (u, v) in p
7               do f [u, v] = f [u, v] + c_f(p)
8                  f [v, u] = - f [u, v]
```

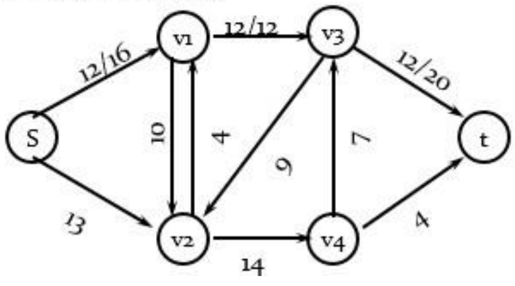
temporary variable:

$c_f(p) = 12$

(residual) network G_f

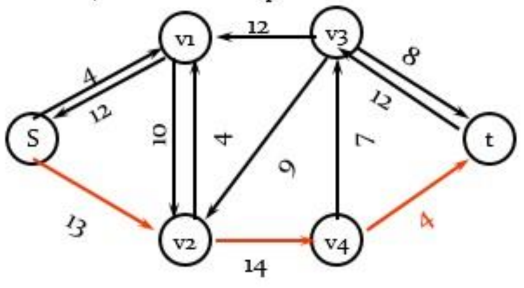


new flow network
G

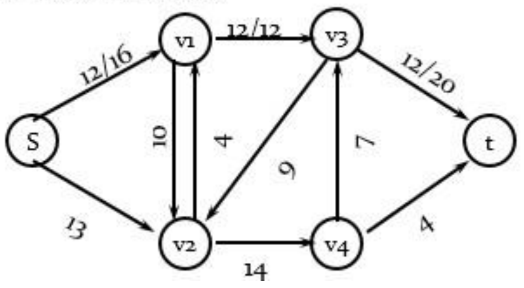


```
1   for each edge (u, v) ∈ E [G]
2       do f [u, v] = 0
3          f [v, u] = 0
4   while there exists a path p from s to t
    in the residual network G_f
5       do c_f(p) = min{c_f(u, v) | (u, v) ∈ p}
6           for each edge (u, v) in p
7               do f [u, v] = f [u, v] + c_f(p)
8                  f [v, u] = - f [u, v]
```

(residual) network G_f



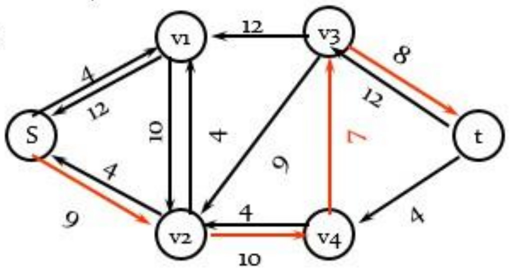
new flow network G



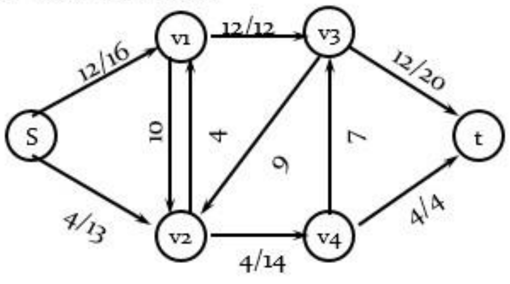
```
1  for each edge $(u, v) \in E[G]$
2     do $f[u, v] = 0$
3        $f[v, u] = 0$
4  while there exists a path $p$ from $s$ to $t$
   in the residual network $G_f$
5     do $c_f(p) = \min\{c_f(u, v) \mid (u, v) \in p\}$
6        for each edge $(u, v)$ in $p$
7           do $f[u, v] = f[u, v] + c_f(p)$
8              $f[v, u] = - f[u, v]$
```

temporary variable:
$c_f(p) = 4$

(residual) network G_f



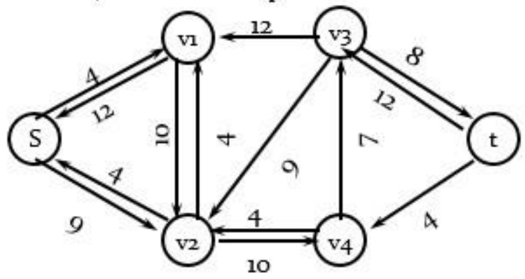
new flow network G



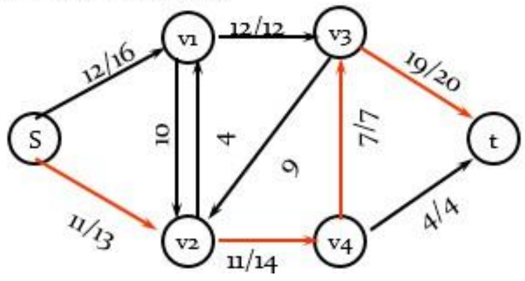
```
1  for each edge $(u, v) \in E[G]$
2     do $f[u, v] = 0$
3        $f[v, u] = 0$
4  while there exists a path $p$ from $s$ to $t$
   in the residual network $G_f$
5     do $c_f(p) = \min\{c_f(u, v) \mid (u, v) \in p\}$
6        for each edge $(u, v)$ in $p$
7           do $f[u, v] = f[u, v] + c_f(p)$
8              $f[v, u] = - f[u, v]$
```

temporary variable:
$c_f(p) = 7$

(residual) network G_f



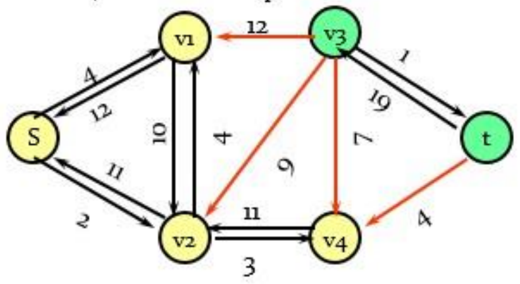
new flow network
G



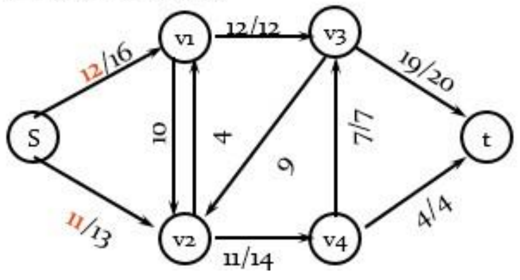
```
1  for each edge $(u, v) \in E[G]$
2     do $f[u, v] = 0$
3        $f[v, u] = 0$
4  while there exists a path p from s to t
   in the residual network $G_f$
5     do $c_f(p) = \min\{c_f(u, v) \mid (u, v) \in p\}$
6        for each edge (u, v) in p
7            do $f[u, v] = f[u, v] + c_f(p)$
8               $f[v, u] = - f[u, v]$
```

temporary variable:
$c_f(p) = 7$

(residual) network G_f



new flow network
G



```
1  for each edge $(u, v) \in E[G]$
2     do $f[u, v] = 0$
3        $f[v, u] = 0$
4  while there exists a path p from s to t
   in the residual network $G_f$
5     do $c_f(p) = \min\{c_f(u, v) \mid (u, v) \in p\}$
6        for each edge (u, v) in p
7            do $f[u, v] = f[u, v] + c_f(p)$
8               $f[v, u] = - f[u, v]$
```

Finally we have:
$|f| = f(s, V) = 23$