01 - Introduction to Design and Analysis of Algorithms

[KOMS119602] & [KOMS120403]

Design and Analysis of Algorithm (2021/2022)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

7-11 February 2022

Practical matters

- **Credit:** 3 SKS
- **Lecturer:** Dewi Sintiari
  - email: nld.sintiari@gmail.com
- **Evaluation:**
  - Presence ($\geq 75\%$) + attitude: 20%
  - Quiz / Take-home assignments (theoretical & programming): 40%
  - Midterm exam (written): 20%
  - Final exam (written): 20%
  - Bonus: writing an article, writing in wikipedia?
  - Grade = 20% Presence + 40% Assignments + 20% Midterm + 20% Final + Bonus

What are algorithms and why do we need them?

What are algorithms and why do we need them?

A simple algorithm:

Recipe of Indomie goreng

Ingredients

Steps

1. .....
2. .....
3. .....
4. .....
5. .....
6. .....
7. .....

Result

What are algorithms and why do we need them?

A simple algorithm:

Recipe of Teh celup

Ingredients

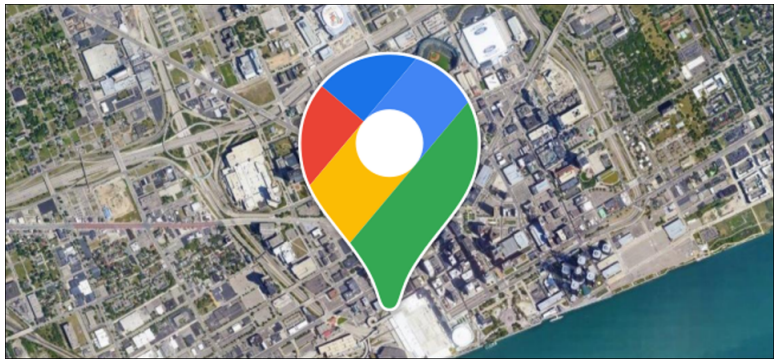
Steps

1. .....
2. .....
3. .....
4. .....
5. .....
6. .....

Result

What are algorithms and why do we need them?

Audio & video compression algorithms allow transmitting live video across internet



Route findings algorithms are used to find a route between two cities

Optimization and scheduling algorithms are used to arrange the schedule of airlines in the world

What are algorithms and why do we need them?

Algorithm = step-by-step procedure

Definition

An *algorithm* is a finite sequence of well-defined instructions, typically used to solve a class of specific problems or to perform a computation.

- In CS, an algorithm gives the computer a specific set of instructions, which allows the computer to do everything.
- Computer programs = algorithms written in programming languages that the computer can understand.
- An algorithm is written in pseudocode.

Algorithmic thinking is the ability to define clear steps to solve a problem. This allows us to break down problems and conceptualize solutions.

Characteristics of algorithms

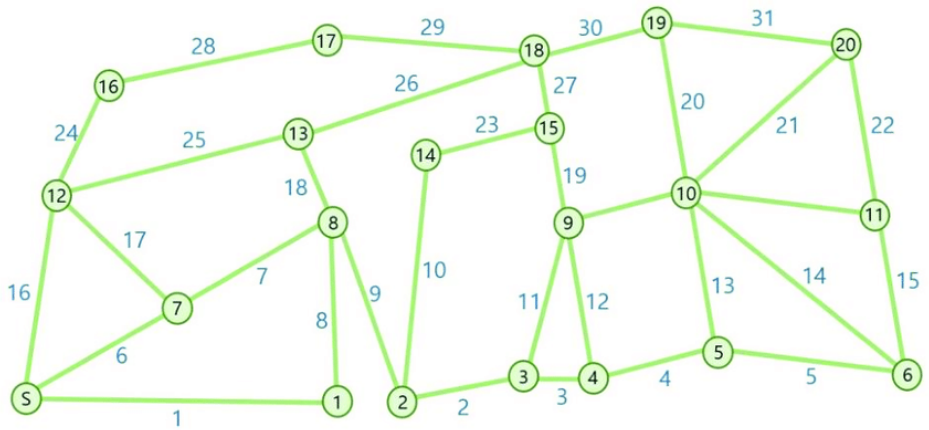
Important components in algorithms:

- Initial situation/condition/input (*Initial state*): can take zero or more inputs.
- Final situation/condition/output (*Output state*): at least one output.
- Definiteness: Each step must be clear, well-defined and precise. There should be no any ambiguity.
- Finiteness: should have finite number of steps and it should end after a finite time.
- Effectiveness: each step must be simple and should take a finite amount of time.
- Constraints given in the beginning and during composing the algorithm (*Constraint and assumption*)

Example of classical algorithmic problems

1. Traveling Salesman Problem (TSP)

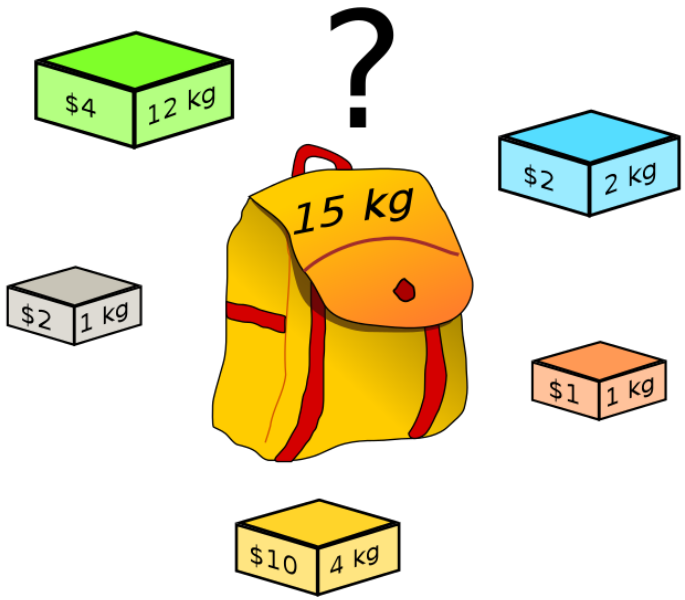
Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?



Example of classical algorithmic problems

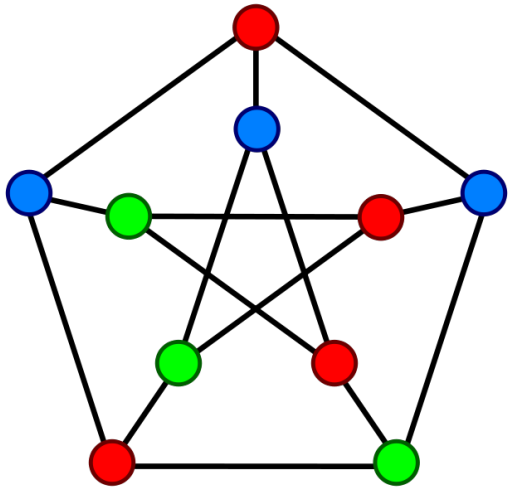
2. Integer Knapsack Problem

Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.



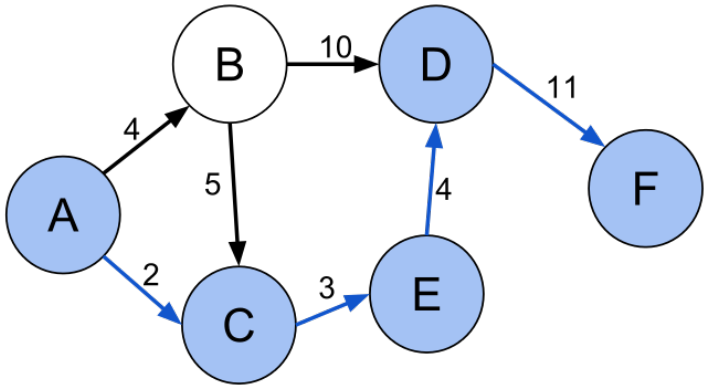
3. Coloring problem

Given a graph of n vertices, determine the minimum number of different colors needed to color the vertices such that no two adjacent vertices are of the same color. We also want to find a way to color the graph.



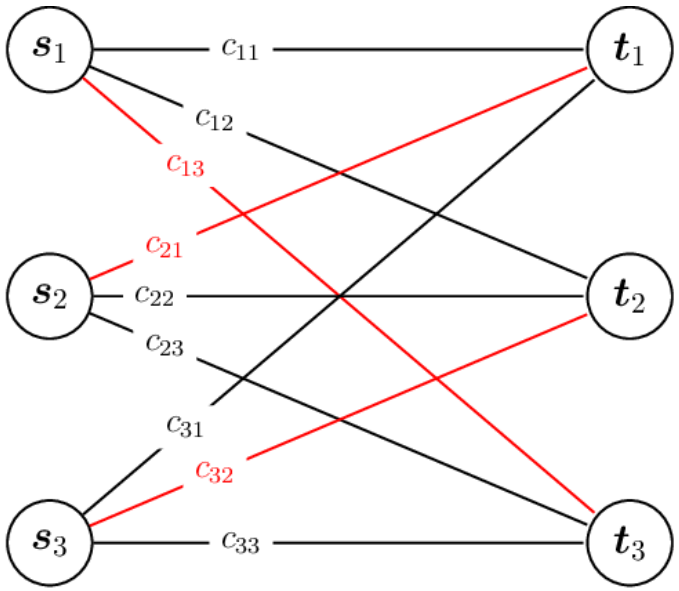
4. Shortest path problem

Given a graph of n vertices, find a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.



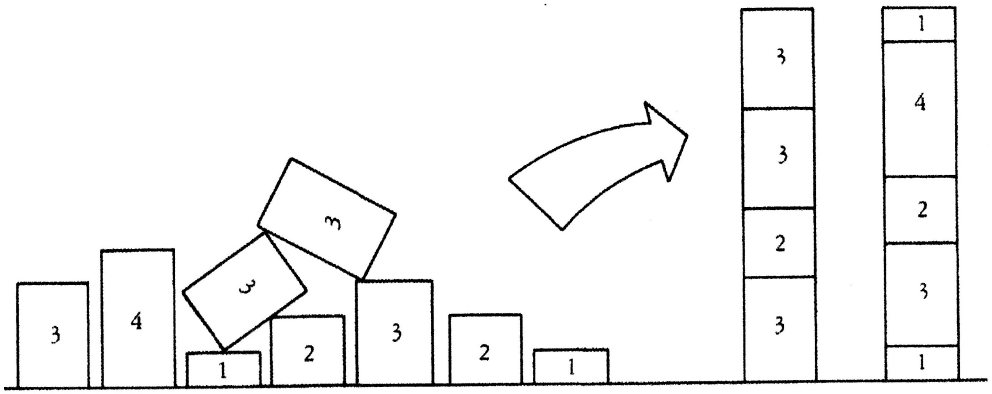
5. Assignment problem

Given n staffs and n tasks. Anyone can be assigned to perform any task, incurring some cost $c(i, j)$ that may vary depending on the staff(s_i)-task(t_j) assignment. It is required to perform as many tasks as possible by assigning one staff to each task, in such a way that the total cost $\sum c(i, j)$ of the assignment is minimized.



6. Partition problem

Given a multiset S of positive integers, decide whether S can be partitioned into two subsets S_1 and S_2 such that the sum of the numbers in S_1 equals the sum of the numbers in S_2.



The study of algorithms

Why do we need an algorithm?

1. To understand the basic idea or the flow of the problem.
2. To find an approach to solve the problem. A good design can produce a good solution.
3. To understand the basic principles of designing the algorithms.
4. Compare the performance of the algorithm w.r.t. other techniques.
5. To improve the efficiency of existing techniques.
6. It is the best method of description without describing the implementation detail.
7. To measure the behavior (or performance) of the methods in all cases (best cases, worst cases, average cases)
8. We can measure and analyze the complexity (time and space) of the problems concerning input size without implementing and running it; it will reduce the cost of design.

The study of algorithms

DAA helps to:

- Design the algorithms for solving problems in CS.
- Design and analyze the logic on how the program will work before developing the actual code.
- Classifying algorithms based on their purpose/design/complexity/etc..

The study of algorithms

Design of algorithms: a method or a mathematical process for problem-solving and engineering algorithms.

Process:

1. Step 1: Obtain a description of the problem. This step is much more difficult than it appears.
2. Step 2: Analyze the problem.
3. Step 3: Develop a high-level algorithm.
4. Step 4: Refine the algorithm by adding more detail.
5. Step 5: Review the algorithm.

The study of algorithms

Analysis of algorithms:

- Correctness
  - Does the input/output relation match algorithm requirements?
- Amount of work done (aka complexity)
  - The number of basic operations to do a task
- Amount of space used
  - Memory used
- Simplicity, clarity
  - Verification and implementation
- Optimality
  - Is it possible to do better?

The study of algorithms

What makes a good algorithm?

- Correctness
- Efficiency

Analysis of algorithms: Correctness

Definition

An algorithm to solve a problem P is correct iff for all the problem instance $i \in I$, it terminates and produces correct output $o \in O$.

Proving correctness is done using formal mathematical proof:

- Counterexample (*indirect* proof)
- Induction (*direct* proof)
- Loop Invariant

Other approaches: proof by cases/enumeration, by chain of iffs, by contradiction, by contrapositive.

Analysis of algorithms: Space/time complexity

Two fundamental parameters based on which we can analysis the efficiency of an algorithm:

- Space Complexity: the amount of space (storage) required by an algorithm to run to completion.
- Time Complexity: a function of input size n that refers to the amount of time needed by an algorithm to run to completion.
- or other resources needed to execute them

The study of algorithms

How to compute the gcd of two numbers m and n?

Middle-school procedure:

1. Find the prime factors of m.
2. Find the prime factors of n.
3. Identify all the common factors in the two prime expansions found in Steps 1 and 2. (If p is a common factor occurring p_m and p_n times in m and n, respectively, it should be repeated $\min\{p_m, p_n\}$ times.)
4. Compute the product of all the common factors and return it as the gcd of the numbers given.

Euclid algorithm:

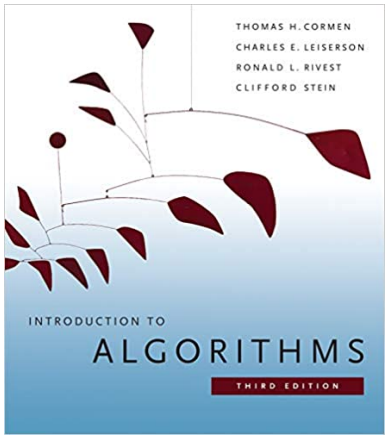
1. Assign the value of $\min\{m, n\}$ to t.
2. Divide m by t. If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.
3. Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step 4.
4. Decrease the value of t by 1. Go to Step 2.

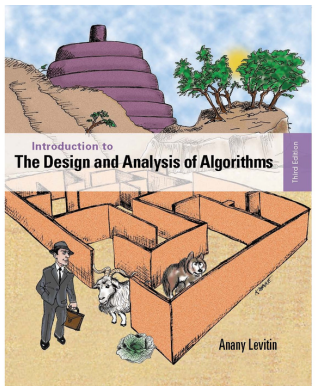
Outline of the semester

1. Introduction to design and analysis of algorithms
2. Complexity analysis of algorithms
3. Brute Force algorithm
4. Greedy algorithm
5. Recursive algorithm
6. Divide-and-Conquer algorithm
7. Decrease-and-Conquer algorithm
8. Transform-and-Conquer algorithm
9. BFS and DFS algorithms
10. Backtracking algorithm
11. Branch & Bound
12. Dynamic programming
13. Sorting algorithms
14. Graph algorithms
15. Computational complexity theory (P, NP, NP-C)

References

- Introduction to Algorithms (Thomas H. Cormen, C. E. Leiserson, R. Rivest, C. Stein), The MIT Press, 1989.
- Introduction to the Design and Analysis of Algorithms (Anany Levitin), Pearson, 2012.





- Kuliah pengantar Strategi Algoritma (Rinaldi Munir ITB)
- e-Modul Struktur Data dan Analisis Algoritma (Made Windu A. Kesiman, PTI Undiksha)

2. Complexity analysis

Determining a functions of time complexity and space complexity. An algorithm is said to be *efficient* when this function's values are small, or grow slowly compared to the growth in the size of input.

- n: the size of input
- $T(n)$: the number of computations/steps (comparison, arithmetic operations, accessing an array, etc.)
- $S(n)$: memory/storage space

- Measuring complexity: best-case, worst-case, and average-case
- We usually estimate the complexity *asymptotically*, i.e., to estimate the complexity function for arbitrarily large input. We use Big O (*upper-bound*), Big-omega (*lower-bound*) and Big-theta (*tight-bound*) notations.

3. Brute Force algorithm / Exhaustive search

This is the most basic and simplest type of algorithm. It systematically enumerate all possible candidates for the solution and check whether each candidate is the solution of the problem's statement.

Example: Given a lock of 4-digit PIN, where the digits to be chosen from 0-9. The brute force will be trying all possible combinations one by one like 0001, 0002, 0003, 0004, and so on until we get the right PIN (there are at most 10000 trials).

4. Greedy algorithm

It is an algorithmic paradigm that builds up a solution piece by piece, by always choosing, at each step, the next piece that offers the most obvious and immediate benefit (i.e. locally optimal choice).

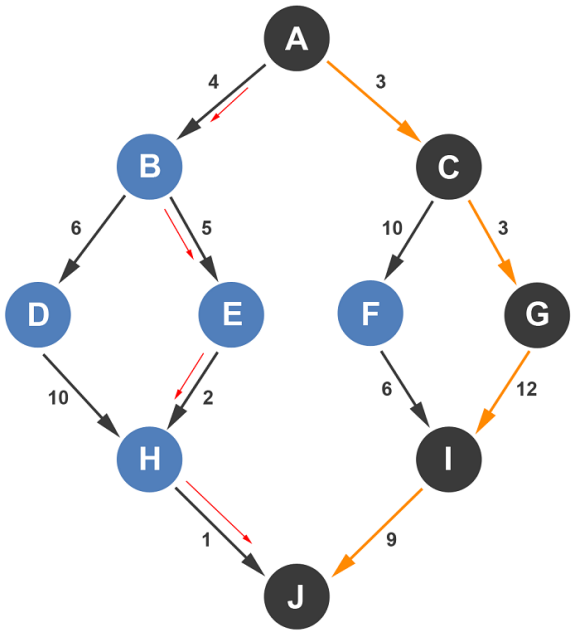
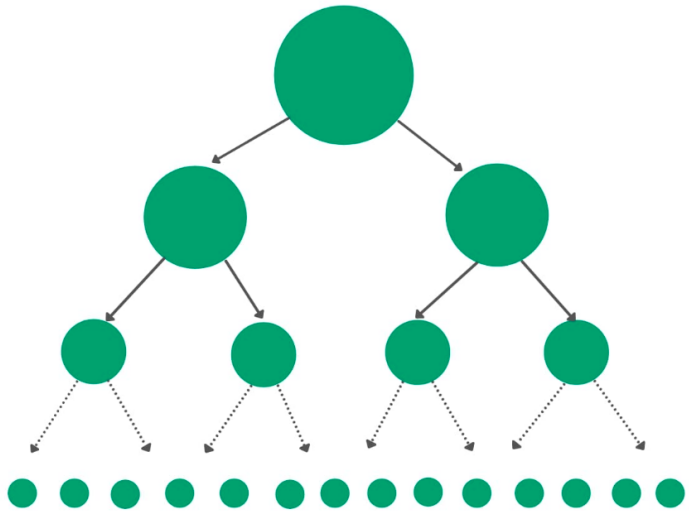


Figure: Greedy algorithm to find a path from A to J

5. Recursive algorithm

In CS, *recursion* is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem.

This is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input.



6. Divide-and-conquer algorithm

It works by recursively breaks down a problem into two/more sub-problems of the same/related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

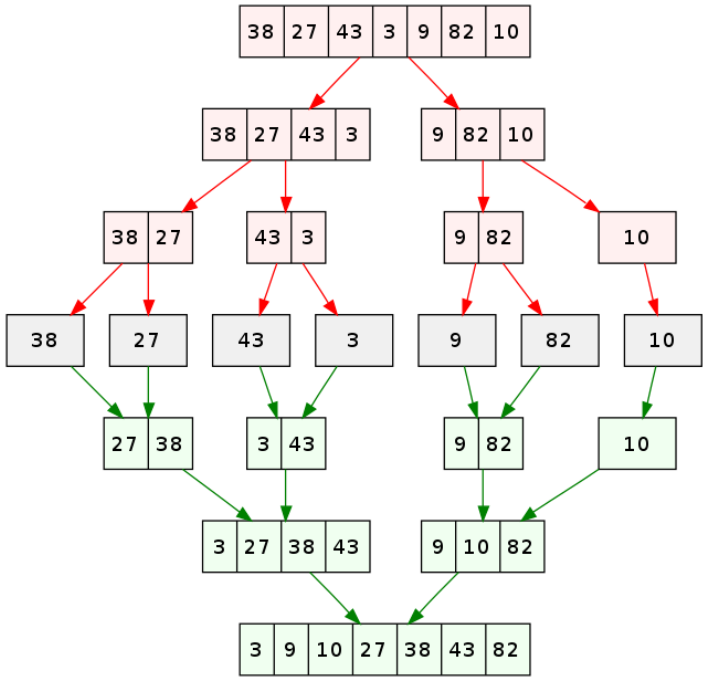


Figure: Divide-and-conquer algorithm to sort a sequence of numbers

7. Decrease-and-conquer algorithm

Three main steps:

- Decrease or reduce problem instance to smaller instance of the same problem and extend solution.
- Conquer the problem by solving smaller instance of the problem.
- Extend solution of smaller instance to obtain solution to original problem.

Basic idea: exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance.

8. Transform-and-conquer algorithm

The technique used in designing algorithms that will transform (to transform) a case into another form, then determine a solution (to conquer) from the new form of the case.

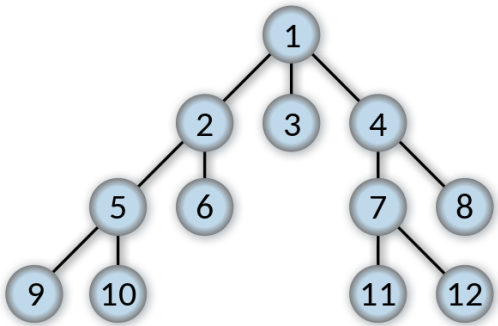
1. First stage involves the transformation to another problem that is more amenable for solution.
2. Second stage involves solving the new problem where the transformed new problem is solved. Then the solutions are converted back to the original problem

Example: multiplying two simple numbers XII and IV (in Roman number system).

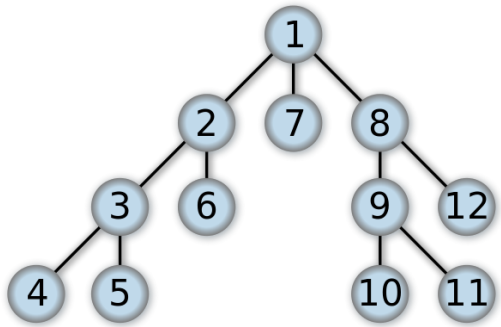
- 1st stage: the numbers XII and IV is transformed to another problem of 12×4.
- 2nd stage: the actual multiplication is done as 48, then the result is converted to Roman number as XLVIII.

9. BFS and DFS algorithms

Breadth-first search (BFS) is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. While for Depth-first search (DFS) starts at the root node and explores as far as possible along each branch before backtracking.



(a) BFS

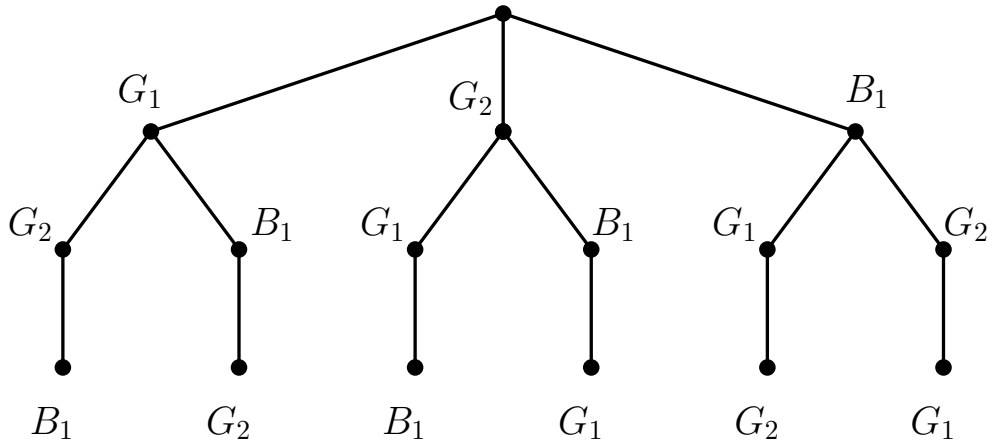


(b) DFS

10. Backtracking algorithm

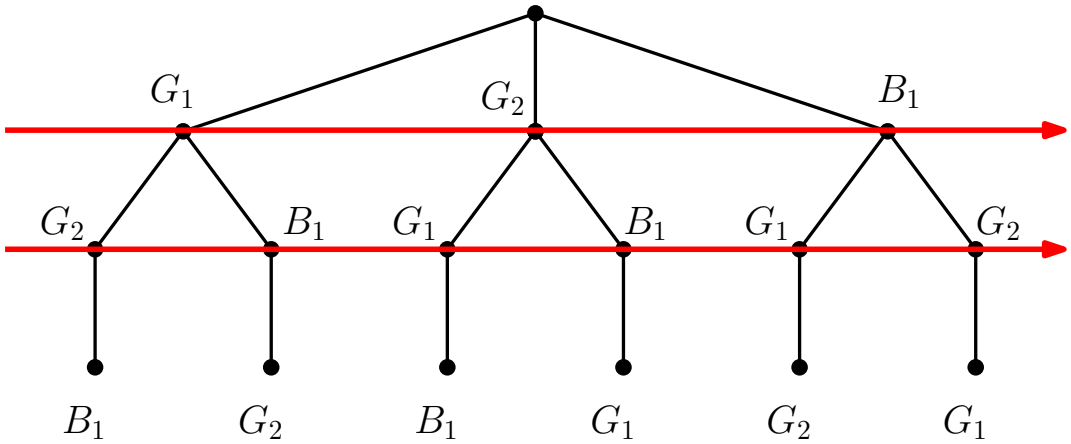
Backtracking is an algorithmic technique where the goal is to get all solutions to a problem using the brute force approach.

Example: We want to enumerate the different ways to order 2 girls and a boy in a row.



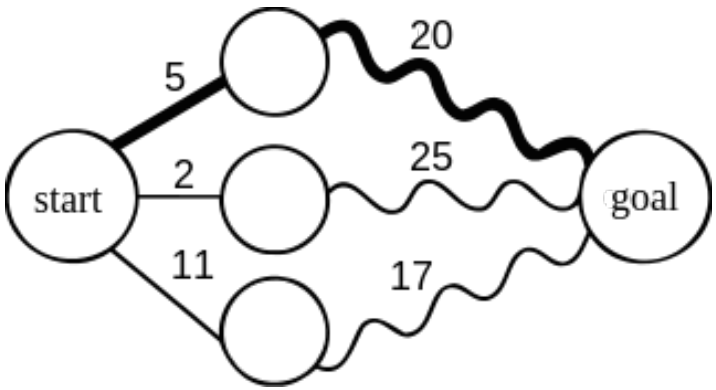
11. Branch and bound algorithm

Similar to the backtracking since it also uses the state space tree. It is used for solving the optimization problems and minimization problems. If we have given a maximization problem then we can convert it using the Branch and bound technique by simply converting the problem into a maximization problem.



12. Dynamic programming This is a technique that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property.

Dynamic programming is mostly applied to recursive algorithms.



13. Sorting algorithms

This is an algorithm that puts elements of a list into an order.

The output of any sorting algorithm must satisfy two conditions:

1. The output is in monotonic order (each element is not smaller/larger than the previous element, according to the required order).
2. The output is a permutation (a reordering, yet retaining all of the original elements) of the input.

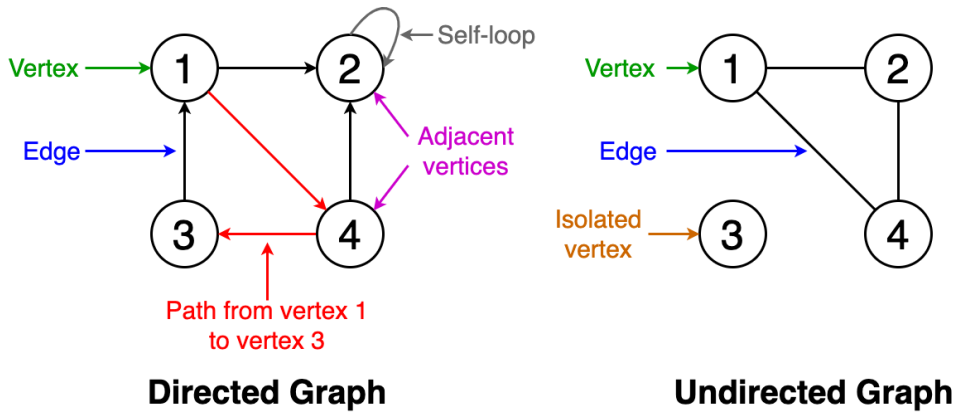
Example:

$$\text{COMPUTERSCIENCE} \rightarrow \text{CCCEEEIMNOPRSTU}$$

14. Graph algorithms

A graph is an abstract notation used to represent the connection between pairs of objects. A graph consists of:

- **Vertices/Nodes:** interconnected objects in a graph.
- **Edges:** links that connect the vertices.



Example: shortest path, minimum spanning tree, cycle detection, strongly connected component, graph coloring, maximum flow

15. Computational complexity theory (P, NP, NPC)

Focuses on classifying computational problems according to their *resource usage*, and the *relation* between classes.

A computational problem is a task solved by a computer.

The computational complexity or simply complexity of an algorithm is the amount of resources (*time* and *memory*) required to run it.

The complexity of a problem is the complexity of the best algorithms that allow solving the problem.

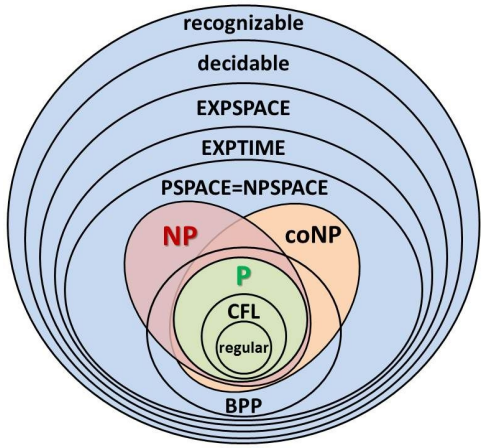


Figure: Computational complexity classes

Classification of algorithms based on the strategy

1. Direct solution: brute-force, greedy
2. Space-state base: backtracking, branch and bound
3. Top-down solution: divide-and-conquer, decrease-and-conquer, transform-an-conquer, dynamic programming, BFS & DFS
4. Bottom-up solution: dynamic programming