Algorithm Design Strategies

**Algorithmic Efficiency**

- Running time not useful for comparing algorithms, due to:

1. Speed of particular computers;
2. Chosen computer language;
3. Quality of programming implementation;
4. Compiler optimizations

- Solution: count the “basic operations”:

1. Count arithmetic operations/compartions
2. Find closed formula
3. Identify best, wost and average case situations
4. For iterative algorithms, count how many iterations
5. For recursive algorithms, count how many recursive calls

**Algorithms**

- Algorithm is a sequence of non-ambiguous instructions, that execute on a finite amount of time. Its input is na instance of a problema that it tries to solve.

- It can be classified based on:

1. Type of problems solved
2. Design techniques
3. Deterministic vs Non-Deterministic

**Deterministic Algorithms**

- Obtains, always, the same answer for the same data, as well as, taking the exact same steps to do it.

**Non-Deterministic Algorithms**

- Can exhibit diferente behavior/output, for the same data, on different runs. Generally used to get approximate solutions.

- Factors of the non-deterministic behavior:

1. External state other than input (timer values, random, values, etc)
2. Time-sensitive operations
3. Hardwares errors

**Types of Problems**

- Searching

- Sorting

- String processing

- Graph/Network problems

- Combinatorial Problems

- Bioinformatics

**Algorithm Design Techniques**

- Brute-Force

-> The most straightforward approach to solving problems

-> Directly based on the problema statment and the definitions involved

-> Strengths:

1. Simplicity
2. Applicable to different kinds of problems

-> Weaknesses:

1. Low efficiency in some cases
2. Useful only for instances of relatively small size

-> Examples:

1. Selection sort
2. Sequentials sort

- Divide-and-Conquer

-> Recursive decomposition into “smaller” prob. instances

-> Framework:

1. Divide a problem instance into similiar, smaller instances
2. Solve smaller instances recursively
3. Combine solutions of smaller instances to get the original one

-> Problems:

1. Recursion is slow
2. Not the best approach for simple problems
3. Sub-problems might overlap

-> Examples:

1. Mergesort
2. Quicksort
3. Multiplying large integers
4. Strassen matrix multiplication

- Decrease-and-Conquer

-> Successive decomposition into a “smaller” problem instance

-> Framework (Top-Down):

1. Identify one similar and smaller problema instance
2. The smaller instance is solved recursively
3. Solutions for smaller instances are processed to get the original one

-> How:

1. Decrease-by-one
2. Decrease by a constant factor
3. Variable-size decrease

-> Example:

1. Binary search
2. Interpolation search
3. Fake-coin problema

- Transform-and-Conquer

-> Solve a different problem to get desired result (problem reduction)

-> May perform some pre-processing on the data

-> Examples:

1. Search on ordered and balanced trees
2. Heapsort

- Dynamic Programming

-> Decomposition into overlapping (smaller) sub-problems

-> Use recurrence but starting from the smallest problema

-> Get intermmidiate solutions from smaller problems

-> Store overlapping solutions

-> Proceed bottom-up

-> Examples:

1. Computing Fibonacci numbers
2. Computing binomial coefficients
3. Graphs: Warshall algo; Floyd alf; etc
4. Knapsack

- Greedy Algorithms

-> Construct a solution through a sequence of steps

-> The choice at each step is:

1. Feasible (satisfies constraints)
2. Locally optimal
3. Irrevocable

-> Examples:

1. Coin-changing problem
2. Graphs:
   1. Dijkstra’s shortest-path algorithm
   2. Prim’s minimum-spanning tree algorithm
   3. Kruskal’s minimum-spanning tree algorithm

**Limitations of Algorithmic Power**

- Solutions:

1. Backtracking
2. Branch-And-Bound
3. Aproximation Algorithms

**Fundamental Data Structures**

- Arrays

- Linked Lists

- Trees

**Common Abstract Data Types**

- Stack

- Queue

- Priority Queue

- Ordered List

- Binary Search Tree

- Graph / Network

**Efficiency Analysis**

- Time: How fast does na algorithm run?

- Space: Does na algorithm require additional memory

**Input Size**

- Relate operations count / running time to input size. Relate size metric to the main operations of na algorithm.

**Worst, Best and Average Cases**

- Running time depends on input size, but it may also depend on particular data configurations, for instance, for a sequential search on a n-element array is the array ordered? Increasing or decreasing order?

- Worst case: W(n)

-> input(s) of size n for which an algorithm runs longest

-> upper bound for operations count

- Best case: B(n)

-> input(s) of size n for which na algorithm runs fastest

-> lower bound for operations count

- Average case: A(n)

-> behavior for “typical” inputs

-> sometimes, much better than worst case

**Growth Rate**

- Identify algorithm efficiency for large input sizes. How much does it grow, when input size becomes larger?

**Asymptotic Notations**

- This order of growth of oeprations count indicates efficiency, this also allows to compare algorithms for the same problems?

- Hides unimportant details about how fast a function grows

- Useful notations: О(n), Ω(n), Ө(n)

**Big-Oh Notation**

- Asymptotic upper bound;

- О(g(n)): set of all functions with smaller or same order of growth as g(n);

**Big-Omega Notation**

- Asymptotic lower bound;

- Ω(g(n)): set of all functions with larger or same order of growth as g(n);

**Big-Theta Notation**

- Asymptotic tight bound;

- Ө(g(n)): set of all functions with same order of growth as g(n)

**Efficiency Classes**

- О(1): constant (e. g. hashing)

- О(log n): logarithmic (e. g. decrease-and-conquer)

- О(n): linear (e. g. processing all elemento of a collection)

- О(n log n): n-log-n (e. g. divide-and-conquer)

- О(nk): polynomial (e. g. k nested loops)

- О(2n): exponential (e. g. generate all subsets of an n-element set)

- О(n!): factorial (e. g. generate all permutations of an n-element set)

**Empirical Analysis**

- Run the algorithm on a sample of test inputs:

1. Input data should represente all possible cases
2. Input data should encompass large (set) sizes
3. Pseudo-random data

- Record and analyze: tables (operation counts, running time, etc)

- Identify best, worst and average case

- Identify complexity classes

- Problems:

1. Inadequate sample input data
2. Dependence of running times
3. Some problems / instances cannot be solved quickly enough

- Advantages:

1. Avoid difficult formal analysis
2. Allow predicting the running time for different input data sets

**Memoization**

- Turning results of a function into something to be remembered

- Use a collection to store previous results

- Time vs space trade-off