Lesson 2 : Problem Solving Paradigms

CPC UTEC - Lecturers Notes

October 2, 2020

Scope & Objectives:

- Have a formal understanding of Big-Oh notation.
- Learn about the popular problem solving paradigms.
- Understand Complete Search and brute force.

1 Algorithmic Analysis

Big-Oh notation is a tool we can use to measure how the complexity of our solution scales as the input of our program grows. This is done by measuring how our complexity compares to other functions as the input size goes to infinity. For this reason this is know as *Asymptotic Analysis*.

Formally, we can define Big-Oh notation as:

$$f(n) = O(g(n)) \longleftrightarrow f(n) \leq kg(n)$$

$$\text{Where } n > n_0,\ k > 0 \text{ and } n_0 \in \mathbb{R}$$

This definition might seem complex but is very simple to understand. We can use the aid of the following graph:

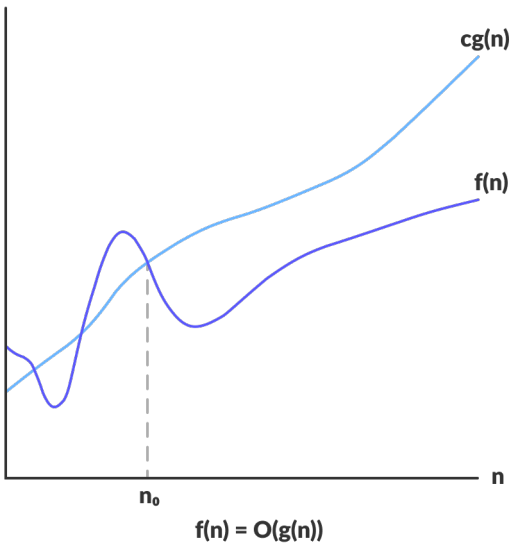

Figure 1: Graphical understanding of Big-Oh

First of all we must understand that $O(g(n))$ is not a function but a family (or set) of functions. For this reasons many believe that the '$\in$' symbol would be more appropriate than the '$=$' symbol. Therefore, it is a good idea to read the '$=$' sign as *belongs* or *is in*.

2 Problem Solving Paradigms

2.1 Paradigm vs Algorithm

Algorithm

- Solves a specific problem.
- Gives steps and instructions.

Paradigm

- Solves a family of problems.
- Gives ideas and methodologies.
- Is a conceptual framework for problem solving.

2.2 Popular Paradigms

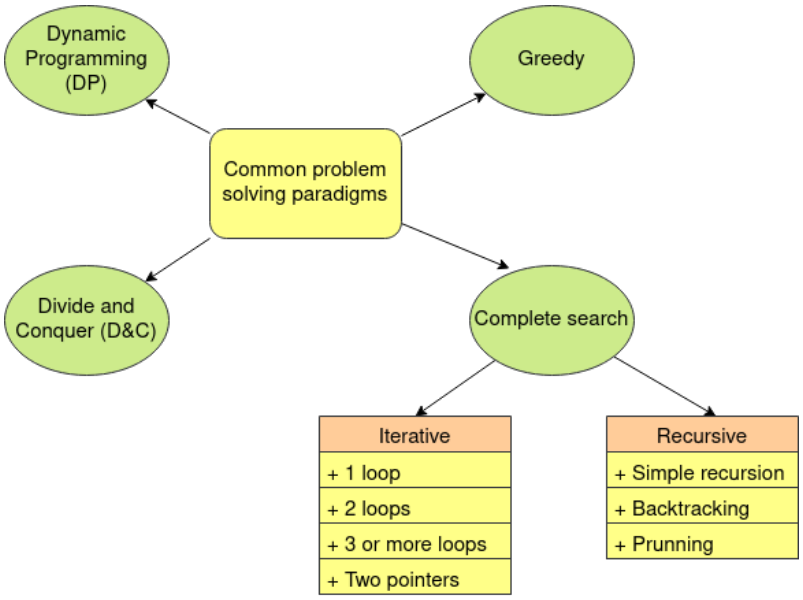

Figure 2: Popular problem solving paradigms.

3 Complete Search

Most competitive programming problems (and most problems in general) can be solved by listing all the possible solutions and checking each of them until we find the correct one. This method of problems solving in which we exhaustively check all possible solutions is called *complete search*.

This paradigm is extremely important as we can almost always find a complete search solutions to any given problem (even though this solution will probably violate the time constrains). Being

good at designing complete search solutions is extremely important to learn the other problems solving paradigms and understanding the importance of optimizations.

3.1 Key Terms

- **Solution-Space:** All the candidate or possible solutions there are.
- **Brute Force:** Iterative complete search.
- **Fixing:** Assuming a *variable* has a constant value.

3.2 Example: Number of solutions

Lets consider the following problem. Given the positive integers a, b, c and d, find how many solutions the equation $x + y + z = d$ has, given that $x, y, z \in \mathbb{N}^+$, $x \leq a$, $y \leq b$ and $z \leq c$.

A first approach for solving this problem is to fix the values of x, y and z. We can then find the value of $x + y + z$ and check if it equals d.

```
cnt ← 0;
for x ← 1 to a do
    for y ← 1 to b do
        for z ← 1 to c do
            if x + y + z = d then
                cnt ← cnt + 1;
return cnt;
```

It is simple to see that this solution has a complexity of $O(n^3)$. In general, for complete search our time complexity can be estimates as:

$$T(n) = |\text{Solution Space}| \times \text{time to check a solution}$$

In this example we take $O(1)$ to check each candidate solution, but we visit $O(n^3)$ possible solutions, giving the total complexity of $O(n^3)$. Can we improve this solution?

One important observation is to realize that if we fix the values of a and b, then we can re-write the equation as: $z = d - x - y$. This means that for every pair of values (x, y), there exists a single possible value of z that can satisfy the equation. Now we all have to do is verify that the value of z is valid.

```
cnt ← 0;
for x ← 1 to a do
    for y ← 1 to b do
        z ← d − x − y;
        if 1 ≤ z ≤ c then
            cnt ← cnt + 1;
return cnt;
```

Our time taken to verify each solution is still $O(1)$, however the size of our solution space is now $O(n^2)$. This drastically improves the solutions complexity, taking it from $O(n^3)$ to $O(n^2)$. Further improvements can be done to make in order to find the answer in $O(n)$ time or faster.

The most important takeaway of this example should be that a brute force algorithm is not necessarily a dumb or easy algorithm, many time brute force solutions need to be tightly optimized in order to yield a correct solution that works within the time constrains.