The Hitchhiker's Guide to Design and Analysis of Algorithms

CSE 2222

Nakul Bhat

Department of Computer Science and Engineering

Manipal Institute of Technology

Email: nakulbhat034@gmail.com

Phone: +91 8660022842

February 6, 2025

Abstract

This course will serve as an introduction to algorithms and their workings. We will cover the different types of asymptotic notations, some widely used algorithm design techniques and analyse their efficiency.

Syllabus

1. **Module 1** **8 Hours**
   Introduction
   Fundamentals of Analysis of Algorithmic Efficiency
2. **Module 2** **10 Hours**
   Brute Force Techniques
   Decrease and Conquer
3. **Module 3** **10 Hours**
   Divide and Conquer
   Transform and Conquer
4. **Module 4** **10 Hours**
   Space and Time Tradeoffs
   Dynamic Programming
5. **Module 5** **10 Hours**
   Greedy Techniques
   Limitations of Algorithmic Power

Contents

Chapter 1

Fundamentals of Algorithmic Efficiency

1.1 The Concept of Basic Operations

We consider the most repeated operation of an algorithm to be its basic operation. By doing this, we can effectively mimic the working of the algorithm without much of the unnecessary complexity.

For example, in bubble sort, we compare each element to the next element, even if we do not swap them. This implies that the basic operation of bubble sort is comparision.

This approach also helps us to approximate the time taken by an algorithm, by separating the execution and number of operations.

$$T(n) \approxeq c_{op} \cdot C(n)$$

Where $T(n)$ is the time taken for the execution of an algorithm for n inputs, c_{op} is the time required for the execution of the basic operation and $C(n)$ is the number of basic operations for n inputs.

1.2 The Three Notations

There can by many algorithms designed around a problem statement. This, would necessitate a framework for comparision of algorithms. However, due to their virtue of being 'methods', we have to eliminate the factor of 'speed of operations' from such a comparision. For this, we are using three notations borrowed from mathematics.

$$\mathcal{O}\left(n\right) \qquad \text{(Upper Bound)}$$
$$\Omega\left(n\right) \qquad \text{(Lower Bound)}$$
$$\Theta\left(n\right) \qquad \text{(Tight Bound)}$$

It is important to stress here that these notations do not represent the best/worst/average cases. These are just mathematical notations, and a best/worst/average case can only be fixed for a particular input size.

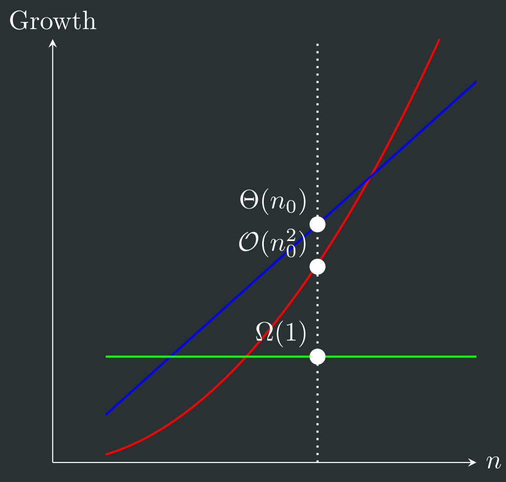


Figure 1.1: Orders of Growth

Informally, we can think of $\mathcal{O}(n)$ as representing the changes in performance of the algorithm with the worst cases at different sizes. Similarly, we can consider $\Omega\,(n)$ as tracking the best case growth rate, and $\Theta\,(n)$ doing the same for the average cases.

Figure 1.1 illustrates this point clearly. In this algorithm, the orders of growth are given by

$$\mathcal{O}(n^2), \Theta(n) \text{ and } \Omega(1) \qquad \text{(These are notations)}$$

And for a particular choice of $n = n_0$

$$\Omega(1) < \mathcal{O}(n_0^2) < \Theta(n_0) \qquad \text{(These are cases)}$$

Note: This is not conventionally considered, as for the above case, $n_0 < 1$ which is meaningless in terms of computational inputs. But this is theoretically possible, and highlights the difference well.

1.3 Formal Definitions

Definition 1.1: $\mathcal{O}(g(n))$

A function $f(n)$ is said to be in $\mathcal{O}(g(n))$ denoted as

$$f(n) \in \mathcal{O}(g(n))$$

If there exists some positive constant c and some non-negative integer n_0 such that

$$f(n) \leq c \cdot g(n) \qquad \forall\, n \geq n_0$$

Definition 1.2: $\Omega(g(n))$

A function $f(n)$ is said to be in $\Omega(g(n))$ denoted as

$$f(n) \in \Omega(g(n))$$

If there exists some positive constant c and some non-negative integer n_0 such that

$$f(n) \geq c \cdot g(n) \qquad \forall\, n \geq n_0$$

Definition 1.3: $\Theta(g(n))$

A function $f(n)$ is said to be in $\Theta(g(n))$ denoted as

$$f(n) \in \Theta(g(n))$$

If there exists some positive constants c_1 and c_2 and some non-negative integer n_0 such that

$$c_2 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n) \qquad \forall n \geq n_0$$

1.4 Worked Examples

Example 1.1: Prove that $f(n) = 3n^2 + 2n + 7$ belongs to $\mathcal{O}(n^3)$.

From the question, we can infer that $g(n) = n^3$. According to definition 1.1, we need to find constants c and n_0 such that

$$f(n) \leq c \cdot g(n) \qquad \forall n \geq n_0$$
$$3n^2 + 2n + 7 \leq 1 \cdot n^3 \qquad \forall n \geq 3 \qquad n_0 = 3, c = 1$$

Example 1.2: Prove that $f(n) = 5n^3 - 4n + 8$ belongs to $\Omega(n^3)$.

From the question, we can infer that $g(n) = n^3$. According to definition 1.2, we need to find constants c and n_0 such that

$$f(n) \geq c \cdot g(n) \qquad \forall n \geq n_0$$
$$5n^3 - 4n + 8 \geq 1 \cdot n^3 \qquad \forall n \geq 2 \qquad n_0 = 2, c = 1$$

Example 1.3: Prove that $f(n) = 4n^2 + 10n + 20$ belongs to $\Theta(n^2)$.

From the question, we can infer that $g(n) = n^2$. According to definition 1.3, we need to find constants c_1, c_2 and n_0 such that

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \qquad \forall n \geq n_0$$
$$1 \cdot n^2 \leq 4n^2 + 10n + 20 \leq 5 \cdot n^2 \qquad \forall n \geq 5 \qquad n_0 = 5, c_1 = 1, c_2 = 5$$

1.5 Theorems for Asymptotic Notations

Here are some useful theorems listed without proof for asymptotic notations.

Theorem 1.1: Addition of $\mathcal{O}(g(n))$

If $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$, then their sum satisfies

$$f_1(n) + f_2(n) \in \mathcal{O}(\max(g_1(n), g_2(n))).$$

Theorem 1.2: Addition of $\Omega(g(n))$

If $f_1(n) \in \Omega(g_1(n))$ and $f_2(n) \in \Omega(g_2(n))$, then their sum satisfies

$$f_1(n) + f_2(n) \in \Omega(\min(g_1(n), g_2(n))).$$

Theorem 1.3: Addition of $\Theta(g(n))$

If $f_1(n) \in \Theta(g_1(n))$ and $f_2(n) \in \Theta(g_2(n))$, then their sum satisfies

$$f_1(n) + f_2(n) \in \Theta(\max(g_1(n), g_2(n))).$$

Chapter 2

Brute Force Techniques

2.1 Selection Sort

Algorithm 2.1: Selection Sort

Given an array A of length n
for $i = 0$ to $n-2$ **do**
  Let $minIndex = i$
  for $j = i$ to $n-1$ **do**
    if $A[j] < A[minIndex]$ **then**
      $minIndex = j$
    end if
  end for
  Swap $A[i]$ and $A[minIndex]$
end for

Example 2.1: Selection Sort

Consider the array $A = [64, 25, 12, 22, 11]$.

1. Start with $i = 0$, find the minimum element in $[64, 25, 12, 22, 11]$. The minimum is 11, swap it with $A[0]$.
   $\Rightarrow [11, 25, 12, 22, 64]$
2. Move to $i = 1$, find the minimum in $[25, 12, 22, 64]$. The minimum is 12, swap with $A[1]$.
   $\Rightarrow [11, 12, 25, 22, 64]$
3. Move to $i = 2$, find the minimum in $[25, 22, 64]$. The minimum is 22, swap with $A[2]$.
   $\Rightarrow [11, 12, 22, 25, 64]$
4. Move to $i = 3$, find the minimum in $[25, 64]$. The minimum is 25, no swap needed.
   $\Rightarrow [11, 12, 22, 25, 64]$
5. Array is now sorted.

2.2 Bubble Sort

Algorithm 2.2: Bubble Sort

Given an array A of length n
for $i = 0$ to $n - 2$ **do**
  for $j = 0$ to $n - i - 2$ **do**
    if $A[j] > A[j + 1]$ **then**
      Swap $A[j]$ and $A[j + 1]$
    end if
  end for
end for

Example 2.2: Bubble Sort

Consider the array $A = [64, 25, 12, 22, 11]$.

1. Pass 1: Compare and swap adjacent elements:
   $[64, 25, 12, 22, 11] \Rightarrow [25, 64, 12, 22, 11] \Rightarrow [25, 12, 64, 22, 11] \Rightarrow [25, 12, 22, 64, 11] \Rightarrow [25, 12, 22, 11, 64]$
2. Pass 2: Ignore last element, repeat for first four elements:
   $[25, 12, 22, 11, 64] \Rightarrow [12, 25, 22, 11, 64] \Rightarrow [12, 22, 25, 11, 64] \Rightarrow [12, 22, 11, 25, 64]$
3. Pass 3: Ignore last two elements, repeat:
   $[12, 22, 11, 25, 64] \Rightarrow [12, 11, 22, 25, 64]$
4. Pass 4: Final swap:
   $[11, 12, 22, 25, 64]$
5. Array is now sorted.

2.3 Sequential Search

Algorithm 2.3: Sequential Search

Given an array A of length n and target value x
for $i = 0$ to $n - 1$ **do**
  if $A[i] == x$ **then**
    Return i (index of x)
  end if
end for
Return -1 (not found)

Example 2.3: Sequential Search

Consider the array $A = [4, 2, 9, 7, 1, 5]$ and target $x = 7$.

1. Compare $A[0] = 4$ with x, not a match.
2. Compare $A[1] = 2$ with x, not a match.
3. Compare $A[2] = 9$ with x, not a match.
4. Compare $A[3] = 7$ with x, match found.
5. Return index 3.

2.4 Brute Force String Matching

Algorithm 2.4: Brute Force String Matching

```
Given text T of length n and pattern P of length m
for i = 0 to n − 1 do                                  ▷ Move one character at a time
    Initialize j = 0
    while j < m and T[i + j] == P[j] do
        Increment j
    end while
    if j == m then
        Return index i (pattern found)
    end if
end for
Return −1 (pattern not found)
```

Example 2.4: Brute Force String Matching

Consider searching for pattern $P =$ "hello" in text $T =$ "ahehello".

1. $T[0] =$ 'a' does not match $P[0]$, move to next letter.
2. $T[1] =$ 'h' matches $P[0]$, continue checking.
3. $T[2] =$ 'e' matches $P[1]$, continue checking.
4. $T[3] =$ 'h' does not match $P[2]$, move to next letter.
5. $T[2] =$ 'e' does not match $P[0]$, move to next letter.
6. $T[3] =$ 'h' matches $P[0]$, continue checking.
7. $T[4] =$ 'e' matches $P[1]$, continue checking.
8. $T[5] =$ 'l' matches $P[2]$, continue checking.
9. $T[6] =$ 'l' matches $P[3]$, continue checking.
10. $T[7] =$ 'o' matches $P[4]$, full match found at index 5.

2.5 Exhaustive Search

Algorithm 2.5: Exhaustive Search

```
Given an array A of length n
Initialize min = A[0], max = A[0]
for i = 1 to n − 1 do
    if A[i] < min then
        min = A[i]
    end if
    if A[i] > max then
        max = A[i]
    end if
end for
Return min, max
```

Example 2.5: Exhaustive Search

Consider finding the minimum and maximum in $A = [7, 2, 9, 1, 5]$.

1. Start with $min = 7$, $max = 7$.
2. Compare $A[1] = 2$, update $min = 2$.
3. Compare $A[2] = 9$, update $max = 9$.
4. Compare $A[3] = 1$, update $min = 1$.
5. Compare $A[4] = 5$, no update needed.
6. Return $(1, 9)$.

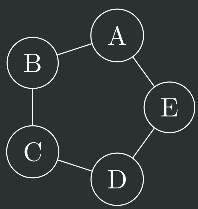
2.6 Depth-First Search (DFS)

Algorithm 2.6: Depth-First Search

Given a graph $G = (V, E)$ and starting node s
Initialize stack with s, mark s as visited
while stack is not empty **do**
  Pop node v from stack
  for each unvisited neighbor w of v **do**
    Mark w as visited
    Push w onto stack
  end for
end while

Example 2.6: Depth-First Search

Consider a graph:



DFS starting from A follows:

$$A \to B \to C \to E \to D$$

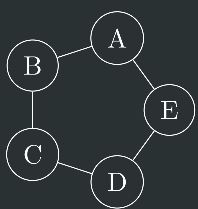
2.7 Breadth-First Search (BFS)

Algorithm 2.7: Breadth-First Search

Given a graph $G = (V, E)$ and starting node s
Initialize queue with s, mark s as visited
while queue is not empty **do**
  Dequeue node v from queue
  for each unvisited neighbor w of v **do**
    Mark w as visited
    Enqueue w
  end for
end while

Example 2.7: Breadth-First Search

Consider a graph:



BFS starting from A follows:

$$A \to B \to D \to C \to E$$