Loop Invariant :

1. Initialization
   - rule applies before the first iteration
2. Maintenance
   - rule holds true between iterations
3. Termination
   - rule holds true after the last iteration as well

Growth of functions

Psuedocode

```
1 time (c1)         public static int sumArray (int [] arr) {
1 time (c2)             ... do something
N+1 times (c3)          for (int i = 0; i < N; i++) {
N times (c4)                ... do something
                        }
1 time (c5)             ... do something
                    }
```

Adding up costs :

$c1 + c2 + N+1\,(c3) + N\,(c4) + c5$ [N is arr. length]

⋮

$N(c3 + c4) + c1 + c2 + c3 + c5 \longrightarrow ax + b$

$a = c3 + c4$

$b = c1 + c2 + c3 + c5$

```
1 time              someFunction () {
N times                 for (int i = 0; i < N; i++) {
(N)(N+1) times              for (int j = 0; j < N; j++) {
(N)(N) times                    ... do something
                            }
                        }
                    }
```

Adding up costs : ⋮

$ax^2 + bx + c$

Asymptotic Notation

3 Notations :

- Big - Oh : $O()$
- Big - Omega : $\Omega()$
- Theta : $\Theta()$

$O()$:

- upper bound of an algorithm's run time
- Formal Definition :
- $f(n) = O(g(n))$ if
  - $\exists$ constants c and n_0 such that
    - $f(n) \le c * g(n)$
    - for all $n \ge n_0$
- usually think about $n \ge 1$
- notation example :
  - $f(n) = 10n + 100$
    - one possible solution :
      - $10n + 100 \le 110n, \forall n \ge 1$
      - $\therefore 10n + 100 = O(n)$ [$c = 110$, $g(n) = n$]
- ex.
  - $4n + 7$
    - $4n + 7 \le 4n + 7n$
      - $4n + 7 \le 11n, \forall n \ge 1$
      - $\therefore 4n + 7 = O(n)$
- ex.
  - we can say for example :
    - $10n + 100 \le 110n^2, \forall n \ge 1$ i.e. $f(n) = O(n^2)$

      or

      $10n + 100 \le 110n^3, \forall n \ge 1$ i.e. $f(n) = O(n^3)$
    - while we can say this, we want to use a close $g(n)$, it's not helpful to say I am shorter than an elephant

$\Omega()$

- lower bound of an algorithm's run time
- Formal Definition :
  - $f(n) = \Omega(g(n))$ if
    - $\exists$ constants c and n_0 such that
      - $c * g(n) \le f(n)$
      - $\forall n \ge n_0$
- ex.
  - $f(n) = 10n + 100$
    - $n \le 10n + 100, \forall n \ge 1$

$\Theta()$

- Both upper and lower bound of an algorithm's run time
- Formal Definition :
  - $f(n) = \Theta(g(n))$ if :
    - $\exists$ constants c_1, c_2, and n_0 such that
      - $c_1 * g(n) \le f(n) \le c_2 * g(n)$
      - $\forall n \ge n_0$
  - ex.
    - $f(n) = 10n + 100$
      - $n \le 10n + 100 \le 110n, \forall n \ge 1$
- Practice finding $O(n)$, $\Omega(n)$, $\Theta(n)$
  - ex.
    - $f(n) = n^2 + 2n + 1$ ($n_0 = 1$)
    - $O(n)$: $n^2 + 2n + 1 \le n^2 + 2n^2 + n^2 \rightarrow f(n) = O(n^2)$
    - $\Omega(n)$: $n^2 \le n^2 + 2n + 1 \rightarrow f(n) = \Omega(n^2)$
    - Since $f(n) = O(n^2)$ and $f(n) = \Omega(n^2)$
      - $f(n) = \Theta(n^2)$
  - ex.
    - $f(n) = n!$ ($n_0 = 1$)
    - $O()$: $n * (n-1) * (n-2) \ldots 1 \le n * n * n \ldots * n \rightarrow f(n) = O(n^n)$
    - $\Omega()$: $1 * 1 * 1 \ldots * 1 \le n * (n-1) * (n-2) \ldots 1 \rightarrow f(n) = \Omega(1)$
    - Since $O()$ and $\Omega()$ are different, we cannot define $\Theta()$
      - we can only say $f(n) = O(n^n)$ and $f(n) = \Omega(1)$
  - ex.

```
1 time (c1)         public static int sumArray (int [] arr) {
1 time (c2)             ... do something
N+1 times (c3)          for (int i = 0; i < N; i++) {
N times (c4)                ... do something
                        }
1 time (c5)             ... do something
                    }
```

Polynomial form : $ax + b$

Order of growth : $O(n)$

Best Case, Worst Case, Average Case

Psuedocode

```
def linearSearch (arr, key) :
    for i in range (len(arr)) :
        if arr[i] == key :
            return i
    return -1
```

- ex.

start

7	4	3

1 comparison

7	4	3	5

1 comparison

7	4	3	5	6

1 comparison

Best case : $O(1)$
$\Omega(1)$
$\Theta(1)$

It's best case when 7 is at the start because we don't need to compare to another element, and size of array does not change number of comparisons

Worst case ex.

1	4	7	3	5

3	4	7

3 comparisons

5	4	3	7

4 comparisons

6	3	4	5	7

5 comparisons

- $O(n)$
- $\Omega(n)$
- $\Theta(n)$

Input is changing based on size of the array

- **Average Case**

1	4	7	3	2

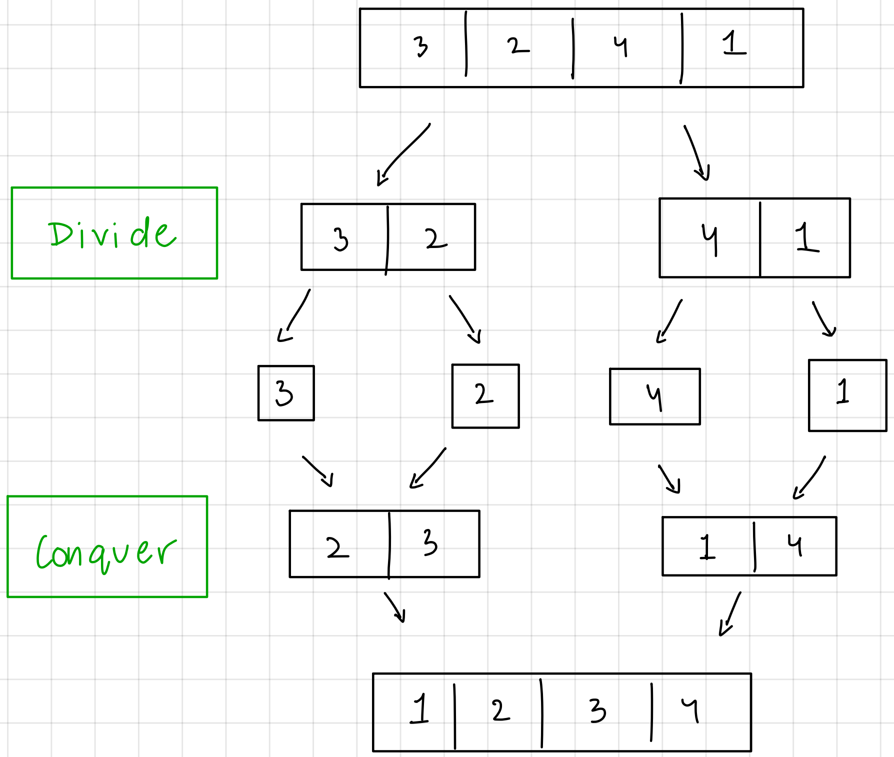
- $O(n)$
- $\Omega(n)$
- $\Theta(n)$

We get the average over all possible combinations

$$= \frac{(1+2+3 \ldots n)}{n} = \frac{(n)(n+1)}{n}$$

Divide and Conquer

1. **Divide** problem into one or more subproblems that are similar
2. **Conquer** the subproblems by solving them recursively
3. **Combine** the subproblem solutions to form a solution to the original problem



Recurrence Relation

Assume time complexity for someFunc is $T(N)$

```
T(N)    void someFunc (int n) {
c1          if (n > 0) {
c2              System.out.println("n");
T(N-1)          someFunc(n-1);
            }
        }
```

For convenience, let's say $c_1 + c_2 = 1$

- Then for someFunc, we can say that

$T(N) = T(N-1) + 1$

$$T(N) \begin{cases} 1 & n = 0 \\ T(N-1) + 1 & n > 0 \end{cases}$$

$\because T(N) = T(N-1) + 1$ — Eq. 1
$\therefore T(N-1) = T(N-2) + 1$ — Eq. 2
$\because T(N-2) = T(N-3) + 1$ — Eq. 3

Substitute Eq. 2 into Eq. 1

$T(N) = (T(N-2) + 1) + 1 = T(N-2) + 2$

Similarily, Eq. 3 → Eq. 2

$T(N) = T(N-3) + 3$

If we perform k such substitutions, we get

$T(N) = T(N-k) + k$

Let's say $N - k = 0$

$\therefore N = k$
$\therefore T(N) = T(0) + N$
$\therefore T(N) = 1 + N$

- So, someFunc has an order of growth of $f(n) = N + 1$
  - Time complexity of $\Theta(n)$

Master Theorem

- For "decreasing" functions
  - For the recurrence relation

$$T(N) = aT(N-b) + f(n)$$

where $a, b > 0$

$f(n) = O(N^k)$

$k \geq 0$

1. If $a < 1$ then $T(n) = O(n^k)$ or $O(f(n))$
2. If $a = 1$ then $T(n) = O(n^{k+1})$ or $O(n * f(n))$
3. If $a > 1$ then $T(n) = O(n^k a^{\frac{n}{b}})$

- For divide and conquer algorithms

$$T(N) = aT\left(\frac{N}{b}\right) + f(n)$$

where $a \geq 1$

$b > 1$

1. $f(n) = O(n^{\log_b a - \epsilon})$ where $\epsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a} \log^k n)$ where $k \geq 0$ then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3. $f(n) = \Omega(n^{\log_b a + \epsilon})$ where $\epsilon > 0$ then $T(n) = \Theta(f(n))$

Additional condition for case 3: $a * f(\frac{n}{b}) \leq c * f(n)$, $c > 1$, $n > n_0$

Alternate Divide and conquer Master Theorem

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

where $a > 0$

$b > 1$

$d \geq 0$

1. $\Theta(n^d)$ if $d > \log_b a$
2. $\Theta(n^d \log n)$ if $d = \log_b a$
3. $\Theta(n^{\log_b a})$ if $d < \log_b a$