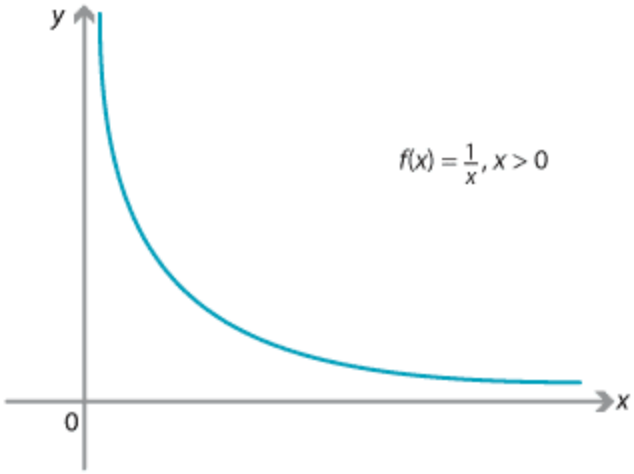
Complexity Analysis

Asymptotic Behavior

- **Definition**: Studying the behavior of an algorithm as its input size approaches to infinity (∞).
- We are interested in addressing the ***tail behavior***.



Tail Behavior of infinity graph

- **Question:** Why we drop constants ? ***Addressing Behavior***
  i. Because we are concerned with the rate of growth and how the algorithm would behave in the large input.
  ii. Focusing on dominating factors, as input growth rate is more dominant than how many times it's repeated.
  ex: $O(2n^2 + n) = O(n^2)$

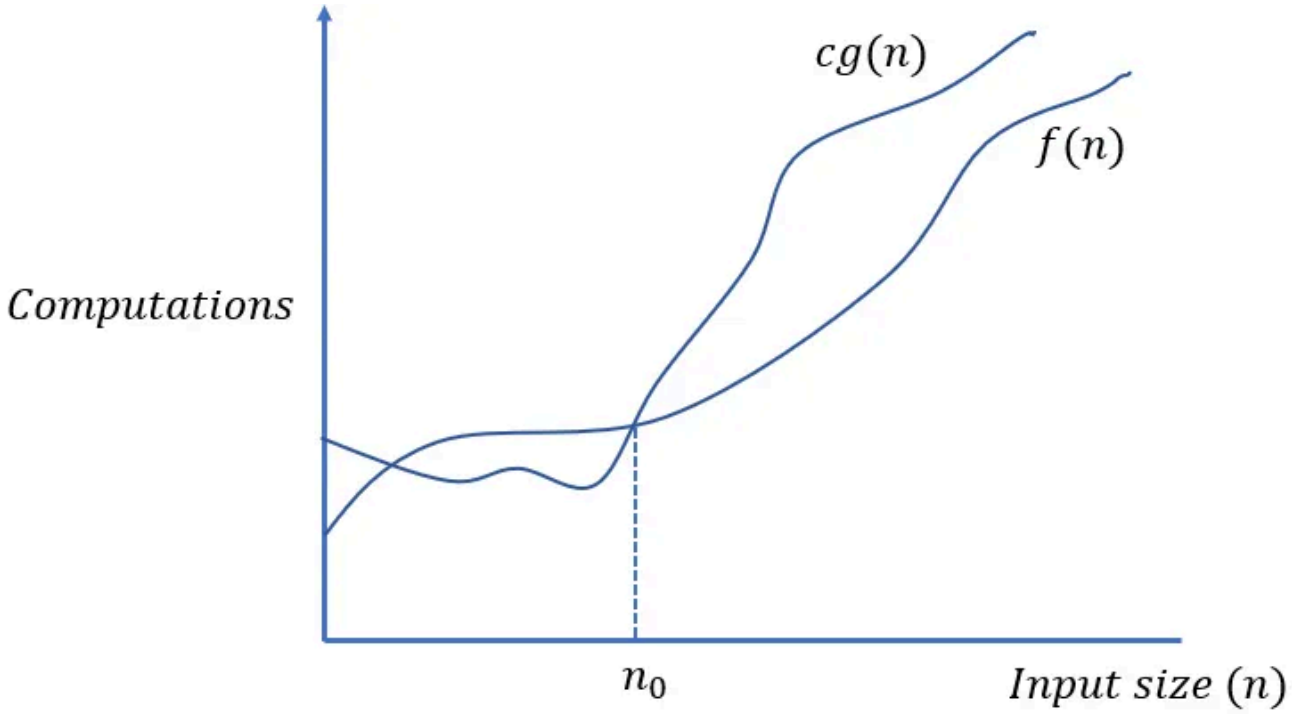
Big-O, Big-Ω and Big-Θ Notation

Big-Oh (O) Notation: Upper Bound (Worst Case)

- Used to describe the asymptotic upper bound of an algorithm, i.e. its worst-case scenario.
- An algorithm with a running time of $f(n)$ is equivalent to $O(g(n))$ if and only if there are constants c and $n0$ such that the below expression is satisfied.
- Equation:

$$0 \leq f(n) \leq cg(n) \quad for\ n \geq n_0$$

- The graph of this upper bound:

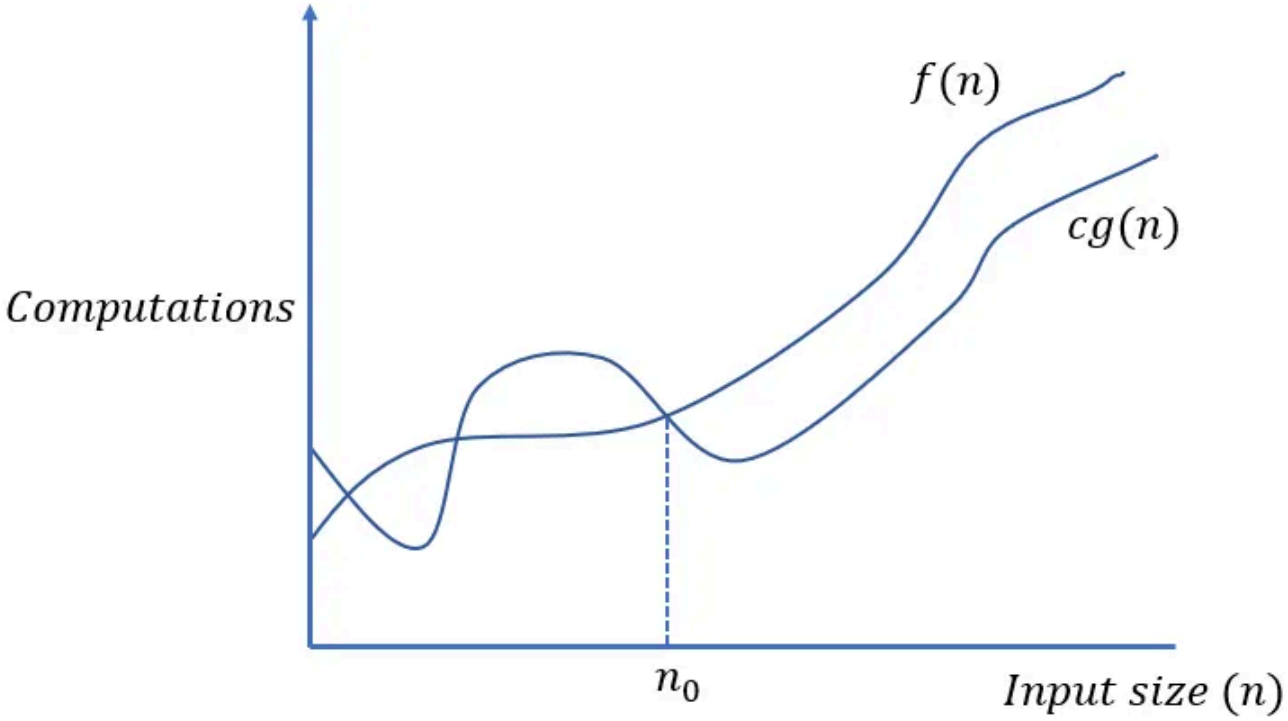


Big-Omega (Ω) Notation: Lower Bound (Best Case)

- Used to describe the asymptotic lower bound of an algorithm, i.e. its best-case scenario.
- An algorithm with a running time of $f(n)$ is equivalent to $\Omega(g(n))$ if and only if there are constants c and $n0$ such that the below expression is satisfied.
- Equation:

$$0 \leq cg(n) \leq f(n) \quad for\ n \geq n_0$$

- The graph of this lower bound:

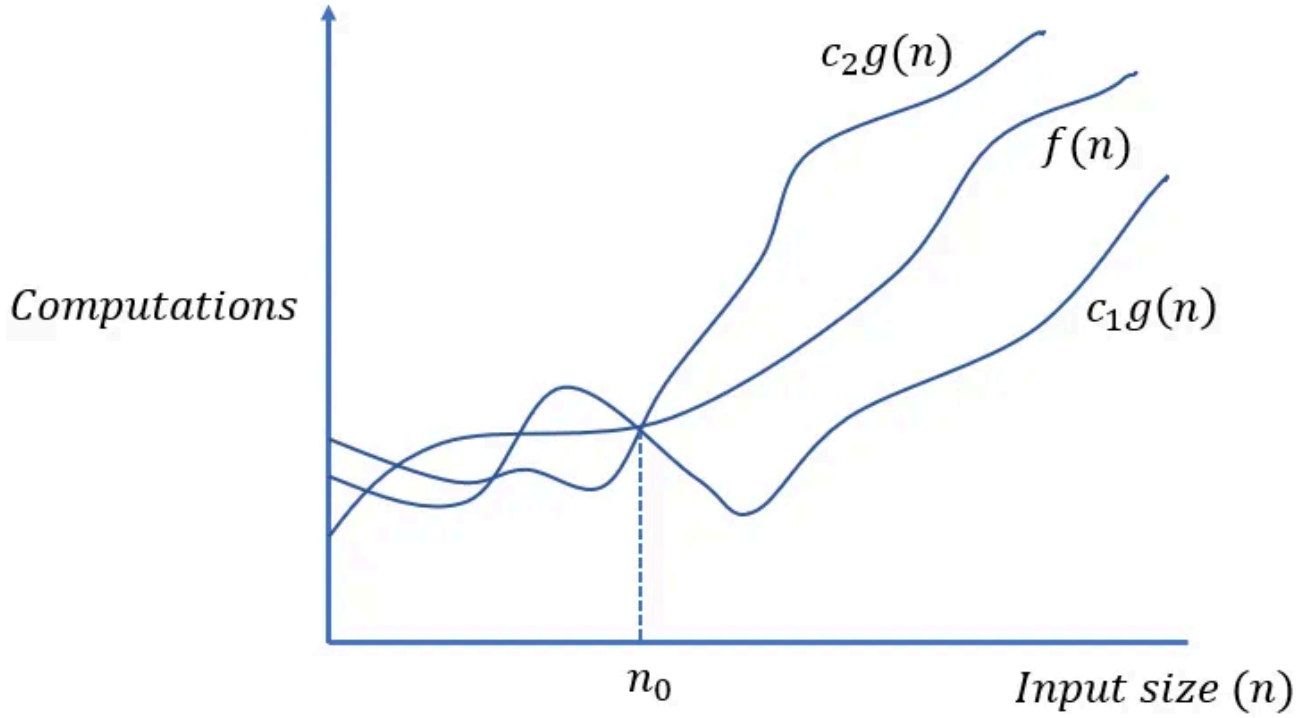


Big-Theta (Θ) Notation: (Average Case)

- Describes both the upper and lower bound of a function and is often referred to as the average time complexity.
- Equation:

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

- This can be visualized in the graph below:



How to compare complexities of different functions?

- Using this equation:

$$lim_{x \to \infty} \frac{f(n)}{g(n)}$$

- There are 4 cases for the result:
  i. if result is $Zero$ then: $g(n) > f(n)$.
  ii. if result is ∞ then: $g(n) < f(n)$.
  iii. if result is $constant$ then: $g(n) = f(n)$.
  iv. if result is ∞/∞ then use L'Hôpital's rule until the result is one of the 3 cases.

Exercise: Order the time complexity for these function from fastest to slowest.

1. $log(\sqrt{n})$
2. $\sqrt{n}$
3. $log(n)$
4. $log^2(n)$
5. $log(log(n))$
6. $n * log(n)$
7. n

8. $\sqrt{n} * log(n)$
9. $n * n$

Answer:

1. $log(log(n))$
2. $log(\sqrt{n}), log(n)$
3. $log^2(n)$
4. $\sqrt{n}$
5. $\sqrt{n} * log(n)$
6. n
7. $n * log(n)$
8. n^2

Recurrence Relation

- **Definition:** Mathematical way to define terms of sequence with respect to term that come before them.
- Solving recurrence relation means that we find a close expression of non-recurrence formula for the n^{th} term of the sequence without using any subscripts.
- Example:

```
void f(int n) {
  if (n == 0) return;
  f(n - 1);
  for (int i = 0; i < n; i++) cout << "text\n";
}
```

- Recurrence Relation:

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

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

$$T(n-2) = T(n-3) + n - 2$$

By Substitution:

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

$$T(n) = T(n-3) + n - 2 + n - 1 + n$$

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

$so,$

$$T(n) = T(n-k) + 3k$$

at $k = n$:

$$T(n) = T(0) + n^2$$

where:

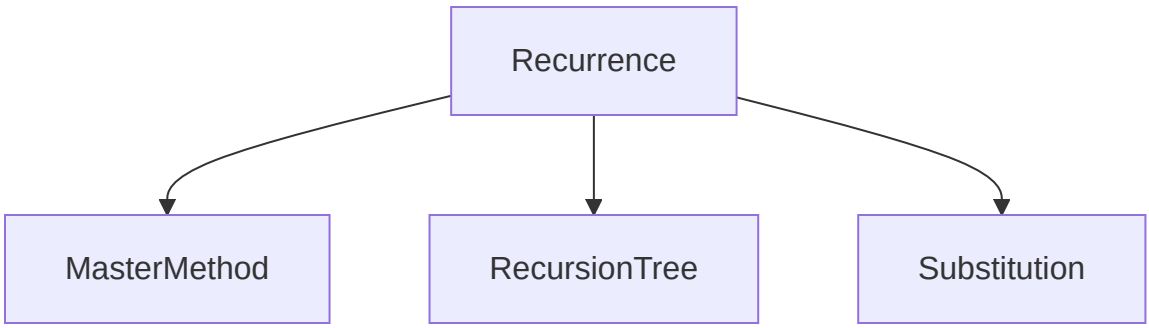
$$T(0) = 1$$

$so,$

$$T(n) = n^2$$

Time Complexity for this function: $\Theta(n^2)$

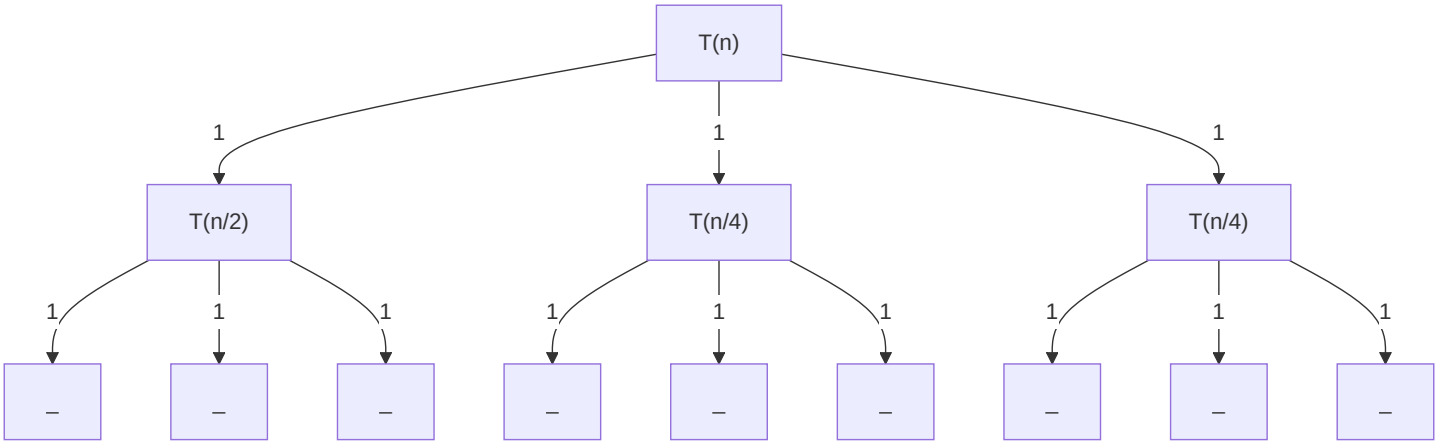
Recurrence Relation Types



Examples:

1. $T(n) = T(\frac{n}{2}) + 2T(\frac{n}{4}) + 1$

Solution



We have 2 heights:

- *worst* = $log_2(n)$ for the $T(\frac{n}{2})$ branch.
- *best* = $log_4(n)$ for the $T(\frac{n}{4})$ branch.
- The equation that describe the recurrsion tree:

$$T(n) = \sum_{k=0}^{height} 3^k = \frac{1 * (3^{height} - 1)}{2}$$

$so,$

- In the worst case:

$$T(n) = O(3^{log_2(n)}) = O(n^{log_2(3)})$$

- In the best case:

$$T(n) = \Omega(3^{log_4(n)}) = \Omega(n^{log_4(3)})$$

2. $T(n) = 2T(n-1) + 1$

Solution

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

$$T(n-2) = 2T(n-3) + 1$$

$so,$

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

$$T(n) = 2(2T(n-2)+1)+1 = 4T(n-2)+3$$

$$T(n) = 2(2T(n-3)+1)+2 = 8T(n-3)+7$$

so,

$$T(n) = 2^kT(n-k)+2^k-1$$

at $k = n$,

$$T(0) = 1$$

so,

$$T(n) = 2^nT(0)+2^n-1$$

$$T(n) = 2(2^n)-1$$

$$T(n) = 2^{n+1}-1$$

Time Complexity for this function: $\Theta(2^n)$

3. $T(n) = T(\sqrt{n})+1$

Solution

$$T(n) = T(n^{\frac{1}{2}})+1$$

$$T(n^{\frac{1}{2}}) = T(n^{\frac{1}{4}})+1$$

$$T(n) = [T(n^{\frac{1}{4}})+1]+1 = T(n^{\frac{1}{4}})+2$$

$$T(n) = T(n^{\frac{1}{8}})+3$$

so,

$$T(n) = T(n^{\frac{1}{2^k}})+k$$

will exit condition at $k = log(log(n))$

$$T(n) = T(2)+log(log(n))$$

Time Complexity for this function: $\Theta(log(log(n)))$

Mathematical Explanation:

$$n^{\frac{1}{2^k}} = 2$$

$$log_2(n^{\frac{1}{2^k}}) = log_2(2)$$

$$\frac{1}{2^k} log_2(n) = 1$$

$$log_2(n) = 2^k$$

$$log(log(n)) = klog(2) = k$$

$$k = log(log(n))$$

4. What is the time and space complexity for this code ?

```
void f(int n) {
    if (n == 0) return;
    int *arr = new int[n];
    f(n - 1);
    for (int i = 0; i < n; i++) memcpy(size n);
}
```

Solution

- Runtime: $T(n) = T(n-1) + n^2$
- Heap: $T(n) = T(n-1) + n^2$
  - if we use `delete[] arr` it will be $T(n) = \Theta(n)$
- Stack: $T(n) = T(n-1) + n$
  - in case of `for (i -> n) f();` then $T(n) = nT(n-1) + 1$

5. What is the time complexity for these functions ?
   i. $T(n) = T(n-1) + 1$
   ii. $T(n) = T(n-1) + n$
   iii. $T(n) = 2T(n-1) + 1$

Solution

1. $\Theta(n)$
2. $\Theta(n^2)$
3. $\Theta(2^n)$

6. How to get the k^{th} node from the last ?

Solution

- Solution 1: store nodes in stack.
  - Time: $O(n)$
  - Space: $O(n)$
- Solution 2: two passes loops.
  - Time: $O(2n)$
  - Space: $O(1)$
- Solution 3: two pointers (fast and slow), one is ahead by k.
  - Time: $O(n)$
  - Space: $O(1)$

7. How to delete **Node 2** without pointer to the previous ?

Solution

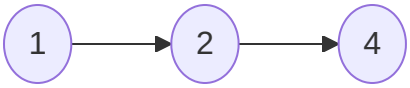
Copy the data from **node 3** to **node 2** and then modify the next pointer of **node 2** to point to **node 4** (hence **node 3** will be deleted and the sequence of the data will be correct)

8. How to modify the list to make all nodes that have values less than pivot (5) on the left of it and all nodes that have values greater than pivot on the right of it ?

Solution

Create 3 pointers:

- `node *smaller = null` then smaller:



- `node *piv = null` then pivot:

- `node *larger = null` then larger: