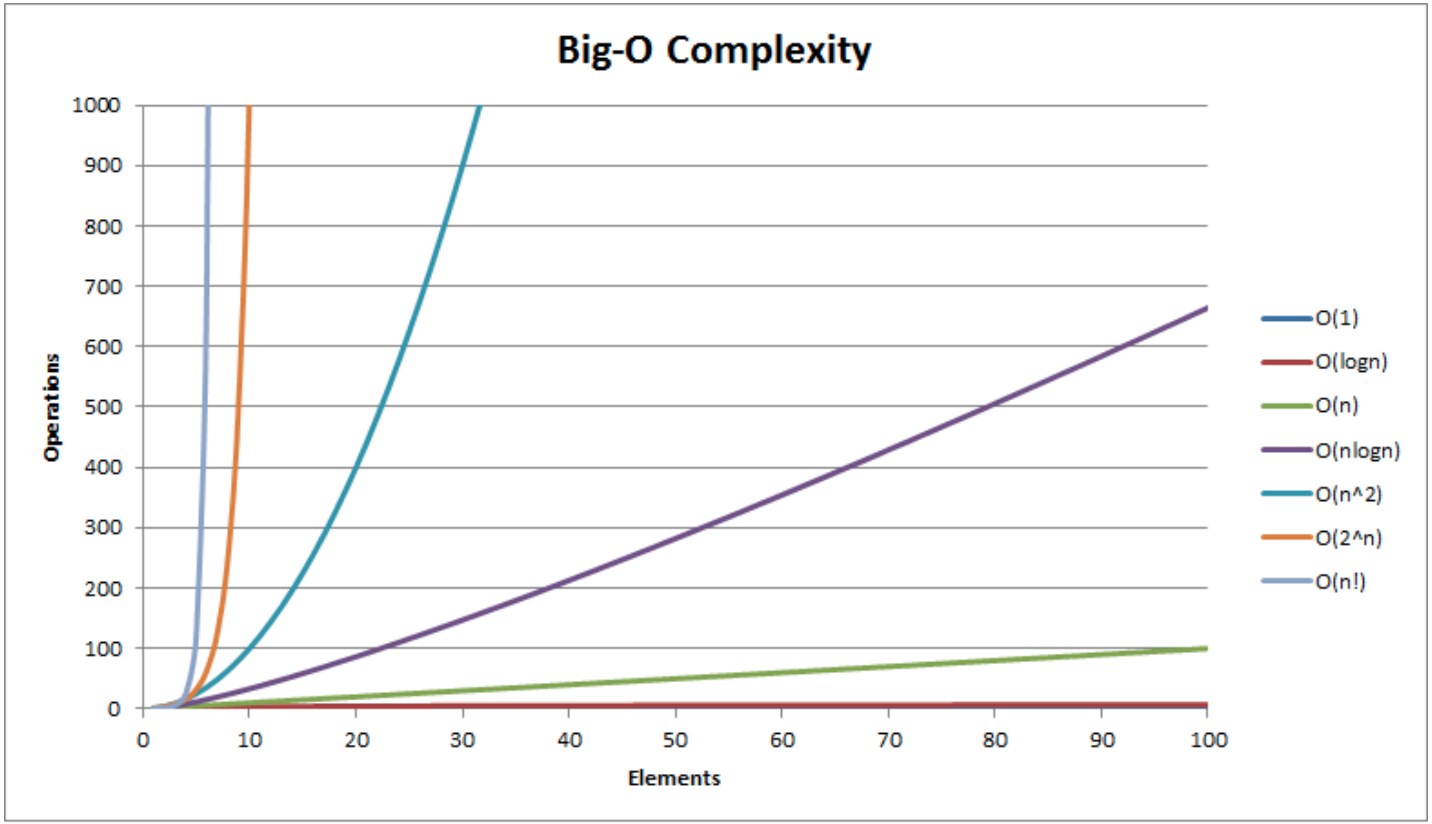
Big-O Analysis

CSE373 - Help Section

Big-O Basics


Big-O Complexity
Operations
1000
900
800
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
70
80
90
100
Elements
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

Big-O Basics

- Basic Order:

  O(1) < O(logn) < O(n) < O(nlogn) < O(n^2) < O(2^n) < O(n!)

- $n^3 > n^2 log(n)$
- $3^n < n!$
- $log(log(n)) < log(n)$

Big-O Basics

- Useful Rules:
  Lower order terms doesn't matter
  $a_0 + a_1 n + \ldots + a_d n^d$ is $O(n^d)$ if $a_d > 0$
- $n^2 + n$
- $2^n + n^5 + 2n$
- $n + log(n)$

Big-O Basics

- Useful Rules:

  Log base doesn't matter.

  $O(\log_a n) = O(\log_b n)$ for any a, b > 0
- *$O(\log_2(n)) = O(\log_4(n))$*

Big-O Basics

- Useful Rules:

  Log grows slower than every polynomial

  $\log(n) = O(n^x)$ for any $x > 0$

- *$\log(n) < n^{0.01}$*

Big-O Basics

- Useful Rules:

  Every exponential grows faster than every polynomial.

  $n^d = O(r^n)$ for all $r > 1$ and all $d > 0$.

- $N^{100} < 1.01^n$

Big-O Definition

- g(n) is in O(f(n)) if there exist positive constants c and n_0 such that

  g(n) <= c * f(n) for all n >= n_0

- $g(n) = 4n$, $f(n) = n$, $c = 5$, $n_0 = 1$
- $g(n) = 100n$, $f(n) = n^2$, $c = 1$, $n_0 = 100$
- $g(n) = n^3$, $f(n) = 2^n$, $c = 2$, $n_0 = 2$

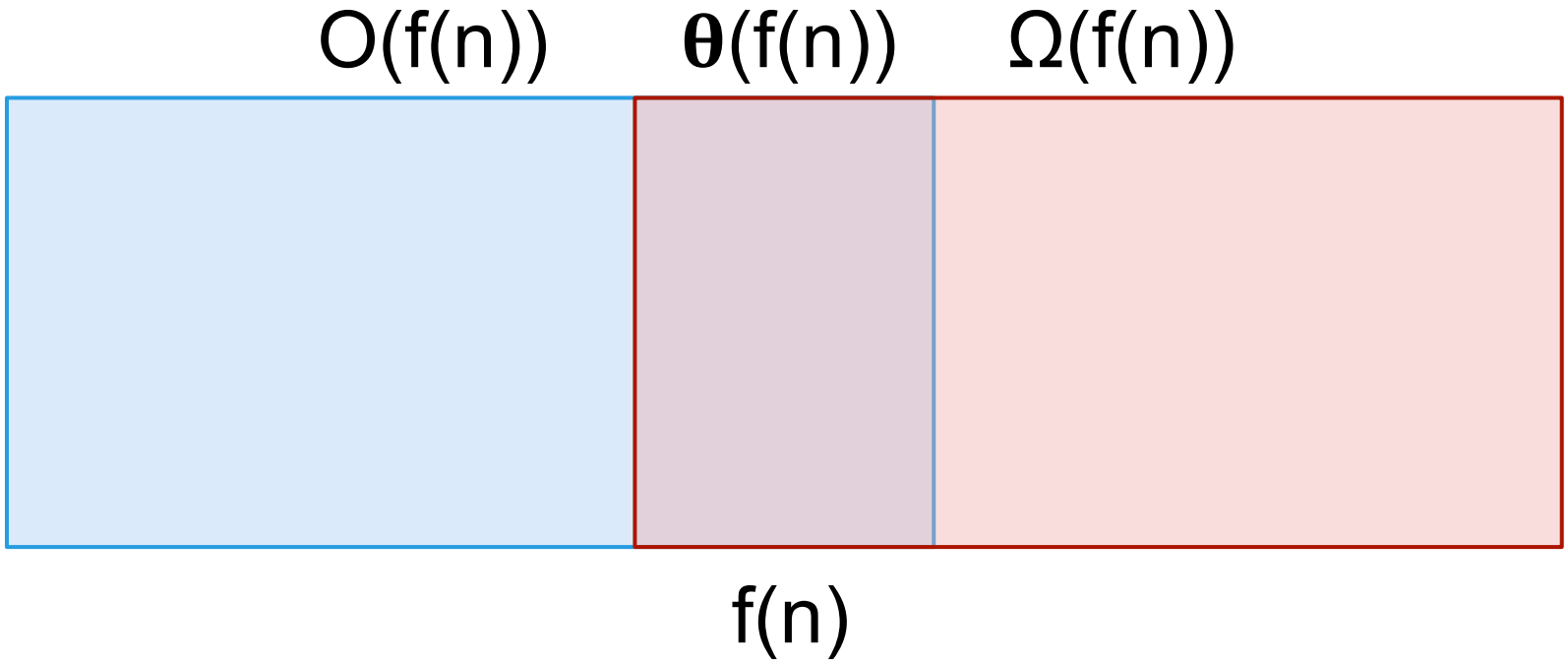
About Ω & θ

- g(n) is in Ω(f(n)) if there exist positive constants c and n_0 such that

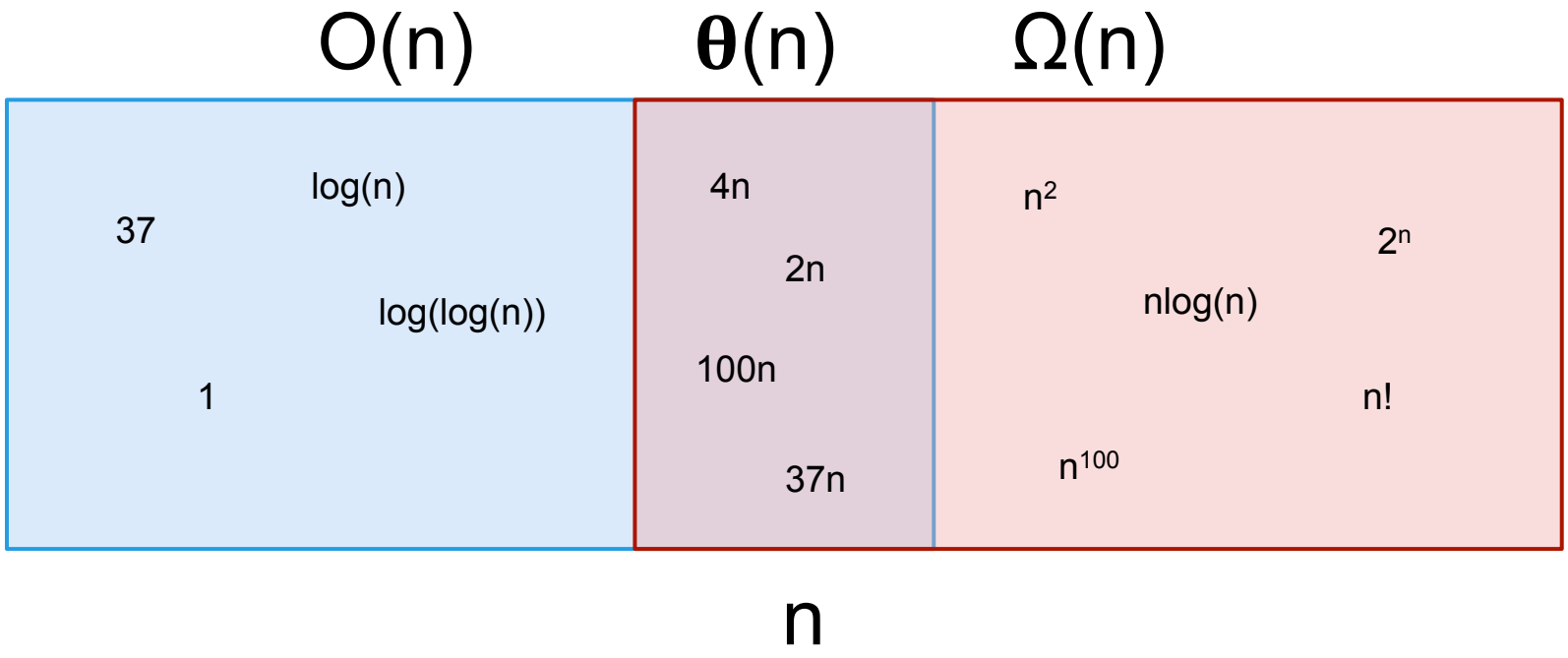
  g(n) **>=** c * f(n) for all n >= n_0

- g(n) is in **θ**(f(n)) if:
  - g(n) is in O(f(n))
  - g(n) is in Ω(f(n))

About Ω & θ



About Ω & θ



Algorithm Analysis

```
int min (int[] arr) {
    int min = arr[0];
    for (int i = 1; i < arr.length; i++)  {
        if (arr[i] < min) {
            min = arr[i]
        }
    }
    return min;
}
```

O(n)

Algorithm Analysis

```
int min (int[][] arr) {
    int min = arr[0][0];
    for (int i = 0; i < arr.length; i++)  {
            for (int j = 0; j < arr[0].length; j++){
            if (arr[i][j] < min) {
                    min = arr[i][j];
            }
        }
    }
    return min;
}
```

$O(n^2)$

Algorithm Analysis

```
int sum (int n) {
    int sum = 0;
    for (int i = 0; i < n; i++)  {
            for (int j = 0; j < i * i * i; j++){
            sum += j;
        }
    }
    return sum;
}
```

$$\sum_{i=1}^{n} i^3 = \left(\sum_{i=1}^{n} i\right)^2$$

$O(n^4)$

Algorithm Analysis

```
int factorial (int n) {
    if (n == 1){
        return 1;
    } else {
        return factorial(n - 1) * n;
    }
}
```

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

$O(n)$

Algorithm Analysis

```
boolean find(int[]arr, int k){
    return help(arr,k,0,arr.length);
}
boolean help(int[]arr, int k, int lo, int hi) {
    int mid = (hi+lo)/2;
    if(lo==hi)                          return false;
    if(arr[mid]==k)          return true;
    if(arr[mid]< k)          return help(arr,k,mid+1,hi);
    else                                return help(arr,k,lo,mid);
}
```

$$T(n) = C_2 + T(n / 2)$$

Algorithm Analysis

- $T(n) = C_2 + T(n / 2)$
  $= C_2 + C_2 + T(n / 4)$
  $= C_2 + C_2 + C_2 + T(n / 8)$
  $= C_2 * k + T(n / 2^k)$
- Let $n / 2^k = 1$
  $\Rightarrow n = 2^k$
  $\Rightarrow k = \log_2(n)$
- $T(n) = C_2 * \log_2(n) + T(1)$
  $= O(\log(n))$

Algorithm Analysis

- $T(n) = n * C_2 + T(n / 2)$
  $= n * C_2 + n * C_2 + T(n / 4)$
  $= n * C_2 + n * C_2 + n * C_2 + T(n / 8)$
  $= C_2 * n * k + T(n / 2^k)$
- Let $n / 2^k = 1$
  $\Rightarrow n = 2^k$
  $\Rightarrow k = \log_2(n)$
- $T(n) = C_2 * n * \log_2(n) + T(1)$
  $= O(n \log(n))$