Algorithm Analysis

- The running time analysis of algorithms involves the counting and finding the final sum $f(n)$ of the "atomic" operations required by the algorithm with input size n
- The definition of an atomic operation depends on what the algorithm does in principle, and what we need to count in order to understand and quantify its running time
- For example, for a searching algorithm, we will need to count the number of "comparisons", because that what searching does: finding the index of x in a list L if x is the in list, and finding -1 if x is not in the list
- At the it step, we compare x to the ith element of the list: if $L_i = x$, we found x and its index is i

Algorithm Analysis

- The atomic operation will be different for other algorithms, for example, we will have to count the number of "swap" operations in a sorting algorithm, and the number of integer additions and multiplications in an algorithm that computes the product of two $n \times n$ matrices
- Furthermore, when we count the number of operations, we will need to consider "the worst case" or "the average case" or "the best case" situations, in terms the input configuration, values or other factors
- Therefore, we would be performing the the worst case, the average case or the best case analysis of the algorithm we are considering
- In most situations, we will have to consider the worst case analysis in order to make a fair assessment of the algorithm, and to compare it to other algorithms computing the same output

Binary Exponentiation Algorithm

- The following algorithm computes x^e given the input x (which can be an integer, or a real number, or even a matrix) and the exponent e (which is a positive integer)
- e_i be the ith bit of the exponent, for $0 \leq i < n$, in other words $e = (e_{n-1} e_{n-2} \cdots e_1 e_0)$
- The input size of the algorithm is n which is the number of bits in e

```
Step 1:    y = 1
Step 2:    for i=n-1 down to 0
Step 3:        y = y * y
Step 4:        if (e_i=1) y = y * x
Step 5:    return y
```

Analysis of Binary Exponentiation Algorithm

- Obviously the atomic operation in this algorithm is the multiplication or squaring operation: `y = y * y` and `y = y * x`
- We will count the number of multiplications (assuming squaring is the same operation)
- The running time of the binary exponentiation algorithm $f(n)$ is the number of multiplications, which depends on the bit configuration of the exponent
- The exponent e could be $(100\cdots0)$ or $(111\cdots1)$ or $(1010\cdots10)$, giving us different $f(n)$ values
- The more 1s in the binary expansion of e implies, the more we need to perform multiplications in Step 4

Analysis of Binary Exponentiation Algorithm

- The best case analysis: there exists only a single 1 in the binary expansion of the exponent: $e = (1000 \cdots 0)$
  In Step 3, we will have n multiplications, and in Step 4, we will have only 1, therefore, $f_{best}(n) = n + 1$
- The worst case analysis: there exists only n 1s in the binary expansion of the exponent: $e = (1111 \cdots 1)$
  In Step 3, we will have n multiplications, and in Step 4, we will also have n multiplications, therefore, $f_{worst}(n) = 2n$
- The average case analysis: Assuming each exponent bit is equal 1 or 0 with equal probability, there will be about $n/2$ 1s and $n/2$ 0s in the binary expansion of the exponent
  In Step 3, we will have n multiplications, and in Step 4, we will have $n/2$ multiplications, therefore, $f_{average}(n) = 1.5n$

Big-O, Big-Omega, and Big-Theta

- Once we have the (best, average, worst) running time function $f(n)$, we can then compute its order:
  - Big-O: $f(n) = O(g_1(n))$
  - Big-Omega: $f(n) = \Omega(g_2(n))$
  - Big-Theta: $f(n) = \Theta(g_3(n))$
- The growth of functions helps to obtain the order of $f(n)$
- This is a separate step from the counting operation
- It helps us to quickly ascertain the performance of the algorithm we have analyzed

Definition of Big-O

- Let $f(n)$ be a positive valued function for $n = 1, 2, 3, \ldots$
- We say $f(n) = O(g(n))$ if there exists a positive constant C and index n_0 such that

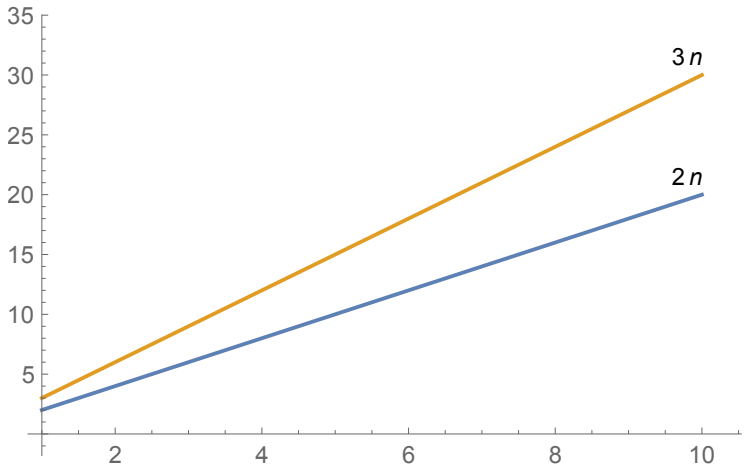
$$f(n) \leq C \cdot g(n) \qquad \text{for} \quad n \geq n_0$$

Big-O Examples

- From the previous slides we found the (worst case) number of multiplications in computing x^e for an n-bit exponent as $f(n) = 2n$
- We can write $f(n) = O(n)$ since

$$f(n) \leq C \cdot n \qquad \text{for} \quad n > n_0$$

is true for $C = 3$ and $n_0 = 1$



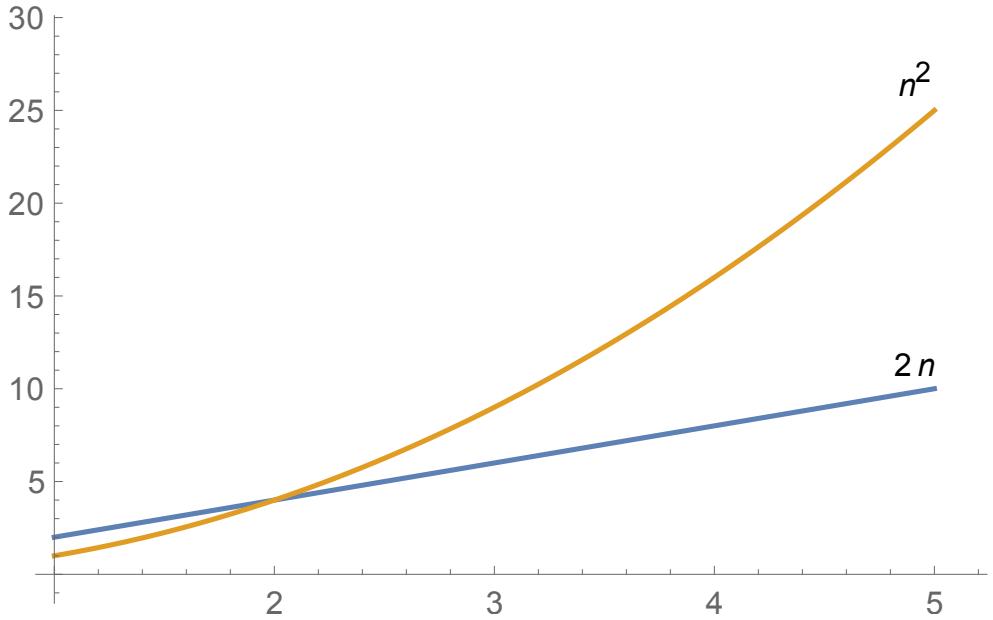
- $f(n) = O(n)$ is also true $C = 2.1$ and $n_0 = 1$
- We only need to find one C and it must be true for all $n \geq n_0$

Big-O Examples

- On the the other hand, it is also true that $f(n) = O(n^2)$ since

$$f(n) \leq C \cdot n^2 \qquad \text{for} \quad n > n_0$$

is true for $C = 1$ and $n_0 = 2$

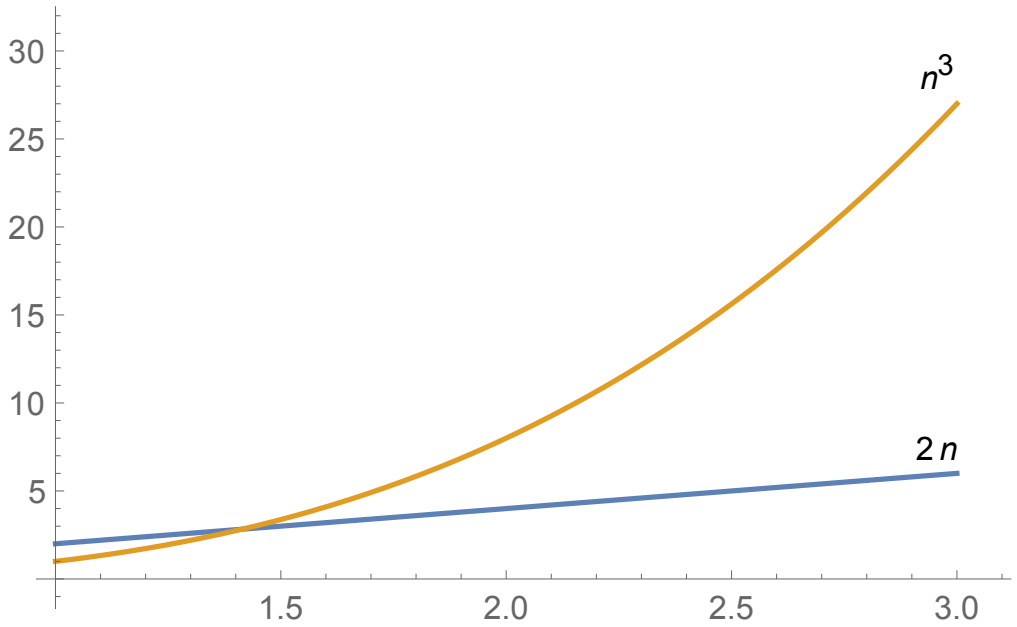


Big-O Examples

- On the the other hand, it is also true that $f(n) = O(n^3)$ since

$$f(n) \leq C \cdot n^3 \qquad \text{for} \quad n > n_0$$

is true for $C = 1$ and $n_0 = 2$

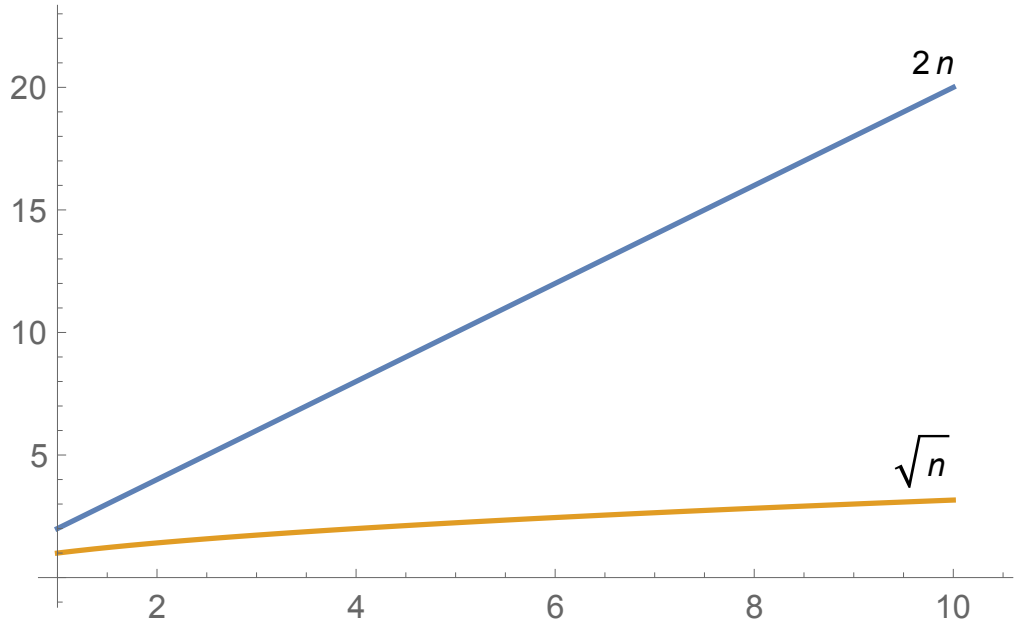


Big-O Examples

- However, there does NOT exist a C and n_0 such that

$$2n \leq C \cdot \sqrt{n} \quad \text{for} \quad n > n_0$$

- Therefore, $2n \neq O(\sqrt{n})$



Big-O Examples

$f(n) = 2n$

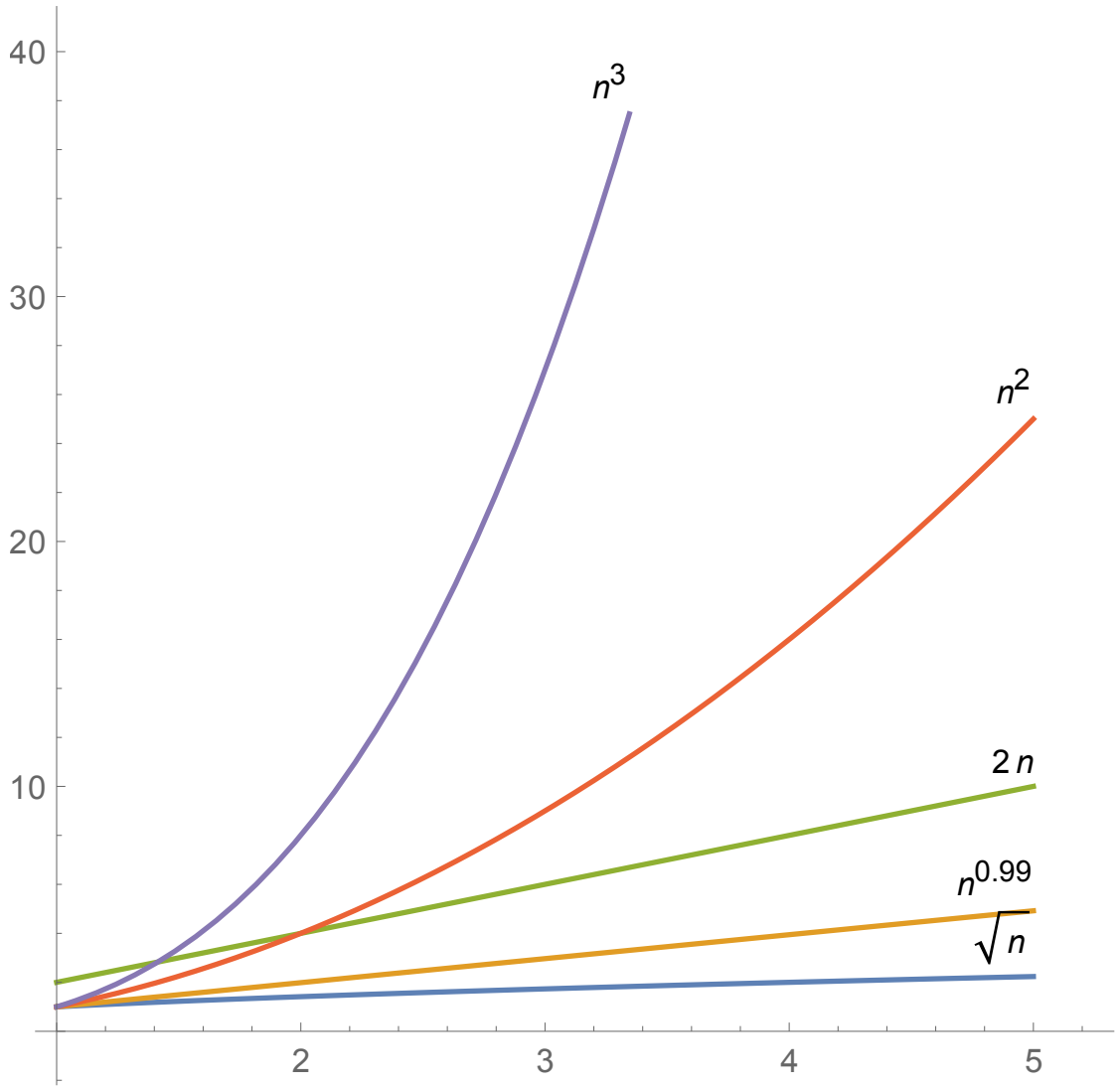
$f(n) = O(n)$

$f(n) = O(n^2)$

$f(n) = O(n^3)$

$f(n) \neq O(n^{0.99})$

$f(n) \neq O(\sqrt{n})$



Definition of Big-Omega

- Let $f(n)$ be a positive valued function for $n = 1, 2, 3, \ldots$
- We say $f(n) = \Omega(g(n))$ if there exists a positive constant C and index n_0 such that

$$f(n) \geq C \cdot g(n) \qquad \text{for} \quad n \geq n_0$$

- For $f(n) = 2n$, we can write $f(n) = \Omega(n)$ since

$$f(n) \geq C \cdot n \qquad \text{for} \quad n > n_0$$

  is true for $C = 1$ and $n_0 = 1$
- Therefore, $2n = \Omega(n)$

Big-Omega Examples

- For $f(n) = 2n$, we can write $f(n) = \Omega(n^{0.99})$ since

$$f(n) \geq C \cdot n \quad \text{for} \quad n > n_0$$

  is true for $C = 1$ and $n_0 = 1$
- Therefore, $2n = \Omega(n^{0.99})$

- For $f(n) = 2n$, we can write $f(n) = \Omega(\sqrt{n}))$ since

$$f(n) \geq C \cdot n \quad \text{for} \quad n > n_0$$

  is true for $C = 1$ and $n_0 = 1$
- Therefore, $2n = \Omega(\sqrt{n})$

Big-Omega Examples

- However, $f(n) = 2n$, there does NOT exist C and n_0 such that

$$f(n) \geq C \cdot n^2 \qquad \text{for} \quad n > n_0$$

- Therefore, $2n \neq \Omega(n^2)$

- However, $f(n) = 2n$, there does NOT exist C and n_0 such that

$$f(n) \geq C \cdot n^3 \qquad \text{for} \quad n > n_0$$

- Therefore, $2n \neq \Omega(n^3)$

Big-Omega Examples

$f(n) = 2n$

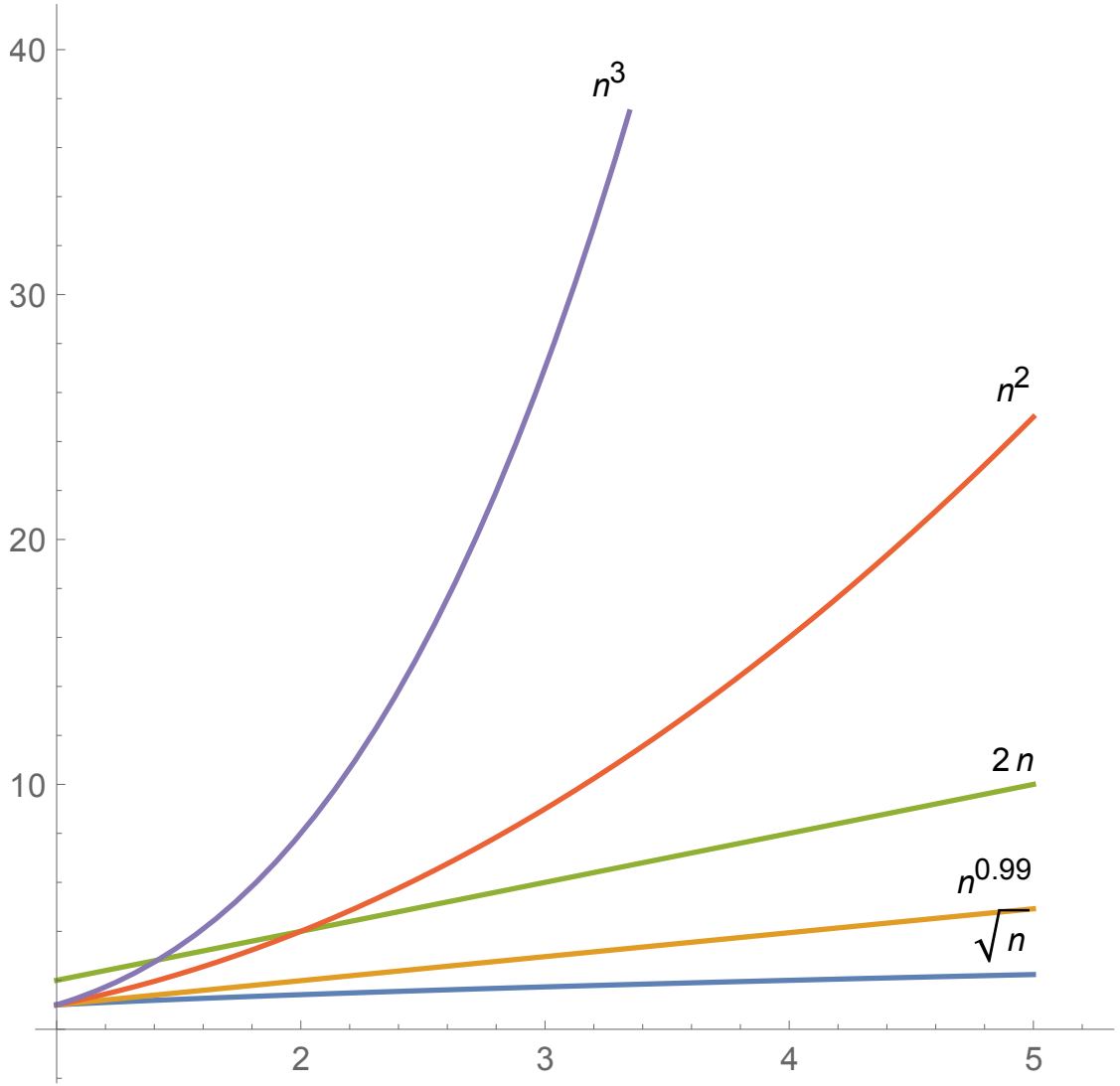
$f(n) = \Omega(n)$

$f(n) = \Omega(\sqrt{n})$

$f(n) = \Omega(n^{0.99})$

$f(n) \neq \Omega(n^2)$

$f(n) \neq \Omega(n^3)$



Definition of Big-Theta

- Let $f(n)$ be a positive valued function for $n = 1, 2, 3, \ldots$
- If there exists a function $g(n)$ such that $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then we say $f(n) = \Theta(g(n))$
- In other words: if there exist positive constants C_1 and C_2, and index n_0 such that

$$C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n) \qquad \text{for} \quad n \geq n_0$$

- We discovered that for $f(n) = 2n$, we have

$$f(n) = O(n) \quad \text{and} \quad f(n) = \Omega(n)$$

Therefore, $f(n) = \Theta(n)$