Asymptotic analysis

Asymptotic analysis or "big O" notation is a way of describing the growth of the runtime of an algorithm without without having to worry about different computers, compilers, or implementations.

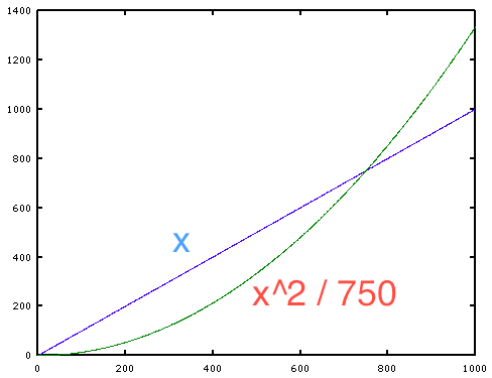
For functions $f(n)$, $g(n)$, $O(g(n))$ is a class of functions such that $f(n) \in O(g(n))$ if there exist M, x_0 such that

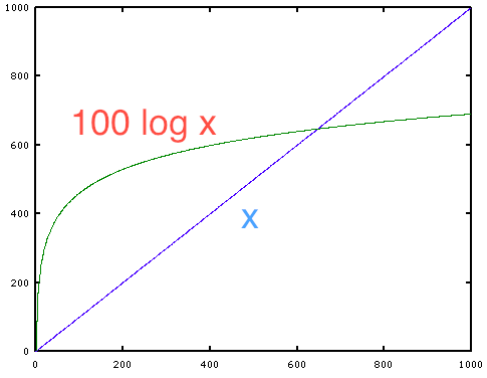
$$|f(n)| \leq M \cdot |g(n)| \text{ for all } x > x_0.$$

Similarly, $f(n) \in \Omega(g(n))$ if there exist M, x_0 such that

$$|f(n)| \geq M \cdot |g(n)| \text{ for all } x > x_0.$$

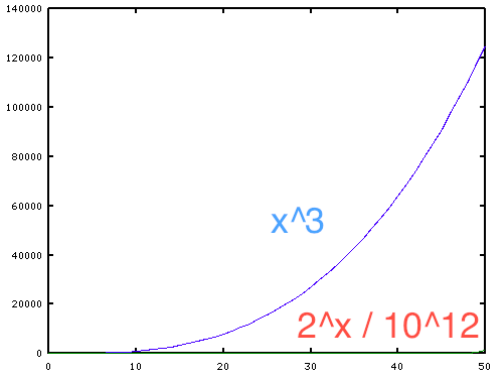
If $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$, then we write $f(n) \in \Theta(g(n))$.

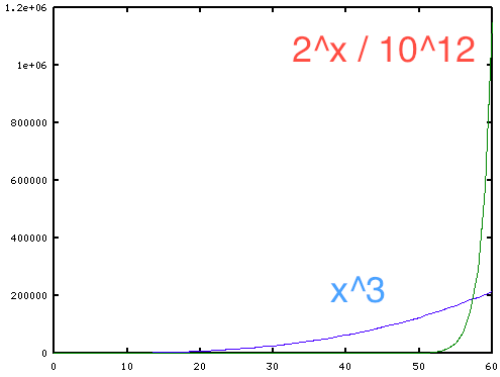




$x =\_\_\_ (x^2)$ $M =\_\_\_, x_0 =\_\_\_$ (1)

$x =\_\_\_ (\log x)$ $M =\_\_\_, x_0 =\_\_\_$ (2)





$x^3 =\_\_\_ (2^x)$ $M =\_\_\_, x_0 =\_\_\_$ (3)

Python

This class uses Python 2.6. Do not use Python 3. If you're not familiar with Python, there are numerous resources available on the Internet:

- Python tutorial: `http://docs.python.org/tutorial/`
- Python libraries: `http://docs.python.org/library/`
- 6.006 resources page: `http://courses.csail.mit.edu/6.006/spring11/resources.shtml`

Docdist code samples

Insertion sort

```
def insertion_sort(A):
    for j in range(len(A)):
        key = A[j]
        # insert A[j] into sorted sequence A[0..j-1]
        i = j-1
        while i>-1 and A[i]>key:
            A[i+1] = A[i]
            i = i-1
        A[i+1] = key
    return A
```

Count frequency

```
def count_frequency(word_list):
    """
    Return a list giving pairs of form: (word,frequency)
    """
    L = []
    for new_word in word_list:
        for entry in L:
            if new_word == entry[0]:
                entry[1] = entry[1] + 1
                break
        else:
            L.append([new_word,1])
    return L
```

Improved count frequency

```
def count_frequency(word_list):
    """
    Return a dictionary mapping words to frequency.
    """
    D = {}
    for new_word in word_list:
        if new_word in D:
            D[new_word] = D[new_word]+1
        else:
            D[new_word] = 1
    return D
```

Get words from line list

```
def get_words_from_line_list(L):
    """
    Parse the given list L of text lines into words.
    Return list of all words found.
    """
    word_list = []
    for line in L:
        words_in_line = get_words_from_string(line)
        word_list = word_list + words_in_line
    return word_list
```

Improved get words from line list

```
def get_words_from_line_list(L):
    """
    Parse the given list L of text lines into words.
    Return list of all words found.
    """
    word_list = []
    for line in L:
        words_in_line = get_words_from_string(line)
        word_list.extend(words_in_line)
    return word_list
```

Inner product

```
def inner_product(L1,L2):
    """
    Inner product between two vectors, where vectors
    are represented as alphabetically sorted (word,freq) pairs.
    Example: inner_product(
       [["and",3],["of",2],["the",5]],
       [["and",4],["in",1],["of",1],["this",2]]) = 14.0
    """
    sum = 0.0
    i = 0
    j = 0
    while i<len(L1) and j<len(L2):
        # L1[i:] and L2[j:] yet to be processed
        if L1[i][0] == L2[j][0]:
            # both vectors have this word
            sum += L1[i][1] * L2[j][1]
            i += 1
            j += 1
        elif L1[i][0] < L2[j][0]:
            # word L1[i][0] is in L1 but not L2
            i += 1
        else:
            # word L2[j][0] is in L2 but not L1
            j += 1
    return sum
```

Improved inner product

```
def inner_product(D1,D2):
    """
    Inner product between two vectors, where vectors
    are represented as dictionaries of (word,freq) pairs.
    Example: inner_product(
       {"and":3,"of":2,"the":5},
       {"and":4,"in":1,"of":1,"this":2}) = 14.0
    """
    sum = 0.0
    for key in D1:
        if key in D2:
            sum += D1[key] * D2[key]
    return sum
```