ECE244

Wael Aboulsaadat

Analysis of Algorithms

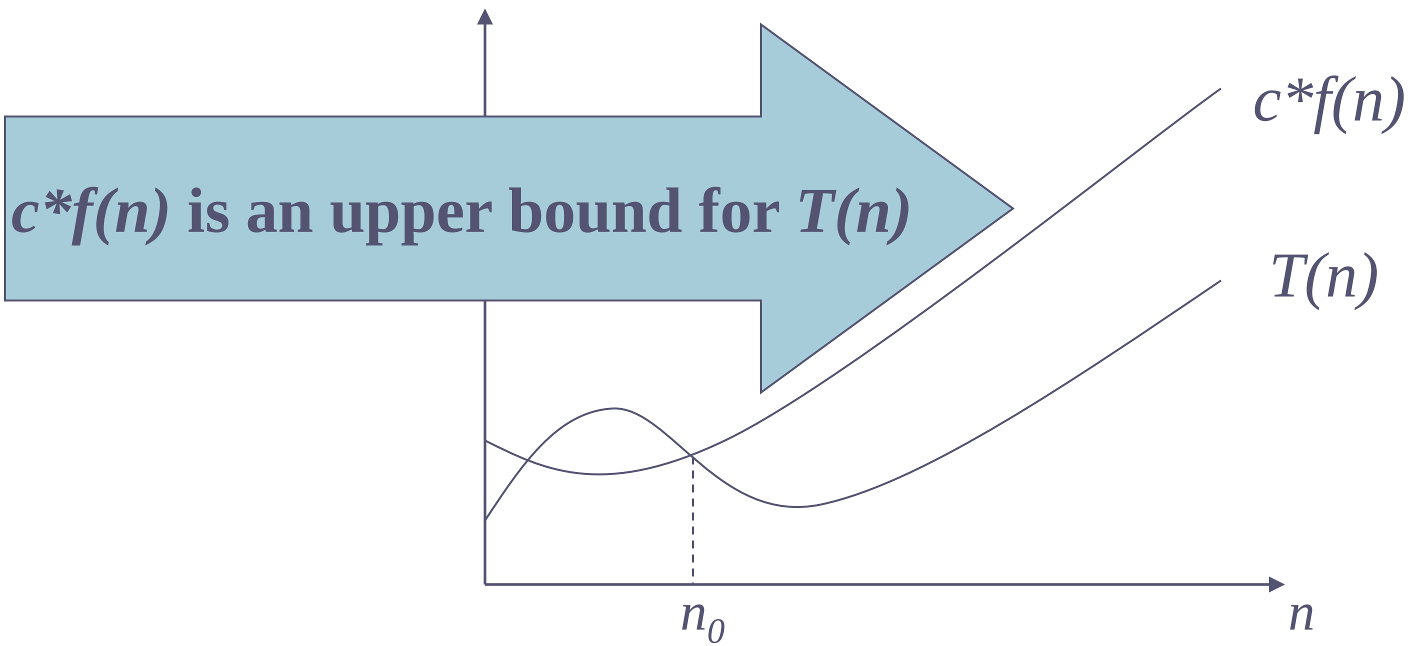
Big-Oh

Acknowledgment: these slides are partially based on slides by; Prof. Schmidt from Drexel U., Prof. Shewchuk from UC Berkely, Kruse & Ryba Data Structure and Program Design in C++, Prof. Savitch Problem Solving in C++ and others

Big-Oh Defined

- The O symbol was introduced in 1927 to indicate relative growth of two functions based on asymptotic behavior of the functions
- It is now used to classify functions and families of functions

T(n) = O(f(n)) if there are constants c and n0 such that T(n) < c*f(n) when n ≥ n0



Major Notations

- **O(g(n)), Big-Oh of g of n, the Asymptotic Upper Bound.**

- **Ω(g(n)), Big-Omega of g of n, the Asymptotic Lower Bound.**

Asymptotic Analysis

$$T(n) = 13n^3 + 42n^2 + 2n\log n + 4n$$

- **Ignoring constants in T(*n*)**
- **Analyzing T(*n*) as *n* "gets large"**

As n **grows larger,** n^3 **is MUCH larger than** n^2**,** $n\log n$**, and** n**, so it dominates** $T(n)$

The running time grows "roughly on the order of n^3**"**

Notationally, $T(n) = O(n^3)$

The big-oh (*O*) Notation

Can we justify Big O notation?

- Big O notation is a *huge* simplification; can we justify it?
  - It only makes sense for *large* problem sizes
  - For sufficiently large problem sizes, the highest-order term swamps all the rest!
- Consider $R = x^2 + 3x + 5$ as x varies:

x = 0	x2 = 0	3x = 10	5 = 5	R = 5
x = 10	x2 = 100	3x = 30	5 = 5	R = 135
x = 100	x2 = 10000	3x = 300	5 = 5	R = 10,305
x = 1000	x2 = 1000000	3x = 3000	5 = 5	R = 1,003,005
x = 10,000				R = 100,030,005
x = 100,000				R = 10,000,300,005

Table of growth rates

- The order of the algorithmic is more important than the speed of the processor

N	$\log_2 N$	$n*\log_2 N$	N^2	N^3	2^N	3^N
1	0	0	1	1	2	3
2	1	2	4	8	4	9
4	2	8	16	64	16	81
8	3	24	64	512	256	6561
16	4	64	256	4096	65,536	43,046,721
32	5	160	1024	322,768	4,294,967,296	…
64	6	384	4096	262,144	(Note 1)	…
128	7	896	16,384	2,097,152	(Note 2)	…
256	8	2048	65,536	1,677,216	????????	…

Note1: The value here is approximately the number of machine instructions executed by a 1 gigaflop computer in 5000 years.

Note 2: The value here is about 500 billion times the age of the universe in nanoseconds, assuming a universe age of 20 billion years.

O(1)

- **The no-growth curve**
- **Independent of the size of the data set on which it operates**
- **E.g.**
  - Sum first and last elements in an array
- **Constant time**

```
int sum_first_last(int arr[], int Size)
{
        int nSum;
        nSum = arr[0] + arr[Size-1];
        return nSum;
}
```

O(c)

O(N)

- Algorithm's performance is directly proportional to the size of the data set being processed

- E.g.
  - Scanning an array or linked list takes O(N) time.
  - Probing an array is still O(N)

```
for (i=0;  i< N; i++ )
{
        val = a[i];
        cout << val;
}
```

O(N)

- Linear Time

O(N+M)

- O(N+M) is just a way of saying that two data sets are involved, and that their combined size determines performance

$O(N^2)$

- algorithm's performance is proportional to the square of the data set size

- This happens when the algorithm processes each element of a set, and that processing requires another pass through the set.

- E.g.
  - Printout char one by one in a string of length N
  - Bubble Sort is $O(N^2)$.

- Quadratic Time

```
for (i=0;  i< strlen(str); i++ )
{
        c = str[i];
        cout << c;
}
```

$O(N^2)$

```
N = strlen(str);
for (i=0;  i<N; i++ )
{
        c = str[i];
        cout << c;
}
```

$O(N)$

$O(N^2)$

- algorithm's performance is proportional to the square of the data set size

- This happens when the algorithm processes each element of a set, and that processing requires another pass through the set.

- E.g.
  - Bubble Sort is $O(N^2)$.

$O(N \cdot M)$

- indicates that two data sets are involved, and the processing of each element of one involves processing the second set.

- If the two set sizes are roughly equivalent, some people just say $O(N^2)$ instead.

- E.g.
  - Text search/replace

$O(N^3)$

- ………..
- Lots of inner loops!
- Cubic Time

$O(2^N)$

- You have an algorithm with exponential growth behavior.
- In the 2 case, time or space double for each new element in data set.
- There's also $O(10^N)$ ☹ etc.
- Exponential time

O(log N) and O(N log N)

- **log N implies $\log_2 N$, which means, roughly, the number of times you can partition a set in half, then partition the halves, and so on, while still having non-empty sets.**

- **Think backward!**

  $2^{10} = 1024$

  $\log_2 1024 = 10$

- **E.g.**
  - **It takes O(log N) time to search a balanced binary tree**

  $$\underbrace{1024 \rightarrow 512 \rightarrow 256 \rightarrow 128 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2}_{10}$$

- **Logarithmic time**

Comparison of Different Orders

Size of Input Data (N) vs. Time