cmpt-225

Algorithm Efficiency

Timing Algorithms

- It can be very useful to time how long an algorithm takes to run
  - In some cases it may be essential to know how long a particular algorithm takes on a particular system
- However, it is not a good general method for comparing algorithms
  - Running time is affected by numerous factors
    - How are the algorithms coded?
    - What computer should we use?
      - CPU speed, memory, specialized hardware (e.g. graphics card)
      - Operating system, system configuration (e.g. virtual memory), programming language, algorithm implementation
      - Other tasks (i.e. what other programs are running), timing of system tasks (e.g. memory management)
    - What data should we use?

Cost Functions

- Because of the sorts of reasons just discussed for general comparative purposes we will count, rather than time, the **number of operations** that an algorithm performs
  - Note that this does not mean that actual running time should be ignored!

Cost Functions

- For simplicity we assume that each operation take one unit of time.
- If algorithm (on some particular input) performs ***t*** operations, we will say that it runs in **time *t***.
- Usually running time ***t*** depends on the data size (the input length).
- We express the time ***t*** as a **cost function** of the data size ***n***
  - We denote the cost function of an algorithm A as $t_A()$, where $t_A(n)$ is the time required to process the data with algorithm A on input of size ***n***
- Typical example of the input size: number of nodes in a linked list, number of disks in a Hanoi Tower problem, the size of an array, the number of items in a stack, the length of a string, …

Nested Loop

```
for (i=1 through n){
   for (j=1 through i){
        for (k=1 through 5){
               Perform task T;
        }
   }
}
```

- If task T requires t units of time, the inner most loop requires 5*t time units and the loop on j requires 5*t*i time units.
- Therefore, the outermost loop requires

$$\sum_{i=1}^{n}(5.t.i) = 5.t.(1+2+...+n) = 5.t.n.(n+1)/2$$

Algorithm Growth Rates.

- We often want to compare the performance of algorithms
- When doing so we generally want to know how they perform when the problem size (*n*) is large
- So it's simpler if we just find out how the algorithms perform as the input size **grows- *the growth rate*.**

- E.g.
  - Algorithm A requires $n^2/5$ time units to solve a problem of size n
  - Algorithm B requires 5*n time units to solve a problem of size n
- It may be difficult to come up with the above conclusions and besides they do not tell us the exact performance of the algorithms A and B.
- It will be easier to come up with the following conclusion for algorithms A and B
  - Algorithm A requires time proportional to n^2
  - Algorithm B requires time proportional to n
- From the above you can determine that for large problems B requires significantly less time than A.

Algorithm Growth Rates

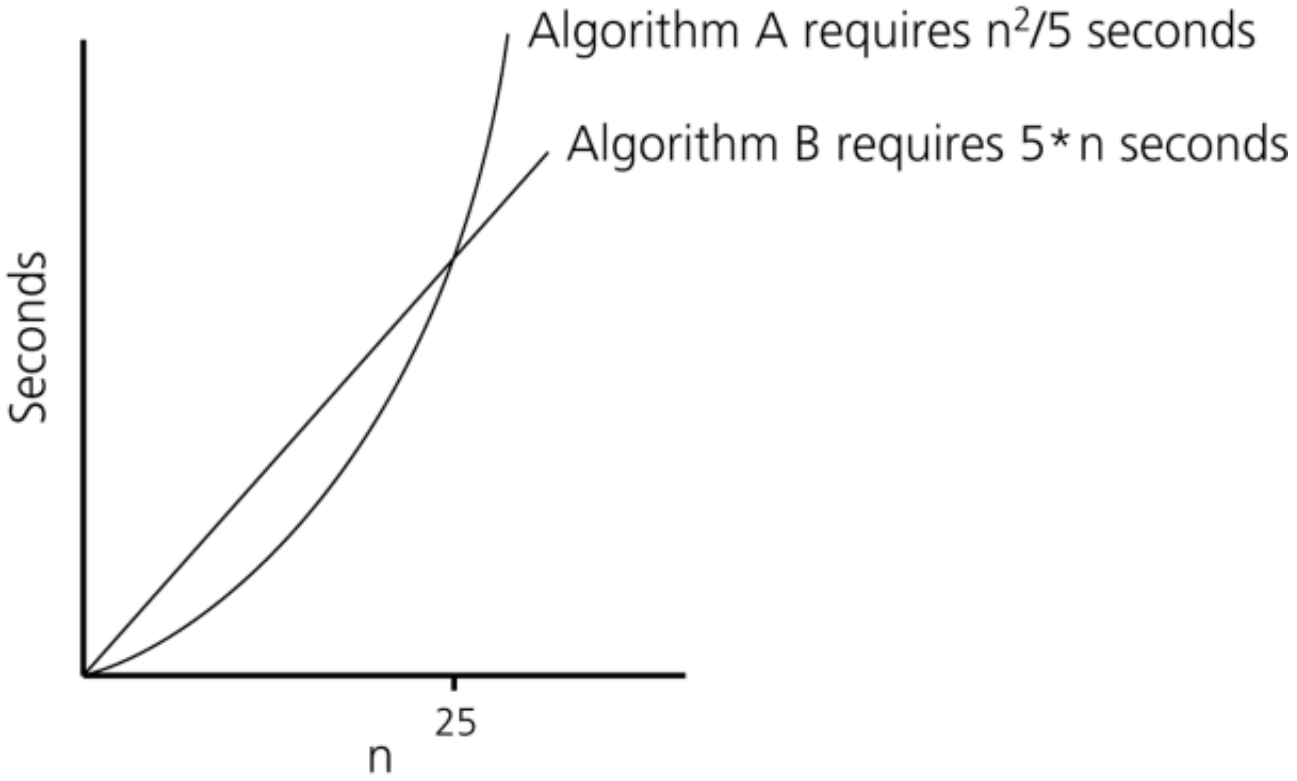


Figure 10-1

Time requirements as a function of the problem size *n*

- Since cost functions are complex, and may be difficult to compute, we approximate them using O notation – O notation determines the growth rate of an algorithm time.

Example of a Cost Function

- Cost Function: $t_A(n) = n^2 + 20n + 100$
  - Which term dominates?
- It depends on the size of n
  - $n = 2$, $t_A(n) = 4 + 40 + \underline{100}$
    - The constant, 100, is the dominating term
  - $n = 10$, $t_A(n) = 100 + \underline{200} + 100$
    - $20n$ is the dominating term
  - $n = 100$, $t_A(n) = \underline{10{,}000} + 2{,}000 + 100$
    - n^2 is the dominating term
  - $n = 1000$, $t_A(n) = \underline{1{,}000{,}000} + 20{,}000 + 100$
    - n^2 is the dominating term

Big O Notation

- O notation approximates the cost function of an algorithm
  - The approximation is usually good enough, especially when considering the efficiency of algorithm as *n* gets very large
  - Allows us to estimate rate of function growth
- Instead of computing the entire cost function we only need to count the number of times that an algorithm executes its *barometer instruction(s)*
  - The instruction that is executed the most number of times in an algorithm (the highest order term)

Big O Notation

- Given functions $t_A(n)$ and $g(n)$, we can say that the efficiency of an algorithm is *of order g(n)* if there are positive constants c and m such that
  - $t_A(n) < c.g(n)$ for all $n > n0$
- we write
  - $t_A(n)$ is $O(g(n))$ and we say that
  - $t_A(n)$ is of order $g(n)$
- e.g. if an algorithm's running time is $3n + 12$ then the algorithm is $O(n)$. If $c=3$ and $n0=12$ then $g(n) = n$:
  - $4 * n \geq 3n + 12$ for all $n \geq 12$

In English…

- The cost function of an algorithm A, $t_A(n)$, can be approximated by another, simpler, function *g(n)* which is also a function with only 1 variable, the data size *n.*
- The function *g(n)* is selected such that it represents an *upper bound* on the efficiency of the algorithm A (i.e. an upper bound on the value of $t_A(n)$).
- This is expressed using the big-O notation: O*(g(n))*.
- For example, if we consider the time efficiency of algorithm A then "t_A(n) is O*(g(n))*" would mean that
  - A cannot take more "time" than O*(g(n))* to execute or that (more than *c.g(n)* for some constant *c*)
  - the cost function $t_A(n)$ grows *at most as* fast as *g(n)*

The general idea is …

- when using Big-O notation, rather than giving a precise figure of the cost function using a specific data size *n*
- express the behaviour of the algorithm as its data size *n* grows very large
- so ignore
  - lower order terms and
  - constants

O Notation Examples

- All these expressions are O(n):
  - n, $3n$, $61n + 5$, $22n – 5$, …
- All these expressions are O(n^2):
  - n^2, $9\ n^2$, $18\ n^2 + 4n – 53$, …
- In general for polynomial functions
  - $t(n)=a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$ is of $O(n^k)$
  - If $c=(a_k + a_{k-1} + \ldots + a + a_0)$ and $n_0=1$ then $t(n) < cn^k$ for $n>1$
- All these expressions are O(n log n):
  - $n(\log n)$, $5n(\log 99n)$, $18 + (4n – 2)(\log (5n + 3))$, …

Growth-rate Functions

- O(1) – **constant** time, the time is independent of *n*, e.g. array look-up
- O(log *n*) – **logarithmic** time, usually the log is base 2, e.g. binary search
- O(*n*) – **linear** time, e.g. linear search
- O(*n**log *n*) – e.g. efficient sorting algorithms
- O(n^2) – **quadratic** time, e.g. selection sort
- O(n^k) – **polynomial** (where *k* is some constant)
- O(2^n) – **exponential** time, very slow!
- Order of growth of some common functions

  $O(1) < O(\log n) < O(n) < O(n * \log n) < O(n^2) < O(n^3) < O(2^n)$

Order-of-Magnitude Analysis and Big O Notation

(a)

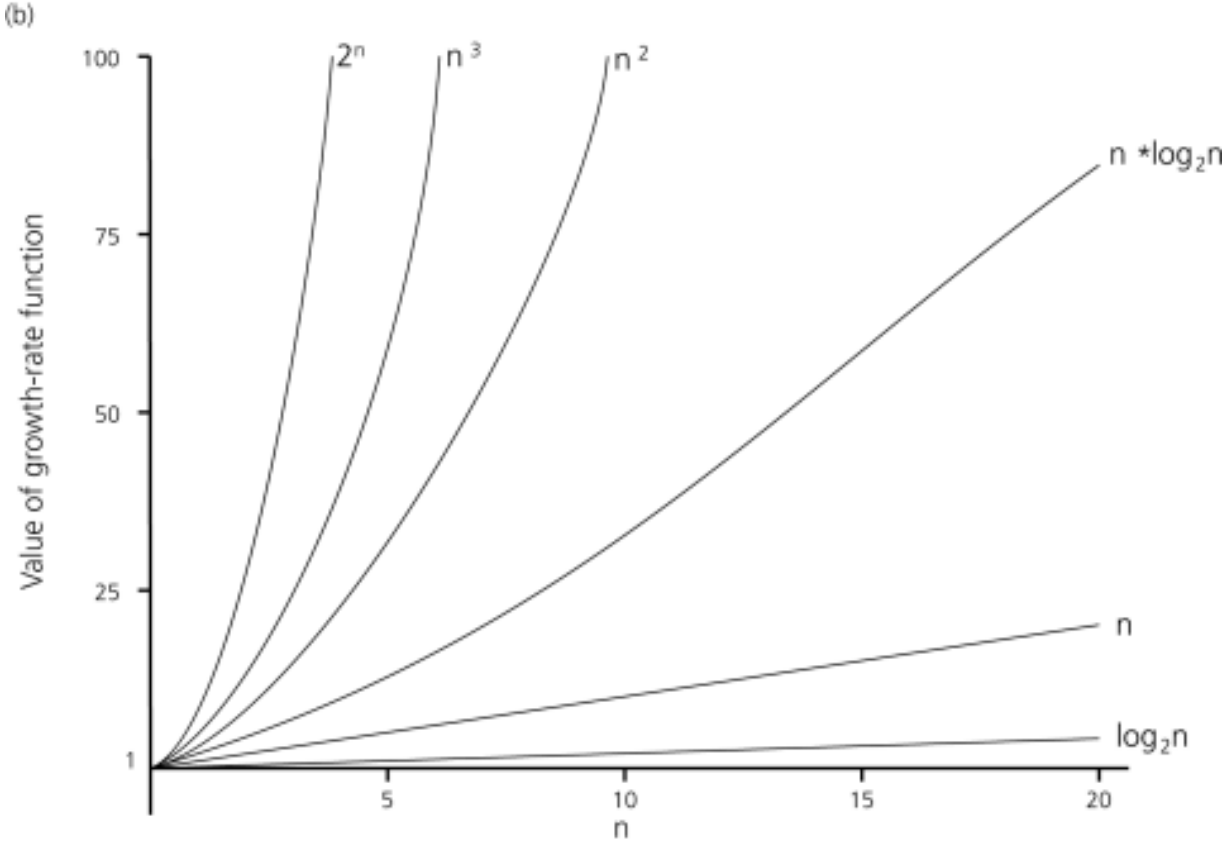
Function	n = 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

A comparison of growth-rate functions: a) in tabular form

An intuitive example

- Suppose the running time of algorithm A and B are 2^n and n^{12} respectively
- For small input size A performs better (n=10)
- For n=100
  - 100^{12} is comparable to the number of molecules in a teaspoon of water
  - 2^{100} is comparable to the number of molecules in a backyard swimming pool.
- For n=1000
  - 1000^{12} is comparable is comparable to the number of molecules in a lake
  - 2^{1000} is already many orders of magnitudes greater than the number of particles in the whole universe!!

Order-of-Magnitude Analysis and Big O Notation



A comparison of growth-rate functions: b) in graphical form

Big O notation

- Note that Big O notation represents an upper bound of a cost function
- E.g
  - $T(n)=3n+5$ is of $O(n)$, $O(n^2)$, $O(n.\log(n))$, $O(2^n)$
- However we usually use the tightest one.

Note on Constant Time

- We write O(1) to indicate something that takes a constant amount of time
  - E.g. finding the minimum element of an ordered array takes O(1) time, because the min is either at the beginning or the end of the array
  - **Important**: constants can be huge, and so in practice O(1) is not necessarily efficient --- all it tells us is that the algorithm will run at the same speed no matter the size of the input we give it
  - 2^{1000} is of O(1)

Arithmetic of Big-O Notation

1) If f(n) is O(g(n)) then c.f(n) is O(g(n)), where c is a constant.
   - Example: 23*log *n is* O(log *n*)

2) If $f_1(n)$ is O(g(n)) and $f_2(n)$ is O(g(n)) then also $f_1(n)+f_2(n)$ is O(g(n))
   - Example: what is order of n^2+n?
     n^2 is $O(n^2)$
     n is O(n) but also $O(n^2)$
     therefore n^2+n is $O(n^2)$

Arithmetic of Big-O Notation

3) If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then $f_1(n)*f_2(n)$ is $O(g_1(n)*g_2(n))$.
   - Example: what is order of $(3n+1)*(2n+\log n)$?
     $3n+1$ is $O(n)$
     $2n+\log n$ is $O(n)$
     $(3n+1)*(2n+\log n)$ is $O(n*n)=O(n^2)$

Using Big O Notation

- Sometimes we need to be more specific when comparing the algorithms.
- For instance, there might be several sorting algorithms with time of order O(n.log n). However, an algorithm with cost function
  2n.log n + 10n + 7log n + 40
  is better than one with cost function
  5n.log n + 2n +10log n +1
- That means:
  - We care about the constant of the main term.
  - But we still don't care about other terms.
- In such situations, the following notation is often used:
  2n.log n + O(n) for the first algorithm
  5n.log n + O(n) for the second one