Module 1: Analysis of Algorithms

Reading Assignment: Chapter 3 of textbook.

Contents

1 Introduction **2**

2 Average Case, Best Case, and Worst Case Analysis of Algorithms **3**

3 Some Mathematical Identities **5**

4 Analysis of Algorithms **7**

4.1 Asymptotic Notation 9

4.1.1 The Big-O Notation 9

4.1.2 Some Identities for Big-O Notation 10

4.1.3 Other Asymptotic Notation 12

5 Computing Recurrence Relations **12**

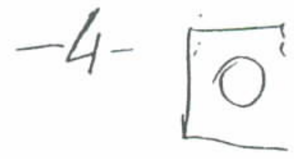
5.1 Expanding Sequences 13

5.2 Recurrence Relations with Full History 14

5.3 Informed Guesses 14

Lecture 1

Example:
F(I) 1. List to find x
F(I)=#n 2. Matrix add
F(I) 3. Sorting



What is an **Algorithm**?

An Algorithm is a finite sequence of instructions which, ~~if followed~~, accomplish a particular task, and which satisfies the following properties:

1) *Input:* has an non-empty input
2) *Output:* produces a non-empty output
3) *Definiteness:* each instruction must be clear and unambiguous.
4) *Effectiveness:* each instruction must be feasable in that it is basic enough to allow execution by a person using only pencil and paper.
5) *Finiteness:* the algorithm will always terminate ~~terminate~~ after a finite number of steps.
6 Optimality

Algorithm Analysis: Given an algorithms — What kind of input we may have

1) Select a *measure of size of input*
   - -Find X in a list of name (# of names in the list)
   - -Multiply 2 matrices (dimension of matrices)
   - -Traverse a binary tree (#of nodes in a tree

2) Select a basic operation.
   - -Find X in a list (#of comparisons)
   - -Multiply 2 matrices (# of multiplications)
   - -Traverse a binary tree (# of nodes visited)

3) Count the number of basic operations executed for a given size of input.

Sorting
Compute
$s = \sum a_i$

Example 1
(1) List
(2) Matrix
(3) Fact

$S = S + a(i,j)$

$S = \sum_{i=1}^{n} \sum_{j=1}^{i} a(i,j)$

Frequencey Count:

Define F(n) as "the number of basic operations executed on input of size n by the algorithm under investigation", to be the *frequency count function*.

example: The frequency count F(n) of how many times the basic operation is executed. Let h(x) be the basic operation, then

Ex. 1a

```
for i := 1 to n do begin
  for j := 1 to i do
     x := X + h(j)
  for k := i + 1 to n do
     x := x * h(k)
end
```

Example 1b

```
for i = 1 to n
  for j = 1 to i
    a(i,j) = a(i,j) + 1
end
```

$\sum_{i=1}^{n} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

$$F(n) = \sum_{i=1}^{n} \left[\sum_{j=1}^{i} 1 + \sum_{k=i+1}^{n} 1 \right] = \sum_{i=1}^{n} \left[i + (n-i) \right] = \sum_{i=1}^{n} n = n^2$$

Frequency Count of Recursive Algorithms:

Consider the following recursive function

Example 2: **Ex. 2.**

```
function fact(n : Natural) : Natural;
begin
  if n = 0 then
     fact := 1
  else
     fact := fact(n-1)*n
end;
```

```
int fact (int n)
{
  if (n=0)
     fact = 0;
  else fact = fact(n-1)*n;
}
```

Input: n

The frequency function, F(n), is called a recursive (i.e. a *recurrence*) relation and is not in a *closed form*. Thus we must show F(n) in a closed form and verify its correctness by an *inductive proof.*

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 + F(n-1) & \text{o/w} \end{cases}$$

$$\begin{aligned} F(n) &= 1 + F(n-1) \\ &= 1 + 1 + F(n-2) = n + F(0) = n \end{aligned}$$

This may be ambiguous! Sorting

Exercise 1: Draw two graphs, one for machine M1 and one for machine M2. On the x axis, plot the size of the list to be sorted and on the y axis, plot the time taken by algorithms A1 and A2. Examine the two graphs and comment on the performance of algorithms A1 and A2 on machines M1 and M2.

This module deals with quantitative performance measures for algorithms. Specifically, it deals with asymptotic analysis of worst case and average case behavior of memory and time requirements. It introduces the big 'O', omega, and theta notations and uses them to quantify time and space complexity of algorithms. As we go through our discussion of asymptotic analysis, we shall note that an important component of this task is the evaluation of summations and recurrence relations, while counting number of operations. This module illustrates how frequently encountered summations and recurrence relations can be performed. It also provided examples of asymptotic analyses of algorithms.

The objective of this module is to equip the students with a set of tools that will enable them to associate a figure of merit with algorithms before actually implementing them (and then perhaps realizing that they do not perform as well as initially thought!).

2 Average Case, Best Case, and Worst Case Analysis of Algorithms

Let us start with a simple example of an algorithm for sorting a list of numbers:

Example 1: A simple way of sorting a list of numbers is using an algorithm called Bubble sort. In this algorithm, we make repeated passes over the list. In each pass, every number is compared to the next number. If this pair is out of order (i.e., the smaller number follows a larger number), the two are interchanged. If during a complete pass over the list no pair of numbers is interchanged, the list is sorted and the process can be stopped. The program segment for doing this is as follows:

```
// the input list of n integers is stored in list[0] .. list[n-1]

for (i = 0; (i < n) && (exchange == 1); i++) {
    exchange = 0;
    for (j = 0; i < n; j++)
        // if two consecutive numbers are out of order,
        // exchange them
        if (list[i] > list [i + 1]) {
            temp = list[i + 1];
            list[i + 1] = list[i];
            list[i] = temp;
            exchange = 1;
        }
}
```

We show the execution of this algorithm on two lists:

```
Initial list:     5  6  3  0  1  9  8  7
                  5  3  6
                  5  3  0  6
                  5  3  0  1  6
                  5  3  0  1  6  9
                  5  3  0  1  6  8  9
End of pass 1:    5  3  0  1  6  8  7  9

                  3  5
                  3  0  5
                  3  0  1  5
                  3  0  1  5  6
                  3  0  1  5  6  8
                  3  0  1  5  6  7  8
End of pass 2:    3  0  1  5  6  7  8  9

                  0  3
                  0  1  3
                  0  1  3  5
                  0  1  3  5  6
                  0  1  3  5  6  7
                  0  1  3  5  6  7  8
End of pass 3:    0  1  3  5  6  7  8  9

End of pass 4:    0  1  3  5  6  7  8  9
```

Since there are no exchanges in pass 4, the list is sorted and the algorithm terminates.
Consider a second example input list:

```
Initial list:     0  1  3  5  6  7  8  9

End of pass 1:    0  1  3  5  6  7  8  9
```

Since there are no exchanges in pass 1, the list is sorted and the algorithm terminates.
Finally, consider the following input list:

```
Initial list:     9  8  7  6  5  3  1  0
```

running times	$time_1$ 1000 steps/sec	$time_2$ 2000 steps/sec	$time_3$ 4000 steps/sec	$time_4$ 8000 steps/sec
$\log_2 n$	0.010	0.005	0.003	0.001
n	1	0.5	0.25	0.125
$n \log_2 n$	10	5	2.5	1.25
$n^{1.5}$	32	16	8	4
n^2	1,000	500	250	125
n^3	1,000,000	500,000	250,000	125,000
1.1^n	10^{39}	10^{39}	10^{38}	10^{38}

Table 3.1 Running times (in seconds) under different assumptions (n=1000).

while ignoring constants. If there exist constants c and N, such that for all $n \geq N$ the number of steps $T(n)$ required to solve the problem for input size n is at least $cg(n)$, then we say that $T(n)=\Omega(g(n))$. So, for example, $n^2=\Omega(n^2-100)$, and also $n=\Omega(n^{0.9})$. The Ω notation thus correspond to the "$\geq$" relation.

If a certain function $f(n)$ satisfies both $f(n)=O(g(n))$ and $f(n)=\Omega(g(n))$, then we say that $f(n)=\Theta(g(n))$. For example, $5n\log_2 n-10=\Theta(n\log n)$. (The base of the logarithm can be omitted in the expression $\Theta(n \log n)$, since different bases change the logarithm only by a constant factor.) The constants used to prove the O part and the Ω part need not be the same.

The O, Ω, and Θ correspond (loosely) to "$\leq$", "$\geq$", and "$=$". Sometimes we need notation corresponding to "$<$" and "$>$". We say that $f(n)=o(g(n))$ (pronounced "$f(n)$ is little oh of $g(n)$") if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

For example, $n/\log_2 n = o(n)$, but $n/10 \neq o(n)$. Similarly, we say that $f(n)=\omega(g(n))$ if $g(n)=o(f(n))$.

We can strengthen Theorem 3.1 by replacing big O with little o:

□ Theorem 3.3

For all constants $c>0$ and $a>1$, and for all monotonically growing functions f(n), we have $(f(n))^c=o(a^{f(n)})$. In other words, an exponential function grows faster than does a polynomial function. □

The $\mathcal{O}$ Symbol

The O notation has received a lot of criticism over the years. The main objection to it is, of course, that in reality constants do matter. The wide use of the O notation makes it convenient to forget about constants altogether. It is essential to remember that the O notation gives only a first approximation. As such, it serves a useful purpose, and its use

Classification of Functions. Functions can be classified into the following categories: Logarithmic ($O(\log n)$), Linear ($O(n)$), Quadratic ($O(n^2)$), Polynomial ($O(n^k)$, $k \geq 1$), and Exponential ($O(c^n), c > 1$). It is also useful to gather a sense of the growth rates of some commonly encountered functions:

n	log n	sqrt(n)	n log n	n^2	2^n
2	1	1.4	2	4	4
4	2	2	8	16	16
16	4	4	64	256	65,536
256	8	16	2048	65,536	1.15 x 10^77
1024	10	32	10240	1048576	1.79 x 10^308

Exercise 8: For each of the following code samples, determine the runtime in terms of the big-O notation:

(a)

```
// computing the sum of first n integers
sum = 0;
for (i = 0; i < n; i++)
    sum += i;
```

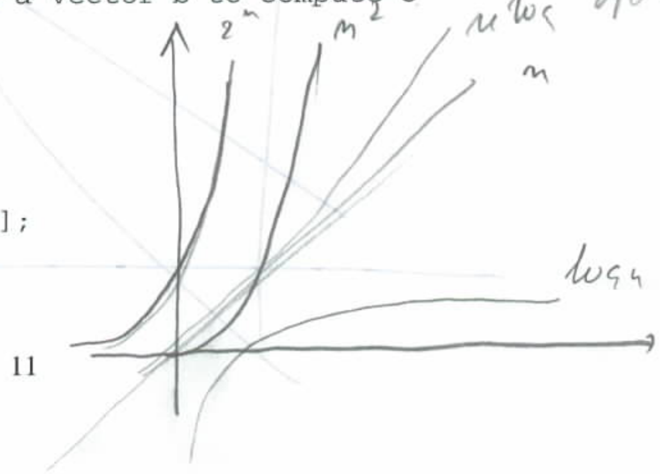
(b)

```
// dot_product of two vectors a and b
dot_product = 0;
for (i = 0; i < n; i++)
    dot_product += a[i] * b[i];
```

(c)

```
// product of a matrix b with a vector b to compute c
for (i = 0; i < n; i++)
    c[i] = 0;
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        c[i] += a[i][j] * b[j];
```

(d)



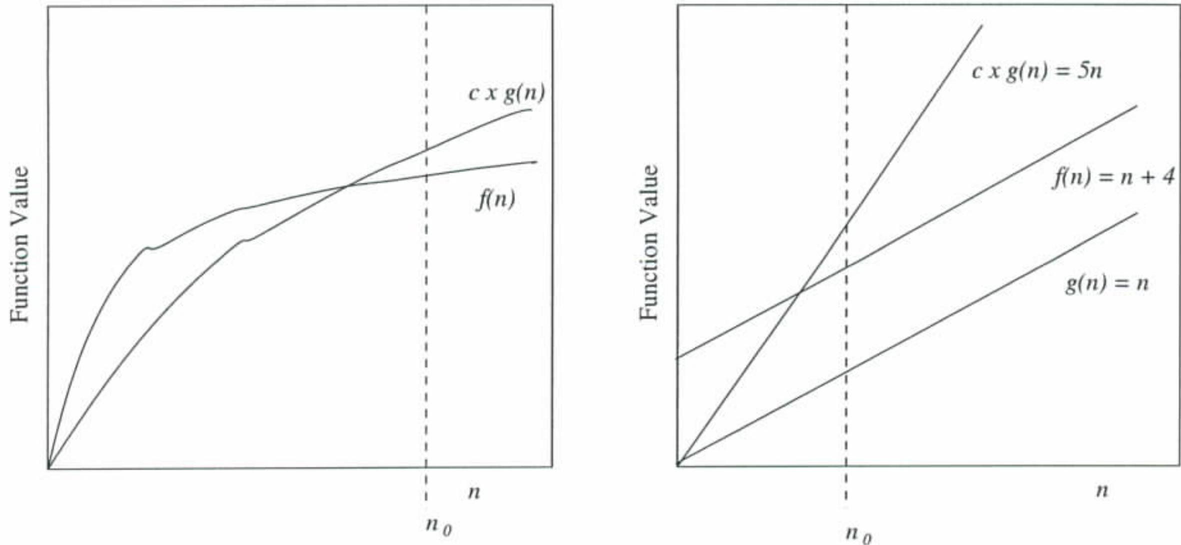


Figure 1: Illustration of the Big-O notation. In each case $f(n)$ is $O(g(n))$.

4.1 Asymptotic Notation

The goal of adopting asymptotic notion is to eliminate lower-level details and focusing on the dominant characteristics of functions. For example, $(n-1) \times 100$ is very close to $n \times 100$ as $n >> 1$. Dealing with $n \times 100$ is, in general, much simpler than $(n-1) \times 100$. Similarly, the value $2 \times c_1 \times n^2$, where c_1 is a constant can be written as $c_2 n^2$, where $c_2 = 2 \times c_1$. This value, $c_2 n^2$ can be further simplified to n^2 with the implicit understanding that there are constants that have been dropped.

4.1.1 The Big-O Notation

Consider two functions $f(n)$ and $g(n)$ that map integers (n) to real numbers. We say that $f(n)$ is $O(g(n))$ if there exist constants c and n_0 such that:

$$f(n) \leq c \times g(n) \text{ for } n \geq n_0$$

This is often also referred to as "$f(n)$ is *order* of $g(n)$".

Let us examine what big-O notation implies in greater detail. Consider the two plots in Figure 1. In the first case, it is easy to see that $f(n)$ is less than $c \times g(n)$ when n exceeds n_0. As an example of this consider $f(n) = 5n$ and $g(n) = n^2$. For $n = 1, 2, 3, 4$, $f(n)$ exceeds $g(n)$. However, for $n \geq 5$, it is easy to see that $f(n) \leq g(n)$. Therefore, in this case, we can say that with $c = 1$ and $n_0 = 5$, the required inequality holds, and therefore $5n$ is in $O(n^2)$.

The second plot in Figure 1 is more interesting. In this case, $f(n) = n + 4$ and $g(n) = n$. It is easy to see that with $c = 1$, there does not exist any n_0 such that $n + 4 \leq n$ for $n \geq n_0$. However, if we select $c = 5$, we can see that $n + 4 \leq 5n$ for $n \geq 1$. While it does seem slightly strange to begin with, we see that $n + 4$ is also $O(n)$.

Asymptotic notation

Definition [Big "oh"] $f(n) = O(g(n))$ if and only if there exist positive constans c and n_0 such that $f(n) \le c\,g(n)$ for all $n \ge n_0$.

Example

$3n+2 = O(n)$ since $3n+2 \le 4n$ for all $n \ge n_0 = 2$

$O(n^2$

$3n+1000 = O(n)$ since $3n+1000 \le 4n$ for $n \ge n_0 = 1000$

$n^2 - 5n$

or $10n^2 + 20n - 100$ $\le 5n^2$

$0 \le n^2 - 2\cdot 10 \cdot n + 100 = (n-10)^2$, $n \ge 10$

→ $10n^2 + 100n - 6 = O(n^2)$ $\quad \square \le 11n^2$ for $n \ge 10$

$98n + 44$ $O(n^3)$

$0 \le n^2 - 100n + 6$

→ $6\cdot 2^n + n^2 = O(2^n)$ since $6\cdot 2^n + n^2 \le 7\cdot 2^n$ for $n \ge 4$ ($n^2 \le 2^n$)

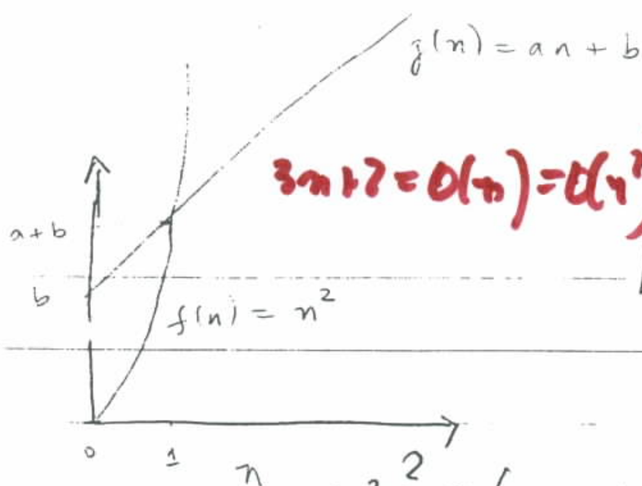
Note that:

$10n^2 + 100n - 6 = O(n^4)$ as $\square \le 10n^4$, $n \ge 2$

$6\cdot 2^n + n^2 = O(n!)$ as $\square \le 1 n!$ for $n \ge 6$

But

$3n+2 \ne O(1)$ since $3n+2 \not\le c\cdot 1$ for every const c and all $n \ge n_0$



$3n+7 = O(n) = O(n^2)$

Example

Let $f(n) = n^2$

$g(n) = an + b$

Is ~~$f = O(g)$~~

~~$g(n) \le f(n)$~~ for $n \ge 1$

true?

so here ($n_0 = 1$, $c = 1$)

there may be many n_0's & c's ~~we only need to~~ find one pair

$$n^2 \overset{?}{=} O(an+b)$$

$\Rightarrow$ g is $O(f)$

~~PROBLEM~~ Solution.

Q: Does $g \in O(f)$ automatically imply that $f \in O(g)$? for this particular example.

Check: is $f(n) \le c' g(n)$ for all $n \ge n_0'$

(If we suspect the answer is no, then we can try a "proof using contradiction")

Assume $f \in O(g)$. Assume it is true?

i.e. $f(n) \le c' g(n)$ for all $n \ge n_0'$.

$n^2 \le c' [an + b]$ for all $n \ge n_0$

$\Rightarrow$ $n^2 \le ac'n + bc'$ for all $n \ge n$

$\Rightarrow$ $n \le ac' + \frac{bc'}{n}$ ~~$\le (a+b)c'$~~ "

$\Rightarrow$ $n \le ac' + bc'$ "

$\Rightarrow$ $n \le$ some const This is impossible