DATA STRUCTURES AND ALGORITHMS

Asymptotic analysis of the algorithms

Summary of the previous lecture

- Algorithm definition
- Representation of the algorithms:
  - Flowchart,
  - Pseudocode
  - Description
- Types of the algorithms
  - Linear
  - Branching
  - Complex

Algorithm analysis

How do you compare two algorithms for solving some problem in terms of efficiency?

The first idea is to implement both algorithms as computer programs and then run them on a suitable range of inputs, measuring how much of the resources in question each program uses.

This approach is often unsatisfactory for the following reasons:

- Empirically comparing two algorithms there is always the chance that one of the programs was “better written” than the other.
- Empirical test cases might unfairly favor one algorithm.
- You could find that even the better of the two algorithms does not fall within your resource budget.

Algorithm analysis

- The critical resource for a program is most often its **running time**.
- However, you cannot pay attention to running time alone. You must also be concerned with other factors such as the **space required** to run the program (both main memory and disk space).
- Typically the **time** required for an algorithm and the **space** required for a data structure are analyzed.

Basic operations

- Primary consideration when estimating an algorithm's performance is **the number of basic operations** required by the algorithm to process an input of a certain size.
- The terms "**basic operations**" and "**size**" are used for algorithms analysis.
- **Size is the number of inputs processed**.

- For example, when comparing sorting algorithms, the size of the problem is typically measured by the **number of records to be sorted**. Size of the problem for adding two matrices is **n x n**.
- A basic operation must have the property that its time to complete does not depend on the particular values of its operands. **Adding or comparing two integer variables are examples of basic operations in most programming languages.**

Example

A simple algorithm to solve the problem of finding the largest value in an array of **n** integers.

```
// Return position of largest value in array A of size n

int largest (int A[ ], int n) {
    int currlarge = 0;
    for ( int i = 1; i < n; i++ )
        if ( A[currlarge] < A[i] )
            currlarge = i;
return currlarge;
}
```

Running time

- Because the most important factor affecting running time is normally size of the input, for a given **input size n** we often express the **time T** to run the algorithm as a function of **n**, written as **T(n).**
- It is always assumed **T(n)** is a non-negative value.
- Let us call **c** the amount of time required to compare two integers in function **largest.** Then function **largest** has a running time expressed by the equation:

$$T(n) = c*n$$

Running time

- The running time of a statement that assigns the first value of an integer array to a variable is simply the time required to copy the value of the first array value.
- We can assume this assignment takes a constant amount of time regardless of the value.
- Let us call $\mathbf{c_1}$ the amount of time necessary to copy an integer. Thus, the equation for this algorithm is simply:

$$\mathbf{T(n) = c_1}$$

indicating that the size of the input **n** has no effect on the running time. This is called a **constant running time**.

Example

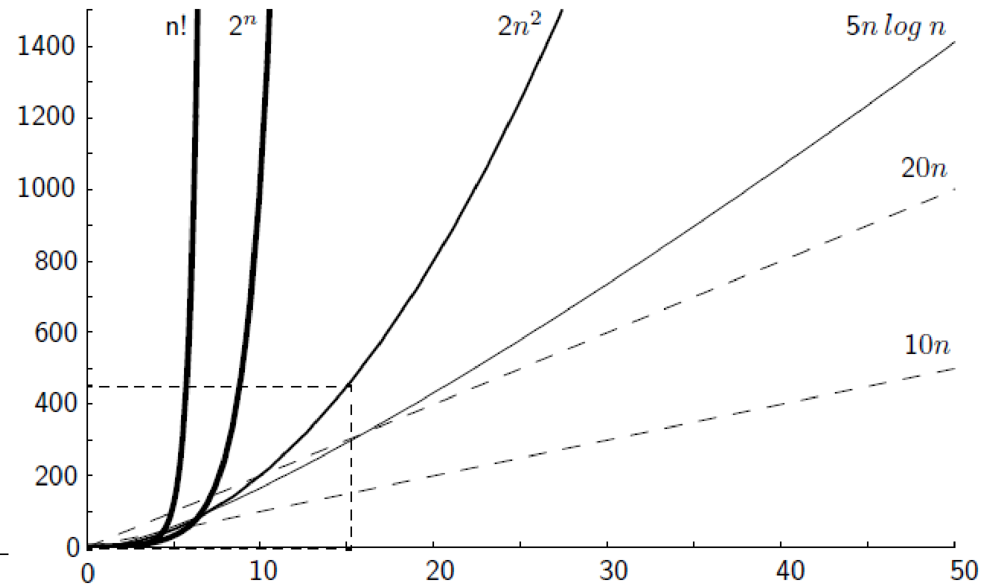
A simple algorithm to count two dimensional matrix elements. Size of matrix n x n.

```
sum = 0;
  for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)
           sum++;
```

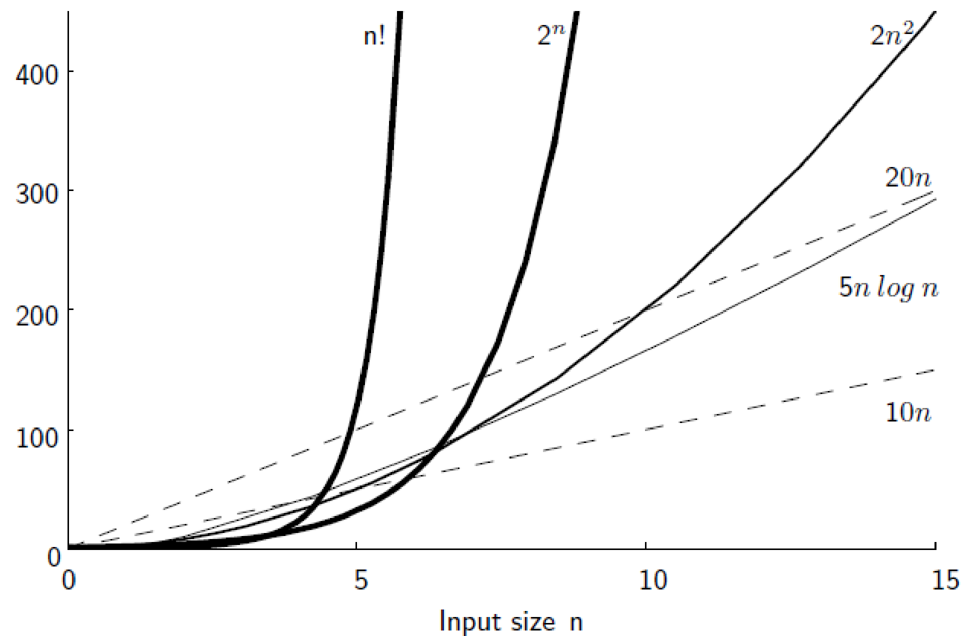
We can assume that incrementing takes constant time. Lets call this time c_2.
The total number of increment operations is n^2. Thus, we say that the running time is:

$$T(n) = c_2 * n^2$$

Growth rate



n	log log n	log n	n	n log n	n^2	n^3	2^n
16	2	4	2^4	$2 \cdot 2^4 = 2^5$	2^8	2^{12}	2^{16}
256	3	8	2^8	$8 \cdot 2^8 = 2^{11}$	2^{16}	2^{24}	2^{256}
1024	≈ 3.3	10	2^{10}	$10 \cdot 2^{10} \approx 2^{13}$	2^{20}	2^{30}	2^{1024}
64K	4	16	2^{16}	$16 \cdot 2^{16} = 2^{20}$	2^{32}	2^{48}	2^{64K}
1M	≈ 4.3	20	2^{20}	$20 \cdot 2^{20} \approx 2^{24}$	2^{40}	2^{60}	2^{1M}
1G	≈ 4.9	30	2^{30}	$30 \cdot 2^{30} \approx 2^{35}$	2^{60}	2^{90}	2^{1G}



Faster PC or algorithms?

$f(n)$	n	n'	Change	n'/n
$10n$	1000	$10,000$	$n' = 10n$	10
$20n$	500	5000	$n' = 10n$	10
$5n \log n$	250	1842	$\sqrt{10}n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	$--$

The increase in problem size that can be run in a fixed period of time on a computer that is ten times faster.

- n is normal computer
- n' is computer that run 10 times faster

Asymptotic analysis

Asymptotic analysis refers to the study of an algorithm as the input size “gets big” or reaches a limit (in the calculus sense).

When comparing algorithms meant to run on small values of **n**, results of analysis can be different.

For example, if the problem is to sort a collection of exactly five records, then an algorithm designed for sorting thousands of records is probably not appropriate, even if its asymptotic analysis indicates good performance.

Asymptotic analysis

- **Asymptotic analysis is a form of estimation for algorithm resource consumption.**
- It provides a simplified model of the running time or other resource needs of an algorithm.
- This simplification usually helps you understand the behavior of your algorithms. Just be aware of the limitations to asymptotic analysis in the rare situation where the constant is important.

Upper Bounds

Asymptotic analysis must evaluate aspect of the algorithm's behavior.

One of evaluation criterion is the **upper bound for the growth** of the algorithm's running time. **It indicates the upper or highest growth rate that the algorithm can have.**

Because the phrase "has an upper bound to its growth rate of f(n)" is long and often used when discussing algorithms, special notation is used - **big-Oh notation** "O(f(n))" .

Upper Bounds

Definition for an upper bound

T(n) represents the true running time of the algorithm.

f(n) is some expression for the upper bound. For T(n) a non-negatively valued function, T(n) is in set O(f(n)) if there exist two positive constants **c** and $\mathbf{n_0}$ such that

$$T(n) <= c*f(n) \text{ for all } n > n_0.$$

Constant n_0 is the smallest value of n for which the claim of an upper bound holds true. Usually n_0 is small, such as 1, but does not need to be.

Example

Consider the sequential search algorithm for finding a specified value in an array of integers. Visiting and examining one value in the array requires **c’** steps where is **c’** a positive number.

$T(n) = c' * n/2$ in average case for all values of $n > 1$,
$c' * n/2 < c' * n$.

Therefore, by the definition, $T(n)$ is in $O(n)$ for $n_0 > 1$ and $c = c'$.

For a particular algorithm, $T(n) = c_1 n^2 + c_2 n$ in the average case where c_1 and c_2 are positive numbers.

Then, $c_1 n^2 + c_2 n < c_1 n^2 + c_2 n^2 < (c_1 + c_2) n^2$ for all $n > 1$.
So, $T(n) < cn^2$ for $c = c_1 + c_2$, and $n_0 > 1$.
Therefore, $T(n)$ is in $O(n^2)$ by the definition.

Example

Lets analyze function $\mathbf{f(n) = (n+2)^2}$

Upper bond of f(n) is

$n^2 + 4n + 4 < n^2 + 4n^2$ when $n > 2$

$n^2 + 4n + 4 < 5n^2$

Constant *5* can be reduced, if n will be larger, as for example

if $n > 5$

then $n^2 + 4n + 4 < 2n^2$

In common case:

$n^2 + 4n + 4 < cn^2$

Therefore, T(n) is in $O(n^2)$ by the definition.

Lower Bounds

Similar notation as **Big-Oh** is used to describe the **least amount of a resource** that an algorithm needs for some class of input.

Like big-Oh notation, this is a measure of the algorithm's growth rate and it works for any resource, but we most often measure the least amount of time required.

The lower bound for an algorithm (or a problem, as explained later) is denoted by the symbol Omega, pronounced "big-Omega"

Definition for an lower bound

For T(n) a non-negatively valued function, T(n) is in set Omega(g(n)) if there exist two positive constants c and n_0 such that T(n) >= c*g(n) for all n > n_0

Example

Assume: $T(n) = c_1n^2 + c_2n$ for c_1 and $c_2 > 0$.

then, $c_1n^2 + c_2n > c_1n^2$ for all $n > 1$.

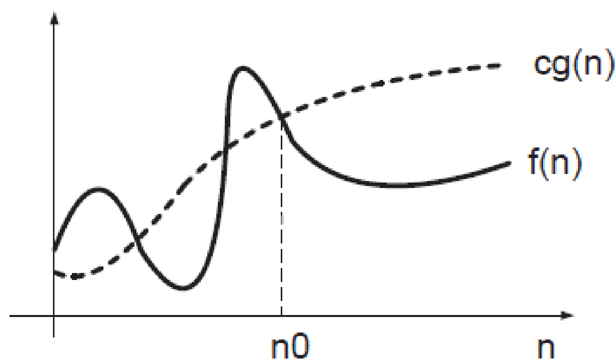
So, $T(n) > cn^2$ for $c = c_1$ and $n_0 > 1$.

Therefore, T(n) is in Omega(n^2) by the definition.

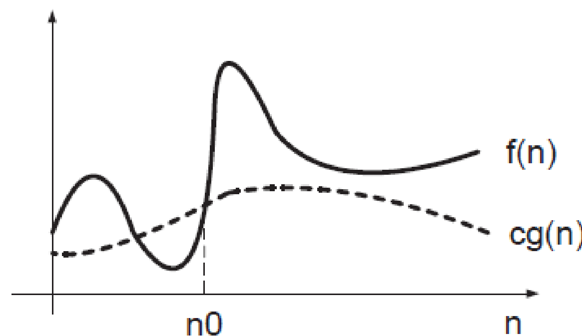
Big-Theta notation

The definitions for big-Oh and Omega give us ways to describe the upper and lower bound for an algorithms. When the upper and lower bounds are the same within a **constant factor**, we indicate this by using (big-Theta) notation.

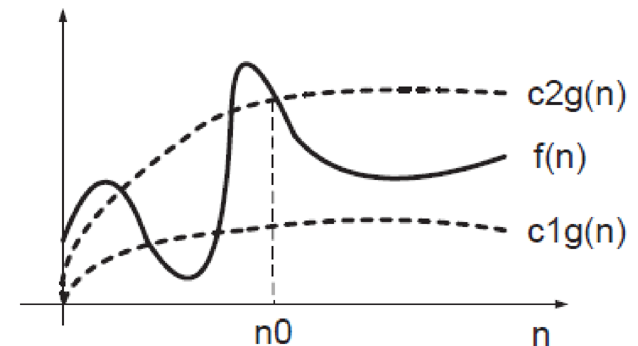
$$0 \le c_1\ g(n) \le f(n) \le c_2\ g(n), \quad n \ge n_0\,.$$



a) $f(n) = \mathcal{O}(g(n))$



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



c) $f(n) = \Theta(g(n))$

Example

```
a = b;
```

Because the assignment statement takes constant time, it is big-Theta(1).

```
sum = 0;
    for ( i=1; i <= n; i++ )
        sum += n;
```

The first line is Big-theta(1). The **for** loop is repeated n times. The third line takes constant time so, the total cost for executing the two lines making up the **for** loop is (n) and the cost of the entire code fragment is also big-Theta(n).

```
sum = 0;
    for (i=1; i<=n; i++)        // loop
    for (j=1; j<=i; j++)        // is a double loop
        sum++;
```

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2},$$ which is Big-theta(n^2)

Simplifying Rules

Following rules are used to determine the simplest form of big-Oh, big-Omega, big-Theta.

1. If $f(n)$ is in $\mathrm{O}(g(n))$ and $g(n)$ is in $\mathrm{O}(h(n))$, then $f(n)$ is in $\mathrm{O}(h(n))$.
2. If $f(n)$ is in $\mathrm{O}(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $\mathrm{O}(g(n))$.
3. If $f_1(n)$ is in $\mathrm{O}(g_1(n))$ and $f_2(n)$ is in $\mathrm{O}(g_2(n))$, then $f_1(n) + f_2(n)$ is in $\mathrm{O}(\max(g_1(n), g_2(n)))$.
4. If $f_1(n)$ is in $\mathrm{O}(g_1(n))$ and $f_2(n)$ is in $\mathrm{O}(g_2(n))$, then $f_1(n)f_2(n)$ is in $\mathrm{O}(g_1(n)g_2(n))$.

Classifying Functions

Given functions **f(n)** and **g(n)** whose growth rates are expressed as algebraic equations, we might like to determine if one grows faster than the other. The best way to do this is to take the limit of the two functions as **n** grows towards infinity:

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

If the limit goes to **infinity**, then **f(n)** is in Omega(g(n)) because f(n) grows faster. If the limit goes to zero, then f(n) is in O(g(n)) because g(n) grows faster.
If the limit goes to some constant other than zero, then f(n) = Big-Theta(g(n)) because both grow at the same rate.

Example

f(n) = 2n log n;
g(n) = n^2,

Is f(n) in O(g(n)), Omega(g(n)), or Big-theta (g(n))?

$$\frac{n^2}{2n\log n} = \frac{n}{2\log n},$$

$$\lim_{n\to\infty} \frac{n^2}{2n\log n} = \infty$$

because **n** grows faster than **2 log n**, thus, $\mathbf{n^2}$ is in Omega(2n log n).

Empirical analysis of algorithms

An alternative to analytical approaches of algorithms evaluations are empirical approaches.

The most obvious empirical approach is simply to run two competitors and see which performs better. In this way we might overcome the deficiencies of analytical approaches.

Empirical analysis of algorithms

Problems

Comparative timing of programs is a difficult business, often subject to experimental errors arising from uncontrolled factors (system load, the language or compiler used, etc.).

The most important point is not to be biased in favor of one of the programs. If you are biased, this is certain to be reflected in the timings.

The most common pitfall when writing two programs to compare their performance is that one receives more code-tuning effort than the other. Code tuning can often reduce running time by a factor of ten. If the running times for two programs differ by a constant factor regardless of input size, then differences in code tuning might account for any difference in running time.

Exercises

Using the definitions of big-Oh and Ω, find the upper and lower bounds for the following expressions. Be sure to state appropriate values for c and n_0.

(a) $c_1 n$

(b) $c_2 n^3 + c_3$

(c) $c_4 n \log n + c_5 n$

(d) $c_6 2^n + c_7 n^6$

(a) What is the smallest integer k such that $\sqrt{n} = O(n^k)$?

(b) What is the smallest integer k such that $n \log n = O(n^k)$?

Exercises

Calculate running time, big-Oh, big-Omega, big-Theta of code fragment with several **for** loops:

```
sum = 0;
    for (i=1; i<=n; i++)           // First for loop
        for (j=1; j<=i; j++)    // is a double loop
            sum++;
    for (k=0; k<n; k++)            // Second for loop
        A[k] = k;
```

Exercises

Built flowchart for the following algorithm.

Find if given number is prime. Define running time of the algorithm.