CSC 8301- Design and Analysis of Algorithms

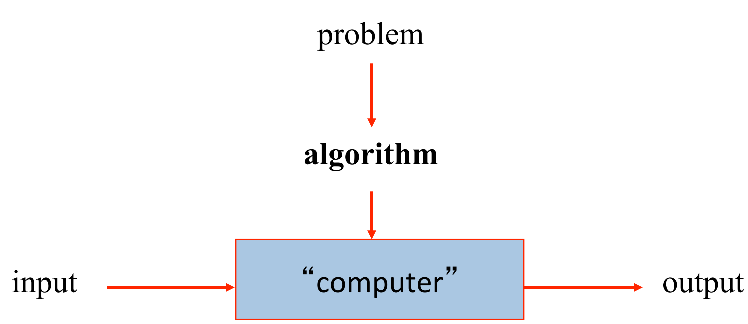
Lecture 1

Introduction

Analysis framework and asymptotic notations

What is an algorithm?

An *algorithm* is a *finite* sequence of *unambiguous* instructions for solving a problem, i.e., for obtaining a required output for any legitimate input.



Euclid's algorithm

Problem: Find gcd(m,n), the greatest common divisor of two nonnegative, not both zero integers m and n

Euclid's algorithm is based on repeated application of equality

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

gcd(24,60) = gcd(60, 24)

Ex.: gcd(60,24) = gcd(24,12) = gcd(12, 0) = 12

```
while n ≠ 0 do
   r ← m mod n
   m ← n
   n ← r
return m
```

Finite: 2^{nd} argument decreases

Unambiguous: well-defined sequence of steps

EFFICIENT!

Why study algorithms?

- Theoretical importance
  - The cornerstone of computer science
- Practical importance
  - a practitioner's toolkit of known algorithms
  - frameworks for designing and analyzing algorithms for new problems

Two main issues related to algorithms

- How to design algorithms
- How to analyze algorithm efficiency

Major Algorithm Design Techniques/Strategies

- Brute force
- Decrease and conquer
- Divide and conquer
- Transform and conquer
- Space-time tradeoff
- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and Bound

Analysis of Algorithms

- How good is the algorithm?
  - correctness (accuracy for approximation alg.)
  - time efficiency
  - space efficiency
  - optimality

- Approaches:
  - empirical (experimental) analysis
  - theoretical (mathematical) analysis

Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of times the algorithm's *basic operation* is executed as a function of *input size*

- *Input size*: number of input items or, if matters, their size n
- *Basic operation*: the operation contributing the most toward the running time of the algorithm

C_{op} = cost of the basic operation

$C(n)$ = # times the basic operation executes

Total time

$T(n) = C_{op} * C(n)$

Example: Searching for a key in a list

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

//Searches for a given value in a given array by sequential search
//Input: An array $A[0..n-1]$ and a search key K
//Output: The index of the first element of A that matches K
// or -1 if there are no matching elements
$i \leftarrow 0$
while $i < n$ **and** $A[i] \neq K$ **do**
  $i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

Input Size? n

Basic Operation? Comparison

Example: Multiplying two *n*x*n* matrices

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** $n-1$ **do**
    $C[i, j] \leftarrow 0.0$
    for $k \leftarrow 0$ **to** $n-1$ **do**
      $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C

Input Size? n

Basic Operation? Multiplication

Best-case, average-case, worst-case

For some algorithms efficiency depends on form of input:

- Worst case: $C_{worst}(n)$ – maximum over inputs of size n
- Best case: $C_{best}(n)$ – minimum over inputs of size n
- Average case: $C_{avg}(n)$ – "average" over inputs of size n
  - Number of times the basic operation is executed on *typical* input
  - NOT the average of worst and best case
  - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs

Example: Searching for a key in a list

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$
//Searches for a given value in a given array by sequential search
//Input: An array $A[0..n-1]$ and a search key K
//Output: The index of the first element of A that matches K
// or -1 if there are no matching elements
$i \leftarrow 0$
while $i < n$ **and** $A[i] \neq K$ **do**
$\quad i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

Worst Case? $T(n) = n+1$

Best Case? $T(n) = 1$

Average Case? (assuming probability of successful search is p and probability of the first match in each position i is same)

Average-case approach

- Divide all instances of size n into several classes, so that in each class the algorithm would take about the same time
- Probability P(I) of input I:
  - how likely input I is
- Random variable C(I):
  - number of steps taken by algorithm on input I
- Average running time is

$$\Sigma_I \, C(I)*P(I)$$

Average Case for Sequential Search (1)

Case 1: The key K is in the list, equally likely to be in any position

$P(K \text{ in } 1^{st} \text{ position}) = 1/n$ (Use $1+2+3+\dots+n = \frac{n(n+1)}{2}$)

$P(K \text{ in } 2^{nd} \text{ position}) = 1/n$

...

$P(K \text{ in } i^{th} \text{ position}) = 1/n$ $(1 \leq i \leq n)$

$T(K \text{ in } 1^{st} \text{ position}) = 1$ (comparison)

$T(K \text{ in } 2^{nd} \text{ position}) = 2$

...

$$T(n) = \frac{1}{n} * 1 + \frac{1}{n} * 2 + \frac{1}{n} * 3 + \dots + \frac{1}{n} * n = \frac{n+1}{2}$$

Average Case for Sequential Search (2)

Case 2: The probability that the key K is in the list is $p \leq 1$

P(key in the list) = p

P(key not in the list) = $1-p$

T(key not in the list) = $n+1$

T(key in the list) = $\frac{n+1}{2}$ (see prev. slide)

$$T(n) = p \cdot \frac{n+1}{2} + (1-p)(n+1) = n+1 - p\,\frac{(n+1)}{2}$$

$$= (n+1)(1-p/2)$$

Example: Multiplying two *n*x*n* matrices

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** $n-1$ **do**
    $C[i, j] \leftarrow 0.0$
    for $k \leftarrow 0$ **to** $n-1$ **do**
      $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C

Worst Case?
Best Case?
Average Case?

$\underline{n^3}$

Types of formulas for basic operation count

- Exact formula

  e.g., $C(n) = n(n-1)/2$

- Formula indicating *order of growth* with specific multiplicative constant

  e.g., $C(n) \approx 0.5\, n^2$

- Formula indicating *order of growth* with unknown multiplicative constant

  e.g., $C(n) \approx cn^2$

Order of growth

Most important: Order of growth within a constant multiple as $n \to \infty$

Example:

- $C(n) = cn^2$
- Suppose we double the input size. How much longer will the algorithm take? Compare $C(2n)$ with $C(n)$

$$\frac{C(2n)}{C(n)} = \frac{\cancel{c} * (2n)^2}{\cancel{c} * n^2} = \frac{4n^2}{n^2} = 4$$

Values of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Example

- Order these functions according to their order of growth (from lowest to highest):

$2^n \qquad n^{4/3} \qquad n(\lg n)^3 \qquad n^{\lg n}$

$n(\lg n)^3$

$n^{4/3}$

$n^{\lg n}$

2^n

Compare $\boxed{2^n}$ with $\boxed{n^{\lg n}}$

$\boxed{2^n} \xrightarrow{\log} \boxed{n \log 2}$

$\boxed{n^{\lg n}} \xrightarrow{\log} \boxed{\lg n \log n}$

Main Points of the Analysis Framework

- Both time and space efficiencies are measured as functions of the algorithm's input size.
- Time efficiency is measured by counting the number of times the algorithm's basic operation is executed.
- Space efficiency is measured by counting the number of extra memory units (beyond input and output) used by the algorithm.
- For some algorithms, one should distinguish among the worst, best and average case efficiencies.
- The main concern is the *order of growth* of the algorithm's running time and extra memory units consumed as input size goes to infinity.

Asymptotic order of growth

A way to classify functions according to their order of growth

- *practical* way to deal with complexity functions
- ignores constant factors and small input sizes

- Big-O
  - $O(g(n))$: class of functions $f(n)$ that grow *no faster* than $g(n)$
- Big-Theta
  - $\Theta(g(n))$: class of functions $f(n)$ that grow *at same rate* as $g(n)$
- Big-Omega
  - $\Omega(g(n))$: class of functions $f(n)$ that grow *at least as fast* as $g(n)$

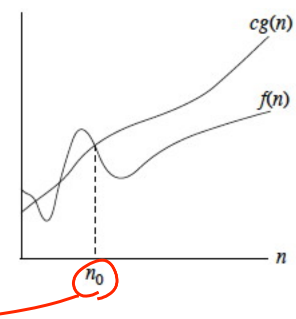
Big-O (asymptotic ≤)

Definition: $f(n)$ is in $O(g(n))$ if order of growth of $f(n) \le$ order of growth of $g(n)$ (within constant multiple),
i.e., there exist positive constant c and non-negative integer n_0 such that

$$f(n) \le c\, g(n) \text{ for every } n \ge n_0$$

Examples:

- $10n^2$ is $O(n^2)$
- $10n$ ~~is $O(n^2)$~~ True
- $5n+20$ is $O(n)$ $5n + 20 \le 10n$ for $n \geq 4$



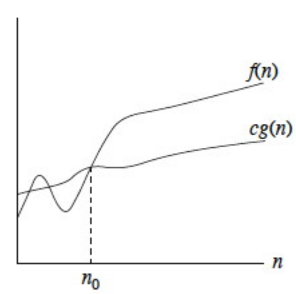
Ω (Omega, asymptotic ≥)

Definition: $f(n)$ is in $\Omega(g(n))$ if there exist positive constant c and non-negative integer n_0 such that

$$f(n) \ge c\, g(n) \text{ for every } n \ge n_0$$

These are all $\Omega(n^2)$:

- n^2
- $n^2 + 100n$
- $1000n^2 - 1000n$
- n^3



These are not:

- $n^{1.999}$
- n
- $\lg n$

Θ (Theta, asymptotic =)

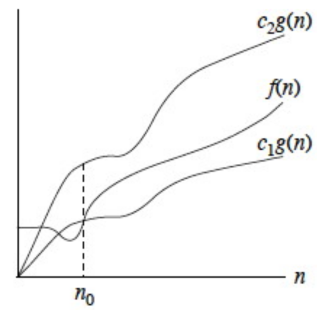
Definition: $f(n)$ is in $\Theta(g(n))$ if there exist positive constants c_1, c_2 and non-negative integer n_0 such that

$$c_1\, g(n) \le f(n) \le c_2\, g(n) \text{ for every } n \ge n_0$$

Example:

- n^2-$2n$ is $\Theta(n^2)$
  - pick $c_1 = 0.5$, $c_2 = 1$, $n_0 = 4$

$\frac{n^2}{2} \le n^2 - 2n \le n^2$ True for $n \geq 4$



Find a tight Θ-bound for:

- $4n^3$
- $4n^3+2n$

$4n^3 \le 4n^3 \le 4n^3$

$4n^3 \le 4n^3 + 2n \le 6n^3$

Establishing order of growth

High level idea: ignore:

- constant factors (too system-dependent)
- low-order terms (irrelevant for large inputs)

For example,

$6n \log n + 2n$ becomes $\Theta(n \log n)$

Establishing order of growth using limits

$$\lim_{n\to\infty} f(n)/g(n) = \begin{cases} 0 & \text{order of growth of } f(n) < \text{order of growth of } g(n) \\ c > 0 & \text{order of growth of } f(n) = \text{order of growth of } g(n) \\ \infty & \text{order of growth of } f(n) > \text{order of growth of } g(n) \end{cases}$$

Examples:

- $10n$ vs. n^2 $\quad \lim_{n\to\infty} \frac{10n}{n^2} = \lim_{n\to\infty} \frac{1}{n} \to \emptyset$

- $n(n+1)/2$ vs. n^2 $\quad \lim_{n\to\infty} \frac{n(n+1)}{2n^2} = \lim_{n\to\infty} \frac{1}{2} + \lim_{n\to\infty} \frac{1}{n} = \frac{1}{2}$

Compare log n with √n

L'Hôpital's Rule and Stirling's Formula

L'Hôpital's rule: If $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$ and the derivatives f', g' exist, then

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

Example: log(*n*) vs. √*n*

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = \lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{2} n^{-\frac{1}{2}}} = \lim_{n\to\infty} \frac{2\sqrt{n}}{n} = \lim_{n\to\infty} \frac{2}{\sqrt{n}} \to \emptyset$$

Stirling's formula: $n! \approx (2\pi n)^{1/2} (n/e)^n$

Orders of growth of some important functions

- All logarithmic functions $\log_a n$ belong to the same class $\Theta(\log n)$ no matter what the logarithm's base $a>1$ is.
- All polynomials of the same degree k belong to the same class: $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 \in \Theta(n^k)$.
- Exponential functions a^n have different orders of growth for different a.
- order $\log n$ < order n^α ($\alpha>0$) < order a^n < order $n!$ < order n^n

Some properties of asymptotic order of growth

- $f(n) \in O(f(n))$
- $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$
- If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$

  Note similarity with $a \le b$
- If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then
  $$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

Homework

- Exercises
  1.1: 6, 9a, 12
  1.2: 1, 2
  1.3: 4, 5
  2.1: 3, 5a, 8, 9
  2.2: 3, 5, 9, 12

- Reading:
  - Preface and Chapter 1 (Sections 1.1-1.4)
  - Sections 2.1 and 2.2