COMP 3761: Algorithm Analysis and Design

Shidong Shan

BCIT

What is an algorithm?

- A recipe, process, method, technique, procedure, routine, $\cdots$
- A sequence of instructions for solving a problem
- Requirements:
  - Finiteness
  - Definiteness
  - Input
  - Output
  - Effectiveness

Why study algorithms

- Theoretical importance
  - The core of computer science
- Practical importance
  - A toolkit of known algorithms
  - Framework for designing and analyzing algorithms for new problems

Basic issues related to algorithm

- How to design algorithms
- How to express algorithms
- Proving correctness
- Efficiency
- Theoretical analysis
- Empirical analysis
- Optimality

Basic design strategies

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Greedy approach
- Dynamic programming
- Backtracking
- Branch and bound
- Space and time tradeoffs

Analysis of Algorithms

- How good is the algorithm?
  - Correctness
  - Time efficiency
  - Space efficiency
- Does there exist a better algorithm?
  - Lower bounds
  - Optimality

Some important problem types

- sorting
- searching
- string processing
- graph problem
- combinatorial problems
- geometric problems
- numerical problems

Input size

- The amount of time required for an algorithm to complete varies according to the size of the input
  e.g. sorting 100 items takes less time than sorting 10000000 items
- $T(n)$ is the time required for an algorithm to solve a problem of size n
- Definition of n is problem specific
  sorting: n is (usually) the number of items to sort
- Primary interest: the order of growth of the algorithm's running time as $n \to \infty$

Theoretical analysis of time efficiency

- Identify the basic operation:
  the operation that contributes most towards the running time of the algorithm
- Determine the number of times the basic operation is executed
- Measure as a function of input size

$$T(n) \approx c_{op} S(n).$$

  - $T(n)$: running time
  - c_{op}: execution time for basic operation
  - $S(n)$: number of times basic operation is executed.

Summation formulas and rules

- Use summation formulas to express $T(n)$ as closed forms, e.g.

$$T(n) = \Sigma_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{1}{2}n(n+1)$$

- Summation rules, e.g.

$$\begin{aligned} \Sigma(a+b) &= \Sigma a + \Sigma b \\ \Sigma_i c x_i &= c \Sigma_i x_i \end{aligned}$$

Empirical analysis of time efficiency

- Select a specific (typical) sample of inputs
- Use physical unit of time (e.g., milliseconds), OR
- Count actual number of executions of basic operations
- Analyze the empirical data

- Difficulties with this approach:
  - implementation-dependent
  - difficult to create a large number of input sets to properly test the programs

Best-case, average-case, worst-case

For some algorithms, efficiency depends on the form of the input:

- best-case $C_{best}(n)$: minimum over inputs of size n
- worst-case $C_{worst}(n)$: maximum over inputs of size n
- average-case $C_{avg}(n)$: "average" over inputs of size n
    - Number of times the basic operation will be executed on typical or random input
    - In general, $C_{avg}(n) \neq \frac{1}{2}(C_{best}(n) + C_{worst}(n))$
    - Under some standard assumptions about the probability distribution of all possible inputs.

Algorithm: Sequential Search

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

Analysis: Sequential Search

Time efficiencies:

- $C_{worst}(n) = n$
- $C_{best}(n) = 1$
- $C_{avg}(n) = \frac{1}{2}p(n+1) + n(1-p)$,
  where p is the probability of a successful search.

Order of growth

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

- How much faster will the algorithm run on a computer that is twice as fast as another?
- How much longer does it take to solve problem of double input size?

Definitions

- Big-oh: $t(n) \in O(g(n))$ if $t(n) \leq cg(n)$, $\forall n \geq n_0$;
  $t(n)$ is bounded above by some constant multiples of $g(n)$
- Big-omega: $t(n) \in \Omega(g(n))$ if $t(n) \geq cg(n)$, $\forall n \geq n_0$;
  $t(n)$ is bounded below by some constant multiples of $g(n)$
- Big-theta: $t(n) \in \Theta(g(n))$ if $c_2 g(n) \leq t(n) \leq c_1 g(n)$, $\forall n \geq n_0$;
  $t(n)$ is bounded both above and below by some constant multiples of $g(n)$

Big-Oh

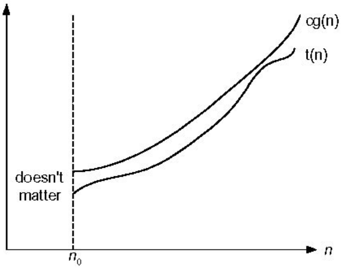

Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Big-Omega

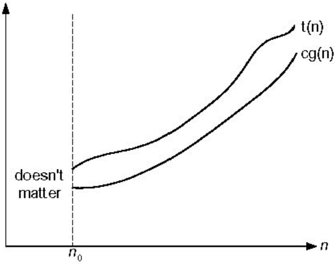

Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-Theta

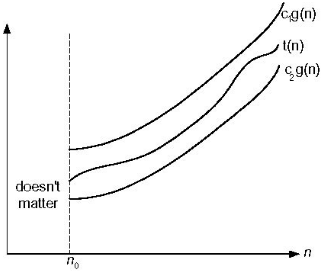

Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Useful properties of asymptotic notations

- $f(n) \in O(f(n))$
- $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$
- If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$
- If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then $f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$

Using limits for comparison

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & t(n) \text{ has a smaller order of growth than } g(n) \\ c > 0 & t(n) \text{ has the same order of growth as } g(n) \\ \infty & t(n) \text{ has a larger order of growth than } g(n) \end{cases}$$

- The first two mean $t(n) \in O(g(n))$
- The last two mean $t(n) \in \Omega(g(n))$
- The second case mean $t(n) \in \Theta(g(n))$

Common efficiency orders

- $O(1)$: constant
- $O(\log n)$: logarithmic
- $O(n)$: linear
- $O(n \log n)$: n-log-n
- $O(n^2)$: quadratic (special case of polynomial)
- $O(n^k)$: polynomial
- $O(2^n)$: exponential
- $O(n!)$: factorial

Order of growth of some important functions

- All logarithmic functions $\log_a n$ belong to the same class $\Theta(\log n)$ regardless of base $a > 1$
- 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's
- $O(\log n) < O(n^k) < O(a^n) < O(n!) < O(n^n)$, for $k > 0$.

Steps for analysis of nonrecursive algorithms

1. Decide on parameter n indicating input size
2. Identify algorithm's basic operation
3. Determine worst, average, and best cases for input of size n
4. Set up a sum for the number of times the basic operation is executed
5. Simplify the sum using standard formulas and rules.

MaxElement

```
ALGORITHM MaxElement(A[0..n − 1])
    //Determines the value of the largest element in a given array
    //Input: An array A[0..n − 1] of real numbers
    //Output: The value of the largest element in A
    maxval ← A[0]
    for i ← 1 to n − 1 do
        if A[i] > maxval
            maxval ← A[i]
    return maxval
```

UniqueElement

ALGORITHM *UniqueElements*$(A[0..n-1])$
//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[i] = A[j]$ **return false**
return true

Algorithm

Implement the following algorithm (Text Page 68 Problem 5) in Java:

Algorithm Secret($A[0..n-1]$)
  //Input: An array $A[0..n-1]$ of n real numbers
  $minval \leftarrow A[0]$; $maxval \leftarrow A[0]$
  for $i \leftarrow 1$ to $n-1$ do
    if $A[i] < minval$
      $minval \leftarrow A[i]$ if $A[i] > maxval$
      $maxval \leftarrow A[i]$ **return** $maxval - minval$

Lab Exercise: Algorithm Implementation and Analysis

Answer the following questions:

1. What does this algorithm compute?
2. What is its basic operation?
3. How many times is the basic operation executed?
4. What is the time efficiency of this algorithm?
5. Run your program on different magnitudes of random input size n (e.g, $n = 10, 20, 100, 500, 1000, \cdots$) and verify its running time.