EECS560　　　　Data Structures　　　　Kong

Topic 1: Introduction to Algorithmic Analysis

Read: Chpt. 1, 3 (Review of Basic DS),
Chpt. 2 (Algorithmic Complexity), Weiss.

Objectives of EECS560:

To study the design, functionality, implementation, and performance of advance *abstract data types* (ADTs) and their implementations for the development of "efficient" programs for solving real-world computational problems.

Remark: EECS560 is not a course on C++, Java, or any other programming language; it is a course on efficient data organizations and their analyses used in problems solving.

Solving a Problem:

To develop an efficient ***program/software*** that can be used to generate a correct output for all possible inputs to the problem.

Data structure:

Organization of data objects so as to facilitate computation in an "***efficient***" manner.

Remark: A carefully designed good data structure provides a framework for the implementation of efficient algorithm.

Observation:

Program = Algorithm + Data Structures

Approach in Data Structures Design and Analysis:

Data Abstractions:

Concentrating only on the structures of the data objects (elements) and the operations (functions) that can be performed (defined) on them.

What is difference between ADT and Data Structure?

Abstract Data Type:

A mathematical system consisting of a collection of data objects together with some operations defined on them.

ADT = {Data, Operations}.

Remarks:

(1) ADT is language independent.
(2) Each operation of an ADT corresponds to one or more algorithms that operate on the data.
(3) In EECS560, we will concentrate on the design and analysis of different classes of Abstract Data Types (ADTs) and their implementations.

Algorithm:

Algorithm is a sequence of step-by-step instructions/statements that can be executed to solve a problem in a finite amount of time and, algorithm always operates on data.

Remark: Statements can be

- **Simple:** assign, compare, return, add, …
- **Structured:** switch, loop, function call, …

Characteristics of an Algorithm:

- Read by human
- Machine/language independent
- Unambiguous
- Must terminate

Algorithm vs. Program:

Program	**Algorithm**
Machine/Language dependent	Independent
May not terminate	Must terminate

Algorithm Specification:

<u>Method</u>	***<u>Simplicity</u>***	***<u>Precision</u>***
English	Simplest	Least precise
Pseudo code		
High-Level P.L.	↓	↓
Low-Level P.L.		
Machine Language	Most complex	Most precise

Standard Format in Describing an Algorithm:

Input:
Output:
Algorithm:

Remark: In describing your algorithm, you must first explain your approach in plain English, followed by your algorithm in pseudo code, then C++ code if required.

Q: How good/bad is a given algorithm?
More importantly, how do we know that this algorithm is even correct?

Analysis of Algorithms:

- Correctness,
- Efficiency.

Measuring the Goodness/Efficiency of an Algorithm:

Compute the amount of computing resources required for the execution of the algorithm.

Some Important Cost Factors:

T(n)	—	Time complexity,
S(n)	—	Space complexity,
		where n is the number/size of inputs.

Approach 1: Experimental Profiling

Implement a given algorithm in a programming language and then execute the program in a machine with a suitable set of input data. Verify the correctness of the algorithm/program and measure the CPU time/memory used.
EECS168/268/560: Concentrating on experimental approach.

Problems: Unreliable; also highly machine/language/input/human dependent.

Approach 2: Analytical approach:

Need to prove the correctness of the algorithm formally using various proof techniques. For efficiency, identify some basic operations in the algorithm that will dominate the execution time complexity of the algorithm and then count them. Memory/space complexity can be computed in a similar fashion.

EECS560/660: Concentrate on analytical approach.

Problems: Mathematically involved; can be extremely difficult to execute. We also need a computational model.

1.Understand the RAM computational model and
2.be able to state its assumptions and
3.use them to analyze the performance of algorithms!
第一个考点！！

Simplest Model of Computation:

Random Access Machine (RAM):

4个假设

1. Only one instruction can be executed at a time. (Sequential machine)
2. Each datum is small enough to be stored in a single memory cell. (Uniform cost criterion)
3. Each stored datum can be accessed with the same constant cost. (Random access model)
4. Each basic operations such as read, write, +, -, ∗, /, compare(x,y), return, …, requires a constant cost. (Normalization)

Remark: More realistic but much more complicate computational models exist.

Let A be an algorithm for a given problem Π. What is the least, most, and average amount of computing resource required in order to execute A?

RAM例子

Algorithmic Fundamentals:

Let D_n be the set of all possible inputs to Π of size n,
$C(I)$ be the amount of computing resource required to execute A with input $I \in D_n$,
$\Pr(I)$ be the probability when I is the input to A,
$R(n)$ be the complexity function of A when executed with any input of size n.

size n is important

1. Best-Case Complexity:

$$R_b(n) = \min_{I \in D_n} C(I).$$

2. Worst-Case Complexity: inFrenquently

$$R_w(n) = \max_{I \in D_n} C(I).$$

3. Average-Case Complexity:

$$R_a(n) = \sum_{I \in D_n} \Pr(I) * C(I).$$

Remark:

$R_a(n)$ is usually very difficult to compute. Different probability functions will lead to different $R_a(n)$.

Computing Complexity Functions:

Recall that an algorithm is a sequence of step-by-step instructions.

```
Algorithm A:        S_1;
                    S_2;
                    •
                    •
                    •
                    S_m;
```

Let $\text{cost}(S_i)$ be the cost in executing the statement S_i, $1 \le i \le m$. Hence,

$$T(n) = \sum_{1 \le i \le m} \text{cost}(S_i).$$

Some Complications:

1. *S_i is a **conditional statement**:* if-then-else, case, switch, etc.

   $\text{cost}(S_i)$ = cost in evaluating the condition +
   cost in evaluating one of the branches

Q: Which branch should we execute?
How do we compute $R_b(n)$, $R_w(n)$, or $R_a(n)$?

2. *S_i is a **repetition (loop)**:* do-loop, while-loop, repeat-loop, etc.
   cost(S_i)
   = (# times the loop condition is evaluated $*$
   cost in evaluating the loop condition) +
   (# times the loop is evaluated $*$
   cost in evaluating the body of the loop)

   Warning: It can be very tricky in determining how many times a loop will be executed.
   Be very careful when evaluating cost of loops.

3. *S_i is a **recursive call**:*
   May need to set up and solve the recurrence equation for cost(S_i).

Algorithm—> prove —> analysis

no control input n;

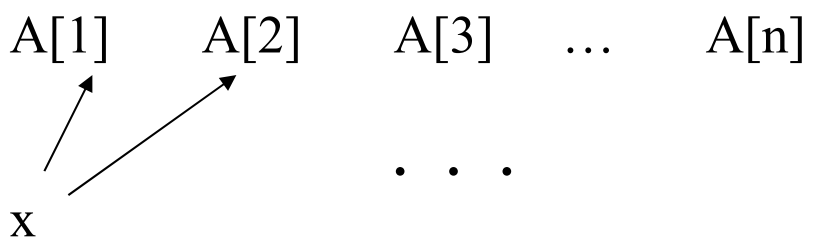
Example: A simple searching problem.

Input: An array A[1..n] of n distinct integers, n ≥ 1, and an integer key x.

Output: Return integer i, 1 ≤ i ≤ n, if A[i] = x. Else, return 0.

RAM的例子：一个值是否在数组中，返回下标或者没找到返回0。。。

Approach: Using sequential search algorithm.
Comparing x with A[1], A[2], …, A[i] successively until x = A[i] (return i) or x ≠ A[i], for all i, 1 ≤ i ≤ n, (return 0).



how to prove algorithm terminate?

Algorithm: Pseudo code

```
Sequential_Search(A:array, x: integer);
    i = 1;                                    (assignment)
    while $i \le n$ and $A[i] \ne x$ do   // find x sequentially   (two comparsion and logic and)
        i = i + 1   addition and assignment
    endwhile;
    if $i \le n$                              (Comparsion)
        then return(i)        // A[i] = x found   (return:)
        else return(0)        // $x \notin A$
    endif;
end Sequential_Search;
```

comparsion: max: n+1
min: 1

1. Detailed Analysis:
   Approach: Assign a constant cost to each and every operation and then compute the total cost in executing the algorithm.

   Three statements with the following operations and costs:

Operations	**Cost**	
assignment	c_1	c1 is constant
comparison	c_2	have two c2 (i<=n ,A[i]!=x)
logical-and	c_3	keyword "and" in pseudo code
addition	c_4	c4 in while loop
return	c_5	c5 is constant

Q: How many times will the while-loop be executed?

min # times = 0 $(x = A[1])$

max # times = n $(x = A[n] \text{ or } x \notin A)$

average # times = ?

Best-case Complexity:

$$T_b(n) = c_1 + (2c_2 + c_3) + (c_2 + c_5)$$
$$= c_1 + 3c_2 + c_3 + c_5$$

Worst-case Complexity:

$$T_w(n) = c_1 + [(n+1)(2c_2 + c_3) + n(c_1 + c_4)] + (c_2 + c_5)$$
$$= (c_1 + 2c_2 + c_3 + c_4)n + (c_1 + 3c_2 + c_3 + c_5)$$

方法二：找出次数多的的基础操作

2. Simplified Approach using Basic Operation(s):
Approach: Instead of counting all operations, identify most important basic operation(s) and just count them. Relative efficiencies of algorithms will be determined based on their numbers of basic operations used.

Most important (basic) operations:
Number of comparisons between x and A[i].

Minimum number of comparisons between x and A[i]:
$T_b(n) = 1$.

Maximum number of comparisons between x and A[i]:
$T_w(n) = n$. 最多比较n+1次

HW: If the given array A[1..n] is sorted, modify the above sequential search algorithm for searching a key x in A. Give your sequential search algorithm in pseudo code and then compute its complexities.

方法三：只看最主要的循环

3. Simplified Approach by Combining Basic Operations in Dominating Steps:
   Approach: We first combine all related steps with constant cost and assign them a constant cost. Next, identify steps that will dominate the execution of the algorithm and compute the total cost in executing these dominating steps. If only few basic operations are involved in these steps, we may further simplify the computation by assuming that all these basic operations in these steps have the same constant cost.

   Dominating step in sequential search algorithm:

```
while $i \le n$ and $A[i] \neq x$ do    // find x
    $i = i + 1$
endwhile;
```

   Best-case complexity:

$$T_b(n) = C.$$

   Worst-case complexity:

$$T_w(n) = \sum_{i=1}^{n} C = Cn.$$

More Examples in using the Dominating Steps:

1.

```
x = 2;
y = 5;
for i = 1 to n do
    for j = 1 to n do
        for k = 1 to n do
            y = x * y / 2;
            x = x + y – 10;
        endfor;
    endfor;
endfor;
```

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} K$$

$$= \sum_{i=1}^{n} \sum_{j=1}^{n} Kn$$

$$= \sum_{i=1}^{n} Kn^2 = Kn^3.$$

2.
```
x = 5;
y = 60;
for i = 1 to n do
    for j = 1 to i do
        x = 2*x + 1;
    endfor;
    for k = 1 to n do
        y = x*y/2;
    endfor;
endfor;
```

T(n)

$$= \sum_{i=1}^{n} (\sum_{j=1}^{i} + \sum_{k=1}^{n})C$$

$$= C \sum_{i=1}^{n} (i + n)$$

$$= C[\frac{n(n+1)}{2} + n^2].$$

3.
```
k = 1;
x = 2;
y = 3;
while k <= n do
    for i = 1 to n do
        k = k + i;
    endfor;
    x = x + y;
endwhile;
```

Observe that the while-loop will only be executed once. Hence,

$$T(n) = \sum_{i=1}^{n} C = Cn.$$

4.
```
k = 1;
x = 6;
y = 60;
while k <= n^2 do
    x = (x*y + 2*x)/4;
    k = k*k;
endwhile;
```

Observe that k is always equal to 1. Hence, we have an infinite loop and $T(n) = \infty$.

From all previous examples, observe that all T(n)'s are being represented by a very simple mathematical expression (elementary function). These are called the ***closed-form expression*** of T(n).

Remark: If T(n) = f(n), where f(n) is an elementary function, then T(n) can be computed exactly by substituting n into f(n).

Q: What if such an expression can't be found (either doesn't exist or much too difficult to compute) to represent T(n)?

Use approximation!

In order to provide a guarantee on how much computing resource is needed in executing A, we may want to find an elementary function f(n) such that $T(n) \leq f(n)$ for all n.

We can simplify our computation even further by finding an elementary function f(n) such that $T(n) \leq kf(n)$ for sufficiently large n, where k is any constant > 0.

如果累加太困难，可以考虑使用约等于

Review: Asymptotic Analysis of Algorithms

Defn: A function f: N → R is a ***positive function*** if f(n) > 0 for all n; f(n) is an ***eventually positive function*** if f(n) > 0 for all $n \geq n_0$.

Remark: Observe that all complexity functions are eventually positive functions. Also, unless specified otherwise, all functions considered in this course are eventually positive functions.

upper bound

Defn: f(n) = O(g(n)) iff ∃ constants k > 0, $n_0 > 0$ such that

$$f(n) \leq k(g(n)) \ \forall\ n \geq n_0.$$

Examples:

Text

1. $2n^2 - 3n + 10 = O(n^2)$.
2. $2n^2 - 3n + 10 = O(n^3)$.
3. $3\lg n! = O(n\lg n)$.
4. $n^2 - 3n^{16} + 2^n = O(2^n)$.
5. $n^2 - 36n\lg n - 1024 = O(n^2)$.
6. $\dfrac{n^4 - 8n^3 + 18n^2 - 15n}{3n^2 - 2n + 1024} = O(n^2)$.
7. $2^{n+1} = O(2^n)$.
8. $4^n \neq O(3^n)$.

Defn: f(n) = Ω(g(n)) iff $\exists$ constants k > 0, n_0 > 0 such that f(n) $\geq$ k(g(n)) $\forall$ n $\geq$ n_0.

Theorem: f(n) = Ω(g(n)) iff g(n) = O(f(n)).

Examples:

1. $2n^2 - 3n + 10 = \Omega(n^2)$.
2. $2n^2 - 3n + 10 \neq \Omega(n^3)$.
3. $3\lg n! = \Omega(n\lg n)$.
4. $n^2 - 3n^{16} + 2^n = \Omega(2^n)$.
5. $n^2 - 36n\lg n - 1024 = \Omega(n^2)$.
6. $\frac{n^4 - 8n^3 + 18n^2 - 15n}{3n^2 - 2n + 1024} = \Omega(n^2)$.
7. $2^{n+1} = \Omega(2^n)$.
8. $4^n = \Omega(3^n)$.

Defn: $f(n) = \Theta(g(n))$ iff $\exists$ constants $k_1 > 0$, $k_2 > 0$, $n_0 > 0$ such that $k_2 g(n) \le f(n) \le k_1 g(n)$, $\forall\ n \ge n_0$.

Theorem: The following statements are equivalence:

1. $f(n) = \Theta(g(n))$.
2. $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
3. $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Examples:

1. $2n^2 - 3n + 10 = \Theta(n^2)$.
2. $2n^2 - 3n + 10 \ne \Theta(n^3)$.
3. $3\lg n! = \Theta(n\lg n)$.
4. $n^2 - 3n^{16} + 2^n = \Theta(2^n)$.
5. $n^2 - 36n\lg n - 1024 = \Theta(n^2)$.
6. $\dfrac{n^4 - 8n^3 + 18n^2 - 15n}{3n^2 - 2n + 1024} = \Theta(n^2)$.
7. $2^{n+1} = \Theta(2^n)$.
8. $4^n \ne \Theta(3^n)$.

HW: Review big-O, big-Ω, and big-Θ.
For various functions f(n) and g(n), try to prove or disprove any asymptotic relation(s) between f(n) and g(n).

Some Important Properties of big-O, big-Ω, and big-Θ:

1. **Reflexive property:**
   $f(n) = O(f(n))$,
   $f(n) = \Omega(f(n))$,
   $f(n) = \Theta(f(n))$.

2. **Symmetric property:**
   $f(n) = \Theta(g(n))$ implies $g(n) = \Theta(f(n))$.

3. **Transitive property:**
   If $f(n) = O(g(n))$ and $g(n) = O(h(n))$,
   then $f(n) = O(h(n))$.
   If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$,
   then $f(n) = \Omega(h(n))$.
   If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$,
   then $f(n) = \Theta(h(n))$.

4. **Sum Rule:**
   If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$,
   then $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$.
   If $f_1(n) = \Omega(g_1(n))$ and $f_2(n) = \Omega(g_2(n))$,
   then $f_1(n) + f_2(n) = \Omega(\min\{g_1(n), g_2(n)\})$.

5. **Product Rule:**
   Given two (positive) functions $g_1(n) > 0$, $g_2(n) > 0$, $\forall\ n \geq n_0$.
   If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$,
   then $f_1(n) * f_2(n) = O(g_1(n)*g_2(n))$.
   If $f_1(n) = \Omega(g_1(n))$ and $f_2(n) = \Omega(g_2(n))$,
   then $f_1(n) * f_2(n) = \Omega(g_1(n)*g_2(n))$.
   If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$,
   then $f_1(n) * f_2(n) = \Theta(g_1(n)*g_2(n))$.

Remark: The sum rule and product rule in (4) & (5) can be extended to k functions, where k is a fixed integer constant.

6. If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n^1 + a_0$, where a_i′s are constants with $a_m > 0$, then $f(n) = \Theta(n^m)$.

Some Useful Function in Complexity Analysis:

f(n)	***Growth Rate***	***Algorithmic Performance***
n^n	Fastest	Worst
$n!$		
•		
•		
•		
3^n		
2^n	↑	↓
•		
•		
•		
$n^k, k \geq 2$		
n^2		
$n\lg n$		
n		
$\lg n$		
c	Slowest	Best

HW: Review logarithmic and exponential functions.

Importance of Efficient Algorithms:

When an algorithm A is used to compute a problem Π with input S, it requires 0.5ms (10^{-3}s) to execute A when |S| = 1,000. If the complexity of the algorithm A is given by the following closed-form expressions, compute the time required to execute the A when |S| = 1,000,000 = 10^6.

(a) $T(n) = 560\log_{10}n$.
(b) $T(n) = 660\log_{10}n$.
(c) $T(n) = 718n\log_{10}n$.
(d) $T(n) = n^2$.
(e) $T(n) = n^3$.
(f) $T(n) = 2^n$.

Solution:

Observe that

$$\frac{T(n)}{C(n)} = \frac{T(n^*)}{C(n^*)},$$

$$C(n^*) = \frac{T(n^*)}{T(n)} * C(n) = \frac{T(n^*)}{T(n)} * 0.5ms.$$

(a) $T(n) = 560\log n.$

$$C(n^*) = \frac{T(n^*)}{T(n)} * 0.5ms = \frac{560\log 10^6}{560\log 10^3} * 0.5ms = \frac{6}{3} * 0.5ms = 1.0ms.$$

(b) $T(n) = 660n.$

$$C(n^*) = \frac{T(n^*)}{T(n)} * 0.5ms = \frac{660 * 10^6}{660 * 10^3} * 0.5ms = 10^3 * 0.5ms = 0.5s.$$

(c) T(n) = 718n log n.

$$C(n^*) = \frac{T(n^*)}{T(n)} * 0.5ms = \frac{718 * 10^6 * \log 10^6}{718 * 10^3 * \log 10^3} * 0.5ms = 10^3 * 2 * 0.5ms = 1s.$$

(d) $T(n) = n^2$.

$$C(n^*) = \frac{T(n^*)}{T(n)} * 0.5ms = \frac{(10^6)^2}{(10^3)^2} * 0.5ms = 10^6 * 0.5ms$$

$$= 10^3 * 0.5s \simeq 8.3 \min.$$

(e) $T(n) = n^3$.

$$C(n^*) = \frac{T(n^*)}{T(n)} * 0.5ms = \frac{(10^6)^3}{(10^3)^3} * 0.5ms = 10^9 * 0.5ms$$

$$= 5 * 10^5 s \simeq 5.79 \text{ days}$$

(f) $T(n) = 2^n$.

$$C(n^*) = \frac{T(n^*)}{T(n)} * 0.5ms$$

$$= \frac{2^{10^6}}{2^{10^3}} * 0.5ms$$

$$= 2^{10^6 - 10^3} * 0.5ms$$

$$= 2^{999000} * 0.5ms.$$

Remark: Since 2^{64} =18,446,744,073,709,551,616,

$C(n^*) = 2^{64} * 0.5ms > 1.07 \times 10^{11}$ days = 293,150,684 years.

Hence, an algorithm with exponential complexity will never be acceptable when n is large.

Q: What if n is “small?”

Example: Given two algorithms A_1 and A_2 with $T_1(n) = 2^{18}n^2$ and $T_2(n) = 2^n$. Find smallest input size $n_0 > 0$ such that algorithm A_1 will perform faster than A_2 for all $n \geq n_0$.

We need to find smallest integer $n > 0$ such that $2^{18}n^2 \leq 2^n$.

Consider

$$\begin{aligned} 2^{18}n^2 &\leq 2^n \\ \lg 2^{18}n^2 &\leq \lg 2^n \\ \lg 2^{18} + \lg n^2 &\leq n \\ 18 + 2\lg n &\leq n \\ 0 &\leq n - 2\lg n - 18 \end{aligned}$$

Take $n = 2^4$, we have $2^4 - 2\lg 2^4 - 18 = -10$
Take $n = 2^5$, we have $2^5 - 2\lg 2^5 - 18 = 4$.

Hence, $2^4 < n < 2^5$.

Q: How do you find the smallest n that will satisfy the above inequality?

Apply binary search to the region $(2^4, 2^5)$.

HW: Find smallest n* such that algorithm A_1 outperforms algorithm A_2 for all $n \geq n^*$.

Given an algorithm A,

Ideal Case:

Compute T(n) in closed-form.

First Approximation:

Compute a function f(n) such that f(n) = Θ(f(n)).

Second Approximation:

Compute a function f(n) such that f(n) = O(f(n)).

Warning: One should never compare the performance of two algorithms using their big-O information.

Example: Consider two algorithms A_1 and A_2 with complexity $T_1(n) = O(n^3)$ and $T_2(n) = O(n^{1000})$. Even though $n^3 = O(n^{1000})$, you can never conclude that algorithm A_1 is more efficient than algorithm A_2 for sufficiently large n.

If $T_1(n) = n^3 = O(n^3)$ and $T_2(n) = n^{1000} = O(n^{1000})$. Clearly algorithm A_1 is more efficient than algorithm A_2 for all $n > 1$. However, if $T_1(n) = n^3 = O(n^3)$ and $T_2(n) = n = O(n^{1000})$, algorithm A_1 is clearly **not** more efficient than algorithm A_2 even when n is large! Hence, one can never compare the performance of two algorithms using their big-O information.

Remark: You can only compare the performance of two algorithms using their closed-form expression (or big-Θ information).

8/20/2018