Algorithm Analysis

CSE 2011
Fall 2009

Introduction

- What is an algorithm?
  - a clearly specified set of simple instructions to be followed to solve a problem
    - Takes a set of values, as input and
    - produces a value, or set of values, as output
  - May be specified
    - In English
    - As a computer program
    - As a pseudo-code
- Data structures
  - Methods of organizing data
- Program = algorithms + data structures

Introduction

- Why need algorithm analysis ?
  - writing a working program is not good enough
  - The program may be inefficient!
  - If the program is run on a large data set, then the running time becomes an issue

3

Example: Selection Problem

- Given a list of N numbers, determine the k^{th} largest, where $k \leq N$.
- Algorithm 1:
  (1) Read N numbers into an array
  (2) Sort the array in decreasing order by some simple algorithm
  (3) Return the element in position k

4

Example: Selection Problem (2)

- Algorithm 2:
  - (1) Read the first *k* elements into an array and sort them in decreasing order
  - (2) Each remaining element is read one by one
    - If smaller than the k^{th} element, then it is ignored
    - Otherwise, it is placed in its correct spot in the array, bumping one element out of the array.
  - (3) The element in the k^{th} position is returned as the answer.

5

Example: Selection Problem (3)

- Which algorithm is better when
  - N = 100 and k = 100?
  - N = 100 and k = 1?
- What happens when N = 1,000,000 and k = 500,000?
- There exist better algorithms

6

Algorithm Analysis

- We only analyze correct algorithms
- An algorithm is correct
  - If, for every input instance, it halts with the correct output
- Incorrect algorithms
  - Might not halt at all on some input instances
  - Might halt with other than the desired answer

7

Algorithm Analysis (2)

- Analyzing an algorithm
  - Predicting the resources that the algorithm requires
  - Resources include
    - Memory
    - Communication bandwidth
    - Computational time (usually most important)

8

Algorithm Analysis (3)

- Factors affecting the running time
  - computer
  - compiler
  - algorithm used
  - input to the algorithm
    - The content of the input affects the running time
    - typically, the input size (number of items in the input) is the main consideration
      - E.g. sorting problem ⇒ the number of items to be sorted
      - E.g. multiply two matrices together ⇒ the total number of elements in the two matrices
- Machine model assumed
  - Instructions are executed one after another, with no concurrent operations ⇒ not parallel computers

9

Worst- / Average- / Best-Case

- Worst-case running time of an algorithm
  - The longest running time for **any** input of size n
  - An upper bound on the running time for any input
    ⇒ guarantee that the algorithm will never take longer
  - Example: Sort a set of numbers in increasing order; and the input is in decreasing order
  - The worst case can occur fairly often
    - E.g. in searching a database for a particular piece of information
- Best-case running time
  - sort a set of numbers in increasing order; and the input is already in increasing order
- Average-case running time
  - May be difficult to define what “average” means

10

Example

- Given an array of integers, return true if the array contains number 100, and false otherwise.
  - Best case: ?
  - Worst case: ?
  - Average case: ?

11

Running Time of Algorithms

- Bounds are for algorithms, rather than programs
  - Programs are just implementations of an algorithm, and almost always the details of the program do not affect the bounds.
- Bounds are for algorithms, rather than problems
  - A problem can be solved with several algorithms, some are more efficient than others.

12

Analysis Model

- It takes exactly one time unit to do any calculation such as addition, multiplication, assignment, comparison, etc.
- There is infinite amount of memory.
- It does not consider the cost associated with page faulting or swapping.
- It does not include I/O costs (which is usually one or more orders of magnitude higher than computation costs).

13

An Example

```
int sum ( int n )
{
     int partialSum;
     partialSum = 0;/* 1 */
     for (int i=0; i <= n-1; i++)  /*2*/
          partialSum += i*i*i; /* 3 */
     return partialSum;  /* 4 */
}
```

14

An Example (cont'd)

- Lines 1 and 4: one unit each
- Line 3: 4N
- Line 2: 1+(N+1)+N=2N+2
- Total: 6N+4 $\Rightarrow$ O(N)

15

Running Time Calculations

- Throw away leading constants
- Throw away low-order terms
- Compute a Big-Oh running time:
  - An upper bound for running time
  - Never underestimate the running time of a program
  - The program may end earlier, but never later (worst-case running time)

16

General Rules for Big-Oh

- *for* loops
  - at most the running time of the statements inside the *for* loop (including tests) times the number of iterations.
- Nested *for* loops

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

  - the running time of the statement multiplied by the product of the sizes of all the *for* loops.
  - $O(N^2)$

17

General Rules for Big-Oh (cont'd)

- Consecutive statements

```
for (i=0;i<n;i++)
    a[i]=0;
for (i=0;i<n;i++)
    for (j=0;j<n;j++)
        a[i] += a[j]+i+j;
```

  - These just add.
  - $O(N) + O(N^2) = O(N^2)$
- **`if C then S1`**
  `else S2`
  - never more than the running time of the test plus the larger of the running times of S1 and S2.

18

Strategies

- Analyze from the inside out.
- If there are method calls, analyze these first.
- Recursive methods (later):
  - Could be just a hidden “for” loop ⇒ simple.
  - Solve a recurrence (more complex)

19

Example: Insertion Sort

1) Initially p = 1
2) Let the first p elements be sorted
3) Insert the (p+1)th element properly in the list so that now p+1 elements are sorted.
4) Increment p and go to step (3)

20

Insertion Sort: Example

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

21

Insertion Sort: Algorithm

```
for (int p=1; p<a.size(); p++)
{
        int tmp=a[p];
        for (j=p; j> 0 && tmp < a[j-1]; j--)
                a[j] = a[j-1];
        a[j] = tmp;
}
```

see applet http://www.cis.upenn.edu/~matuszek/cse121-2003/Applets/Chap03/Insertion/InsertSort.html

- Consists of N - 1 passes
- For pass p = 1 through N - 1, ensures that the elements in positions 0 through p are in sorted order
  - elements in positions 0 through p - 1 are already sorted
  - move the element in position p left until its correct place is found among the first p + 1 elements

22

Example 2

To sort the following numbers in increasing order:

34 8 64 51 32 21

p = 1; tmp = 8;

34 > tmp, so second element a[1] is set to 34: {8, 34}…

We have reached the front of the list. Thus, 1st position a[0] = tmp=8

After 1st pass: 8 34 64 51 32 21

(first 2 elements are sorted)

23

P = 2; tmp = 64;

34 < 64, so stop at 3rd position and set 3rd position = 64

After 2nd pass: 8 34 64 51 32 21

(first 3 elements are sorted)

P = 3; tmp = 51;

51 < 64, so we have 8 34 64 64 32 21,

34 < 51, so stop at 2nd position, set 3rd position = tmp,

After 3rd pass: 8 34 51 64 32 21

(first 4 elements are sorted)

P = 4; tmp = 32,

32 < 64, so 8 34 51 64 64 21,

32 < 51, so 8 34 51 51 64 21,

next 32 < 34, so 8 34 34, 51 64 21,

next 32 > 8, so stop at 1st position and set 2nd position = 32,

After 4th pass: 8 32 34 51 64 21

P = 5; tmp = 21, . . .

After 5th pass: 8 21 32 34 51 64

24

Analysis: Worst-case Running Time

```
for (int p=1; p<a.size(); p++)
{
        int tmp=a[p];
        for (j=p; j> 0 && tmp < a[j-1]; j--)
                a[j] = a[j-1];
        a[j] = tmp;
}
```

- Inner loop is executed p times, for each p=1..N-1
  $\Rightarrow$ Overall: 1 + 2 + 3 + . . . + N-1 = …= $O(N^2)$
- Space requirement is O(?)

25

Analysis

- The bound is tight $\Theta(N^2)$.
- There exist inputs that actually use $\Omega(N^2)$ time.
- Consider a reversed sorted list as input:
  - When a[p] is inserted into the sorted a[0..p-1], we need to compare a[p] with all elements in a[0..p-1] and move each element one position to the right
    $\Rightarrow$ $\Omega(i)$ steps
  - The total number of steps is $\Omega(\Sigma_1^{N-1} i) = \Omega(N(N-1)/2) = \Omega(N^2)$

26

Analysis: Best-case Running Time

- The input is already sorted in increasing order:
  - When inserting a[p] into the sorted a[0..p-1], only need to compare a[p] with a[p-1] and there is no data movement.
  - For each iteration of the outer for-loop, the inner for-loop terminates after checking the loop condition once $\Rightarrow$ O(N) time
- If input is *nearly sorted*, insertion sort runs fast.

27

Next time …

- Growth rates
- O, Ω, Θ, o

28