HST 952

Computing for Biomedical Scientists

Lecture 10

Recap: Searching

- Linear search is most appropriate for searching through a collection of unsorted items
- It is not very efficient, but is easy to program
- Binary search is a more efficient search method than linear search
- It works for arrays/collections that are already sorted (is essentially the strategy that humans use for searching a phone book or dictionary)
  - this strategy is often called a *divide-and-conquer* strategy
- Strategy for searching for a name in one section of a phone book is the same as initial strategy for searching for the name in the entire phone book
- This implies that we can solve the binary search problem using recursion

Recap: Sorting

- Selection sort is one of the easiest sorting methods to understand and code
- Interchanges smallest number in unsorted portion of an array with first location in unsorted portion of array
- It is not the most efficient sorting method
- Merge sort uses a divide-and-conquer strategy for sorting
- More efficient than selection sort

Read

Foundations of Computer Science by Aho and Ullman

- Chapter 2
- Chapter 3

Outline

- Time complexity of algorithms

Time complexity of algorithms

- As can be seen from searching and sorting examples, different algorithms may exist for a particular problem
- In order to choose an algorithm for solving the problem we often need to consider its *performance*
  - how quickly it runs
  - whether it uses computing resources efficiently
- This means that we need to be able to measure and compare the performance of different algorithms for the same problem

Time complexity of algorithms

- When you need to write a program that is used only once on small amounts of data
  - okay to select the easiest to implement algorithm
- When you need to write a program that needs to be reused many times other issues may arise
  - how much times does it take to run it?
  - how much storage space do its variables use?
  - how much network traffic does it generate?
- For large problems the *running time* is what really determines whether a program should be used

Time complexity of algorithms

- To measure the running time of a program, we
  - Select different sets of inputs that it should be tested on (for *benchmarking*). Such inputs may correspond to
    - the easiest case of the problem that needs to be solved
    - the hardest case of the problem that needs to be solved
    - a case that falls between these two extremes
  - Analyze its running time on the set of inputs
    - The "Big-Oh" notation is a measure that is used in estimating this
- We will focus on analysis of running time

Analyzing a program's running time

- First we determine the size of its input
  - for a program that sorts n numbers, n is the size of its input
- We use the function T(n) to represent the number of units of time taken by the program on an input of size n
- Example: a sequential search program on an input of size n has a running time

  $T(n) = c*n$
  - c is a constant (greater than 0)
  - sequential search has a running time that is linearly proportional to the size of its input: it is a *linear time* algorithm

Analyzing a program's running time

- In many cases, the running time of a program depends on a particular type of input, not just the size of the input
  - the running time of a factorial program depends on the particular number whose factorial is being sought because this determines the total number of multiplications that need to be performed
  - the running time of a search program may depend on whether the value being sought occurs in the collection of items to be searched
- In these cases, we define T(n) to be the *worst-case* running time

Analyzing a program's running time

- The *worst-case* running time is the maximum possible running time on any input of size n
- The *average-case* running time is the average running time of a program over all possible inputs of size n
  - this is often a more realistic measure of performance than worst-case running time
  - it is much harder to compute than worst-case time
  - it assumes that each possible input of size n is equally likely (this is often untrue)
- The best-case running time is the minimum possible running time on any input of size n

Analyzing a program's running time

- The worst-case running time is what is most commonly used to measure a program's running time
- To assess the running time, we have to accept the idea that certain programming operations take a fixed amount of time (independent of the input size):
  - arithmetic operations (+, -, *, etc.)
  - logical operations (and, or, not)
  - comparison operations (==, <, >, etc.)
  - array/vector indexing
  - simple assignments (n = 2, etc.)
  - calls to System methods such as println

Analyzing a program's running time

Let's analyze the running time of the factorial program fragment below:

```
findFactorial(n) {
        int factorial = 1;  // set initial value of factorial to 1
        int iterator = 1;    // set initial value of loop iterator to 1
        while (iterator <= n) {
           factorial = factorial * iterator;
           iterator = iterator + 1;
        } // end of while ()
        System.out.println("The factorial is " + factorial);
}
```

Analyzing a program's running time

```
findFactorial(n) {
        int factorial = 1;  // set initial value of factorial to 1
        int iterator = 1;    // set initial value of loop iterator to 1
        while (iterator <= n) {
          factorial = factorial * iterator;
          iterator = iterator + 1;
        } // end of while ()
        System.out.println("The factorial is " + factorial);
}
```

- We perform two variable initializations and two assignments before the while loop
- We check the loop condition n+1 times
- We go into the while loop n times
- We perform two assignments, and two arithmetic operations each time
- We perform one print statement
- The running time, $T(n) = 4 + (n+1) + n*(4) + 1$

  $$T(n) = 5n + 6$$

Analyzing a program's running time

Imagine that for a problem we have a choice of using program 1 which has a running time

$T_1(n) = 40*n + 10$

and program 2 which has a running time of

$T_2(n) = 3*n^2$

Let's examine what this means for different values of n

Analyzing a program's running time

$T_1(n) = 40*n + 10$

$T_2(n) = 3*n^2$

Running times for $T_1(n)$ and $T_2(n)$:

n	$T_1(n)$	$T_2(n)$
1	50	3
2	90	12
…		
10	410	300
…		
13	530	507
14	570	588
15	610	675
…		
20	810	1200
21	850	1323
22	890	1452

Analyzing a program's running time

$T_1(n) = 40*n + 10$

$T_2(n) = 3*n^2$

If program 1 and 2 are two different methods for finding a patient ID within the database of a small practice with 12 patients (i.e., n = 12) which program would you choose?

Would your choice be different if you knew that the practice would expand to include up to 100 patients?

Analyzing a program's running time

- Program 2 has a running time that increases fairly quickly as n gets larger than 12
- Program 1 has a running time that grows much more slowly as n increases
- Even if the speed of the computer hardware on which we are running both programs doubles, $T_1(n)$ remains a better choice than $T_2(n)$ for large n
- For large collections of data such as can be found in electronic medical records, etc. improving hardware speeds is no substitute for improving the efficiency of algorithms that may need to manipulate the data in such collections

Analyzing a program's running time

- The precise running time of a program depends on the particular computer used. *Constant* factors for a particular computer include:
  - the average number of machine language instructions the assembler for that computer produces
  - the average number of machine language instructions the computer executes in one second
- The Big-Oh notation is designed to help us focus on the non-constant portions of the running time
- Instead of saying that the factorial program studied has running time $T(n) = 5n + 6$, we say it takes $O(n)$ time (dropping the 5 and 6 from $5n + 6$)

Analyzing a program’s running time

The Big-Oh notation allows us to

- ignore unknown constants associated with the computer
- make simplifying assumptions about the amount of time used up by an invocation of a simple programming statement

If

- f(n) is a mathematical function on the non-negative integers (i.e., n = 0,1,2,3,4,5,…), and
- T(n) is a function with a non-negative value (possibly corresponding to the running time of some program)

We say that

T(n) is O(f(n)) if T(n) is at most a constant times f(n) for most values of n greater than some base-line n_0

Analyzing a program's running time

Formally:

T(n) is O(f(n)) if there exists a non-negative integer n_0 and a *constant* c > 0 such that

for all integers n >= n_0, T(n) <= c*f(n)

For program 1 in our previous example T(0) = 10, T(1) = 50, and T(n) = 40n + 10 generally. We can say that T(n) is O(n) because for n_0= 10, n >= n_0 and c = 41,

40n + 10 <= 41n

(this is because for n>=10, 40n + 10 <= 40n + n)

Analyzing a program's running time

For program 2 in our previous example $T(0) = 0$, $T(1) = 3$, and $T(n) = 3n^2$ generally. We can say that $T(n)$ is $O(n^2)$ because for $n_0 = 0$, $n >= n_0$ and $c = 3$,

$$3n^2 <= 3n^2$$

Analyzing a program's running time

- Binary search iterative algorithm analysis (board)

Analyzing a program's running time

Common program running times and their names:

Big-Oh	Informal name	Example we've seen
$O(1)$	constant time	
$O(\log n)$	logarithmic time	binary search
$O(n)$	linear time	sequential search
$O(n \log n)$	n log n time	merge sort
$O(n^2)$	quadratic time	selection sort
$O(n^3)$	cubic time	
$O(2^n)$	exponential time	