

COMPUTER PROGRAMMING IN JAVA

COLUMBIA UNIVERSITY HIGH SCHOOL SCIENCE HONORS PROGRAM

2006 Feb 24 Sat

1. Review

Questions about recursion.

Questions about any previous lecture.

Questions about any previous project.

2. Asymptotic Analysis

In computer science, we are often dealing with large datasets. There are many kinds of tasks we would like to do, and many different ways to do it. How do we know if one way is better than another way?

For example, let's say that I want to sort a set of numbers (as we'll see, this is a very common task for which there are many different algorithms). We have at our disposal 2 algorithms, call them A1 and A2 respectively. A1 takes 3 milliseconds to sort a set of 100 integers (call this set S1) on my machine (call it M1), while A2 takes 4 milliseconds to sort 90 numbers (call this set S2) on someone else's machine (call it M2). Does this mean A1 is better?

Not necessarily. There are innumerable factors that could affect the runtime of a single execution of an algorithm. What if machine M2 was under heavy load when I did the testing. What if my test set S1 was already sorted, i.e. all the work was already done, algorithm A1 just need to check it was sorted.

What we are getting at is that we need a formal way to compare the performance of 2 algorithms, while at the same time abstracting away all the details that we don't care about (like machine load), and we want to generalize this to any dataset of size n. This formal measure is called asymptotic (algorithmic) analysis.

3. The Basics

We want to find a mathematical function which describes the amount of work the algorithm performs in terms of the size of the dataset. What is a unit of work? This is actually a highly qualitative question. For now we will just consider “simple” code statements as one unit of work. For example,

```
int i = 0;
i ++;
int j = 1;
boolean b = false;
b = i < j;
```

Each of these statements is one unit of work. Now how about the following?

```
int sum = 0;
for (int i = 0; i < n; i ++ )
{
    sum = sum + i;
}
```

```
int sum = 0;
for (int i = 0; i < n; i ++ )
{
    for (int j = 0; j < 3; j ++ )
    {
        sum = sum + i;
    }
}
```

```
int sum = 0;
for (int i = 0; i < n; i ++ )
{
    for (int j = 0; j < n; j ++ )
    {
        sum = sum + i;
    }
}
```

4. Big-Oh

Given that we now know how to measure the amount of work a block of code performs, we can talk about comparisons.

Remember that we only want a general idea of how fast an algorithm is, because it is infeasible (or impossible) to get the “true” runtime. Also, we want to be able to talk about general datasets of indeterminate size.

For this, we have “Big-Oh notation”. Big-Oh is a way to be “formally imprecise”. All we care about in with Big-Oh is the highest order polynomial. Examples:

```
2n = O(n)
n + 100,000 = O(n)
n^2 - n + 2 = O(n^2)
(1/2)n^2 + 100,000,000n = O(n^2)
```

5. Sequential Search

Given an array of n elements and some item k , I want to find out whether k is in the list or not. What is the simplest way? Start at the beginning of the array and look at each element until I find k , or I know that k is not in the array if I reach the end.

What is the running time of this algorithm? $O(n)$

6. Binary Search

We have the same problem as before with one difference: the list is sorted in ascending order. Can we improve on sequential search?

Sure we can: think of the number guessing game. If we're trying to guess a number from 1 to 100, we always start at 50, because as part of the response we're told if we're too high or too low. We can apply the same idea here: start by looking at the median value m . Is it the value k we're looking for? If not, then is k bigger or smaller than the median value m ? If k is smaller, then it must be in the lower half. If k is bigger, then it must be in the upper half.

Note that binary search is a nice candidate for a recursive implementation.

What is the running time of this algorithm? $O(\log_2 n)$

7. Insertion Sort

The basic idea behind insertion sort is to separate the number into two groups: one that is sorted, and one that is not. At the beginning of the algorithm, the sorted group is empty, and the unsorted group is equal to the original group. For each number in the unsorted group, we want to insert it into the right place in the sorted group.

What is the runtime of this algorithm? $O(n^2)$

8. Selection Sort

Selection sort uses the same idea as insertion sort, but in reverse (kind of). We still have the idea of two groups of data, one sorted and one not, but here instead of inserting each unsorted element into the right place, we select (from unsorted elements) the largest element each time, and append it to the end of the sorted group.

What is the runtime of this algorithm? $O(n^2)$

9. Assignments

- Implement Sequential Search
- Implement Binary Search

A template class file has been provided for you for both of these assignments. The program loads in 10000 integers in the range -1,000,000,000 to 1,000,000,000, in both unsorted and sorted order. You should also add code that prompts the user for a number to search for, and then use both sequential and binary search to look for it. After you've found the number (or determined it's not in the list), print out the execution time.

- Pig

Pig is a two player game played with a single die. The object of the game is to accumulate 100 points. The player to roll first is randomly determined by coin flip. During each turn a player repeatedly rolls the die earning points equal to the face value on each roll until:

- A one is rolled, in which case the turn is over and the player forfeits all points earned on that turn.
- The player holds, takes the points earned so far and hands the dice over to the opponent.

Part 1:

Implement the game of Pig so that 2 human players can play. Try to make the interface nice: print the scores each time, allow the players to make typos, etc.

Part 2:

Modify your implementation so that the computer can play as a player. Try to make it so that you can have "Player vs Player", "Player vs Computer", and "Computer vs Computer". The computer can use a very simple strategy: for example, always stop after 2 dice rolls.

Part 3:

Further modify your implementation so that the Computer can play with a variety of strategies or difficulties, for example "Simple" (the strategy in Part 2), "Intermediate" (something more intelligent), and "Advanced" (really intelligent), or "Risky" (keeps rolling until some high score), "Win Right Away" (try to win right away), and "Slow and Steady" (doesn't push the odds).

Be creative! There are many sites online for you to read about the game, as well as different strategies, just search "pig dice game". Also, feel free to work with a partner on coming up with different strategies.

Credit goes to the Columbia's CS1004 course for this problem (Professor Adam Cannon and Chris Murphy).