CMSC 250: Introduction to Algorithm Analysis

Justin Wyss-Gallifent

April 20, 2023

1 Introduction

The goal of these notes is to see how some of the topics we learned in the course can be used to analyze algorithms.

Primarily we will focus on counting-type arguments, including:

- How many of a certain operation does an algorithm do, depending on its input?
- How long does an algorithm take to run, depending on its input?

In all three of these there are three classic notions:

- The best case. This is typically the "fastest case" or the case that requires the least of a certain operation.
- The worst case. This is typically the "slowest case" or the case which requires the most of a certain operaton.
- The average case. This is perhaps the hardest to understand because the meaning is vague. Average over what? All possible input? All possible real-world input? Mostly likely real-world input?

2 Selecting the First Element in a List

Here is the pseudocode:

```
% Assume A is a list with n elements index 0 through n-1.
first = A[0]
```

This requires exactly one assignment. If an assignment takes 3 seconds then so does the entire pseudocode. Neither of these depends upon the input A.

Easy but worth mentioning!

3 Selecting the Smallest Element in a List

Now consider the process of selecting not the first but the smallest element in a list.

Here is the pseudocode. For simplicity we will assume that the logistics of the loop require no assignments, time, etc., just the code in the body of the loop.

```
% Assume A is a list with n elements index 0 through n-1.
min = A[0]
for i = 1 to n-1
  if A[i] < min:
    min = A[i]
  end
```

```
end
```

Note that we don't need to start the loop at $n = 0$ because $A[0]$ is already assigned as a pre-emptive first minimum.

Question: How many comparisons does this require?

Answer: The loop runs $n - 1$ times and consequently $n - 1$ comparisons will be made.

Question: How many assignments does this require in the best case?

Answer: The best case here will occur when the first element is the minimum, the inequality will fail each time and there will be just the first assignment, so 1.

Question: How many assignments does this require in the worst case?

Answer: The worst case here will be when each successive element is smaller, meaning the list is decreasing, since this will force the inequality to always be satisfied and the body of the loop will always run. Then we will have the assignment before the loop and $n - 1$ more for a total of n.

Question: How many assignments does this require in the average case?

Answer: As we mentioned earlier we cannot answer this without some notion of what "average" means. Here are some ideas for this problem.

(a) Suppose our list will always contain exacctly three distinct elements. Since the functioning of the code doesn't depend on the actual values but the comparisons between them, for the sake of discussion this means we can consider just six inputs:

$$[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]$$

We can analyze each:

- $[1, 2, 3]$ takes 1 assignment.
- $[1, 3, 2]$ takes 1 assignment.
- $[2, 1, 3]$ takes 2 assignments.
- $[2, 3, 1]$ takes 2 assignments.
- $[3, 1, 2]$ takes 2 assignments.
- $[3, 2, 1]$ takes 3 assignments.

On average $(1 + 1 + 2 + 2 + 2 + 3)/6 = 11/6$ assignments.

Observe we can think about this as an expected value problem whereby there is a $1/6$ probability of each input and the expected number of assignments is then $11/6$.

(b) Suppose our list tends to be increasing where "tends to" means that as we go further along the list we are less likely to encounter a smaller element. Let's say that element $A[j]$ has a $1/(2^j)$ probability of being smaller than all previous elements. How many assignments should we expect now?

Let's have a think:

- There is the first assignment, probability 1.
- There is a $1/2^1$ probability that $A[1]$ is smaller than all previous elements and in such a case an assignment will occur.
- There is a $1/2^2$ probability that $A[2]$ is smaller than all previous elements and in such a case an assignment will occur.
- ...
- There is a $1/2^{n-1}$ probability that $A[n-1]$ is smaller than all previous elements and in such a case an assignment will occur.

The expected number of assignments is then:

$$1 + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{2^{n-1}}$$

We can rewrite this:

$$\sum_{j=0}^{n-1} \frac{1}{2^j} = \sum_{j=0}^{n-1} \left(\frac{1}{2}\right)^j = \frac{1 - \left(\frac{1}{2}\right)^n}{1 - \frac{1}{2}} = ...\text{algebra}... = \frac{2^n - 1}{2^{n-1}}$$

For example a list of length 7 will require, on average:

$$\frac{2^7 - 1}{2^6} = \frac{127}{64} \approx 1.98 \text{ assignments}$$

This second example was pretty simple and still demanded a geometric sum formula. We can imagine how hard this could get for more complicated algorithms and data.

Question: Suppose a comparison takes 2 seconds no matter what and assignments takes 1 second. However suppose our list structure is a bit more convoluted than it seems and assigning from values later in the list is more expensive, timewise. It turns out that the line `min = A[i]` takes $1+0.5i$ seconds. Moreover suppose our list tends to be increasing as with the previous calculation. How long should we expect our algorithm to take?

Answer: There are $n-1$ comparisons so that's $2(n-1)$ seconds, that's easy. As for the assignments and accesses:

- The first assignment takes $1 + 0.5(0)$ seconds, probability 1.

- There is a $1/2^1$ probability that $A[1]$ is smaller than all previous elements and in such a case an assignment will occur, taking $1 + 0.5(1)$ seconds.
- There is a $1/2^2$ probability that $A[2]$ is smaller than all previous elements and in such a case an assignment will occur, taking $1 + 0.5(2)$ seconds.
- ...
- There is a $1/2^{n-1}$ probability that $A[n-1]$ is smaller than all previous elements and in such a case an assignment will occur, taking $1+0.5(n-1)$ seconds.

The expected time is then:

$$2(n-1)+\left(\frac{1}{2^0}\right)(1+0.5(0))+\left(\frac{1}{2^1}\right)(1+0.5(1))+\left(\frac{1}{2^2}\right)(1+0.5(2))+...+\left(\frac{1}{2^{n-1}}\right)(1+0.5(n-1))$$

We can rewrite this:

$$2(n-1)+\sum_{j=0}^{n-1}\left(\frac{1}{2^j}\right)(1+0.5j) = 2(n+1)+\sum_{j=0}^{n-1}\left(\frac{1}{2}\right)^j+\frac{1}{2}\sum_{j=0}^{n-1}\frac{j}{2^j}$$

We can simplify the first sum, just like we did earlier. We do not currently, in this course, have a formula for the second sum. We will see one in CMSC351.

4 Selecting the Smallest Element in a Square 2D Array

Here is the pseudocode:

```
% Assume A is a 2D array with nxn elements.
min = A[0,0]
for i = 0 to n-1
  for j = 0 to n-1
    if A[i,j] < min:
      min = A[i,j]
    end
  end
end
```

Note that without making the pseudcode more complicated we are required to compare $A[0,0]$ unnecessarily.

Question: How many comparisons does this require?

Answer: This is easy, it will require n^2 comparisons.

Question: How many assignments does this require in the best case?

Answer: As with the previous code, in the best case just 1.

Question: How many assignments does this require in the worst case?

Answer: In the worst case $1 + n^2$ because there will be the initial assignment followed by $n^2 - 1$ assignments. Note that the comparison `A[0,0] < min` inside the loops will never be satisfied and so inside the loops the assignment will only happen $n^2 - 1$ times.

Question: How many assignments does this require in the average case?

Answer: As is on par with out discussions, this is a loaded question. What does average mean, etc.?

Question: Suppose it takes $i + 2^j$ seconds to run the code `min = A[i,j]`. In the worst case how long will it take the code to run?

Answer: The assignment before the loops will take $0 + 2^0 = 1$ seconds. Including the loops, the time required will be:

$$1 + \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \left(i + 2^j\right)$$

This simplifies:

$$\begin{aligned}
1 + \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \left(i + 2^j\right) &= \sum_{i=0}^{n-1} \left[\sum_{j=0}^{n-1} i + \sum_{j=0}^{n-1} 2^j \right] \\
&= 1 + \sum_{i=0}^{n-1} \left[ni + 2^n - 1\right] \\
&= 1 + n \sum_{i=0}^{n-1} i + \sum_{i=0}^{n-1} \left[2^n - 1\right] \\
&= 1 + n \left(\frac{(n-1)(n)}{2} \right) + n(2^n - 1)
\end{aligned}$$

5 A Fun Algorithm

Consider the following algorithm in the form of a function:

```
def f(n):
  if n = 0:
    return(0)
  else:
    return(factorial(n)+f(n-1))
  end
```

```
end
```

First of all let's plug some numbers in to see how it works.

- If we input $f(0)$ it just returns 0.
- If we input $f(1)$ it returns $1! + f(0) = 1$.
- If we input $f(2)$ it returns $2! + f(1) = 2 + 1 = 3$.
- If we input $f(3)$ it returns $3! + f(2) = 6 + 3 = 9$.
- If we input $f(4)$ it returns $4! + f(3) = 24 + 9 = 33$.
- And so on.

Question: How many additions does this perform?

Answer: The call $f(0)$ performs none. A call to $f(n)$ will perform 1 addition and a call to $f(n-1)$. Therefore a call to $f(n)$ will perform n additions.

Question: How many multiplications does this perform?

Answer: Assume $k!$ perform $k-1$ multiplications because it's $1 \cdot 2 \cdot ... \cdot k$. Then we have:

- $f(0)$ performs 0
- $f(1)$ performs $1! + f(0)$ so 0.
- $f(2)$ performs $2! + f(1)$ so $1 + 0 = 1$.
- $f(3)$ performs $3! + f(2)$ so $2 + 1 = 3$.
- $f(4)$ performs $4! + f(3)$ so $3 + 3 = 6$.
- $f(5)$ performs $5! + f(4)$ so $4 + 6 = 10$.
- And so on.

In general $f(n)$ performs $1 + 2 + ... + (n-1) = (n-1)n/2$ multiplications.

Question:

Suppose each addition takes 2 seconds and each multiplication takes 6 seconds. More over suppose the input is a number from 1 to 100 inclusive with each equally likely. How long should we expect the algorithm to run on average?

Answer:

Denote by p_i the probability of the input being i and by x_i the time required for input i.

Then we have:

$$p_i = \frac{1}{100}$$

And we have:

$$x_i = 2i + 6\left(\frac{(i-1)i}{2}\right) = 3i^2 - i$$

The expected value will then be:

$$\begin{aligned} p_1x_1 + ... + p_{100}x_{100} &= \sum_{i=1}^{100} \frac{1}{100}(3i^2 - i) \\ &= \frac{1}{100}\left[3\sum_{i=1}^{100} i^2 - \sum_{i=1}^{100} i\right] \\ &= \frac{1}{100}\left[3\left(\frac{100(100+1)(2(100)+1)}{6}\right) - \frac{100(100+1)}{2}\right] \\ &= ... \\ &= 10100 \end{aligned}$$

So on average 10100 seconds.

6 Bubble Sort

6.1 Introduction

Bubble Sort comes in a variety of versions. It's not the best sorting algorithm but it's a good introduction because it's easy to understand, especially compared to some others.

6.2 Pseudocode

Here is the pseudocode for one particular version.

```
% Assume A is a list of length n.
didaswap = True
for i = 0 to n-2
  didaswap = False
  for j = 0 to n-i-2
    if A[j] > A[j+1]
      swap A[i] and A[i+1]
      didaswap = True
    end
  end
  if didaswap == False
    break
  end
end
```

Take some time to understand how this algorithm works.

When $i = 0$ it passes through the list, from $j = 0$ to $j = n - 0 - 2 = n - 2$, comparing each $A[j]$ to $A[j+1]$ and swapping if the left one is larger. Effectively this will move the largest element to the end of the list. If any swaps were made, a flag is set which indicates that another pass should be made.

When $i = 1$ we don't need to worry about the last element since it's the largest, so we pass through the list again but ignore that last element, we see we pass from $j = 0$ to $j = n - 1 - 2 = n - 3$. otherwise the idea is the same as $i = 0$ except in this case when we finish the largest two elements are in order at the end of the list.

We continue this until no swaps are made, which indicates the list is in order.

Since, after the ith iteration, the final $i + 1$ elements are in order, we know that when $i = n - 2$, the final $n - 2 + 1 = n - 1$ elements are in order, so they are all in order, and no swaps will occur.

Now, as for analyzing this, let's suppose that the if-statement takes c seconds for some constant c, no matter if true or false. How much times does the code take to run?

6.3 Best Case

If the list is already sorted then we have one pass through for `i=0` with $n-i-1$ iterations and since no swaps occur we exit. The time is then:

$$T(n) = \sum_{j=0}^{n-i-2} c = (n - i - 1)c$$

6.4 Worst Case

If the list is in reverse order then each `i` loop iteration results in $n - i - 1$ comparisons and swaps. To see this observe that if we start with $[5, 4, 3, 2, 1]$ then the `i=0` pass has $n-i-1 = 5-0-1 = 4$ comparisons and swaps and results in $[4, 3, 2, 1, 5]$. The `i=1` pass has $n - i - 1 = 5 - 1 - 1 = 3$ comparisons and swaps and results in $[3, 2, 1, 4, 5]$, and so on. We therefore use all of `i=0,...,n-1` iterations (no `break`) and the time is then:

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-i-2} c \\
&= \sum_{i=0}^{n-2} (n-i-1)c \\
&= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-1} i \\
&= (n-1)(n-1) - \frac{(n-2)(n-1)}{2} \\
&= n^2 - 2n + 1 - \frac{1}{2}n^2 + \frac{3}{2}n - 1 \\
&= \frac{1}{2}n^2 - \frac{1}{2}n
\end{aligned}
$$