NETWORK OPTIMIZATION

Megan Bryant
mrbryant@email.wm.edu

Department of Mathematics, The College of William and Mary, P.O. Box 8795, Williamsburg, VA 23187

Executive Summary

Homework 2.
Collaborators:

2.1 *Compute* 3^{18} *using only* 5 *multiplications. You are allowed to use a calculator only to multiply two numbers, but you are not allowed to use functions such as* x^y *and* e^x*.*

Report the multiplications you performed, and all intermediate answers you obtained.

We can utilize the following exponentiation pseudo code to derive the answer:

expon(a, n)

if n is odd then
    p = expon$(a, \lfloor \frac{n}{2} \rfloor)$
        return $p \times p \times a$
else
    p = expon$(a, \frac{n}{2})$
    return $p \times p$
end if

Now, we can use that pseudocode to solve expon$(3, 18)$. We know that $a = 3$, $n = 18$ and n is even. Therefore, we have the following

$$3^{18} = 3^9 \times 3^9$$

Now, we must call the exponential function again for 3^9 and arrive at the following

$$3^9 = 3^4 \times 3^4 \times 3$$

Again, we must call the exponentiation function and arrive at

$$3^4 = 3^2 \times 3^2$$

Once more, we have

$$3^2 = 3 \times 3$$

Succinctly, we arrive at the following

$$\begin{aligned} 3^{18} &= 3^9 \times 3^9 \\ 3^9 &= 3^4 \times 3^4 \times 3 \\ 3^4 &= 3^2 \times 3^2 \\ 3^2 &= 3 \times 3 \end{aligned}$$

Notice that we have only five multiplications which is a significant reduction over the 18 required if the algorithm was not employed. Therefore, we will be able use our algorithm to determine the desired exponentiation in only 5 steps.

$$\begin{aligned} 3^2 &= 3 \times 3 = 9 \\ 3^4 &= 9 \times 9 = 81 \\ 3^9 &= 81 \times 81 \times 3 = 19683 \\ 3^{18} &= 19683 \times 19683 = \mathbf{387420489} \end{aligned}$$

2.2 *Suppose you have algorithms with the six running times listed below. (Assume these are the exact number of operations performed as a function of the input size* n*).*

Suppose you have a computer that can perform 10^{10} operations per second, and you need to compute a result in at most one hour of computation.

For each of the algorithms, what is the largest input size n for which you would be able to get the result in one hour?

Let us first compute the number of operations that our computer can perform in an hour.

$$10^{10}/\text{sec} \times 60 \text{ sec/min } \times 60\text{min/hour} = 3.6 \times 10^{13}$$

Therefore, we know that we can perform a total of 3.6×10^{13} computations per hour. Now, to determine what the largest input size n for which we can solve an algorithm with the following running times in one hour, we must solve relationship between running time and computations per hour.

Let $n_i, i \in \{1, \ldots, 6\}$ represent the maximum input size.

1.) n^2
$n_1^2 = 3.6 \times 10^{13} = \sqrt{3.6 \times 10^{13}}$
$\implies n_1 = 6 \times 10^6$

2.) n^3
$n_2^3 = 3.6 \times 10^{13} = \lfloor \sqrt[3]{3.6 \times 10^{13}} \rfloor$
$\implies n_2 = 33019$

3.) $100n^2$
$100n_3^2 = 3.6 \times 10^{13}$
$n_3^2 = 3.6 \times 10^{11}$
$\implies n_3 = 6 \times 10^5$

4.) $\log_2 n$
$\log_2 n_4 = 3.6 \times 10^{13}$
$2^{3.6 \times 10^{13}} = n_4$
$\implies n_4 = \infty$ as $2^{3.6 \times 10^{13}}$
This is larger than any feasible input size.

5.) 2^n
$2^{n_5} = 3.6 \times 10^{13}$
$\log(2^{n_5}) = \log(3.6 \times 10^{13})$
$n_5 \log(2) = \log(3.6 \times 10^{13})$
$n_5 = \lfloor \frac{\log(3.6 \times 10^{13})}{\log(2)} \rfloor$
$\implies n_5 = 45$

6.) 2^{2^n}
$2^{2^{n_6}} = 3.6 \times 10^{13}$
$\log(2^{2^{n_6}}) = \log(3.6 \times 10^{13})$
$2^{n_6} \log(2) = \log(3.6 \times 10^{13})$
$2^{n_6} = \frac{\log(3.6 \times 10^{13})}{\log(2)}$
$\log(2^{n_6}) = \log\left(\frac{\log(3.6 \times 10^{13})}{\log(2)}\right)$
$n_6 \log(2) = \log\left(\frac{\log(3.6 \times 10^{13})}{\log(2)}\right)$
$n_6 = \lfloor \frac{\log(\frac{\log(3.6 \times 10^{13})}{\log(2)})}{\log(2)} \rfloor = \lfloor 5.49 \rfloor$
$\implies n_6 = 5$

Note: we are making the assumption that the input sizes are discrete and have thus take the floor of any non-integer. We see that polynomial time algorithms are able to accommodate very large input sizes whereas exponential time algorithms have difficulty handling even relatively small input sizes.

Our answers are summarized below for readability.

Runtime	Input Size
n^2	6×10^6
n^3	33019
$100n^2$	6×10^5
$\log_2 n$	∞
2^n	45
2^{2^n}	5

2.3 *Consider the following basic problem. You're given an array A consisting of n integers $A[1], A[2], \ldots, A[n]$.*

You'd like to output an $n \times n$ matrix B in which $B[i, j]$ (for $i < j$) contains the sum of array entries $A[i]$ through $A[j]$; that is, the sum $A[i] + A[i + 1] + \ldots + A[j]$. (The value of entry $B[i, j]$ is left unspecified whenever $i \geq j$.)

Here is a simple algorithm to solve this problem:

ConstructB

For $i = 1$ to n
    For $j = i + 1$ to n
        $B[i, j] = A[i] + A[i + 1] + \ldots + A[j]$
    end for
end for

1.) *Find an exact expression for the number of additions done by the algorithm for an input array consisting of n integers.*

We want to find an exact expression for the number of additions done by the algorithm. We know that we require one operation in order to add two numbers together. Given the parameters of our algorithm, we know that j must be in the range of $i + 1$ to n. Therefore, let us begin by enumerating a few instances of j and evaluting the number of operations the addition action requires.

$j =$	# Operations = j - i
$i + 1$	1
$i + 2$	2
$i + 3$	3
.	.
.	.
.	.
n	$n - i$

Therefore, we have a general idea of how many operations to expect for each j. Now, let's consider how many operations to expect for each i, with the maximum number of operations as $n - i$. Therefore, in each iteration of the inner 'For' loop, the number of additions made can be denoted as follows

$$\sum_{k=1}^{n-i} k = \frac{(n-i)((n-i)+1)}{2}.$$

However, we are concerned with how many additions are done by the algorithm, not the number done in a single inner loop. Therefore, we must consider how many times the inner loop runs to determine how many times the addition operation is executed.

Now we must determine how many times the inner loop is executed. We know that the maximum length of the inner loop is $n - 1$ since the inner loop states 'For $j = i+1$ to n' and i ranges from 1 to n but is initialized to 1. Therefore, with $j = i + 1$ ranging from2 to n, the longest that the inner loop may be is $n - 1$. This means that total number of times the inner loop is executed is

$$\sum_{i=1}^{n-1} i = \frac{(n-1)((n-1)+1)}{2} = \frac{n(n-1)}{2}.$$

Finally, we can consider the number of times the outer loop is executed, which is simply n since the loop runs from $i = 1$ to n.

Now that we have determined how many times each loop is run, we know that the number of additions can be exactly expressed by multiplying the number of additions in each inner loop by the number of inner loops for each outer loop and finally then number of outer loops. This leaves us with the following computation:

$$\frac{(n-i)((n-i)+1)}{2} \times \frac{n(n-1)}{2} \times n = \frac{1}{4}n^2(n-1)(n-i)(n-i+1)$$

This is an exact expression for the number of times which the addition operation is executed.

2.) *Find the right function $f(n)$ and show that your answer in (a) is $\Theta(f(n))$.*

We know that in order for the algorithm to be $\Theta(f(n))$ it must be both $O(f(n))$ and $\Omega(f(n))$.

Let us recall the following theorem regarding $O(f(n))$running time.

Theorem 1. *An algorithm is said to run in $O(f(n))$ time if for some numbers c and n_0, the time taken by the algorithm is at most $cf(n)$ for all $n \geq n_0$.*

We know that the exterior 'for' loop ('For $i = 1$ to n') runs a total of n times and is $O(n)$.

We also know that the interior 'for' loop ('For $j = i + 1$ to n') runs at most n times per loop and is $O(n)$.

Finally the inner most operation, the addition, adds at most n items and is $O(n)$.

Therefore, we conclude that the algorithm is $O(n^3)$.

Now, we must determine if the algorithm is also $\Omega(n^3)$.

We are given the following theorem regarding $\Omega(f(n))$, a lower bound on the running time.

Theorem 2. *An algorithm is said to be $\Omega(f(n))$ if for some numbers c' and n_0 and all $n \geq n_0$, the algorithm takes at least $c'f(n)$ time on some problem instance.*

Thus, we are searching for a value c' that satisfies the theorem. Let us examine each aspect of the algorithm carefully as before.

We should first note that the outer loop will run n times. That is already $\Omega(n)$ (and $\Theta(n)$).

We now must determine a lower bound on how many times the inner loop will run. We have done this in part (1.) by showing that the inner loop must have

$$\frac{(n-1)((n-1)+1)}{2} = \frac{n(n-1)}{2}$$

total iterations and is thus the aglorithm is at least $\Omega(n^2)$.

Now we must determine a lower bound on the number of operations of the addition action. We know from part (1.) that the number of additions per inner loop is

$$\frac{(n-i)((n-i)+1)}{2}.$$

Remember, we want to determine a lower bound for the running time, and thus a lower bound for the number of additions that must be made. This can be done be evaluating the summation at the midpoint, $i = \frac{n}{2}$.

$$\begin{aligned}\sum_{k=1}^{n-\frac{n}{2}} k &= \frac{(n-\frac{n}{2})(n-\frac{n}{2}+1)}{2} \\ &= \frac{\frac{n(n+2)}{4}}{2} = \frac{n^2+2n}{8} \\ &= \frac{n^2}{8} + \frac{n}{4}\end{aligned}$$

This means that there must be at least $\frac{n^2}{8}$ additions for the $\frac{n}{2}$ iterations. Therefore, the number of additions is bounded on the low end by

$$\frac{n^2}{8} \times \frac{n^2}{2} = \frac{n^3}{16}$$

.

Therefore, the algorithm is bounded by $cf(n) = \frac{n^3}{16}$ which is Ωn^3.

Since the algorithm is both $O(n^3)$ and Ωn^3 it is $\Theta(n^3)$.

3.) *The algorithm given is very natural, but it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with a better asymptotic running time (you should be able to lose a factor n compared to your answer in (a)).*

In order for our new algorithm to have a better asymptotic running time, we know that the algorithm must run in less than $O(n^3)$ time. Given the hint that we should be able to lose a factor of n, this leads us to believe that there exists an algorithm that runs in $O(n^2)$ time.

For this to be true, the algorithm would have to lose one of the three steps that run in $O(n)$ time. We know that in order to create a $n \times n$ matrix we must retain both 'for' loops. Therefore, the only aspect of the algorithm that is easily eligible for improvement is the inner additive loop.

Recall that in our earlier problem 2.1, we were able to reduce the number of steps needed

to compute an exponentiation by carefully considering our methodology. We shall apply a similar paradigm here.

We will introduce the parameter S into our pseudo code which will stand for the current value of the summation. This will be initialized at 0 to represent an empty sum. It will then be set to $A[i]$ in the initial 'for' loop, beginning of course with $A[1]$. Then, each step of the interior 'for' loop need only make one addition that is iterative in nature; S will be reinitialized to $S + A[i]$.

ConstructB Refined

```
S = 0 For i = 1 to n
        S = A[i]
        For j = i + 1 to n
              S = S + A[j]
              B[i, j] = S
        end for
end for
```

Now we must compare the performance of our refined algorithm with the original. We have retained both the interior and outer 'for' loops, which both run in $O(n)$ time. However, the addition in the inner for loop now runs in $O(1)$ time since it is only one addition and one assignment. Therefore, we have eliminated an order of n and the algorithm runs in $O(n^2)$.