CS 5084 - Assignment 2

Basics of Algorithm Analysis

Ehu Shubham Shaw

February 10, 2025

1. (30 points) Suppose you have the algorithms with the five running times listed below, How much slower do each of these algorithms get when you do the following:

(a) (15 points) Impact of Double the input size.

For this investigation, we change n with 2n in each running time function and determine how much slower the algorithm becomes.

Algorithm (i) - n^2 (original: n^2).
Modified: (2n)^2 = 4n^2.
Result: four times slower.

Algorithm (ii) - n^3 (original: n^3).
Modified: (2n).^3 = 8n^3
Result: eight times slower.

Algorithm (iii) - 100n^2 (original: 100n^2)
Modified: 100 (2n)^2 = 400n^2
Result: four times slower.

Algorithm (iv) - nlogn (original: nlogn).
Modified: 2n log(2n) = 2n (log n + log 2) = 2n log n + 2n log 2.
The result is 2(1 + log 2/log n) times slower.

so as n grows larger, this approaches twice as slow.
Algorithm (v) – 2^n (Original: 2^n).
Modified: 2^2n = (2^n)^2.
Result: 2^n times slower.

(b) (15 points) Increase the input size by 1.

For this investigation, we change n with (n+1) in each running time function and determine how much slower the algorithm becomes.

Algorithm (i) - n^2 (original: n^2).
Modified: (n+1).^2 = n^2 + 2n + 1
Result: 1 + (2n+1)/n^2 times slower.
so ass n grows large, this approaches one time slower.

Algorithm (ii) - n^3 (original: n^3).
Modified: (n+1).^3 = n^3 + 3n^2 + 3n + 1
The result is 1 + (3n^2 + 3n + 1)/n^3 times slower.
So an grows large, this approaches one time slower.

Algorithm (iii) - 100n^2 (original: 100n^2)
Modified: 100 (n+1)^2 = 100(n^2 + 2n + 1)
Result: 1 + (200n + 100) / 100n^2 times slower.
So an n grows large, this approaches one time slower.
Algorithm (IV): nlogn

Original: nlogn.
Modified: (n+1) log(n+1)
The result is ((n+1)log(n+1))/(nlogn) times slower.
So an n grows large, this approaches one time slower.

Algorithm (v) – 2^n
Original: 2^n.
Modified: 2^n+1= 2 x 2^n.
Result: two times slower.

2. (20 points) Take the following list of functions and arrange them in ascending order of growth rate. That is, if function g(n) immediately follows function f(n) in your list, then it should be the case that f(n) is O(g(n))?

Let's arrange these functions in ascending order of growth rate.
$F_3(n)$ = n + 10 (slowest growing)
This is the fundamental linear function involving a constant, It increases slower than functions with linear coefficients.
$f_4(n)$ = 10n
Linear function with coefficient ten, grows quicker than n + 10, but slower than 100n.
$f_5(n)$ = 100n
A linear function with coefficient 100, Grows faster than other linear functions but slower than n^2 × log(n).
$f_6(n)$ = n^2 × log(n)
Product of quadratic and logarithmic functions, It grows faster than any linear function, but slower than polynomial functions.
$f_1(n)$ = n^2.5
The Polynomial function, It grows faster than n^2 × log(n), but slower than exponentials.
$F_2(n)$ = root(2^n) (fastest increasing)
Exponential function with square root, any exponential function will eventually grow faster than any polynomial the order is right since each function f(n) is O(g(n)), and g(n) is the next function in the list. We can confirm this by observing that:

- n + 10 is O(10n)
- 10n is O(100n)
- 100n is O(n^2 × log(n))
- n^2 × log(n) is O(n^2.5)
- n^2.5 is O(root(2^n))

As a result, this indicates the appropriate ascending order of growth rates for the supplied functions.

3- (30 points) Analysis of Array Sum Algorithm.

(3.a) - Running Time Upper Bound :

To study the running time of the provided algorithm, let's look at its structure.

The approach utilizes two nested loops and a summing operation
For i = 1, 2,..., n
    For j=i+1, i+2,...,n.
        Sum A[i] through A[j].

In the outer loop i, we iterate n times.
For each i, the inner loop j extends from i+1 to n.
Within these loops, we total the elements from i to j.
This establishes three tiers of operation:
    Outer loop: with n iterations.
    Inner loop: for each i, about (n-i) iterations
    Summery: for each (i, j) pair, (j-i+1) additions
In the worst-case scenario, the method executes O(n^3) operations, hence f(n) = n^3 provides an O(f(n)) constraint.

(3.b) Lower Bound Proof:

To prove ohm(n^3), we can show that the algorithm must do at least cn^3 operations for some constant c.
Consider when i varies between 1 and n/2:
- For each such i, j ranges between i and n.
- For each (i,j) pair, we execute (j-i+1) additions
Even if we only count operations in which:
i goes from 1 to n/2, while j goes from n/2 to n.
We add the (j-i+1) elements.
This allows for at least (n/2)×(n/2)×(n/4) = n^3/16 operations.
being a result, the running time is ohm(n^3), with O(n^3) being the tight bound.

(3.c) Improved Algorithm.

A more efficient solution can be achieved using prefix sums:

// prefix sums

Initialize P[0] = 0

For i = 1 to n:

    P[i] = P[i-1] + A[I]

// array B

For i = 1 to n:

    For j = i+1 to n:

        B[i,j] = P[j] - P[i-1]

Time Complexity Analysis:

Phase 1 (Prefix Sum Computation) involves O(n) operations, single loop through array A each loop executes constant-time operations.
Phase 2 (Array B Computation) involves $O(n^2)$ operations, two nested loops: $O(n^2)$ iterations each B[i,j] computation is O(1) using prefix sums. Total time complexity is $O(n) + O(n^2)$.
Proof for Asymptotic Improvement:
Consider $g(n) = n^2$ (new algorithm) and $f(n) = n^3$ (old algorithm).
As n approaches infinity, the limit of g(n)/f(n) equals the limit of n^2/n^3. $1/n = 0$
This limit shows that g(n) grows asymptotically slower than f(n), indicating that our new approach outperforms the previous $O(n^3)$ solution now space complexity $O(n^2)$, same as the original approach.
A single O(n) array for prefix sums, A single $O(n^2)$ array for output B. This enhancement reduces time complexity from $O(n^3)$ to $O(n^2)$ while preserving the same space complexity, resulting in much higher efficiency for larger inputs.

4. (20 points) Glass Jar Stress Testing Analysis

(4.a) Two-Jar Strategy.

So to find jars With k=2, develop a strategy to find the highest safe rung requiring fewer than linear drops. Let's think about how we can use our two jars strategically rather than examining each rung sequentially or using pure binary search, we can take a hybrid approach like divide the rungs into root(n) parts. Now test the first jar at root(n) intervals, such as root(n), 2root(n), 3root(n), and so on. When the first jar breaks at position i root(n), we know the maximum safe rung is in the interval (i-1)root(n) + 1, i root(n).
Using the second jar, test each rung in this interval successively. For a mathematics Analysis
Total drips = root(n) (first jar) + root(n) (worst scenario for second jar) = 2root(n).
To show that f(n) = 2root(n) grows slower than linearly:
lim(n—> infinite) 2root(n/n) equals lim(n—> infinite). 2/root(n) = 0
Thus, f(n) = 2root(n) meets our criteria for sub-linear growth.

(4.b) k-Jar Strategy.

For k>2 jars, develop a strategy where each fk(n) grows asymptotically slower than fk-1(n) we could approach this as we may generalize our technique by employing the following strategy For K Jars Divide the range into segments with size n^(1/k). Use the first jar to test at places n^(1/k), 2n^(1/k), 3n^(1/k). When it breaks, recursively apply the remaining k-1 jars to the smaller interval. Then we are left with $f_1(n)$ equals n (linear search) $f_2(n)$ = 2root(n), as demonstrated in Part 1. $f_3(n) = 3n^{(1/3)}$ $f_4(n) = 4n^{(1/4)}$ And so on…. We can do this mathematical Proof To prove fk(n)/fk-1(n) —> 0:

lim(n—>infinite) kn^(1/k)/(k-1)n^(1/(k-1))

= lim(n—>infinite) k/(k-1) * n^(1/k - 1/(k-1))

= 0 (since 1/k - 1/(k-1) < 0).

The analysis shows that raising our jar budget k allows us to develop increasingly efficient jar-dropping tactics. Each method fk(n) develops asymptotically slower than the preceding one, resulting in higher efficiency for bigger values of n.