Analysis

June 11, 2020

1 Ram Model of Computation

RAM Model: - Each simple operation $ (_, *, -, =, if, call) $ takes exactly one time step

- Loops and subroutines are not considered simple operation. They are the composition of many single-step operation
- Each memory acess takes exactly one time step. Secondly we have as much memory as we need
- Note this algorithm is not perfect as multiplication takes more time then addition, howerver its as close to exelent as we can get

2 Best, Worst and Average Complexity

Worst-Case Complexity: > the worst-case complexity of the algoithm is the function defined by the maximum number of steps taken in any instance of size $ n $. This represents the curve passing through the highest point in each column

Best-Case Complexity: > The best-case complexity of the algorithm is the function defined by the minimum number of steps taken in any instance of size $ n $. This represents the curve passing through the lowest point of each column

Average-Case Complexity: > The average-case complexity of the algorithm which is the function defined by the average number of steps over all instances of size n

3 Big Oh Notation

The Big-Oh notation ignores the difference between multiplicative constanct. the function $ f(n) = 2n $ and $ g(n) = n $ are identical in Big Oh analysis.

Suppose an algorithm ran twice as fast in $ C $ then it did in $ Python $. The $ 2 $ is not important because it does not scale

3.0.1 Big-O Definition

- $ f(n) = O(g(n)) $ means $ c * g(n) $ is an *upper bound* on $ f(n) $. Thus there exists some constant $ c $ such that $ f(n) $ is always $ <= c*g(n) $ for larger enough $ n $ (i.e, $ n >= n_0 $ for some constant $ n_0 $
- $ f(n) = Ω(g(n))$ means $ c * g(n) $ is a *lower bound* on $ f(n) $. Thus there exists some constant $ c $ such that $ f(n) $ is always $ >= c*g(n) $ for all $ n >= n_0 $

- $ f(n) = Θ(g(n)) $ means $ c_1 * g(n) $ is an upper bound on $ f(n) $ and $ c_2 * g(n) $ is a lower bound on $ f(n) $, for all $ n >= n_0$. Thus there exist constants $ c_1 $ and $ c_2 $ such that $ f(n) <= c_1 * g(n) $ and $ f(n) >= c_2 * g(n) $. This means that $ g(n) $ provides a nice, tight bound on $ f(n) $

The big O notation and worst-case analysis are tools that greatly simplify out ability to compare the effciency of algorithms

Figure 2.3: Illustrating the big (a) $ O $, (b) $ Ω $ and (c) $ Θ $ notation

4 Growth Rates and Dominance Relations

Figure 2.4: Growth rates of common functions measured in nanoseconds

Key Takeaways from Table: - All such algorithms take roughly the same time for $ n = 10 $ - Any algorithm with $ n! $ running times becomes usless for $ n 20 $ - Algorithms whos running time is $ 2^n $ have a greater operating range, but become impractical for $ n > 40 $ - Quadratic-time algorithms whos running time is $ n^2 $ remains usable to about $ n = 10,000 $ but quickly deteriorate with larger inputs. They are likely to be hopeless for $ n > 1,000,000 $ - Linear-time and $ n lg n $ algorithms remain pratical on imputs of one billion items - An $ O(lg n) $ algorithm hardly breaks a serat for any imaginable value for $ n $

4.1 Dominance Relations

In many functions, the highest power will dominate. This is simmilar to taking the limit of something

Constant Functions: > $ f(n) = 1 $: Such functions might measure the cost of adding two numbers, printing out a string or the growth realized by functions such as $ f(n) = min(n, 100) $. In the big picture there is no dependence on the parameter $ n $

Log Functions: > $ f(n) = log(n)$: Log time-complexity shows up in algorithms such as binary search. Such functions grow quite slowly as $ n $ gets big, but faster then the constant function (which is standing still, after all)

Linear Functions: > $ f(n) = n$: Such functions mesure the cost of looking at each item onece (or twie, or ten times) in an $ n$-element array, say to identify the biggest item, the smallest item, or compute the average value

Superlinear Functions: > $ f(n) = n lg n $: This important functions arises in such algorithms as Quicksort and mergesort. They grow just a little faster than linear, just enough to be a different dominance class

Quadratic Functions: > $ f(n) = n^ 2 $: such functions measure the cost of looking at most or all *pairs* of tiems in an n-element universe. This arises in algorithms such as insertion sort and selection sort

Cubic Functions: > $ f(n) = n^3 $: Such functions enumerate through all *tiples* of items in an n-element universe. there also arise in certain dynamic programming algorithms

Exponential Functions: > $ f(n) = c^n $ for a given constant $ c > 1 $: Functions like $ 2^n $ arise when enumerating all subsets of $ n $ items. As we have seen, exponential algorithms become useless fast

Factorial Function: > $ f(n) = n! $: Functions like $ n! $ arise when generating all permutations or ordering $ n $ items

$$n! > 2^n > n^3 > n^2 > nlog(n) > n > log(n) > 1$$

5 Working with Big-O

5.1 Adding Functions

The sum of two functions is governed by the dominant one, namely:

$$O(f(n)) + O(g(n)) \rightarrow O(max(f(n), g(n)))$$

$$(f(n)) + (g(n)) \rightarrow (max(f(n), g(n)))$$

$$(f(n)) + (g(n)) \rightarrow (max(f(n), g(n)))$$

The expressions tell us that it is useful to simplify terms because everything else does not matter except the dominant terms

5.1.1 Example

$$f(n) = O(n^2)$$

$$g(n) = o(n^2)$$

$$f(n) + g(n) = O(n^2)$$

5.2 Multiplying Functions

Multiplication is repeated addition. Consider multiplication by any constant c > 0, be it 1 or 1 million. Multiplying a function by a constant can not affect its asymptotic behavioir

$$O(c * f(n)) \rightarrow O(f(n))$$

$$(c * f(n)) \rightarrow (f(n))$$

$$(c * f(n)) \rightarrow (f(n))$$

On the other hand, when two functions in a product are increasing, both are important. The function $ O (n! logn)$ dominates $ n! $ just as much as $ log n $ dominates 1

$$O(f(n)) * O(g(n)) \rightarrow O(f(n) * g(n))$$

$$(f(n)) * (g(n)) \rightarrow O(f(n) * g(n))$$

$$(f(n)) * (g(n)) \rightarrow (f(n) * g(n))$$

5.2.1 Big-O relationships are transitive

If $ f(n) = O(g(n)) $ and $ g(n) = O(h(n))$, the $ f(n) = O(h(n)) $

6 Reasoning About Efficiency

6.1 Selection Sort

Here we will analyze the selection sort algorithm, which repeatedly identifies the smallest remaining unsorted element and puts at the end of the sorted portion of the array

Figure 2.5: Animation of selection sort in action

```
selection_sort(int s[], int n)
{
    int i,j; /* counters */
    int min; /* index of minimum */
        for (i=0; i<n; i++) {
            min=i;
            for (j=i+1; j<n; j++)
                if (s[j] < s[min]) min=j;
            swap(&s[i],&s[min]);
        }
}
```

The outer loop goes around $ n $ times. The nested inner loop goes aroung $ n - i - 1 $ times, where $ i $ is the index of the outer loop. The exact number of times the $ if $ statement is executed is given by:

What the sum is doing is adding up the integers in decreasing order starting from $ n - 1$ i.e

$$S(n) = (n-1) + (n-2) + (n-3) + ... + 2 + 1$$

But with the Big-O, we are only intersted in the *order* of the expression. One way to think about it is that we are adding up $ n - 1 $ temrs, whose average value is about $ n / 2 $. This yeilds

$$S(n) = n(n-1)/2$$

Another way to think about it is in terms of upper and lower bounds. We have $ n $ terms at most, each of which is at most $ n - 1 $. Thus $ S(n) <= n(n -1) = O(n^2)$. We have $ n/2 $ temrs each that are bigger then $ n /2 $. Thus $ S(n) => (n/2) * (n/2) = \Omega(n^2) $. This tells us that the running time is $ $\Theta(n^2)$, meaning that selection sort is quadratic

6.2 Insertion Sort

A basic rule of thumb in Big-O analysis is that worst-case running time follows from multiplying the largest number of times each nested loop can iteratie. Considet the insertion sort algorithm whos inner loops are shown

```
for (i=1; i<n; i++) {
     j=i;
     while ((j>0) && (s[j] < s[j-1])) {
         swap(&s[j],&s[j-1]);
         j = j-1;
     }
```

```
}
```

How often does the inner while loop iterate? Its trickey because there are different stopping conditions: one to prevent us from running off the bounds of the array $ (j > 0)$ and the other ot mark when the element finds its propper place in sorted order $ (s[j] < s[j-1])$. Since worst-case analysis seeks an upper bound on the running time, we ignore the early termination and assume that this loop always goes around $ i $ times.

Infact we can assume it always goes around $ n $ times since $ i < n $. Since the outer loop goes around $ n $ times, insertion sort must be a quadratic-time algorithm $ O (n^2) $

6.3 Sting Pattern Matching

Pattern matching is the most fundamental algorithmic operation on text strings. This algorithm implements the find command available in any web browser or text editor:

Problem: Substring Pattern Matching

Input: A string $ t $ and $ a $ apttern string $ p $

output: Does $ t $ contain the pattern $ p $ as a substring, and if so where?

Figure 2.6: Searching for the substring *abba* in the text *aababba*

```
int findmatch(char *p, char *t)
{
    int i,j; /* counters */
    int m, n; /* string lengths */
    m = strlen(p);
    n = strlen(t);
    for (i=0; i<=(n-m); i=i+1) {
        j=0;
        while ((j<m) && (t[i+j]==p[j]))
            j = j+1;
        if (j == m) return(i);
    }
    return(-1);
```

The inner while loop goes around at most $ m $ times and potentially for less when the patern match fails. This plus two other statements, lie within the outter $ for $ loop. The outer loop goes around at most $ n - m $ times. Since no complete alignment is possible once we get too far to the right of the text

The time complexity of nested loops multiplie, so this gives a worst-case running time of $ O ((n - m)(m + 2) $

We still have the problem where we do not know the runtime of `strlen`. SO we must guess, and we update out complexity $ O(n + m + (n-m)(m + 2))$

If we multiply out the complexity and reduce it, we get $ O (n + m + nm - m^2)$.

If we make multiple other observations, such as knowing that $ n > m $ because its impossible to have $ p $ as a substring of $ t $ for any pattern lower than the text itsef, we can get the final big-o to $ O(n +nm - m^2) $

Finally, observe that the $ -m^2 $ temr is negative and thus only serves to lowet the value. Since Big-O gives an upper bound, we can drop any negative terms without invalidating the upper rbound

So finally, we are left with $ O(nm) $.

6.4 Matrix Multiplication

Nested summations often arise in the analysis of algorithms with nested loops. Consider the problem of matrix multiplication

Problem: Matrix Multiplication

Input: Two matrices $ A $ (of dimension $ x * y $) and $ B $ (dimension $(y * z)$

Output: An $ x * y $ matrix $ C $ where $ C[i][j] $ is the dot product of the $ ith $ row of $ A $ and the $ jth $ column of $ B $

The elemntary algo for matrix multiplication is implemented as a tight product of three nested loops:

```
for (i=1; i<=x; i++)
    for (j=1; j<=y; j++) {
        C[i][j] = 0;
        for (k=1; k<=z; k++)
            C[i][j] += A[i][k] * B[k][j];
        }
```

The number of multiplications $ M(x, y, z) $ is given by the following summation

Sums get evaluated from the right inward. The sum of $ z $ is ones is $ z $ so:

The sum of $ y $ $ zs $is just as simple, $ yz $ so:

Finally, the sum of $ x $ $ yzs $ is $ xyz $

Thus the running of this matrix multiplication algorithm $ O(xyz) $. If we consider the common case where all three dimensions are the same, this becomes $ O (n^3)$, i.e.., a cubic algorithm

7 Logarithms and Their Applications

A logarithm is simply an inverse exponential function

Saying $ b^x = y $ is the equvalent to saying that $ x = log_b(y) $

Further, this definition is the same as saying $ b^{log_b(y)} = y$

7.1 Logs and Binary Search

Binary search is a good example of an $ O(log n) $ algorithm. The number of steps the algorithm take equals the number of times we can halve $ n $ untill only one name is left

Bt definition, this is exactly $ log_2(n) $

7.2 Logarithms and Trees

Figure 2.7: A height $ h $ tree with $ d $ children per node as $ d^h $ leaves. Here $ h = 2 $ and $ d = 3 $

A binary tree of height $ 1 $ can have up to $ 2 $ leaf nodes, while a tree of height $ 2 $ can have up to four leaves. What is the height of $ h $ of a rooted binary tree with $ n $ leaf nodes?

Note the number of leaves doubles every time we increase the height by one.

To account for $ n $ leaves, $ n = 2^h $ which implies that $ h = log_2(n) $

What if we generalize to trees that have $ d $ children, where $ d= 2 $ for the case of binary trees? A tree of height $ 1 $ can have up to $ d $ leaf nodes, while one of height $ 2 $ can have up to $ d^2 $ leaves.

The number of possible eleaves multiplies by $ d $ every time we increase the height by one, so to account for $ n $ leaves, $ n = d^h $, which implies that $ h = log_d(n) $

Takeaway is that short trees can have very many leaves and thus is the main reason binary trees prove fundamental to the design of fast data structures

7.3 Logarithms and Bits

If there are two bit patterns of lenght $ 1 (01) $ and four of lenght $ 2 (00, 01, 10, 11) $. How many bits $ w $ do we need to represent any of one of $ n $ different possibilities, be it one of $ n $ items or the integers from $ 1 $ to $ n $?

The key observation is that there mist be at least $ n $ different bit patters of lenght $ w $. Since the number of different bit patterns doubles as you add each bit, we need at least $ w $ bits where $ 2^w = n $, i.e., we need $ w = log_2(n) $ bits

7.4 Logarithms and Multiplication

Logarithms are useful for multiplication, particularly for exponentation

$$log_a(xy) = log_a(x) + log_a(y)$$

the log of a product is the sum of logs property is a direct consequence of the following

$$log_a(n^b) = b * log_a(n)$$

So how can we compute $ a^b $ for any $ a $ and $ b $ using the $ exp(x) $ and $ ln(x) $ functions on your calculator, where $ exp(x) = e^x $ and $ ln(x) = log_e(x) $?

We know $ a^b = exp(ln(a^b)) = exp(bln(a)) $

so the problem is reduced to one multiplication plus one call to each of these functions

7.5 Fast Exponentation

what if we wanted to compute exact value for $ a^n $ for some large $ n $

the simplest algorithm performs $ n - 1 $ multiplications by computing $ a * a * … a $. But we can do better by noting that $ n = [n /2] + [n/2] $

If $ n $ is even, then $ a^n = (a^{\{n/2\}})2 $

If $ n $ is odd, then $ a^n = a(a^{\{[n/2]\}})2 $

In either case, we have halved the size of our exponent at the cost of, at most, two multiplications so $ (lg n) $

```
function power(a, n)
    if (n = 0) return(1)
    x = power(a, n/2 )
    if (n is even) then return(x2)
        else return(a × x2)
```

7.6 Logarithms and Summations

The *Harmonic number* arise as a special case of arithmetic progression, namely $ H(n) = S(n, -1) $. They reflect the sum of the progression of simple reciprocals, namely

This shows up in places such as the federal sentencing guidelines for fraud

The harmonic number is important because they usually explain "where the log comes from" when one magically pops out from algebraic manipulation. The average complexity of quicksort is the sumation

8 Properties of Log

$$b^x = y$$

$$x = log_b(y)$$

Important Bases: - Binary Logarithm - Base $b = 2$; we have seen how the base arises whenever repeated halving (binary search) or doubling (nodes in tree) occur. Most algorithmic applications of logarithms imply binary logarithms - Natural Log - Base $b = e$; usually denoted $ ln(x) $, is base $ e = 2.71828… $ logarithm. The inverse of $ ln(x) $ is the exponential function $ exp(x) = e^x $ on your calculator - Common Logarithm Base $ n = 10 $; less common today is the base-10, usually denoted as $ log(x)$. The base was employed in slide rules and algorithm books in days before pockey calculators

$$log_a(xy) = log_a(x) + log_a(y)$$

$$log_a(b) = log_c(b)/log_c(a)$$

Key Ideas: - The base of the logarithm has no real impact on the growth rate. Compate the following three values; $ log_2 (1,000,000) = 19.93) $, $ log_3 (1,000,000) = 12.575 $ and $ log_100 (1,000,000) = 3 $.A big change in the base of the log produces little difference in the value

of the log. The conversion factor is lost to Big-O notation whenever $ a $ or $ c $ are constants - Logarithms cut any function down to size: the growth rates of the logatihm of any polynomial function is O (lg n): this is because $ log_a(n^b) = b*log_a(n) $. The power of binary search on wide range of problems is a consequence of this observation. Note that doing a binary search on a sorted array of $ n^2 $ things requires only twice as many comparisons as a binary search on a $ n $ things. Same with factorals

9 Advanced Analysis

$ n! > c^n > n^2 > n^2 > n^1+e > n log n > n > sqrt(n) > log^2(n) > log(n) > log(n)/log log(n) > log log (n) > a(n) > 1 $