CSE4081 Analysis of Algorithms

Final Exam

Grant Butler | gbutler2020@my.fit.edu

Table of Contents

Loops, Counting, and Summations

There are two basic loop types: `for` and `while`.

1. Dot Product as a For Loop

The syntax for *for* loops (usually) describe how many times statements in its scope executes. An analysis skill is to be able to use summation notation to countthe number of times instructions executes in a for loop.
Here is an example from linear algebra: Given two vectors

$$\vec{A} = \langle a_1, a_2, \ldots, a_n \rangle \text{ and } \vec{B} = \langle b_1, b_2, \ldots, a_n \rangle$$

Their *inner* (dot) product is:

$$\langle a | b \rangle = \sum_{k=n}^{k=1} (a_k \cdot b_k)$$

Write this sum as a ***for*** loop and state its *big-O* time complexity.

Code

```
// given vectors a and b of length n
int n;
float a[n], b[n];
float dot_product = 0.0;
for (int k = 1; i < n; i++) {
    dot_product += a[k] * b[k];
}
```

Time Complexity

Since there is only one for loop, the time complexity of the dot product is `O(n)`, because it depends solely on `n`, or the size of the vectors.

2. Matrix Multiplication as Nested For Loops

For loops can be nested, giving rise to a sequence of summations with one (perhaps) dependent on previous sums.

Given two *n* x *n* matrices A and B, the standard algorithm to compute their product, given as $C = A \times B$ is to computer the dot product of each row of A_i with each column of B_j.

Write this description as nested *for* loops, express the operational complexities as summations and simplify the results.

Code

```
// given:
int n;
float a[n][n], b[n][n];

// the resultant matrix is output to:
float result[n][n];

// using these for loops:
// given that all of the elements of the resultant matrix are initialized to 0
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            result[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

Summation

As a summation in sigma notation, the multiplication of two $n \times n$ matricies A and B can be written as such:

$$c_{ij} = \sum_{k=0}^{n} a_{ik} \cdot b_{kj}$$

Time Complexity

What is the big-*O* time complexity of this standard matrix multiplication algorithm?

The time complexity of this algorithm is $O(n^3)$ because there are 3 for loops being used. Variable `i` is used for each row A_i, `j` is used for each column B_j, and `k` is used to increment through the elements in each. Thus, for input of two $n \times n$ martrices, this will run in n^3 time, since each loop runs n times, and they are nested.

Summations

3. Geometric Summation

Geometric summations and their variations often occur because of the natureof recursion. What is a simple expression for the following sum?

$$\sum_{i=0}^{i=n-1} 2^i$$

Solution

$$\sum_{i=0}^{i=n-1} 2^i = 2^0 + 2^1 + 2^2 + 2^3 + ... + 2^{n-1}$$
$$= 1 + 2 + 4 + 8 + ... + 2^{n-1}$$

$$\text{Sum} = \frac{a_1(1 - r^m)}{1 - r} = \frac{1(1 - 2^n)}{1 - 2}$$
$$= 1 + 2 \cdot \frac{1 - 2^{n-1}}{1 - 2}$$

Not the most elegant solution.

4. Differentiation/Integration of a Sum

Differentiation and integration under a summation is formally useful. What are expressions for the derivative and integral of the following sum?

$$\sum_{i=0}^{i=n-1} 2^i$$

Solution

$$\begin{aligned}\int \left(\sum_{i=0}^{i=n-1} 2^i\right) di &= \sum_{i=0}^{i=n-1} \left(\int (2^i)\, di\right) \\ &= \sum_{i=0}^{i=n-1} \left(\frac{2^i}{\ln 2} + C\right)\end{aligned}$$

where C is the integration constant

Recursion

5. Binary Search

What recursion describes *binary search* and what is its solution?

$$
\begin{aligned}
T(n) &= T\left(\frac{n}{2}\right) + 1 \\
T\left(\frac{n}{2}\right) &= T\left(\frac{n}{4}\right) + 1 \\
T\left(\frac{n}{4}\right) &= T\left(\frac{n}{8}\right) + 1 \\
T\left(\frac{n}{2^{k-1}}\right) &= T\left(\frac{n}{2^k}\right) + 1(k) \\
\implies T(n) &= T\left(\frac{n}{2^k}\right) + 1(k) \\
\frac{n}{2^k} &= 1 \\
n &= 2^k \\
\log_2 n &= k \\
T(n) &= T(1) + \log_2 n \\
T(n) &= \log_2 n + 1 \\
&\implies O(\log n)
\end{aligned}
$$

6. A Common Recurrence

What is the solution and what algorithms does the following recurrence describe?

$$T(n) = 2 * T\left(\frac{n}{2}\right) + n$$

Solution

$$\begin{aligned}
T(n) &= 2 * T\left(\frac{n}{2}\right) + n \\
&= 2[2T(n/4) + n/2] + n \\
&= 2^2 * T(n/4) + 2n \\
&= 2[2^2 * T(n/8) + n/4] + 2n \\
\implies T(n) &= n \log n
\end{aligned}$$

Algorithm

This is the recurrence for the *mergesort* algorithm! This is because it sorts each half, and then has to do n calculations to merge the result.

7. Strassen's Matrix Multiplication

The recursion for Strassen’s (1966) matrixmultiplication algorithm is:

$$T(n) = 7T(n/2) + cn^2 \quad \text{with initial condition } T(1) = 1$$

Assume that all matrices have sized $2^{n-1} \times 2^{n-1}$. Solve the Strassen's reccurence to derive the running time of Strassen's matrix multiplication algorithm.

Solution

$$\begin{aligned}
T(n) &= 7 * T(n/2) + cn^2 \\
&= 7[7 * T(n/4) * c(n/2)^2] + cn^2 \\
&= 7^2 * T(n/4) + 7c(n/2)^2 + cn^2 \\
&= 7^3 * T(n/8) + 7^2 c * (n/4)^2 + 7c(n/2)^2 + cn^2 \\
&\vdots \\
T(n) &\le 7 * T(n/2) + cn^2 \text{ and } T(1) = 1 \\
\implies T(n) &= O(n^{\log_2 7})
\end{aligned}$$

Therefore, the time complexity of Strassen's Matrix Multiplication is $O(n^{\log_2 7})$.

New Developments

There have been other developments since 1966. Summarize these.

Newer algorithms, like the **Coppersmith-Winograd** algorithm, have been developed since and can solve this problem much faster.

Algorithmic Paradigms

8. List and Describe Paradigms

1. Brute Force
    - Try all solutions to find the best one. Slow and usually the worst possible choice.
2. Divide and Conquer
    - Divide a larger problem into smaller ones, usually with recursion.
3. Dynamic Programming
    - Use globally optimal solutions to solve sub-problems with *cached* informations.
4. Greedy
    - Find globally optimal solutions with locally optimal decisions.
5. Backtracking
    - Incrementally build on candidates and backtrack to previous candidates when the candidate cannot be a possible solution.
        - Monte Carlo Backtracking is my favorite, since it often has the prettiest visualizations.

Applications

9. Algorithms with Applications

An interesting question is: Which questions/problems have algorithms that can be applied to compute solutions? We know there are questions with “yes or no” answers for which there is no algorithm.

I would argue that most problems have some solution, and the very point that a yes/no answer can be given for some problems, for instance the Twin Prime Conjecture, have a solution already. This question is begging me to give a humorous approach by referring to how some people will post joke videos of them 'programming' a even/odd number checker where they have switch cases for every single number possible, and complain about how much longer their code will have to be. There might be an 'algorithm' there, but it's not a very good one. Thus, I present that even questions with binary answers still have an 'algorithm' of sorts, but then again, not a very good one.

10. Undecidable Questions

1. Name and describe some of these undecidable questions.
   - Twin Prime Conjecture
   - Continuum Hypothesis
2. I like string problems and their algorithms (I know little about “String theory” but believe it is an interesting attempt to describe our universe (whatever that is) take time and look up string theory.
3. Name some questions about strings and describe algorithms that answer these questions.
   - There is a question of using a bombardment of particles to get a 'picture' of something when they bounce back, and anything that is smaller than a photon is not seeable, yet there are supposed to be quantum ripples that are smaller than a photon. String Theory is supposed to be able to meld together Relativity and Quantum by having the photon be a 'string', and thus there must be a string that is smaller to use to detect quantum ripples. Some sort of divide and conquer approach probably could be used to model something like that, increasing the effective detail of the resultant 'picture'.

Computational Complexity

11. Class *P* Problems

Class P problems are problems that are solvable in polynomial time, that is, that they are solvable in $O(n^k)$ time in their worst case. They are tractable problems, and can be solved in practice. An intractable problem cannot be feasibly be done in practice.

12. Class *NP* Problems

Class NP problems are problems that are *veryfiable* in polynomial time. This does not mean that there is a way to find the solution in that time, just to check the solution's validity.

13. *NP* Problem Characterization

The class of NP Problems can be characterized as a checkable proof that is probabilistically verifiable where one uses random bits to check the solutions. See here.

14. Reductions of Instances

This is problem solving technique of reducing a new problem to an already solved problems. What time restrictions should be placed take on the run time of a reduction?

Only problems easier than the new problem should be chosen to reduce the new problem to, never something harder than the new problem. It is called *Reduction*, is it not?

15. There was no question here.

```
An N P-complete problem, let’s call it Y,
```

16. Recursive Languages

A recursively enumerable language is a *Turing-acceptable* language if it is a subset in the set of all possible words over the alphabet of the language, or that there is a Turing machine that will enumerate all valid strings of the language.

A recursive language is one that is a recursive subset of all possible finite sequences over the alphabet of that language.