ECE 592 – Topics in Data Science

Test 2: Scientific Programming – Fall 2022

October 3, 2022

Question 1 (Computational complexity.)
What is the computational complexity of the following pseudo code? Express your answer as a function of N using $\Theta(\cdot)$ notation. Make sure to justify your answer.

```
a = 0 ## initialize the variable a with 0
for i = 1 to N { ## loop over i
    for j = 1 to ceiling(square_root(i)) ## loop over roughly sqrt(i) values
        a = a + my_function(i,j) ## increment by function (runs in constant time)
    } ## end loop over j
} ## end loop over i
```

Solution. The number of times that the inner loop runs is roughly

$$\sqrt{1} + \sqrt{2} + \ldots + \sqrt{N}.$$

This expression is $\theta(N^{1.5})$. One way to see this is to approximate it by an integral,

$$\int_{x=1}^{N} \sqrt{x} dx = \frac{x^{1.5}}{1.5}\Big|_{x=N} - \frac{x^{1.5}}{1.5}\Big|_{x=1} = \frac{N^{1.5} - 1^{1.5}}{1.5} = \Theta(N^{1.5}).$$

Another way is to realize that for i (the outer loop) from $N/2$ to N, we have $\Theta(N)$ instances of the outer loop where the inner loop performs $\Theta(\sqrt{N})$ operations. The product of N and $\sqrt{N}$ gives us the same $\Theta(N^{1.5})$. (When the outer loop is less than $N/2$, fewer operations are performed, hence it doesn't increase the Θ term.)

Question 2 (Algorithms.)
The ordered sequence of numbers $(1, 2, \ldots, N)$ is modified as follows.

1. One of the numbers is removed. (We now have $N-1$ numbers in our ordered sequence.)
2. All remaining $N-1$ numbers are permuted randomly. (We now have $N-1$ numbers in our randomly-ordered sequence.)

For example, if $N = 5$, then we begin with $(1, 2, 3, 4, 5)$, can remove 2, resulting in $(1, 3, 4, 5)$, and a random permutation could be $(3, 4, 1, 5)$.
You are given N and the permuted sequence, and your goal is to identify the number that was removed. (In our example, the number 2 was removed.) Describe an efficient algorithm and provide its computational complexity. (There is no need to use pseudocode; describing in words is fine if your algorithm is simple.) More credit will be provided for algorithms with lower complexity.

Solution. A naive solution involves sorting the $N-1$ numbers, and then scanning through them in order to identify the missing one. Sorting is $O(N \log(N))$, and scanning them is $O(N)$, hence this naive approach is $O(N \log(N))$. (If the numbers are already sorted, then realizing that they're sorted is $\Omega(N)$, hence this naive solution is $\Omega(N)$.)
We can do better by realizing that $1 + 2 + \ldots + N = N(N-1)/2$. Therefore, we can sum the $N-1$ numbers and subtract the sum from $N(N-1)/2$, resulting in the missing value. Summing is $\Theta(N)$, and performing the subtraction is $\Theta(1)$, hence this improved approach is $\Theta(N)$.

Question 3 (Constants matter.)
The purpose of this question is to show that details of a computing system can impact what algorithms we prefer to use. Consider a computing system where memory accesses to the cache are 20 times faster than random access memory (RAM) access. Now consider the memory resources that two algorithms require for processing an input of size N.

- Algorithm 1 requires $f_{c1}(N) = 10N$ cache accesses and $f_{R1}(N) = \sqrt{N}$ RAM accesses.
- Algorithm 2 requires $f_{c2}(N) = 3N$ cache accesses and $f_{R2}(N) = N$ RAM accesses.

Algorithm 1 uses more cache accesses than Algorithm 2, but fewer RAM accesses.

(a) Because RAM is 20 times slower than cache, we can interpret RAM accesses as analogous to 20 cache accesses. Compute $f_1(N) = f_{c1}(N) + 20f_{R1}(N)$ and $f_2(N) = f_{c2}(N) + 20f_{R2}(N)$.

Solution. $f_1(N) = f_{c1}(N) + 20f_{R1}(N) = (10N) + 20(\sqrt{N}) = 10N + 20\sqrt{N}$. $f_2(N) = f_{c2}(N) + 20f_{R2}(N) = (3N) + 20(N) = 23N$.

(b) For small values of N, $f_2(N)$ is faster (smaller); for large values of N, $f_1(N)$ is faster. Compute N^* for which $f_1(N^*) = f_2(N^*)$. You need not compute an actual number for N^*; an expression such as $N^* + 100/\sqrt{N^*} = \log_2(N^*)$ is fine.

Solution. We want N^* such that

$$f_1(N^*) = 10N^* + 20\sqrt{N^*} = f_2(N^*) = 23N^*,$$

meaning that $20\sqrt{N^*} = 13N^*$, hence $N^* = (20/13)^2$.

(c) This question assumes that the cache is 20 times faster than RAM. What if it the cache is 10 times faster than RAM, 40 times faster? Discuss (there is no need for detailed calculations) whether the break-even point, N^*, will become larger or smaller.

Solution. The main idea in this question is that $\sqrt{N}$ in $f_{R1}(N)$ is much faster than N in $f_{R2}(N)$, but one must pay more in $f_{C1}(N)$ in order to benefit from the faster f_{R1}. If the ratio increases from 20 to 40, then the advantage of using f_{R1} is even more pronounced, and we will use Algorithm 1 even for smaller problems (the break-even N^* becomes smaller.) If the ratio decreases to 10, then the advantage of f_{R1} is smaller, and Algorithm 1 becomes advantageous only for larger break-even points.