Problem Set Three

Zoe Farmer
Jeremy Granger
Ryan Roden

February 26, 2024

1. Professor Snape has n computer chips that are supposedly both identical and capable of testing each other's correctness. Snape's test apparatus can hold two chips at a time. When it is loaded, each chip tests the other and reports whether it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the answer of a bad chip cannot be trusted. Thus, the four possible outcomes of a test are as follows:

Chip A says	Chip B says	Conclusion
B is good	A is good	both are good, or both are bad
B is good	A is bad	at least one is bad
B is bad	A is good	at least one is bad
B is bad	A is bad	at least one is bad

Table 1: Chip Statements

   (a) Prove that if $n/2$ or more chips are bad, Snape cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool Snape.
   Hint: Let B denote the set of all bad chips and G the set of all good chips. Observe that there are only three types of comparisons: within B, within G, and between B and G. What kind of results will each of these types of comparisons yield? How big are the sets?

      i. Let B denote the set of all bad chips, and G the set of all good chips where $A = B \cap G$.

      We first take a random chip, called m, and iterate through A, identifying good-good responses. Every time we get a good-good response, it indicates that chip m is in the same set as its pair, and we add it to set M. For every chip added to our similar chips we then test with every other chip not in M, again identifying good-good responses. Any responses besides good-good are ignored.

      We now remove all elements in M from A, and repeat the process using M_n as our new good-good pair set each time until no more chips remain in A, and we have p sets of similar chips. At this point we can now combine the chips using the same method. Take any chip in set p_1 and match it with any chip in any p_n. If the answer is good-good, then we can group these chips together. This process is repeated until we have q sets. In the case of $|G| > |B|$, there will be a set in q with cardinaility greater than $|A|/2$, meaning that this set is the set of good chips, and the rest are bad.

Now let's consider the case where $|B| = |G|$. Using the same process as before we can identify p distinct sets that can be reduced into two sets using the same method. When these sets are checked with each other they will always return either good-bad or bad-good. If their cardinalities are the same, then there is no way to prove which set is the set of good chips and which is the bad, given their ability to conspire. If the bad chips conspire to fool Snape, then the bad chips will always strive to emulate the good chips, meaning no decision can be made about the quality of the chips.

Finally, considering the case where $|B| > |G|$, we can reduce the problem into q sets again with all good chips grouped together, however since there is no "largest" set, there is no way to identify any set. Given that the bad chips can conspire, they will refuse to join together, and are unable to join with the good chips, therefore all sets will have the same behavior.

(b) Consider the problem of finding a single good chip from among the n chips, assuming that more than $n/2$ of the chips are good. Prove that $n/2$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.
Hint: To reduce the problem's size, you will need to discard some chips. Only discard a pair that definitely includes a bad chip.

i. In order to solve this, we can use an algorithmic approach.

```python
def find_good(A):
    B = []        # New set to add to
    n = len(A)    # Length of set
    if n <= 2:    # If we hit the end, is good
        return A # Return the input and don't call
    else
        flag = False                # Flag for worst
        for i in range(0, n, 2):    # Step by two
            if i >= n/2:            # If covered half
                if len(B) >= n/2: # and setsize is n/2
                    flag = True    # switch tactics
                    break
            if testsetpair(A[i],A[i+1]) == 'gg':# test pairs
                B.append[A[i]]
        if flag:                    # switch to here
            for i in range(0, n, 2): # Again step the same
                # test two setpairs at halfway points
                if testsetpair(A[i], A[i + (n/2)]) == 'gg':
                    B.append[A[i]]
        return find_good(B)
```

In order to understand this algorithm we need to understand the three cases.

$$\begin{cases} \text{Best Case} & \Rightarrow \text{All of the chips alternate good-bad until the end} \\ \text{Average Case} & \Rightarrow \text{There's a random mix of good and bad chips} \\ \text{Worst Case} & \Rightarrow \text{The first } n/2 \text{ pairs are both bad} \end{cases}$$

For case 1, our algorithm will only accept good-good pairs, and no good-bad pairs will be accepted. Since there are more good chips than bad, we will reach the base case where the length of our array is less than or equal to 2 and both chips will be good.

For case 2, as before the algorithm will only accept good-good pairs. Unlike the first case however, this will take considerably longer to complete. For every good-good pair that is found, only one element will be accepted and then recursively called with our new set. Since we have more good than bad, we will eventually reach our smallest array with one or two good chips.

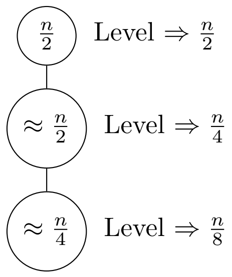
For case 3, we run into a slight problem. If all chips examined in the first $n/4$ operations are bad, then they will lie and always return a good-good pair when they can. If this occurs, the length of our new array will be greater than $n/2$. This needs to be avoided. In this case, we need to check that our array is not over that size, and if it is we need to break out of the loop and start checking again, however instead of i and $i + 1$, we need to check i and $i + n/2$ concurrently. This will remove all of those good-good pairs, and allow the algorithm to finish properly.

(c) Prove that the good chips can be identified with $\Theta(n)$ pairwise tests, assuming that more than $n/2$ of the chips are good. Give and solve the recurrence that describes the number of tests.

i. We can describe our algorithm above with a recurrence relation

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

Solving this using the recurrence tree method we obtain the following recursion tree.


$\frac{n}{2}$
Level $\Rightarrow \frac{n}{2}$
$\approx \frac{n}{2}$
Level $\Rightarrow \frac{n}{4}$
$\approx \frac{n}{4}$
Level $\Rightarrow \frac{n}{8}$

This can be rewritten as a sum with depth d.

$$T(n) = \sum_{i=0}^{d} \frac{n}{2^i}$$

We can determine d based off of the depth of the recursion tree.

$$T(n) = \sum_{i=0}^{\lg\left(\frac{n}{2}\right)} \frac{n}{2^i} = 2(n-1)$$

2. Consider the following basic problem. Professor Dumbledore gives you an array A consisting of n integers $A[1], A[2], \cdots, A[n]$ and asks you to output a two-dimensional $n \times n$ array B in which $B[i,j]$(for $i < j$) contains the sum of array elements $A[i]$ through $A[j]$, i.e., the sum $A[i] + A[i+1] + \cdots + A[j]$. (The value of array element $B[i,j]$ is left unspecified whenever $i \geq j$, so it does not matter what is output for these values.)
Dumbledore also gives you the following simple algorithm to solve this problem.

```
for i in range(1, n):
    for j in range(i + 1, n):
        s = sum(A[i:j])
        B[i,j] = s
```

(a) For some function f that you should choose, give a bound of the form $O(f(n))$ on the running time of this algorithm on an input of size n (i.e., a bound on the number of operations performed by the algorithm).

i. To start, since at worst case the two nested for-loops run 1 to n, and then 1 to n they produce $O(n^2)$. $B[i,j] = s$ is a constant time operation. Summing

$A[i:j]$ will use $(j-i)$ operations, and $j-i$ depends on the size of n, so it is essentially another nested for-loop of $O(n)$. Overall, the function has $O(n^3)$ runtime.

(b) For this same function f, show that the running time of the algorithm on an input of size n is also $\Omega(f(n))$. (This shows an asymptotically tight bound of $\Theta(f(n))$ on the running time.)

i. We can write the number of operations in Dumbledore's algorithm as a double sum, and then solve the sum to determine the lower bound.

$$\begin{aligned}
\text{Our original algorithm} \rightarrow \quad & \sum_{i=1}^{n}\left(\sum_{j=i+1}^{n}(j-i+1)\right) \\
\text{Substitute} \rightarrow \quad & k = i+1 \\
\text{Yielding} \rightarrow \quad & \sum_{i=1}^{n}\left(\sum_{j=k}^{n}(j-k)\right) \\
\text{The inner sum equals} \rightarrow \quad & \sum_{i=1}^{n}\left(\sum_{j=1}^{n} j\right) \\
\text{We can rewrite this as} \rightarrow \quad & \sum_{i=1}^{n}\left(\frac{1}{2}n\,(n+1)\right) \\
\text{The sum doesn't depend on } i \rightarrow \quad & \frac{1}{2}n\,(n+1)\left(\sum_{i=1}^{n} 1\right) \\
\text{We can rewrite the sum} \rightarrow \quad & \frac{1}{2}n^2\,(n+1) \rightarrow \frac{n^3+n^2}{2} \\
n^3 \text{ dominates} \rightarrow \quad & \sum_{i=1}^{n}\left(\sum_{j=i+1}^{n}(j-i+1)\right) = O(n^3) \\
\text{We've already proven the upper bound} \rightarrow \quad & \sum_{i=1}^{n}\left(\sum_{j=i+1}^{n}(j-i+1)\right) = \Theta(n^3)\blacksquare
\end{aligned}$$

(c) Although Dumbledore's algorithm is the most natural way to solve the problem – after all, it just iterates through the relevant elements of B, filling in a value for each – it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time and prove its correctness.

i. Our algorithm has better run time than Dumbledore's.

```python
old = None
for i in range(1, n):
    for j in range(i + 1, n):
        if old is None:
            old = A[i] + A[i + 1]
            B[i,j] = old
        else:
            B[i,j] = old + A[j]
```

The proof of this is below. Again, like before we can write our algorithm as a double sum. Since we always only perform one operation on the innermost loop, we have constant time which is represented as c.

$$\text{Our improved algorithm} \rightarrow \quad \sum_{i=1}^{n}\left(\sum_{j=i+1}^{n}(c)\right)$$

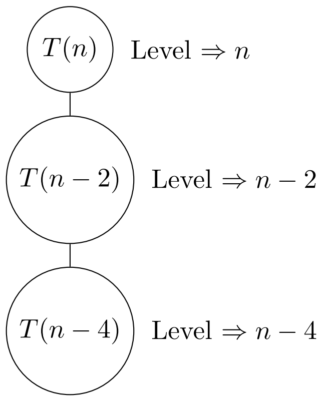
$$\text{The inner sum can be rewritten} \rightarrow \quad \sum_{i=1}^{n} n$$

$$\text{And the outer can now be rewritten} \rightarrow \quad n^2$$

$$\text{Therefore} \rightarrow \quad \sum_{i=1}^{n}\left(\sum_{j=i+1}^{n}(c)\right) = O(n^2) < O(n^3)\blacksquare$$

3. Why do we analyze the average-case performance of a randomized algorithm and not its worst-case performance? Succinctly explain.
   (a) When we use randomized quicksort we are only interested in the average case performance because when we use a randomized pivot the entry is pulled from a uniform distribution. In other words, every entry in the list has an equal chance of being selected as a pivot for that level of the recurrence tree. This means that the chance of encountering a worst-case scenario is greatly reduced.
4. Solve the following recurrence relations using the recurrence tree method; include a diagram of your recurrence tree. If the recurrence relation describes the behavior of an algorithm you know, state its name.

(a) $T(n) = T(n-2) + n$

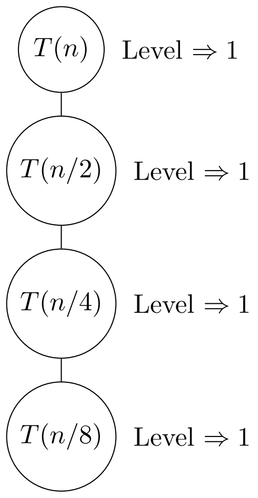


This can equivalently written as a sum with depth d.

$$n + \sum_{i=0}^{d-1} n - 2(i+1)$$

Therefore the complexity is $O(n^2)$. This describes the worst-case condition of quicksort, where the worst pivot is chosen each time, and the algorithm finishes in $O(n^2)$ time as well.

(b) $T(n) = T(n/2) + 1$

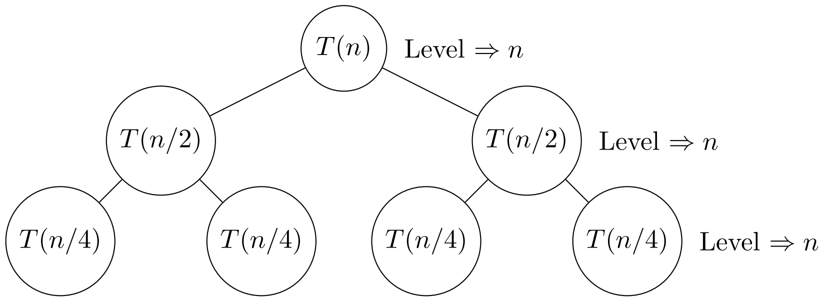


This can also be rewritten as a sum with depth d.

$$\sum_{i=1}^{d} i$$

Therefore the complexity is $O(\lg(n))$. Both quicksort's and mergesort's space complexity are described by this recurrence relation.

(c) $T(n) = 2T(n/2) + n$

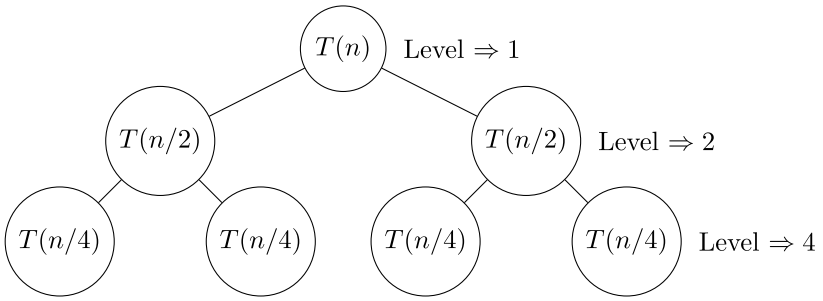


This can be rewritten as a sum with depth d.

$$\sum_{i=1}^{d} n$$

Therefore the complexity is $O(n \lg(n))$. Both Mergesort and Heapsort time complexity are described by this recurrence relation.

(d) $T(n) = 2T(n/2) + 1$



This can be rewritten as a sum with depth d.

$$\sum_{i=1}^{d} i^2$$

Therefore the complexity is $O(2^{\lg(n)})$, which is equal to $O(n)$. This describes best runtime complexity for quicksort, bubblesort, and binary tree sort.