TCSS 343 - Week 3

Jake McKenzie

August 5, 2018

Divide and Conquer

"Common misconception that *fun* is relaxing.
If it is, youre not doing it right."
...
Iain Banks

"Writing is nature's way of letting you know how sloppy your thinking is."
...
Guindon

"Master method will never be sufficient for the detailed 'knuthian' analysis of algorithms, but they can free algorithm designers from mundane analyses to let them work on more interesting problems."
...
Jon Bentley, Dorothea Blostein et al (creators of the master method)

0. Use the master method to solve the following recurrences. If the master method does not apply indicate that it does not.

I) $T(n) = 7T(\frac{n}{7}) + O(n)$

II) $T(n) = 5T(\frac{n}{3}) + O(n)$

III) $T(n) = 3T(\frac{2}{n}) + O(1)$

IV) $T(n) = 16T(\frac{n}{4}) + O(n^3)$

V) $T(n) = 4T(\frac{n}{9}) + O(n \log n)$

VI) $T(n) = T(\frac{n}{9}) + O(\sqrt{n})$

VII) $T(n) = 7T(\frac{n}{2}) + O(n^2)$

VIII) $T(n) = 4T(\frac{n}{2}) + O(16^{\log n})$

IX) $T(n) = 2T(n) + O(n^3)$

X) $T(n) = 2T(\frac{n}{3}) + O(n^{-\log n})$

XI) $T(n) = 2T(\frac{n}{2}) + O(n^{0.51})$

XII) $T(n) = 5T(\frac{n}{25}) + O(n^{-1})$

XIII) $T(n) = 2.1T(0.3n) + O(2^{\log \sqrt{n}})$

1. Consider the problem of searching an element x in an array arr[] of size n. The problem can be solved in $O(\log n)$ time if:

I) Array is sorted

II) Array is sorted and rotated by k. k is given to you and $k \leqslant n$

III) Array is sorted and rotated by k. k is NOT given to you and $k \leqslant n$

IV) Array is not sorted

a) I

b) I and II

c) I,II and III

d) I,II,III and IV

2. What is the probability that a number is composite numbers in first 50 natural numbers inclusive? (hint: 1 is neither composite nor prime and composite). (I've seen this question in pools of interview questions for algorithms)

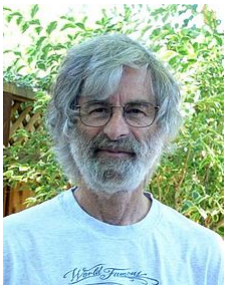
a) 0.58

b) 0.62

c) 0.68

d) 0.72

3. Solve the following recurrence: $T(n) = T(\frac{6n}{7}) + T(\frac{n}{7}) + O(n)$. The master method cannot be used on this one. Use the tree method or repeated substition to find the runtime.


Thinking doesn't guarantee that we won't make mistakes. But not thinking guarantees that we will. ~ Leslie Lamport (turing award winnder, both devised important algorithms and ways of formal modeling)

4. **Preferably by using your notes, write down a formal specification for quicksort**. Formal specifications are important, they give us the language to describe subtle problems that you can't get from simply writing code. When students hear "specifications" they typically panic. You have to learn and use these funny symbols and such, but if you don't get it right that's okay. The world isn't going to come crashing down if you don't do get it perfectly right but please try.


Writing formal specifications won't catch coding errors or bugs, but it will catch algorithm errors. ~ Leslie Lamport (the guy from the previous page)

5. You typically write specifications while thinking about the problem, before you write code. I'm assuming you probably wrote a recursive specification for quicksort on the previous page. If you didn't, great you're done you can move on. **If you did write a recursive specification for quicksort, please now attempt to write a non-recursive specification for quicksort**. This is more challenging but please, please try to do so. It requires more thought but thinking is a really good idea; don't trust anyone who tells you otherwise. **Come up with one reason why you think it's a good idea to write a spec**.

A really important algorithm that's used on the backend that you've probably never heard of is known as *rotate*. It's is a fundamental tool used behind the scenes in all sorts of computer graphics from updating frame buffers on embedded devices to websites. We can even give a formal mathematical **specification**:

$$\text{A permutation of } n \text{ elements by } k \text{ where } k \geqslant 0 \quad :$$

$$(k \mod n, k+1 \mod n, \ldots, k+n-1 \mod n, k+n-1 \mod n)$$

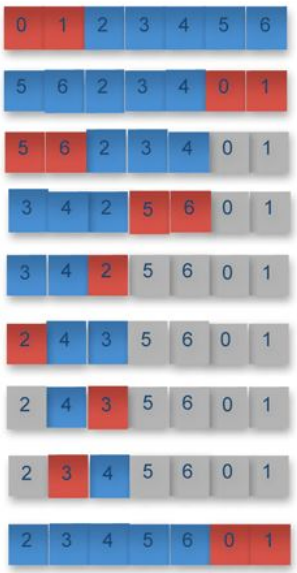
Mathematical specifications can be nice compact ways of specifying algorithms when we can find them, but this isn't always possible. My favourite way that I've seen this implemented is the Gries-Mills Block Swapping algorithm which is an inplace algorithm:

Gries and Mills Block Swapping

Bentley also describes a block-swapping algorithm by Gries and Mills in his book and the article. The algorithm swaps the largest equal-sized non-overlapping blocks available at each step. The main idea is to use swap as an operation as follows from his article:

1. A is the left subarray, B is the right subarray – that is, the starting point is AB
2. If A is shorter, divide B into B_L and B_R, such that length of B_R equals the length of A
3. Swap A and B_R to change AB_LB_R into B_RB_LA
4. Recur on the two pieces of B
5. Once A and B are of equal lengths, swap A and B

Figure 6 shows the steps:




6. Can you write down an algorithm that does what is described on the last page, it need not be Gries-Mills'(pictured ←) block swapping algorithm, that attempts to do *rotate*?

```
def binary_sum(x):
    if len(x) == 0: return 0;
    if len(x) == 1: return x[0];

    y = []
    for i in range(1,len(x),2):#range(state,stop,step)
        y.append(x[i - 1] + x[i])
    if i + 1 < len(x):
        y[0] += x[i + 1]

    return binary_sum(y)
```

7. Can you write down a recurrence relation for the python code above and then solve for the runtime?

8. Match the complexity class to what "makes sense":

A) $O(n^2)$	1) adding a nested loop for every input you have
B) $O(1)$	2) iterations that use divide and conquer
C) $O(n)$	3) random access to an element in a collection, dependent on indexing
D) $O(\log n)$	4) list iterations
E) $O(n \log n)$	5) nested loops on the same collection
F) $O(n!)$	6) divide and conquer

9.

Master Theorem

The Master Theorem applies to recurrences of the following form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

There are 3 cases:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with[1] $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$. Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

9. Use this different formulation of the master theorem from CLRS to solve these recurrences relations.

a) $T(n) = 4T(\frac{n}{9}) + O(n \log n)$

b) $T(n) = 2T(\frac{n}{3}) + O(n^{-\log n})$

c) $T(n) = 5T(\frac{n}{2}) + O(2^n)$