CS170–Fall 2015 — Homework 2 Solutions

Susanna Souv, SID 24346856

Collaborators: David Lu, Angela Kuo

1. Recurrence Relations

(a) $T(n) = 3T(n/4) + 4n$

Because the recurrence relation is in the form $T(n) = aT(\lceil \frac{n}{b} \rceil) + O(n^d)$ where $a > 0$, $b > 1$, and $d \geq 0$.

We can compare d and $log_b a$.

$d = 1$

$log_b a = log_4 3$

$log_b a < d$ so $T(n) = \Theta(n)$

(b) $T(n) = 45T(n/3) + .1n^3$

Because the recurrence relation is in the form $T(n) = aT(\lceil \frac{n}{b} \rceil) + O(n^d)$ where $a > 0$, $b > 1$, and $d \geq 0$.

We can compare d and $log_b a$.

$d = 3$

$log_b a = log_3 45$

$log_b a > d$ so $T(n) = \Theta(n^{log_3 45})$

(c) $T(n) = T(n-1) + c^n$

We can't use the Master Theorem here so we look at how it branches out and the work done at each level.

The branching factor $a = 1$ so the number of branches at each node is 1 (i.e it's linear). At each level, c^i work is done (where i is an integer s.t. $0 < i \leq n$).

The total running time is all of that summed up so:

$\Theta(\sum i = 1^n c^i)$

which is (when $c \neq 1$) further reduced to:

$\frac{\Theta(c^1 - c^{n+1}}{1-c) = \Theta(c\frac{(1-c^n)}{(1-c)})}$

from Homework 1, we know that

when $c = 1$:

every term is 1 so the run time approximates to $\Theta(n)$

(d) YOUR ANSWER GOES HERE

2. Goldilocks' Problems

Main Idea:

We pick a pivot Goldilocks, i.e. the Goldilocks that the first soup matches with and keep track of the other Goldilocks' other responses in two arrays, one for too hot and one for too cold. We pair off each Goldilocks with the correct soup and place in an array to return. For each soup that we check, we start with an already paired off Goldilocks and depending on whether they find it too hot or cold, we start checking the array full of Goldilocks who thought otherwise.

Let $G = [g_1, ..., g_n]$ be the array of Goldilocks and each g_i is a distinct Goldilocks. Likewise for $S = [s_1, ..., s_n]$ as the array of soups that each Goldilocks has exactly one soup just right for them. At each level, we'll have two arrays H_i ("H" for "hot") and C_i ("C" for "cold"). We take s_1 and categorize each g_i by putting the ones who find it too hot in H_1 and too cold in C_1. We pair off the correct g_1 with s_1. With s_2, we check to see if g_1 finds s_2 too hot or too cold. If g_1 finds it too hot, we check the Goldilocks in C_1; otherwise, we check the ones in H_1. g_1 will not find s_2 just right if they have already found s_1 just right because each g has exactly one unique, correspondng s. As we continue with s_i where $2 < i \leq n$, we check the s_i starting with g_1 and check in the corresponding array, depending on the response in the same way we decided which array to check s_2 in. We continue until we are down to s_n and there is only one g_n left.

Pseudocode

procedure goldilocks-soups(Goldilocks[] G, Soups[] S)
R = Pair[], array of Pairs (Goldilocks g_i, Soup s_i)
hot = Goldilocks[][], array of arrays of Goldilocks; indexed by Soups
cold = Goldilocks[][], array of arrays of Goldilocks; indexed by Soups
current = the current Goldilocks we are checking s_i with
for each soup in S:
if R is empty:
current g = G[0]
current a = G
else:
current g = R[0]
current a = R
status = justright(current g, soup)
while a is not empty:
if status == "too hot":
hot[soup.index].append(current g)
if cold[soup.index] is not empty:
a = cold[soup.index]
else if status == "too cold":
cold[soup.index.append(current g)
if hot[soup.index] is not empty:
a = hot[soup.index]
else:
R.append((current, soup)) status = justright(current g,
return R

1: **procedure** GoldilocksSoup
2: **if** t **then**rue **return** false

3.Stock Market Hindsight

(a) Note that assuming 0 profit on the first day, then buying and then selling each subsequent day (i.e. just totalling as if we bought on the ith day and then selling the very next day) we get an accurate total for each day starting from day 1. Array B is exactly this. Every B[i] will the be profit if you bought on the first day and then sold on day i.

For example, assume A = [2, -1, 8, -7, 3, -9]; then B would = [0, 2, 1, 9, 2, 5].

(b) The construction of array B should take linear time, every single time.

In order to find the maximum profit, we compare the amount of profit between different indexes in B. Time only goes in one direction so we can only compare move from left to right. This is exactly like question 6 from Homework 1 where we took a divide-and-conquer approach to find the highest possible change in elevation. In order to do this, we need to find the lowest possible point that must be left of the highest possible which must be more right. The biggestJump procedure divides the input array in half and finds the minimum value and maximum value, given the leftmost index and the rightmost index. The procedure recurses until we reach one-element arrays. At each level, only the highest height that is valid and corresponding configuration is returned. Given that the runtime of this is $\Theta(n)$ from the previous homework and the time it takes to create array B is $\Theta(n)$ as well, $T(n) = \Theta(n) + \Theta(n) \approx \Theta(n)$. So the runtime is linear.

4.Majority Elements

(a) **Main idea:**

Essentially, we count and tally every unique element by doing divide-and-conquer. We split each array in half at each level until we reach 1-element arrays. At the very end with all of the tallies, we see if the highest tallied element is $> \frac{n}{2}$. If so, it is majority element. Otherwise, there is none.

Pseudocode:

(b) **Main idea:**

5.Local Maxima in Matrices

6.Upper Triangular Matrix Multiplication

Let B be the upper right matrix. If we recurse, this algorithm assumes that every matrix we do this for is an upper triangular matrix but since B is not, it does not hold.