Assignment 2

Benjamin Jakubowski

June 9, 2016

1. Recurrences

First, recall the master theorem provides bounds for recurrences of the form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

a. $T(n) = 4T\left(\frac{n}{3}\right) + n$

In this problem, $a = 4, b = 3$, and $f(n) = n$.
Since $f(n) = n = O(n^{log_3 4 - \epsilon})$, then $T(n) = \Theta(n^{log_3 4})$ (by master theorem case one). We proceed to prove this bound using substitution:
Assume $T(k) \leq c_1 k^{log_3 4} - c_2 k$ for $k < n$. Now consider $T(n) = 4T\left(\frac{n}{3}\right) + n$. Then

$$\begin{aligned} T(N) &= 4T\left(\frac{n}{3}\right) + n \\ &\leq 4\left[c_1\left(\frac{n}{3}\right)^{log_3 4} - c_2\left(\frac{n}{3}\right)\right] + n \\ &= c_1 n^{log_3 4} - \frac{4}{3}c_2 \cdot n + n \\ &= c_1 n^{log_3 4} - \left[\frac{4}{3}c_2 - 1\right] n \end{aligned}$$

Then to conclude the proof, we must show

$$\begin{aligned} & c_1 n^{log_3 4} - \left[\frac{4}{3}c_2 - 1\right] n \leq c_1 n^{log_3 4} - \left[\frac{4}{3}c_2 - 1\right] n \\ \implies \quad & \frac{4}{3}c_2 - 1 \geq c_2 \\ \implies \quad & 1 \leq \frac{1}{3}c_2 \\ \implies \quad & 3 \leq c_2 \end{aligned}$$

Thus, having shown the bound holds for all $c_2 \geq 3$, we've shown $T(n) = \Theta(n^{log_3 4})$.

B. $T(n) = 4T\left(\frac{n}{2}\right) + n^2$

In this problem, $a = 4, b = 2$, and $f(n) = n^2$.
Since $f(n) = n^2 = \Theta(n^{log_2 4}) = \Theta(n^2)$, then $T(n) = \Theta(n^{log_2 4} \log n) = \Theta(n^2 \log n)$ (by master theorem case two). We proceed to prove this bound using substitution:
Assume $T(k) \leq dk^2 \log k$ for $k < n$. Now consider $T(n) = 4T\left(\frac{n}{2}\right) + n^2$. Then

$$\begin{aligned} T(N) &= 4T\left(\frac{n}{2}\right) + n^2 \\ &\leq 4\left[d\left(\frac{n}{2}\right)^2 \log\left(\frac{n}{2}\right)\right] + n^2 \\ &= dn^2 \log n - dn^2 \log 2 + n^2 \\ &= dn^2 \log n - (d-1)n^2 \\ &\leq dn^2 \log n \qquad \text{if } d \geq 1 \end{aligned}$$

Thus, having shown the bound holds for all $d \geq 1$, we've shown $T(n) = \Theta(n^2 \log n)$.

C. $T(n) = 4T\left(\frac{n}{2}\right) + n^2 \log n$

In this problem, $a = 4, b = 2$, and $f(n) = n^2 \log n$.
Next, note $n^{\log_b a} = n^{\log_2 4} = n^2$. Then, while $f(n) = n^2 \log n$ is asymptotically larger than n^2, it is not polynomially larger. Thus, the master theorem is not applicable (and, per the instructions in the class forum, the solution is not found).

2. Exact solution of recurrence

We aim to find the exact solution of the recurrence

$$T(n) = \begin{cases} c & \text{if } n = 0 \\ aT(n-1) + k & \text{if } n > 0 \end{cases}$$

Solution: The solution is

$$T(n) = \sum_{i=0}^{n-1} a^i \cdot k + a^n \cdot c$$

Proof:
We prove by induction- just to build insight, we prove few additional base cases.
First, when $n = 0, T(n) = c$. When $n = 1$,

$$T(1) = a \cdot T(0) + k = a \cdot c + k$$

When $n = 2$

$$T(2) = a \cdot T(1) + k = a \cdot (a \cdot c + k) + k = a \cdot k + k + a^2 c$$

Thus, we hypothesize $T(n) = a^{n-1}k + a^{n-2}k + \cdots + a^0 k + a^n c$.

Formally, assume for n

$$T(n) = \sum_{i=0}^{n-1} a^i \cdot k + a^n \cdot c$$

Then consider $T(n+1)$:

$$\begin{aligned} T(n+1) &= aT(n) + k \\ &= a\left[\sum_{i=0}^{n-1} a^i \cdot k + a^n \cdot c\right] + k \\ &= \left(\sum_{i=0}^{n-1} a^{i+1} \cdot k\right) + k + a^{n+1} \cdot c \\ &= \left(\sum_{i=1}^{n} a^i \cdot k\right) + a^0 k + a^{n+1} \cdot c \\ &= \left(\sum_{i=0}^{(n+1)-1} a^i \cdot k\right) + a^{n+1} \cdot c \end{aligned}$$

3. Iterative binary search

Pseudocode for iterative binary search is given in Algorithm 1.

Algorithm 1 Iterative binary search

```
1: function ITERATIVE_BINARY_SEARCH(A, target)
2:     lower_index = 0
3:     upper_index = len(A) - 1
4:     while upper_index > lower_index do
5:         middle_index = (lower_index + upper_index)/2
6:         if A[middle_index] < target then
7:             lower_index = middle_index+1
8:         else if A[middle_index] > target then
9:             upper_index = middle_index-1
10:        else:
11:            return middle_index
12:    if upper_index == lower_index then
13:        if A[lower_index] == target then
14:            return lower_index
15:        else:
16:            return "Target not in array"
17:    else:
18:        return "Target not in (empty) array."
```

Now we prove the correctness of our algorithm using a loop invariant. First, for the sake of notation, we'll refer to *target* as t, *lower_index* as l, *middle_index* as m, and

upper_index as u. We prove the correctness through two cases: t in A and t not in A. First, if t is in A, we use the following loop invariant: at the start of each iteration of the while loop, if t is in A, then t is in the subarray $A[i, u]$.

- **Initialization**: If t is in A, then prior to the first iteration of the loop t is in $A[l, u]$ since $A[l, u] = A$.
- **Maintenance**: Now assume the invariant is true before an iteration of the loop. Then (assuming the loop doesn't terminate with the correct solution m) note the body of the loop either changes the value of u or l; in either case, the change ensures $A[l, u]$ still contains t (because of the conditions in the **if** and **else if** blocks on line 6 and 8).
- **Termination**: Now note the loop terminates in one of two ways:
  - **Loop terminates due to condition**: In this case, the loop terminates when $l = u$. But then, since t is in $A[l, u]$, $l = u$ implies $A[l, u]$ contains a single element- namely t. Thus, returning l in line 12 returns the correct index of t in the array.
  - **Function returns**: In this case, the function returns (from line 10) with the correct value, since the **else** statement only evaluates when $t = A[m]$.

Finally, to complete the proof, let's consider the case when A does not contain t. Then the algorithm is clearly correct, since either

- A **is the empty array**: In this case, the while loop is not executed (since $u = -1 < 0 = l$), and control jumps from line 4 to line 14, returning the string indicating the target is not in the empty array.
- A **is non-empty**: In this case, the while loop is executed, but clearly $A[m] \neq t$; hence the loop terminates exactly when $u = l$ (note the loop will terminate, since $u - l$ is strictly decreasing with each iteration of the loop). Thus, after the loop terminates, the program evaluates the **else** block on line 13, returning the target is not in the array.

Finally, note the worst case runtime of this implementation of binary search is when t is not in A. Then the **while** loop is executed $O(\log(n))$ times, so the worst-case running time is $O(\log(n))$.

4. Recursive binary search

Pseudocode for recursive binary search is given in Algorithm 2. Note each call to the function spans a single recursive call (with $1/2(u_0 - l_0) \approx u_1 - l_1$). Otherwise the function's runtime is $O(1)$ (all since all other statements run in constant time). Thus, our recurrence is $T(n) = T\left(\frac{n}{2}\right) + c$. sThen, by the master theorem case 2, we have

$$f(n) = c = \Theta(1) = \Theta(n^0) = \Theta(n^{log_2 1}) = \Theta(n^{log_b a})$$

so

$$T(n) = \Theta(n^{log_2 1} \log n) = \Theta(\log n)$$

Algorithm 2 Recursive binary search

```
1: function REC_BINARY_SEARCH(A, target, lower_index = None, upper_index = None)
2:     if lower_index is none then
3:         lower_index = 0
4:     if upper_index is none then
5:         upper_index = len(A) - 1
6:     if upper_index > lower_index then
7:         middle_index = (lower_index + upper_index)/2
8:         if A[middle_index] < target then
9:             return Recursive_binary_search(A, target, middle_index+1, upper_index)
10:        else if A[middle_index] > target then
11:            return Recursive_binary_search(A, target, lower_index, middle_index-1)
12:        else:
13:            return middle_index
14:    else if upper_index == lower_index then
15:        if A[lower_index] == target then
16:            return lower_index
17:        else:
18:            return "Target not in array"
19:    else:
20:        return "Target not in (empty) array."
```

5. Counting inversions with merge sort

Pseudocode for counting inversions using a modification of merge sort is given in Algorithm 3. Note the only real change (beside minor changes in function signatures) between this inversion counting algorithm and vanilla merge sort is **Line 17**, where we increment the inversion count by the number of elements remaining in A if we append an element from B before array A is empty. This is a constant-time addition to the algorithm.

Since merge sort has a worst-case running time of $O(n \log n)$, and we are only adding constant time operations to the algorithm, the worst-case running time of our inversion counting algorithm is also $O(n \log n)$. To be more rigorous, we can formulate and solve a recurrence. First, note `inversion_count_merge`'s worst-case runtime is $O(n)$; in addition, the other (non-recursive) operations in `inversion_count_merge_sort` run in constant time[1]. Thus, `inversion_count_merge_sort` runtime is $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$.

[1] More precisely, the non-recursive operations in `inversion_count_merge_sort` could be implemented in constant time. Note the outlined approach using list slicing is suboptimal. An improved version of the pseudocode would not use list slicing, which is not $O(1)$, and instead change the function signature to include parameters for the lower and upper indices into the original array

Now we apply the master theorem:

$$f(n) = c \cdot n = \Theta(n) = \Theta(n^1) = \Theta(n^{\log_2 2}) = \Theta(n^{\log_b a})$$

Thus, by master theorem case 2,

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$$

Algorithm 3 Counting inversions

```
1: function INVERSION_COUNT_MERGE(A, B, count = 0)
2:     n = len(A) + len(B)
3:     A.append('end')
4:     B.append('end')
5:     merged = []
6:     for i in [0, ··· n − 1] do
7:         if len(A) == 1 then
8:             merged.extend(B[0:-1])
9:             return merged, count
10:        else if len(B) == 1 then
11:            merged.extend(A[0:-1])
12:            return merged, count
13:        else if A[0] ≤ B[0] then
14:            merged.append(A.pop(0))
15:        else
16:            merged.append(B.pop(0))
17:            count += len(A) - 1
18: function INVERSION_COUNT_MERGE_SORT(A)
19:    if len(A) > 1 then
20:        n = len(A)
21:        mid = n/2
22:        L = A[0 : mid]
23:        R = A[mid : n]
24:        L_sorted, L_count = inversion_count_merge_sort(L)
25:        R_sorted, R_count = inversion_count_merge_sort(R)
26:        return inversion_count_merge(L_sorted, R_sorted, L_count + R_count)
27:    else
28:        return A, 0
```