CSCI 3104 Summer 2018
Problem Set 3
Eischen, Parker
10/31
parker.eischen@colorado.edu

1. (10 pts total) For parts (1a) and (1b), justify your answers in terms of deterministic QuickSort, and for part (1c), refer to Randomized QuickSort. In both cases, refer to the versions of the algorithms given in lecture (you can refer to the moodle lecture notes).

(a) What is the asymptotic running time of QuickSort when every element of the input A is identical, i.e., for 1 ≤ i, j ≤ n, A[i] = A[j]?

If each element is the same, the recursion tree becomes either left-side or right-side dominant. After each iteration the left side is decreased by one element and the right side is left empty. This causes the equation to have a run time of O(n^2)

(b) Let the input array A = [9, 7, 5, 11, 12, 2, 14, 3, 10, 6]. What is the number of times a comparison is made to the element with value 3?

A = [9, 7, 5, 11, 12, 2, 14, 3, 10, 6] <-pivot i=-1, j=0
A = [9, 7, 5, 11, 12, 2, 14, 3, 10, 6] <-pivot i=-1, j=1
A = [9, 7, 5, 11, 12, 2, 14, 3, 10, 6] <-pivot i=0, j=2 swap
A = [5, 7, 9, 11, 12, 2, 14, 3, 10, 6] <-pivot i=0, j=3
A = [5, 7, 9, 11, 12, 2, 14, 3, 10, 6] <-pivot i=0, j=4
A = [5, 7, 9, 11, 12, 2, 14, 3, 10, 6] <-pivot i=1, j=5 swap
A = [5, 2, 9, 11, 12, 7, 14, 3, 10, 6] <-pivot i=1, j=6
A = [5, 2, 9, 11, 12, 7, 14, 3, 10, 6] <-pivot i=1, j=7
A = [5, 2, 9, 11, 12, 7, 14, 3, 10, 6] <-pivot i=1, j=8 swap
A = [5, 2, 3, 11, 12, 7, 14, 9, 10, 6] <-pivot i=2, j=8 1 compare
.
.
.
A = [5, 2, 3, 6, 12, 7, 14, 3, 10, 9] <-pivot i=3, j=10

AL = [5, 2, 3] <- pivot i=-1 j =0 2 compare
AL = [5, 2, 3] <- pivot i=-1 j =1 3 compare swap
AL = [2, 5, 3] <- pivot i=-0 j =1 swapped without comparing
AL =[2,3,5] 3 is now gone

3 compares for the element 3

(c) How many calls are made to random-int in (i) the worst case and (ii) the best case? Give your answers in asymptotic notation.

The worst case for a randomized quick sort would be if the randomized pivot always lands on either the largest or smallest number in the list, dividing each list into a one element smaller list. Meaning it would call **n - 1** times. The best case would be if the pivot divided the list perfectly in the middle. Meaning an array of size 5 would take 3 calls to sort, 11 would take 7, 23 would take 15 etc. This means the relationship would be **~ 2n/3**

2. (20 pts total) Use the Master Theorem to solve the following recurrence relations. For each recurrence, either give the asympotic solution using the Master Theorem (state which case), or else state the Master Theorem doesn't apply.

(a) T(n) = T((3n)/4) + 2

A = 1 b = 4/3 f(n) = 2

Case 3:

2 = Ω(n^(log(4/3)(1) - E) = Ω(n^(0-E) E = .000000001

1*f(3n/4) < c(f(n))

1*2 < c*2

C = 5

Works out!

T(n) = Θ(2) = Θ(1)

(b) T(n) = 3T(n/4) + nlgn

a= 3 b=4 f(n) = nlgn

Case 2:

nlgn=Θ(n^(log(4)(3))

nlgn=Θ(n^(.8)

c1n^.8<=nlgn<=c2n^.8

C1 = 1

C2 = 1000

Works out!

T(n) = Θ(nlgn)

(c) T(n) = 8T(n/3) + 2^n

a=8 b = 3 f(n) = 2^n

Case 3:

2^n = Ω(n^(log(3)(8) + E)

2^n = Ω(n^2)

log(3)(8) < 2

E = 2-log(3)(8)

therefore

2^n = Ω(2^n = Ω(n^(log(3)(8) + E) = Ω(n^2)

This part works!

8 * 2^(n/3) < = c*2^(n)

2^3 * 2(n/3) <= c*2^(n)

2^((n+9)/3) <= c* 2^(n)

C = .99

This works!

T(n) = Θ(2^n)

(d) T(n) = T(n/2) + T(n/4) + n^2

This recurrence is **not possible for the master theorem**. The general formula for the theorem is T(n) = a(Tn/b) + f(n). It does not cover if there are two different calls to T(n/b) as this problem has.

(e) T(n) = 100T(n/42) + lg n

a=100 b=42 f(n) = lgn

Case 1:

lg(n) = O(n^log(42)(100) - E)

lg(n) = O(n)

n^log(42)(100) > n

E = log(42)(100) - 1

lg(n) = O(n^log(42)(100) - E) = O(n)

T(n) = Θ(n^log(42)(100))

3. (30 pts total) Thormund the cleric has created n vials of holy water for an upcoming quest into a graveyard – little does he know, Grog has accidentally spilled ale into some of these vials. Harry saw Grog's mistake and tells Thormund he needs to identify which of vials of holy water are tainted. Together with Harry, Thormund has constructed a strange contraption that fits over two vials at a time to perform a test. When the contraption is activated, each vial glows one of two colors depending on whether the other vial is tainted or not. An untainted vial always glows correctly according to whether the other vial is accurate or not, but the glow of a tainted vial cannot be trusted. You quickly notice that there are four possible test outcomes:

vial i glows	*vial j glows*	
red	*red*	*at least one is tainted*
red	*green*	*at least one is tainted*
green	*red*	*at least one is tainted*
green	*green*	*both are untainted, or both tainted*

(a) Prove that if n/2 or more vials are tainted, Thormund cannot necessarily determine which vials are tainted using any strategy based on this kind of pairwise test. Assume a worst-case scenario in which the tainted vials are used, meaning the tainted vials always show the opposite color of the other vial.

(b) Consider the problem of finding a single good vial from among the n vials, and suppose Thormund knows that more than n/2 of the vials are untainted, but not which ones. Prove that [n/2] pairwise tests are sufficient to reduce the problem to one of nearly half the size.

If we know that more than n/2 viles are untainted then at most, n/2 -1 viles are tainted.
The most optimal way of figuring it out would be checking a tainted vile and a singular clean vile each iteration.

Example: " ' " means tainted [a, b', c, d] (3 clean, 1 tainted)
Viles Result
A,b' rg //means either a or b is tainted. Since we know n-2 -1 viles are tainted (1 in this case) Then we know c is an untainted vile (2 tests needed)

[a,b',c',d,e,f] (4 clean, 2 tainted)
Viles Result
A,b' gr //a or b is tainted
a,c' gr //a or c is tainted
A,d gg //a and d are clean or tainted, meaning that at least 2 viles tested have been tainted. So one of the remaining viles are clean.

This formula can be used on any value of n if at least (n/2) + 1 clean vials are known.

(c) Now, under the same assumptions as part (3b), prove that all of the untainted vials can be identified with Θ(n) pairwise tests. Give and solve the recurrence that describes the number of tests.

Since we know (n/2)-1 are tainted at most we can compare all elements with the first element and count the number of Green Green results

Example [a,b,c,d,e,f] (max of 2 tainted, 4 clean)
A,b = gg
A,c =gg // since a b and c are the same, that means they have to all be clean because we cant have >2 tainted.
A,d = gg
A,e = gg
A,f = gg
All clean are found. 6 tests

[a,b,c',d,e,f]
A,b = gg //1
A,c = rg
C,b = rg
D,a = gg//2
D,b = gg //a,b,d are clean meaning c is tainted
D,e = gg/ / e is clean
D,f = gg//f is clean
All viles are known 7 tests, worst case possible.

T(2) = 0 test
T(3) = 1 test
T(4) = 4

T(n) = T(n-1) +1 for n>2
0 otherwise

T(r) = T(n-r) + r-1

T(n) = n

4. (20 pts total) Harry needs your help breaking into a dwarven lock box. The lock box projects an array A consisting of n integers A[1], A[2], . . . , A[n] and has you enter in a two-dimensional n×n array B – to open the box – in which B[i, j] (for i < j) contains the sum of array elements A[i] through A[j], i.e., B[i, j] = A[i]+A[i+1]+· · ·+A[j]. (The value of array element B[i, j] is left unspecified whenever i ≥ j, so it doesn't matter what the output is for these values.) Normally Harry would do the computations himself, but the lock box changes the input array after a few minutes, thus Harry needs a fast way to solve this problem. Harry suggests the following simple algorithm to solve this problem:

```
dwarvenLockBox(A) {
        for i = 1 to n {
                for j = i+1 to n {
                        s = sum(A[i..j]) // look very closely here
                        B[i,j] = s
}}}
```

(a) For some function g that you should choose, give a bound of the form Ω(g(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).
g(n) = n^3

(b) For this same function g, show that the running time of the algorithm on an input of size n is also O(g(n)). (This shows an asymptotically tight bound of Θ(g(n)) on the running time.)
g(n) = n^3, for this algorithm, it will take n iterations for each for loop so that means it will at least take n^2 time. Also, the inner for loop calculates the sum of an array, if i = 1 and j = n, then the sum will be calculated n times as well in the worst case. n^2 n times = n^3. **O(n^3)**

Since both Ω(n^3) and O(n^3) then **T(n) = Θ(n^3)** as well.

(c) Although Harry's algorithm is a 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 an algorithm that solves this problem in time O(g(n)/n) (asymptotically faster) and prove its correctness.

If you use A[1,2…,5] the result of the main algorithm is

[0,3,6,10,15]
[0,0,5,9,14]
[0,0,0,7,12]
[0,0,0,0,9]
[0,0,0,0,0]

The relationship found was, for every increase in j it follows the SUM(K) = n(n+1)/2 relation. The tricky part was trying to figure out how to decrease that value as i increases. It turns out the value of B[i][j] decreases by the same formula as i increases K = n(n+1)/2 just 1 index behind for n

```
def DwarvenLockBoxEfficient(A):
    n = len(A)
    for i in range(0,len(A)):
        for j in range(0,len(A)):
            z = j +1
            h = i
            if i >= j:
                B[i][j] = 0
            else:
                B[i][j] = ((z * (z+1)) / 2) - ((h * (h+1)) / 2)
    return B
```

This has a runtime of **T(n) = O(n^2)** since there are two for loops both running n times.

Proof of Correctness:

Claim:

For an array A[1,2,3…,n] the two dimensional matrix B[n][n], B[i][j] = A[i] + A[i+1] + A[i+2] … A[j] for each iteration of i and j.

Initialization:

Before the loop we define n as being equal to the length of A and initialize B as size [n] [n] of all 0's. A is expected to be integers 1,2,3 … n.

Maintenance:

With each iteration we set a variable z = j +1 and h = i. This will help us with our assignment. A[i] + A[i+1] + A[i+2] … A[j] = (z*(z+1))/2 - ((h * (h+1))/2). We assign B[i][j] to that value. For each iteration of j the value of z increases by 1 and h stays the same. For each iteration of i, h increases by 1. For any iteration in which i >=j that value will = 0.

Termination:

As this loop ends i and j = n. B[n][n] = 0 and B[0,1,...n] [0,1,...n] = A[i] + A[i+1] + … + A[n].

5. (20 pts total) Consider the following strategy for choosing a pivot element for the Partition subroutine of QuickSort, applied to an array A.

• Let n be the number of elements of the array A.
• If n ≤ 24, perform an Insertion Sort of A and return.
• Otherwise:
– Choose 2[n ^½] elements at random from n; let S be the new list with the chosen elements.
– Sort the list S using Insertion Sort and use the median m of S as a pivot element.
– Partition using m as a pivot. – Carry out QuickSort recursively on the two parts.

(a) How much time does it take to sort S and find its median? Give a Θ bound
Selection sort for
The size of S would be 2[n^½] the average time complexity of insertion sort is O(n^2), so since the size of n is the square root of the original n, then the average for the sort would be Θ(n). Finding the median will be which is O(n) T(n) = Θ(n) + Θ(n) = Θ(n)
(b) If the element m obtained as the median of S is used as the pivot, what can we say about the sizes of the two partitions of the array A?
For the worst case, S would contain the top sqrt(n) elements. When sorted, the median would be the lowest of those sqrt(n). This would mean the pivot would partion A into two arrays, one sqrt(n) the other would be n-sqrt(n). (both of those -1 if excluding the pivot element).

Example:
A : [1,4,2,5,9,7,3,6,8]

S : [1,2,6,**7,8,9**] (7,8,9 are the highest elements of A)
M (i.e. pivot) = 7

A1 = [2,3,4,6,1,5] n - sqrt(n) -1
A2 = [9,8] sqrt(n)-1

(c) Write a recurrence relation for the worst case running time of QuickSort with this pivoting strategy.
It would take O(2n^(½)) time to select the units at random = O(n).
Getting the median of 2n^(½) = O(n) + O(n^½) + O(2n^½) = O(n)
After partitioning we know the worst case will get a max of n^(½) elements and the remaining elements on the other side. T(n) = T(n-(n^½)).
The partitioning step takes O(n) + O(n) = O(n).

So the recurrence relation = **T(n) = T(n-(n^½)) + O(n)**