CSCI 3104 Spring 2018 Problem Set 3
Cole Schlisner (02/08)

Problem 1

(5 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 3.

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

The running time is equivalent to the worst-case running time of QuickSort, which is $O(n^2)$.

This is due to the following line in the Partition function:

```
if A[j]<=x {
```

Which implies that an element will be swapped with another element if is less than **or equal to** the pivot. Since every element is the same, all elements are equal to the pivot and this line will always evaluate to True. This means the maximum amount of swaps will be made, which is equivalent to the worst case input.

(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?*

After the first call to partition(), 3 is put into the first half of the array, because it is less than or equal to 6 (comparison 1). It is then chosen as the pivot for the next iteration (of a size 3 array) so there are 2 more comparisons to it - meaning the total comparisons to 3 is 3.

The following is a list of every comparison made (along with the values being compared). If a comparison with 3 was made, the output line is denoted by a "*". The code I used to generate this is included in Appendix 1.

```
[9, 7, 5, 11, 12, 2, 14, 3, 10, 6] a[j]= 9 x= 6
[9, 7, 5, 11, 12, 2, 14, 3, 10, 6] a[j]= 7 x= 6
[9, 7, 5, 11, 12, 2, 14, 3, 10, 6] a[j]= 5 x= 6
[5, 7, 9, 11, 12, 2, 14, 3, 10, 6] a[j]= 11 x= 6
[5, 7, 9, 11, 12, 2, 14, 3, 10, 6] a[j]= 12 x= 6
[5, 7, 9, 11, 12, 2, 14, 3, 10, 6] a[j]= 2 x= 6
[5, 2, 9, 11, 12, 7, 14, 3, 10, 6] a[j]= 14 x= 6
[5, 2, 9, 11, 12, 7, 14, 3, 10, 6] a[j]= 3 x= 6 *
[5, 2, 3, 11, 12, 7, 14, 9, 10, 6] a[j]= 10 x= 6
[5, 2, 3] a[j]= 5 x= 3 *
[5, 2, 3] a[j]= 2 x= 3 *
[12, 7, 14, 9, 10, 11] a[j]= 12 x= 11
[12, 7, 14, 9, 10, 11] a[j]= 7 x= 11
[7, 12, 14, 9, 10, 11] a[j]= 14 x= 11
[7, 12, 14, 9, 10, 11] a[j]= 9 x= 11
[7, 9, 14, 12, 10, 11] a[j]= 10 x= 11
[7, 9, 10] a[j]= 7 x= 10
[7, 9, 10] a[j]= 9 x= 10
[7, 9] a[j]= 7 x= 9
[14, 12] a[j]= 14 x= 12
[2, 3, 5, 6, 7, 9, 10, 11, 12, 14]
```

(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.*

(i) If we just happened to randomly select the lowest (or highest) element each time, then that would be equivalent to calling QuickSort on a sorted array (i.e. the worst case). In this case, we only shave off 1 element of the array during each recursion, resulting in a recursion tree of depth $\Theta(n)$. This means `random-int` would be called $\Theta(n)$ times.

(ii) In the best case, QuickSort will divide the problem in half every recursion, resulting in a balanced binary tree with depth $\Theta(log_2(n))$. This implies `random-int` would be called $\Theta(log_2(n))$ times.

Problem 2

(30 pts total) Professor Trelawney has acquired ***n*** *enchanted crystal balls, of dubious origin and dubious reliability. Trelawney needs your help to identify which crystal balls are accurate and which are inaccurate. She has constructed a strange contraption that fits over two crystal balls at a time to perform a test. When the contraption is activated, each crystal ball glows one of two colors depending on whether the* ***other*** *crystal ball is accurate or not. An accurate crystal ball always glows correctly according to whether the other crystal ball is accurate or not, but the glow of an inaccurate crystal ball cannot be trusted. You quickly notice that there are four possible test outcomes:*

crystal ball i glows	crystal ball j glows		
red	red	$\Rightarrow$	at least one is inaccurate
red	green	$\Rightarrow$	at least one is inaccurate
green	red	$\Rightarrow$	at least one is inaccurate
green	green	$\Rightarrow$	both accurate, or both inaccurate

(a) *Prove that if $n/2$ or more crystal balls are inaccurate, Trelawney cannot necessarily determine which crystal balls are accurate using any strategy based on this kind of pairwise test. Assume a worst-case scenario in which the inaccurate crystal balls contain malicious spirits that collectively conspire to fool Trelawney.*

Suppose n = 4, and balls 1 and 2 are innacurate while balls 3 and 4 are accurate. In the worst case, the inaccurate balls would conspire to make the accurate balls seem inaccurate. That is, if the pairwise tests were put into a table (where G=green, R=red):

	1	2	3	4
1		GG	RR	RR
2	GG		RR	RR
3	RR	RR		GG
4	RR	RR	GG	

Thus there would be no way to distinguish which set of balls is accurate and which are inaccurate, or even if there are any accurate balls at all.

(b) *Suppose Trelawney knows that more than $n/2$ of the crystal balls are accurate, but not which ones. Prove that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.*

Take for example the case of n=6, in which 4 balls are accurate (balls 3-6 in the table) and the remaining 2 (balls 1 and 2) are inaccurate. The worst case (in which the inaccurate balls mimic the accurate ones) would look like:

	1	2	3	4	5	6
1		GR	GR	GR	GR	GR
2	RG		GR	GR	GR	GR
3	RG	RG		GG	GG	GG
4	RG	RG	GG		GG	GG
5	RG	RG	GG	GG		GG
6	RG	RG	GG	GG	GG	

We can choose our $\lfloor n/2 \rfloor = 3$ comparisons in 2 different ways that would yield useful results. Either we (a) take one crystall ball, and compare it with 3 other balls in succession, or (b) arrange the balls in a queue and compare the 1st ball to the 2nd, the 2nd to the 3rd, and the 3rd to the 4th. The issue with (a) is that we are basically choosing comparisons from a single column/row, and run the risk of making comparisons along the 1st two columns/rows in the table, in which all results look the same and we have no useful data. However, in strategy (b): we are essentially running along the diagonal, so we can guarantee no more than $\lfloor n/2 \rfloor$ 'GR' results in our tests. (Note that 'GR' and 'RR' imply the same thing, so the color of the inaccurate balls makes no difference). There are then 3 basic scenarios: either the $\lfloor n/2 \rfloor$ comparisons contain all comparisons between inaccurate balls, all comparisons between accurate balls, or a mix. Each of these allow us to deduce the accuracy of the remaining balls, cutting the problem (nearly) in half. Additionally, if n is even then the first scenario will never occur as $\lfloor n/2 \rfloor < (n/2) + 1$
Successive comparisons for our n=6 example:

- 1 GR 2 GR 3 GG 4 $\Longrightarrow$ one or both of the first two balls is inaccurate, however if the third comparison was 'false' (that is, both balls are inaccurate) it would imply that both the 3rd and 4th balls are inaccurate. Since we know there are at most 2 inaccurate balls, we can conclude that the third comparison is 'true' and balls 3 and 4 are accurate. This implies that at least ball 2 is inaccurate. We now know the accuracy of 3 balls, and the problem is halved.
- 3 GG 4 GG 5 GG 6 $\Longrightarrow$ if any of these comparisons were 'false' then they would all need to be 'false', and balls 3-6 would need to be inaccurate. Since there are at most 2 inaccurate balls, all three comparisons are 'true' and we know the accuracy of balls 3-6 - the problem is now reduced to finding the accuracy of the remaining 2 balls.

If we had instead 5 balls/2 comparisons, and both comparisons resulted in GR (or RR) - we would know that the rest of the comparisons would need to result in GG. Using the logic in the second example above, we would be able to reduce the problem to the first two balls.

(c) *Now, under the same assumptions as part (2b), prove that all of the accurate crystal balls can be identified with $\Theta(n)$ pairwise tests. Give and solve the recurrence that describes the number of tests.*
Assuming a test takes $\Theta(1)$ time:

$$T(n) = T(n/2) + \lfloor n/2 \rfloor$$

Assuming n is a power of 2, T(1) = 0:
Unrolling:

$$\begin{aligned}
T(n) &= T(n/2) + \frac{n}{2} && (1)\\
&= (T(n/4) + \frac{n}{4}) + \frac{n}{2} && (2)\\
&= ((T(n/8) + \frac{n}{8}) + \frac{n}{4}) + \frac{n}{2} && (3)\\
... & && (4)\\
&= T(\frac{n}{2^i}) + \sum_{k=1}^{i} \frac{n}{2^k} && (5)\\
&= T(\frac{n}{2^i}) + n\sum_{k=1}^{i} \frac{1}{2^k} && (6)
\end{aligned}$$

Base Case: $\frac{n}{2^i} = 1 \implies i = lg(n)$
Given $i = lg(n)$:

$$\begin{aligned}
T(n) &= T(\frac{n}{2^i}) + n\sum_{k=1}^{i} \frac{1}{2^k} && (7)\\
&= T(1) + n\sum_{k=1}^{log(n)} \frac{1}{2^k} && (8)\\
&= 0 + nO(1) && (9)\\
&= \Theta(n) && (10)
\end{aligned}$$

Problem 3

(20 pts) Professor Dumbledore needs your help. He gives you an array A consisting of n integers $A[1], A[2] \dots, A[n]$ 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., $B[i,j] = A[i] + A[i+1] + \cdots + A[j]$. (The value of array element $B[i,j]$ is left unspecified whenever $i \geq j$, so it doesn't matter what the output is for these values.) Dumbledore suggests the following simple algorithm to solve this problem:

```
dumbledoreSolve(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 $\Omega(g(n))$ 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).*

Not accounting for the assignment operations, there are $j - i$ operations for each sum inside the nested for loop, so:

$$T(n) > \sum_{i=1}^{n} \sum_{j=i+1}^{n} (j - i)$$

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} (j - i) = \sum_{i=1}^{n} [\sum_{j=i+1}^{n} j - \sum_{j=i+1}^{n} i] \quad (11)$$

$$= \sum_{i=1}^{n} [\sum_{j=1}^{n-i} j - \sum_{j=1}^{n-i} i] \quad (12)$$

$$= \dots.. \quad (13)$$

$$= \frac{n(2 - 3n + n^2)}{6} \quad (14)$$

$$\lim_{n \to \infty} \frac{\frac{n(2-3n+n^2)}{6}}{n^3} = \frac{1}{6} \quad (15)$$

$$\therefore \sum_{i=1}^{n} \sum_{j=i+1}^{n} (j - i) = \Theta(n^3) \quad (16)$$

$$\therefore T(n) = \Omega(n^3) \quad (17)$$

(b) *For this same function g, show that the running time of the algorithm on an input of size n is also $\mathcal{O}(g(n))$. (This shows an asymptotically tight bound of $\Theta(g(n))$ on the running time.)*

The amount of times the sum() function on the inside of the nested for loop is executed is exactly:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

We also know that less than n comparsions are made during each execution of the sum() function (the amount decreases with each successive loop).

Therefore we can state the inequality:

$$T(n) < n\frac{n(n-1)}{2}$$

Now we can analyze:

$$T(n) < n\frac{n(n-1)}{2} = \frac{n^3 - n^2}{2} \tag{18}$$

$$\lim_{n\to\infty} \frac{\frac{n^3-n^2}{2}}{n^3} = \frac{1}{2} \tag{19}$$

$$\therefore n\frac{n(n-1)}{2} = \Theta(n^3) \tag{20}$$

$$\therefore T(n) = O(n^3) \tag{21}$$

(c) *Although Dumbledore'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 ineffciency. Give an algorithm that solves this problem in time $\mathcal{O}(g(n)/n)$ (asymptotically faster) and prove its correctness.*

$O(g(n)/) = O(\frac{n^3}{n}) = O(n^2)$

Pseudocode assuming 0-based indexing:

```
def sumarr(A):
  n = A.length
  m = [[]] // empty matrix
  for i in [0 .. A.length-1):
    m[i][i+1] = A[i] + A[i+1]
    for j in [i+2 .. A.length]:
      m[i][j] = m[i][j-1] + A[j]
  return m
```

Invariant: m[i][j] $= \sum_{k=i}^{j-1} m[i][k]$ for all i,j in [1..n] such that $i < j$ XOR i=n

Initialization: Before the outer loop enters, i=j=0 and the entire matrix is 0. Thus the invariant holds.

Maintenance: Before the second loop is entered, the first element $m_{i,i+j}$ is initialized to be A[i]+A[i+1]. This element is not the sum of the previous elements, however since the invariant states the sum starts at k=i, this element makes no difference to the invariant. After the second loop is entered, it simply increments j and adds A[j] to the previous element of m (in the same row). Since m[i][j-1] is already the sum of the elements A[i..j-1], adding A[j] to this quantity is equivalent to the sum of A[i..j]. This preserves the invariant, as it is essentially the definition of summation, and the inner loop exits when the end of the array is reached. Then i is incremented in the outer loop for the next row of the matrix and the invariant is preserved.

Termination: The outer loop exits when the end of the array is reached and i=n. Because i=n, there cannot be a j $\in$ A such that $i < j$, so the loop invariant holds.

References

1. CLRS
2. https://en.wikipedia.org/wiki/1/2_%2B_1/4_%2B_1/8_%2B_1/16_%2B_%E2%8B%AF

Appendix 1
Python 3 code for generating list in 1b:

```
def qs(a, p, r):
  if (p < r):
    q = part(a, p, r)
    qs(a, p, q-1)
    qs(a, q+1, r)

def part(a, p, r):
  x = a[r]
  i = p-1
  for j in range(p, r):
    print(a[p:r+1], "a[j]=", a[j], "x=",x, ("*" if (x==3 or a[j]==3) else ""))
    if a[j] <= x :
      i += 1
      exc(a, i, j)
  exc(a, i+1, r)
  return i+1

def exc(a, x, y):
  tmp = a[x]
  a[x] = a[y]
  a[y] = tmp

A = [9, 7, 5, 11, 12, 2, 14, 3, 10, 6]
qs(A, 0, len(A)-1)
print(A)
```