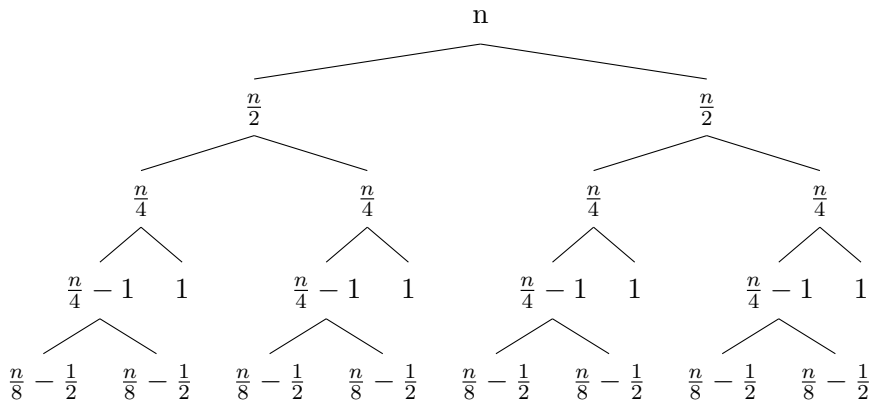
CSCI 3104 Spring 2018 Problem Set 4
Mohamed Al-Rasbi (07/25)

Problem 1

(10 pts) Suppose that we modify the `Partition` *algorithm in QuickSort in such a way that on every third level of the recursion tree it chooses the worst possible pivot, and on all other levels of the recursion tree* `Partition` *chooses the best possible pivot. Write down a recurrence relation for this version of QuickSort and give its asymptotic solution. Then, give a verbal explanation of how this* `Partition` *algorithm changes the running time of QuickSort.*



As you can see from the tree above, if we assume that n is the number of elements in an array. Then:
1. Level 2 is the base case
2. Level 3, 4 and 5 are the recurrence cases.

So, the recurrence relation is $T(n) = 4T(\frac{n}{4}) + 4T(\frac{n-4}{4}) + 8T(\frac{n-4}{8}) + \frac{n}{2}$
The running time is $\Theta(n \log n)$
I drew the tree like that because the children nodes of level 3 are from the worst case, so level 4 is unbalanced. The tree will continue like that and repeat the same thing for level 6, 9, 12, etc.
The running time is $\Theta(n \log n)$ because:
For example, if we have n = 10, then level2 = 5, level3 = 2, level4 = 1, and then the recursion will stop.
Therefore, if we have n elements in an array, then the depth if the tree is almost $\log n$, which is almost a balanced tree. And balanced tree of the recursions is the best case for quick sort, which has the running time $\Theta(n \log n)$. Also, the probability that we sort an array by quicksort and get running time of the best case is higher the the probability that we get the worst case.
Because of all that, we can say that the running time of this quick sort is $\Theta(n \log n)$.

Problem 2

(10 pts) Mr. Ollivander, of Ollivanders wand shop, has hired you as his assistant, to find the most powerful wand in the store. You are given a magical scale which "weighs" wands by how powerful they are (the scale dips lower for the wand which is more powerful). You are given n wands $W_1, \ldots, W_n$, each having distinct levels of power (no two are exactly equal).

(a) *Consider the following algorithm to find the most powerful wand:*

i. *Divide the n wands into $\frac{n}{2}$ pairs of wands.*

ii. *Compare each wand with its pair, and retain the more powerful of the two.*

iii. *Repeat this process until just one wand remains.*

Illustrate the comparisons that the algorithm will do for the following n = 8 input:

$$W_1 : \frac{3}{2},\ W_2 : \frac{5}{2},\ W_3 : \frac{1}{2},\ W_4 : 1,\ W_5 : 2,\ W_6 : \frac{5}{4},\ W_7 : \frac{1}{4}\ , W_8 : \frac{9}{4}$$

First we have:

$$(W_1 : \frac{3}{2},\ W_2 : \frac{5}{2},\ W_3 : \frac{1}{2},\ W_4 : 1,\ W_5 : 2,\ W_6 : \frac{5}{4},\ W_7 : \frac{1}{4}\ , W_8 : \frac{9}{4})$$

Then:

$$(W_1 : \frac{3}{2},\ W_2 : \frac{5}{2}),\ (W_3 : \frac{1}{2},\ W_4 : 1),\ (W_5 : 2,\ W_6 : \frac{5}{4}),\ (W_7 : \frac{1}{4}\ , W_8 : \frac{9}{4})$$

$$(W_2 : \frac{5}{2},\ W_4 : 1),\ (W_5 : 2,\ W_8 : \frac{9}{4})$$

$$(W_2 : \frac{5}{2},\ W_8 : \frac{9}{4})$$

$$(W_2 : \frac{5}{2})$$

(b) *Show that for n wands, the algorithm (2a) uses at most n comparisons.*

As we can see from the illustration above, the number of wands is halved in each round. In the second round it is n/2, then n/4, n/8, etc, until we reach $n/2^m$ which is going to be either 1/2 (in case we only have one wand) or 1 (in case we have 2 or more wands). So, the the number of rounds of comparisons is $m = \log_2 n$. Therefore, we can say that the number of comparisons is:

$(n/2 + n/4 + n/8 + ...) \leq n(1/2 + 1/4 + 1/8 + ...) \leq n(\frac{1/2}{1 - 1/2}) \leq n$

(c) *Describe an algorithm that uses the results of (2a) to find the second most powerful wand, using at most $log_2 n$ additional comparisons. There is no need for pseudocode; just write out the steps of the algorithm like we have written in (2a). Hint: if you follow sports, especially wrestling, read about the* ***repechage****.*

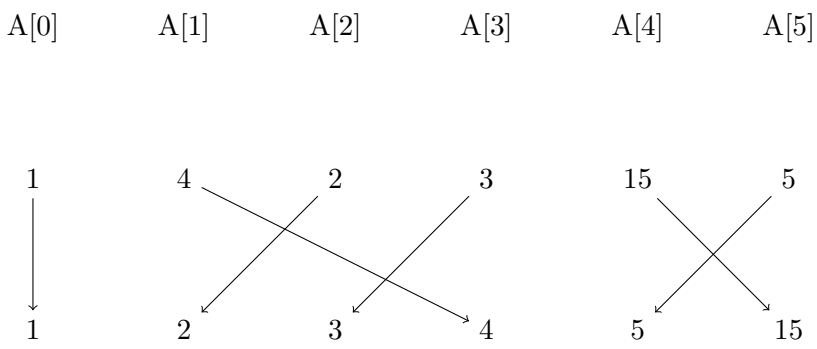
As I showed in part b, we have $\log_2 n$ rounds of comparisons. So what we can do is, take all the wands that was compared to the most powerful wand (in part a, they are W_1, W_4, W_8) and make comparisons between them. And in the example above, we will do 2 more comparisons, which is less than $\log_2 8 = 3$.

(d) *Show the additional comparisons that your algorithm in (2c) will perform for the input given in (2a).*

We will compare W_1 and W_4, W_1 is more powerful. Then will compare W_1 and W_8, W_8 is more powerful. Hence, W_8 is the second most powerful wand after W_2.

Problem 3

(20 pts) For obtuse historical reasons, Prof. Dumbledore asks his students to line up in ascending order by height in a very tight room with little extra space. Similar to Alex the African Grey parrot (look it up!), the students, being bored, decided to play a little trick on Prof. Dumbledore. They lined up in order by height – almost. They made sure that each person was no more than k positions away from where they were supposed to be (in ascending order), but this allowed them to significantly mess up the precise ordering. Here is an example of an array with this property when $k = 2$:



(a) *Write down pseudocode for an algorithm that would sort such an array in place–so it fits in the tight room–in time $\mathcal{O}(nk \log k)$. Your algorithm can use a function $sort(A, l, r)$ that sorts the subarray $A[l], \ldots, A[r]$ in place in $\mathcal{O}((r-l)\log(r-l))$ steps (assuming $r > l$).*

```
def sortArr(A, k):
    length = len(k)
    for i in range(0, length-k):
        sort(A, i, i+k)
    return A
```

The run time of `sort(A, i, i+k)` in each iteration is $O(k \log k)$, since $k = r-l = (k+i)-i$. And the for loop iterates through all the elements of the array, which is n. That means the run time of the function is $O(nk \log k)$

(b) *Suppose you are given to an auxiliary room which can fit k + 1 students. Modify your previous algorithm to sort the given array in time $\mathcal{O}(nk)$.*

```
def sortArr2(A, k):
    for i in range(0, k):
        for j in range(i+1, k+1):
            if A[j] < A[i]:
             tmp = A[i]
             A[i] = A[j]
             A[j] = tmp
```

```
return A
```

The run time we seek is $O(nk) = O(n(n-1)) = O(k(k+1))$. The outer loop starts from 0 to k, and the inner loop starts from i+1 to k+1. This means that the run time is $[k + (k-1) + (k-2) + ... + (k-(k-1))] \leq O(k^2) \leq O(k(k+1)) = O(nk)$

(c) *With the same extra room as in the previous part, modify heap sort using a binary min heap of size $k+1$ to sort the given array in time $\mathcal{O}(n \log k)$.*

```
def heapSort(arr, k):
    n = k+1

    for i in range(n, -1, -1):
        heapify(arr, n, i) #create binary min heap

    # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]   # swap
        heapify(arr, i, 0)
```

I copied the function from here (`https://www.geeksforgeeks.org/heap-sort/`) and modified it

(d) *(5 pts extra credit) Include the correct story about Alex, with proper citation. If you wish, you may copy this story verbatim, but must indicate clearly that you have done so and, of course, still cite your source.*

This is a video for Alex's story from abc news. The brief story is that Alex was a smart parrot and died at the age of 31.
`https://www.youtube.com/watch?v=7yGOgs_UlEc`
I assume the story is correct since it is from a trusted source (abc news).

Problem 4

(20 pts) 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 $n \leq 24$, perform an Insertion Sort of A and return..*
- *Otherwise:*
  - *Choose $2\lfloor n^{1/3} \rfloor$ 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 its median? Give a Θ bound.*

The running time to sort S is $\Theta((2n^{1/3})^2) = \Theta(n^{2/3})$
And finding the median in a sorted array is a constant time.

(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?*

The median m has at least $n^{1/3}$ elements less than or equal to it, and $n^{1/3}$ elements greater than or equal to it.
Therefore, the sizes of the two partitions of A will be: at least $n^{1/3}$ elements on one partition and $n - n^{1/3}$ on the other partition.

(c) *Write a recurrence relation for the worst case running time of QuickSort with this pivoting strategy.*

The recurrence realtion for the worst case is:
For $n \leq 24$: $T(n) = C_1$
For $n > 24$: $T(n) = T(n1/3) + T(n - n1/3) + C_2 n$

Problem 5

(20 pts extra credit) Recall that the Insertion Sort algorithm (Chapter 2.1 of CLRS) is an in-place sorting algorithm that takes $\Theta(n^2)$ time and $\Theta(n)$ space. In this problem, you will learn how to ***instrument*** *your code and how to perform a numerical experiment that the asymptotic analysis of Insertion Sort's running time. There are two functions and one experiment to do this.*

(i) `InsertionSort(A,n)` *takes as input an unordered array A, of length n, and returns both an in-place sorted version of A and a count t of the number of atomic operations performed by* `InsertionSort`*.*

*Recall: atomic operations include mathematical operations like -, +, *, and /, assignment operations like $\leftarrow$ and =, comparison operations like $<$, $>$, and ==, and* `RAM` *indexing or referencing operations like [].*

(ii) `randomArray(n)` *takes as input an integer n and returns an array A such that for each $0 \leq i < n$, A[i] is a uniformly random integer between 1 and n. (It is okay if A is a random permutation of the first n positive integers; see the end of Chapter 5.3.)*

(a) *From scratch, implement the functions* `InsertionSort` *and* `randomArray`*. You may not use any library functions that make their implementation trivial. You may use a library function that implements a pseudorandom number generator in order to implement* `randomArray`*.*

Submit a paragraph that explains how you instrumented `InsertionSort`*, i.e., explain which operations you counted and why these are the correct ones to count.*

Hint: your instrument code should only count the operations of the `InsertionSort` *algorithm and not the operations of the instrument code you added to it.*

```
 1 import random
 2
 3 def randomArray(n):
 4         A = []
 5         for i in range(n):
 6                 A.append(random.randint(1, n))
 7
 8         return A
 9
10 def InsertionSort(A, n):
11         t = 0
12         for i in range(1, n):
13                 elem = A[i]
14                 j = i - 1
15                 while(j >= 0 and elem < A[j]):
16                         A[j+1] = A[j]
```

```
17                          j -= 1
18                          t += 9
19
20                  A[j+1] = elem
21                  t += 9
22
23          return (A, t)
```

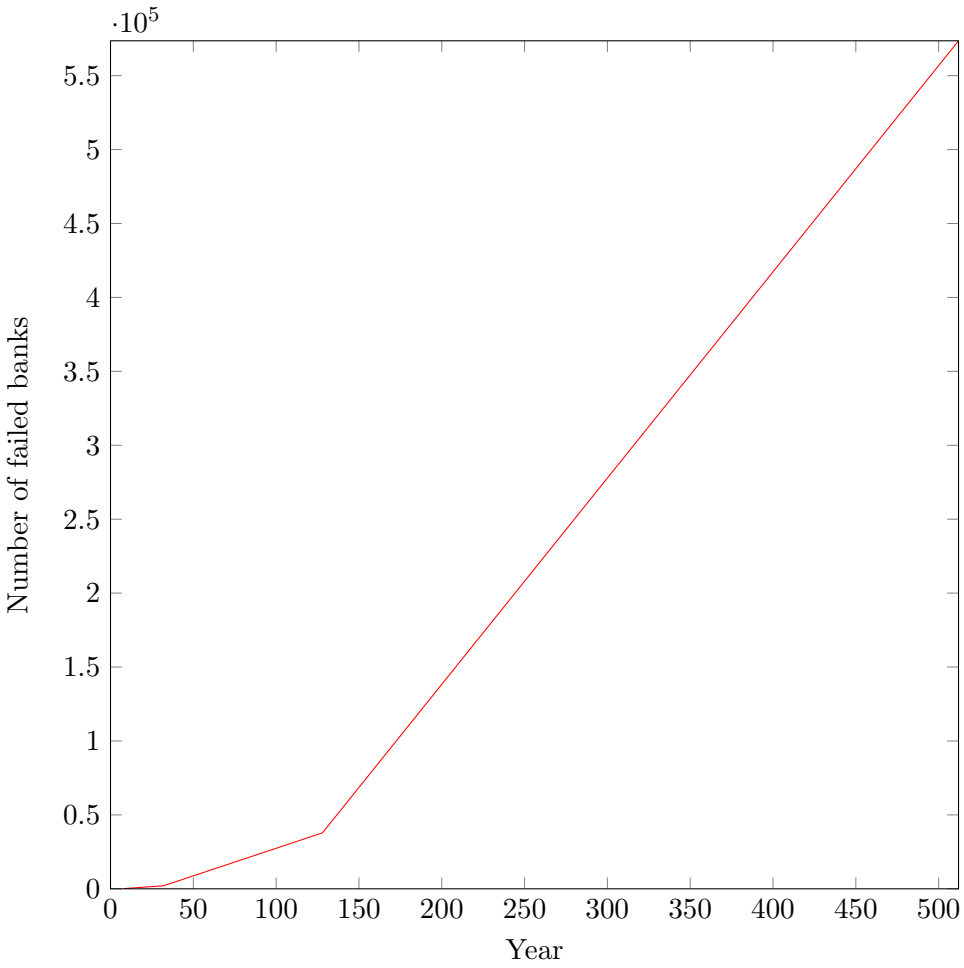
I will list each line number and why I counted the operations on that line.
`line 12:` in the for loop, we assign the value to i and compare if it is in range(n). That is t += 2
`line 13:` I index the element from the array A and assign it to the variable elem. I did that because if I didn't, in line 16 where we use the variable elem, I would have assigned and indexed element from the array in each iteration in the inner loop. This is t += 2.
`line 14:` assign i - 1 to j to keep track of the elements in the array. That is t += 2
`line 20:` These are tree operations. Assign, increment and indexing. t+=3
`This is why we have t +=9 in line 21`

`line 15:` Three operations. $\geq$ and $<$ and indexing. t += 3
`line 16:` Four operations. Indexing twice, assign, and increment. t += 4
`line 17:` Two operations. Assign and decrement. t += 2
`This is why we have t +=9 in line 18`

All these operation are necessary in insertion sort and we can't remove any of them without adding another one. This is why I have to count all of them.

(b) *For each of* $n = 2^3, 2^5, \ldots, 2^{15}, 2^{16}$*, run* ***`InsertionSort(randomArray(n),n)`*** *five times and record the tuple* $(n, \langle t \rangle)$*, where* $\langle t \rangle$ *is the average number of operations your function counted over the five repetitions. Use whatever software you like to make a line plot of these 12 data points; overlay on your data a function of the form* $T(n) = An^2$*, where you choose the constant* A *so that the function is close to your data.*
Hint: To increase the aesthetics, use a log-log plot. These are the values I got, n is length of array, t is number of operations:

```
n=8 t=63
n=32 t=2133
n=1024 t=2306340
n=32768 t=2417314527
n=65536 t=9657067032
```



Using these values, I can get A from $T(n) = An^2$

```
n=2^3 t=144
n=2^6 t=8388
n=2^9 t=589851
n=2^12 t=37513746
```

$\frac{8388}{144} = 58$

$\frac{589851}{8388} = 70$

$\frac{37513746}{589851} = 63$

By taking the average $\frac{63 + 70 + 58}{3}$, then $A = 63$

I worked with:
Omar Mohammed, Mahmoud Almansouri, Firas Al Mahrouky, Yazeed Almuqwishi, Absdullah Bajkhaif