CS 161 **Problem Set 3**
Spring 2017 **Due:** April 28, 2017, 3pm

Please answer each of the following problems. Refer to the course webpage for the **collaboration policy,** as well as for **helpful advice** for how to write up your solutions.

Note: For all problems, if you include pseudocode in your solution, please also include a brief English description of what the pseudocode does.

1. Suppose that p is an unknown value, $0 < p < 1$. Suppose that you can call a function `randP` which returns `true` with probability p and returns `false` with probability $1 - p$. Every call to `randP` is independent. You have no way to generate random numbers except through `randP`.

   (a) (2 pts) Describe an algorithm—using `randP`—that returns `true` with probability 1/2 and `false` with probability 1/2. Your algorithm should in expectation, use $\frac{1}{p(1-p)}$ calls to `randP`. Your algorithm does not have access to the value of p, and does not have access to any source of randomness other than calls to `randP`. **[We are expecting pseudocode, and a short English description of what the algorithm does. Your pseudocode should be detailed enough that a CS106B student could implement it without much thought.]** Hints: (i) Your algorithm does not have to compute p, or an approximation to it. (ii) Notice that in the worst case, your algorithm may use more calls to `randP`, possibly even infinitely many.

```
def rand():
    p1 = randP()
    p2 = randP()
    while p1 != p2:
        p1 = randP()
        p2 = randP()
    return p1
```

   (b) (1pts) Formally prove that your algorithm runs using expected $\frac{1}{p(1-p)}$ calls to `randP`. **[We are expecting a mathematical calculation of the expected value of the total number of calls to `randP`.]**

   The number of 'rounds' is a geometric random variable. Each round has two calls to `randP`, and succeeds (i.e., returns) with probability $1 - (p^2 + (1-p)^2)$, because it goes until they aren't both the same. The expected value of a geometric variable is $\frac{1}{n}$, where n is the probability of success. Thus, this is an expected $\frac{1}{1-(p^2+(1-p)^2)}$ rounds. We can simplify this to $\frac{1}{2p-2p^2} = \frac{1}{2p(1-p)}$. Now since what we want is the expected number of `randP` calls and not the expected number of rounds, we must multiply by 2, because that is the number of `randP` calls per round. Thus, we get $\frac{1}{p(1-p)}$.

   (c) (1 pt) Informally argue that your algorithm returns `true` with probability 1/2 and `false` with probability 1/2. **[We are expecting an informal justification of why the algorithm returns `true` with probability 1/2 and `false` with probability**

1/2. **Your argument should convince the reader that you are correct, but does not have to be a formal proof.**]

We will calculate P(`randP` returns true | `randP` returns). If we know `randP` returns, we know that $p1$ and $p2$ are different. We will show that $P(p1 = \text{true}) = P(p1 = \text{false}) = 1/2$. We will work with the reduced sample space of { ($p1$ = true, $p2$ = false), ($p1$ = false, $p2$ = true) }. $P(p1 = \text{true and } p2 = \text{false}) = p(1-p)$. $P(p1 = \text{false and } p2 = \text{true}) = (1-p)p$. Now since we are working with the reduced sample space, $P(p1 = \text{true and } p2 = \text{false}) + P(p1 = \text{false and } p2 = \text{true}) = 1$. Thus, since they are both equal probabilities, they are both 1/2.

2. Suppose that A and B are sorted arrays of length n, and that all numbers in the arrays are distinct.

   (a) (3pts) Design an algorithm to find the median of all $2n$ numbers in $O(\log n)$ time. For our purposes, we define the median of the $2n$ numbers as the nth smallest number in the $2n$ values. [**We are expecting: pseudocode, and an English description of the algorithm, detailed enough that a CS 106B student could implement it without much thought.**]

```
def find_median(A, B):
    n = A.len
    if n == 1:
        return min(A[1], B[1])
    if n == 2:
        C = MergeSort(A + B)
        return C[2]
    medianA = (A[ceil(n/2)] + A[floor(n/2)])/2
    medianB = (B[ceil(n/2)] + B[floor(n/2)])/2
    if medianA == medianB:
        return medianA
    if medianA < medianB:
        return find_median(A[ceil(n/2)...n], B[1...floor(n/2)])
    return find_median(A[1...floor(n/2)], B[ceil(n/2)...n])
```

   (b) (3pts) Informally argue that your algorithm correctly finds the median of all $2n$ numbers. [**We are expecting a short (paragraph or two) argument that will convince the reader why your algorithm works correctly.**]

   First, we will show that the base case works. If $n = 1$, then the n-th smallest element would be the smallest element, and thus we return the minimum of the two elements left. If $n = 2$, then we are returning the second smallest element, thus we can mergesort the array and return the second smallest element.

   Now we will show that the recursive part works. Without loss of generality, suppose the the median of A is smaller than the median of B. Now we are guaranteed that $\lceil \frac{n}{2} \rceil - 1$ elements in A are smaller than the median of A. The elements smaller than B's median could also possibly be smaller than the median of A, thus adding at most

another $\lceil \frac{n}{2} \rceil - 1$. Within B, we are guaranteed that $\lceil \frac{n}{2} \rceil - 1$ are smaller than the median of B. Within A, we are guaranteed that at least $\lceil \frac{n}{2} \rceil$ elements are smaller than the median of B. Overall, we know that at most $2 \cdot \lceil \frac{n}{2} \rceil - 1$ elements are smaller than the median of A, and that $2 \cdot \lceil \frac{n}{2} \rceil - 1$ elements are smaller than the median of B. This guarantees that at most $n - 1$ elements are smaller than the median of A, and that at least $n - 1$ elements are smaller than the median of B. Thus, since we define median as having $n - 1$ elements smaller than it, the median must be greater than or equal to the median of A, and less than or equal to the median of B. Then, we can recurse on the upper half of A and the lower half of B (where A is the array with the lower median).

(c) (2pts) Prove that your algorithm runs in time $O(\log(n))$ time. **[We are expecting a formal proof.]**

At each iteration, we select half of each array, creating a subproblem of half the size. Given m total elements, the subproblem will always be of at most $\frac{m}{2} + 1$ elements. This is because if the arrays each have an odd length, then it will include one extra element. Since we do a constant amount of work at each iteration, this leaves us with a recurrence relation of $T(m) = T(\frac{m}{2}) + O(1)$. Using the Master Theorem, this is $O(m)$.

3. Suppose you want to sort an array A of n numbers (not necessarily distinct), and you are guaranteed that all the numbers in the array are in the set $\{1, \ldots, k\}$. A **"20-question sorting algorithm"** is any deterministic algorithm that asks a series of YES/NO questions (not necessarily 20 of them, that's just a name) about A, and then *writes down* the elements of A in sorted order. (Specifically, the algorithm does not need to rearrange the elements of A, it can just write down the sorted numbers in a separate location).

   Note that there are many YES/NO questions beyond just comparison-questions—for example, the following are also valid YES/NO questions: "If I ignored A[3] and A[17] would the array be sorted?" and "Did it rain today?"

   (a) (2 pts) Describe a 20-question sorting algorithm that will, for every input, ask only $O(k \log n)$ questions. Feel free to assume that the algorithm is also told n and k, although this isn't necessary. [Hint: If you are stuck, first think about how you would do this with $\log n$ questions if $k = 2$. What would you need to know about the array to write down the sorted list of elements?] **[We are expecting a description of the algorithm and an informal (1-paragraph) argument that it achieves the desired runtime.]**

```
def sort(buckets, k, n):
    for 1 to k:
        num_in_bucket = binary_search_for_bucket_count()
```

   For each of the k buckets, we will do a "binary search" for the count of elements in that bucket. We can start by asking if bucket k has more elements than $\frac{n}{2}$. If so, we would try $\frac{3n}{4}$. If not, we would try $\frac{n}{4}$. Either way, for each bucket this leads to a binary search that takes time $\log(n)$. Thus the total runtime would be $k \log(n)$.

(b) (2 pts) Prove that for *every* 20-question sorting algorithm, there is some array A consisting of n integers between 1 and k that will require $\Omega(k \log \frac{n}{k})$ questions, provided $k \leq n$. [Hints: Why is it sufficient for this problem to lower-bound the number of ordered arrays, instead of counting exactly? Once you have understood this, use a counting argument: how can you lower-bound the number of ordered arrays are there that consist of n integers $\{1, \ldots, k\}$ (not necessarily distinct)? There are a number of ways to do this; we suggest you do NOT use Stirling's approximation: you don't need this in order to prove the result, and it will be complicated.] **[We are expecting a mathematically rigorous proof (which does NOT necessarily mean something long and tedious).]**

Each distinct sorted array must come from a different path. This is because given two arrays as input, if they follow the same path in a decision tree, then they must end at different places or else the algorithm won't know which sorted array to return. Given n and k, the number of sorted arrays is $\binom{n+k-1}{n}$. This is using the buckets formula from CS109. Now to get that many options in our leaves of our decision tree, we need $\log(\binom{n+k-1}{n})$ questions. $\binom{n+k-1}{n}$ is equivalent to $\frac{(n+k-1)!}{n!(k-1)!}$. Then we can cancel out the $n!$ on the bottom and have $\frac{(n+k-1)(n+k-2)\ldots(n+1)}{(k-1)!}$. Now we can find a lower bound of this logarithm, by estimating the top and bottom of the denominator. We estimate with $\log(\frac{(n+1)^{k-2}}{k^k})$. Then we can simplify to see that we are left with $k\log(n+1) - 2\log(n+1) - k\log(k)$. This simplifies to order $k\log(n/k)$, which is what we needed to show.

4. Suppose that on your computer you have stored n password-protected files, each with a unique password. You've written down all of these n passwords, but you do not know which password unlocks which file. You've put these files into an array F and their passwords into an array P in an arbitrary order (so $P[i]$ does not necessarily unlock $F[i]$). If you test password $P[i]$ on file $F[j]$, one of three things will happen:

   1) $P[i]$ unlocks $F[j]$
   2) The computer tells you that $P[i]$ is lexicographically smaller than $F[j]$'s true password
   3) The computer tells you that $P[i]$ is lexicographically greater than $F[j]$'s true password

   You **cannot** test whether a password is lexicographically smaller or greater than another password, and you **cannot** test whether a file's password is lexicographically smaller or greater than another file's password.

   (a) (3pts) Design an randomized algorithm to match each file to its password, which runs in expected runtime $O(n\log(n))$. **[We are expecting: pseudocode and/or an English description of an algorithm.]**

```
def match_passwords(files, passwords):
    if files.len == 0:
        return
    i = random(1, passwords.len)
    rand_password = passwords[i]
```

```
right_files = []
right_passwords = []
left_files = []
left_passwords = []
matched_file = None
for file in files:
    if test(rand_password, file) == LOWER:
        left_passwords += rand_password
    elif test(rand_password, file) == HIGHER:
        right_passwords += rand_password
    else:
        matched_file = file
for password in passwords:
    if test(password, matched_file) == LOWER:
        right_files += matched_file
    if test(password, matched_file) == HIGHER:
        left_files += matched_file
match_passwords(right_files, right_passwords)
match_passwords(left_files, left_passwords)
```

Choose a random password, and then test every file on it. Partition the passwords into two sections: lexicographically greater than the real password and lexicographically smaller than the real password. Once we find the real password, now we test every file against that password. We again partition the files into the two groups. Once we have done that, we recurse on each side. Assuming the lexicographic tester is consistent, the files and passwords in the recursive parts should all match up perfectly. The base case is if there is one password and one file, then they will automatically match up.

(b) (2pts) Explain why your algorithm is correct. [**We are expecting: an informal argument (a paragraph or so) about why your algorithm is correct, which is enough to convince the reader/grader. You may also submit a formal proof if you prefer.**]

If the lexicographic categorizer is correct, then this algorithm is also correct. For a given password, if it has n other passwords greater than it, then it will need n files greater than the file it matches to match each of the passwords greater than it. At each level of recursion, we partition the files and the passwords and take out the password that matches. The base case is somewhat integrated into the actual function, and it's pretty much when the only file left matches the only password, because then that one gets taken out and there are none left.

(c) (2pts) Analyze the running time of your algorithm, and show that it runs in expected runtime $O(n \log(n))$. [**We are expecting: a formal analysis of the runtime.**]

I will show that the expected number of steps is the same asymptotically as that of quicksort. For quicksort, we take order n time to partition the array randomly, splitting the array into two subarrays of sizes $n-k$ and k. For our password algorithm,

we take order n time to partition the array randomly, but splitting the array into two subarrays of sizes $n-k$ and $k-1$. Since the same amount of work is done at each level (order n) in both quicksort and our password algorithm, and since our password algorithm splits the arrays into sections of size at most what quicksort would, we can put an upper bound to the expected runtime of $O(n\log(n))$. Of course, this is assuming the amount of work done at each level for our password program actually is order n. It is, since all we do is iterate through our list of passwords and files once each. Thus $2n$ comparisons.

5. (6pts, 1 pt per part) In the `select` algorithm from class, in order to find a pivot, we break up our array into blocks of length 5. Why 5? In this question, we explore `select-3`, in which we break up our array into blocks of length 3. As you go through this problem, feel free to refer to the course notes from 4/12 and the week 3 recitation notes. In addition, to simplify your logic, you should assume throughout this problem that your array is a power of 3.

   (a) Consider the pseudocode for `select` and `choosePivot` on pages 2 and 4 of the April 12th course notes. In this pseudocode, we break up our array into blocks of size 5. What change(s) would you need to make to this pseudocode in order to write `select-3` and `choosePivot-3`? **[We are expecting a one or two sentence description of changes made, so that a 106b student would know exactly what to do.]**

   Change the 5 to a 3 when it divides.

   (b) Prove that the recursive call to `select-3` inside of `choosePivot-3` is on an array of size at most $n/3$. You should assume that your array is a power of 3. **[We are expecting a rigorous proof.]**

   We split the array into $\frac{n}{3}$ groups. We pick one element from each group to be part of what we recurse on. Thus, we recurse on a size of at most $n/3$.

   (c) Prove that the recursive call inside of `select-3` is on an array of size at most $2n/3+2$. You should again assume that your array is a power of 3. (Hint: it might be helpful to note that $\lceil x \rceil \leq x+1$.) **[We are expecting a rigorous proof.]**

   Let p be the median of medians. The number of elements that must be smaller than p is $2 \cdot \frac{n/3-1}{2} + 1$. This includes all the smaller medians, and the element in the same column as p, but smaller than p. The elements that could possibly be smaller than p are all the elements in the columns of the medians less than p and the smallest element in the other columns. This is $n/3-1$ elements. A similar argument follows for finding the number of possible elements greater than p. If we add all these elements up, then we get that there are at most $\frac{2n}{3}-1$ elements greater than or smaller than p. Now we must recurse on one of those sides, but we know that the recursive call will be of size at most $\frac{2n}{3}$.

   (d) Explain why the work done within a single call to `select-3` and `choosePivot-3` on an array of size n is $\Theta(n)$. **[We are expecting a few sentences of explanation. You do *not* need to do a formal proof with the definition of Θ.]**

Doing mergesort on order n elements of constant size (3) is order n time. Partition is also order n time. When we do a select within the choosePivot-3 method, that also is order n time. Thus overall we are left with a recurrence relation that has the array splitting roughly in half each time, while also performing some other order n tasks.

(e) Using what you proved in (b), (c), and (d), write down a recurrence relation for the runtime of `select-3`. (You can do this problem even if you did not complete all of (b), (c), and (d).) **[We are expecting a one-line answer with your recurrence relation.]**

$T(n) = T(\frac{2n}{3}) + T(\frac{n}{3}) + O(n)$

(f) Is `select-3` $O(n)$? Justify your answer. **[We are not expecting a formal proof, but you should describe clear reasoning, such as analyzing a tree, unraveling the recurrence relation to get a summation, or attempting the substitution method. (Note that succeeding at the substitution method would prove select-3 is O(n), but failing at the substitution method does not prove select-3 is not O(n).)]**

`select-3` is not $O(n)$. First we will show that the substitution method does not work for $O(n)$. For it to work, we would need to show that $cn \leq \frac{2cn}{3} + \frac{cn}{3} + bn$. This simplifies to $c \leq c + b$, which can never be true.

Now we will show that the runtime IS $O(n \log(n))$, which will prove that `select-3` is not $O(n)$. We must show that $cn \log(n) \geq \frac{2cn \log(2n/3)}{3} + \frac{cn \log(n/3)}{3} + bn$. This simplifies to $c(3\log(n) - 2\log(2) - 2\log(n) + 2\log(3) - \log(n) + \log(3)) \geq b$, which is equivalent to $c \geq \frac{b}{3\log(3) - 2\log(2)}$. Thus we can choose a value for c that works for that constraint.