Solution of Second Homework

Han Wu, Yu Chen

1 Different-sized sub-problems (6 points)

Solve the following recurrence relation.

$$T(n) = T(n/2) + T(n/4) + T(n/8) + n,$$

where $T(1) = 1$.[**We are expecting a formal proof. For example, what is the value range of the parameters in your assumption? You may state your final running time with $O()$ notation, but do not use it in your proof. Pay attention to $T(1) = 1$, which should be satisfied in your results.**]
There are three ways of solving this.

1. One is unrolling recursion and finding similarities which can require inventiveness and can be really hard.
2. Another method is to solve the problem by *Substitution*. To prove that $T(n) = O(n)$, we guess:

$$T(n) \leq \begin{cases} d & if \quad n = 1 \\ dn & if \quad n > 1 \end{cases}$$

   For the base case, we use the standard assumption that $T(1) = 1 \leq d$. For the inductive hypothesis, we assume that our guess is correct for any $n < k$, and we prove our guess for k. That is, consider d such that for all $1 \leq n < k, T(n) \leq dn$.
   To prove for $n = k$, we solve the following equation:

$$T(k) \leq T(k/2) + T(k/4) + T(k/8) + k \leq dk/2 + dk/4 + dk/8 + k \leq dk$$

$$(d/8 - 1)k \geq 0$$

$$d \geq 8$$

   Therefore, we can choose $d = max(1, 8) = 8$, which is a constant factor. The induction is completed. By the definition of big-O, the recurrence runs in $O(n)$ time.
3. The other way is to use *Akra-Bazzi* method.*Akra-Bazzi* is used to solve the following recursive formula,

$$T(x) = \begin{cases} \Theta(1) & if \quad 1 \leq x \leq x_0 \\ \sum_{i=1}^{k} a_i T(b_i x) + g(x) & if \quad x > x_0 \end{cases}$$

   , where

   - $x \geq 1$ is a real number,
   - x_0 is a constant such that $x_0 \geq 1/b_i$ and $x_0 \geq 1/(1 - b_i)$ for $1 \leq i \leq k$,
   - a_i is a constant for $1 \leq i \leq k$,
   - $b_i \in (0, 1)$ is a constant for $1 \leq i \leq k$,

- $k \geq 1$ is a constant, and
- $g(x)$ is a non-negative function that satisfies the polynomial-growth condition.

Theorem (. Akra-Bazzi) Given a recurrence of the form specified in Equation above, let p be the unique real number for which $\sum_{i=1}^{k} a_i b_i^p = 1$. Then

$$T(x) = \Theta(x^p(1 + \int_1^x \frac{g(u)}{u^{p+1}} du)).$$

So, in this case

- $g(x) = n$
- $a_1 = a_2 = a_3 = 1$
- $b_1 = 1/2, b_2 = 1/4, b_3 = 1/8$
- $\sum_{i=1}^{k} a_i b_i^p = 1 \Rightarrow \frac{1}{2^p} + \frac{1}{4^p} + \frac{1}{8^p} = 1 \Rightarrow p \approx 0.88$
- $T(n) = \Theta(n^p(1 + \int_1^n \frac{g(u)}{u^{p+1}} du)) = \Theta(n^p(\frac{n^{1-p}}{1-p})) = O(n)$.

2 What's wrong with this proof? (8 points)

Consider the following recurrence relation:

$$T(n) = T(n-5) + 10 \cdot n$$

for $n \geq 5$, where $T(0) = T(1) = T(2) = T(3) = T(4) = 1$. Consider the following three arguments.

(1) **Claim:** $T(n) = O(n)$**.** To see this, we will use strong induction. The inductive hypothesis is that $T(k) = O(k)$ for all $5 \leq k < n$. For the base case, we see $T(5) = T(0) + 10 \cdot 5 = 51 = O(1)$. For the inductive step, assume that the inductive hypothesis holds for all $k < n$. Then

$$T(n) = T(n-5) + 10n,$$

and by induction $T(n-5) = O(n-5)$, so

$$T(n) = O(n-5) + 10n = O(n).$$

This establishes the inductive hypothesis for n. Finally, we conclude that $T(n) = O(n)$ for all n.

(2) **Claim:** $T(n) = O(n)$**.** To see this, we will use the Master Method. We have $T(n) = a \cdot T(n/b) + O(n^d)$, for $a = d = 1$ and

$$b = \frac{1}{1 - 5/n}.$$

Then we have that $a < b^d$ (since $1 < 1/(1-5/n)$ for all $n > 0$), and the master theorem says that this takes time $O(n^d) = O(n)$.

(3) **Claim:** $T(n) = O(n^2)$**.** Imagine the recursion tree for this problem. (Notice that it's not really a "tree," since the degree is 1). At the top level we have a single problem of size n. At the second level we have a single problem of size $n-5$. At the t'th level we have a single problem of size $n - 5t$, and this continues for at most $t = \lfloor n/5 \rfloor + 1$ levels. At the t'th level for $t \leq \lfloor n/5 \rfloor$, the amount of work done is $10(n-5t)$. At the last level the amount of work is at most 1. Thus the total amount of work done is at most

$$1 + \sum_{t=0}^{\lfloor n/5 \rfloor} 10(n - 5t) = O(n^2).$$

(a) (3 points) Which, if any, of these arguments are correct? **[We are expecting a single sentence stating which are correct or all are wrong.]**
Only (3) is True.

(b) (5 points) For each argument that you said was incorrect, explain why it is incorrect. If you said that all three were incorrect, then give a correct argument. **[We are expecting a few sentences of detailed reasoning for each incorrect algorithm; and if you give your own proof we are expecting something with the level of detail of the proofs above—except it should be correct!]**
(1) The main problem is that O () cannot be used in the derivation of substitution method. We can also prove by contradiction as follows. Suppose that, as per the definition of $O()$, there must exist constants n_0 and $c > 0$ so that for all $n > n0$, $T(n) \leq cn$. Therefore, we known that $T(n-5) = O(n-5) \leq c(n-5) \Rightarrow T(n) = T(n-5) + 10n \leq cn + (10n - 5c)$. To ensure that $T(n) \leq cn$, we have $(10n - 5c) \leq 0 \Rightarrow c \geq 2n$. This is in contradiction to the fact that c is a constant independent of n.
(2) The b of the master theorem is a constant independent of n. So we can't use the master theorem to solve this recursion.

3 RandP (4 points)

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` (in other words, run $\frac{1}{p(1-p)}$ times of function `randP` in expectation). 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 description of what the algorithm does.]**

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.
As shown in algorithm 2.

Algorithm 1 Boolean Random

```
while true do
    $b_1 \leftarrow randP()$
    $b_2 \leftarrow randP()$
    if $b_1 == true$ and $b_2 == false$ then
        return true
    if $b_1 == false$ and $b_2 == true$ then
        return false
```

(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`.]**

Proof. We denote that *BooleanRandom* was successfully returned as event A. So, the probability of occurrence of event A is $Pr(A) = Pr_{(b_1\&\&!b_2)||(!b_1\&\&b_2)} = p(1-p) + (1-p)p = 2p(1-p)$. Meanwhile, the test stopped until the event A happened, and the number of tests was X. Therefore, X obeys the geometric distribution and is denoted as $X \sim GE(P)$. From the expected formula $E(X) = 1/p$ of geometric distribution, we have $E(X) = \frac{1}{2p(1-p)}$. In other words, we are expected to make $2 * \frac{1}{2p(1-p)}$ calls of *randP*. □

(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 known that each call of $randP$ is independent. Therefore, the Bernoulli event X in each loop of Algorithm 2 is also independent. At the same time, we know that the probability of $BooleanRandom$ returning true or false is $Pr(true) = Pr(false) = p(1-p)$, and the probability of successfully return is $Pr(A) = 2p(1-p)$. So, we have $Pr(true|A) = Pr(false|A) = \frac{p(1-p)}{2p(1-p)} = 1/2$.

4 Median (8 points)

Suppose that A and B are sorted arrays of length n in ascending order, 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 a description of the algorithm.]**

Algorithm 2 Median(A, B, n)

```
Input: Sorted arrays $A$ and $B$ of length n in ascending order
Output: $n$th smallest number in $A$ and $B$
if $n == 1$ then
    return $\min(A[0], B[0])$
$mid = (n-1)/2$
if $A[mid] < B[mid]$ then
    return Median($A[mid+1:n], B[0:mid], n/2$)
if $A[mid] > B[mid]$ then
    return Median($A[0, mid], B[mid+1:n], n/2$)
return $A[mid]$
```

(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.]**

Proof. To see this, we'll use a proof by Loop Invariant.
Loop Invariant: At the end of the $Median(A, B, n)$, it returns the nth smallest number in A and B.
Initialization: When the size n of the list A and B is 1, the function $Median(A, B, 1)$ returns the minimum one of $A[0]$ and $B[0]$, which is the $1st$ number in $\{A[0], B[0]\}$. So the *Loop Invariant* holds.
Maintenance: Assume that the loop invariant holds at the recursive calls in function $Median(A, B, n)$. So that the function $Median(A[mid+1:n], B[0:mid], n/2)$ returns the median m in $A[mid+1:n]$ and $B[0:mid]$, in other words, there exist $n/2$ items in $A[mid+1:n]$ and $B[0:mid]$ which value no large than m. With the conditions of ascending sorted, we have $\forall_{i \in A[0:mid]} i \leq m$ and $\forall_{i \in B[mid+1:n]} i \geq m$. Therefore, there are n elements not greater than m. So the return value of the function $Median(A, B, n)$ satisfies the loop invariant. We can prove the other case (line8-9) in a similar way.
Termination: The function terminates when the input array size n is 1, or $A[mid] = B[mid]$.

Now the loop invariant gives: The return value of the function $Median(A, B, n)$ returns the nth smallest number in A and B. Therefore the algorithm is correct. □

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

$T(n) = T(n/2) + O(1) \Rightarrow T(n) = O(\log n)$ (By Master Theorem)

5 20-question sorting (4 points)

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 $\{1, \dots, k\}$. 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.]**

Because the elements in the array A are in the set $\{1, \dots, k\}$, for each element $x \in \{1, \dots, k\}$, we can continuously ask the question "Is the frequency of occurrence of x is greater than *some number*?" to obtain the frequency of occurrence of it. After obtaining the frequency of each element, we can easily arrange these k numbers in ascending order and obtain a sorted array. For each element, since the range of its frequency is $[0 : n]$, we only need to ask *logn* questions at most to determine its frequency. Details are shown in Algorithm 3.

Algorithm 3 sort

```
procedure SORT(Array $A = [a_1, \dots, a_n]$ of $n$ elements, each one is in the set $\{1, \dots, k\}$)
    $cnt \leftarrow$ an array of length $k + 1$
    for each element $e$ in set $\{1, \dots, k\}$ do
        $l \leftarrow 0,\ r \leftarrow n$
        while $l < r$ do
            $mid \leftarrow (l + r)/2$
            Ask the question: "Is the frequency of occurrence of $a_i$ is greater than $mid$?"
            if "yes" then
                $l = mid + 1$
            else
                $r = mid$
        $cnt[e] = l$
    $i \leftarrow 0$
    $res \leftarrow$ an array of length $n$
    for each element $e$ in set $\{1, \dots, k\}$ do
        for $i$ in $[0 : cnt[e])$ do
            $res[i++] = e$
    return $res$
```

(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, assuming $k \le n$. [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 $\{1, \dots, k\}$ (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).]**

The number of question we need to ask for an element in $\{1, \dots, k\}$ is determined by the range of its frequency. Thus, the best case scenario is, the frequency of each element in $\{1, \ldots, k\}$ is in $[0 : \frac{n}{k}]$. The upper bound of the frequency range cannot be less than $\frac{n}{k}$, otherwise we cannot use k numbers to form an array with length n. Thus, the lower bound of the question we need to ask is $\Omega(k \log \frac{n}{k})$.

6 Match File with Password (7 points)

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 a description of the algorithm.]**

For the first password $P[0]$, we test it with each of the n files, and record the files whose passwords are lexicographically smaller than it (denoted by L), the file $F[j]$ which matches it, and the files whose passwords are lexicographically greater than it (denoted by R). For the second password $P[1]$, there is no need to test it with the left $n-1$ files. We just have to test it with the file $F[j]$ which $P[0]$ matches, if $P[1]$ is lexicographically smaller than $F[j]$'s password, we test it with files in L, otherwise we test it with files in R. We repeat the above process untill all passwords find the matched files. This process is quite like *quicksort*: password is the pivot, and acoording to it, we can determine the matched file(locate its position in sorted array), and partition the left files into two sets(partition the left elements into two sets in *quicksort*). The worst-case running time of deterministic *quicksort* is $O(n^2)$, so does the above process. So like the *randomized quicksort*, we randomly choose a password in each round. Details are shown in Algorithm 4. Note that the space of L and R can be optimized to $O(n)$. However, for clarity, we won't optimize them in the pseudocode.

Algorithm 4 match

```
procedure MATCH(List $P = [p_1, \dots, p_n]$ of $n$ passwords, List $F = [f_1, \dots, f_n]$ of $n$ files)
    $m \leftarrow$ a binary search tree, each node is a file-password pair
    $L \leftarrow$ a hashmap, key={file-password pair}, value={left files whose password is lexicographically smaller than the corresponding password in key}
    $R \leftarrow$ a hashmap, key={file-password pair}, value={left files whose password is lexicographically greater than the corresponding password in key}
    for $i$ in $[1 : n]$ do
        $p \leftarrow$ randomly choose a password not in $m$
        if $m$ is empty then
            iterate through $F$, find the matched file $f$, and partition the left $n-1$ files into $L'$ and $R'$ according to $p$
            $(f, p)$ becomes the root of $m$
            $L((f, p)) = L'$
            $R((f, p)) = R'$
        else
            test $p$ in $m$ untill a leaf node, denote it by $lf$
            if $p$ is lexicographically smaller than the password of $lf.file$ then
                $S \leftarrow L(lf)$
                iterate through $S$, find the matched file $f$, and partition the left files in $S$ into $L'$ and $R'$ according to $p$
                $(f, p)$ becomes the left child of $lf$
                $L((f, p)) = L'$
                $R((f, p)) = R'$
            else
                $S \leftarrow R(lf)$
                iterate through $S$, find the matched file $f$, and partition the left files in $S$ into $L'$ and $R'$ according to $p$
                $(f, p)$ becomes the right child of $lf$
                $L((f, p)) = L'$
                $R((f, p)) = R'$
    return $m$
```

(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.]**
This algorithm is almost the same as the *randomized quicksort*. As explained in the first question, each round it will match a password with a file. Thus, it will match the n password with n files finally.

(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.]**
$T(n) = P(n) + \frac{1}{n}\sum_{k=1}^{n}(T(k-1) + T(n-k)) \rightarrow T(n) = O(nlogn)$, here $P(n)$ is the partition time. Details can refer to Prove the average time complexity of *quicksort*.

7 select algorithm (6 points)

(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. 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` in the slides of lecture 4. 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.]**
We have to make a little change in page 25 of lecture 4.
Old: Split A into $m = \lceil \frac{n}{5} \rceil$, of size ≤ 5 each
New: Split A into $m = \lceil \frac{n}{3} \rceil$, of size ≤ 3 each

(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.]**
Because n is a power of 3, in each recursive call, we can split current array with length n into $\frac{n}{3}$ groups. We take the median of each group, so in next recursive call, the new array consists of $\frac{n}{3}$ elements, each one is the median of corresponding group.

(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.]**
To prove that the recursive call inside of select-3 is on an array of size at most $2n/3 + 2$, we need to prove that $|A_<| \leq 2n/3 + 2$ and $|A_>| \leq 2n/3 + 2$. Next we will prove that $|A_>| \leq 2n/3 + 2$.
Suppose $g = \lceil \frac{n}{3} \rceil$ is the number of groups, and p is the median of $p_1, \ldots, p_g$. Because p is the median of g elements, the median of $\lceil \frac{g}{2} \rceil - 1$ groups p_i are smaller than p. If p is larger than a group median, it is larger than at least two elements in that group (the median and the smaller one number). This applies to all groups excpt the remainder group, which might have fewer than 3 elements. Accounting for the remainder group, p is greater than at least $2 \cdot (\lceil \frac{g}{2} \rceil - 2)$ elements of A.

$$\begin{aligned}
|A_>| &\leq (n-1) - (2 \cdot (\lceil \frac{g}{2} \rceil - 2) + 1) \\
&= n + 2 - 2 \cdot \lceil \frac{g}{2} \rceil \\
&\leq n - 2n/6 + 2 \\
&= 2n/3 + 2
\end{aligned}$$

By symmetry, $|A_<| \leq 2n/3 + 2$ as well.
Thus, we can conclude that the recursive call inside of select-3 is on an array of size at most $2n/3 + 2$.

(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.]**
In a single call to select-3 and choosePivot-3, we first partition the array with length n into $\lceil \frac{n}{3} \rceil$ groups, this step requires $O(n)$, and we find median of each group, this step needs $O(\lceil \frac{n}{3} \rceil) = O(n)$, so the overall time complexity is $\Theta(n)$.

(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.]**

$$T(n) \leq \begin{cases} T(n/3 + 1) + T(2n/3 + 2) + \Theta(n) & , if \ n > 3 \\ c & , if \ n \leq 3 \end{cases}$$

(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 $O(n)$. From above recurrence relation we can observe that $T(n) = T(n/3) + T(2n/3) + \Theta(n)$. This means that in each level of the recursion tree we have to do $O(n)$ job. And in the worst case, we always choose the $T(2n/3)$ branch to walk down, which will result in a recurrence tree with $log_{\frac{3}{2}} n = \frac{log_2 n}{log_2 \frac{3}{2}} = c \cdot logn (c = \frac{1}{log_2 \frac{3}{2}})$ height. Thus, the overall time complexity is $O(nlogn)$.