Problem Set 1 Solutions

This problem set is due **at 10:00pm** on **Wednesday, February 12, 2020**.

EXERCISES (NOT TO BE TURNED IN)

Asymptotic Analysis, Recursion, and Master Theorem

- Do Exercise 4.3-7 in CLRS on page 87.
- Do Exercise 4.3-9 in CLRS on page 88.

Randomized Algorithms

- Do Exercise 7.4-4 in CLRS on page 184.
- Do Exercise 9.2-4 in CLRS on page 220.

Problem 1-1. Recurrences and Asymptotics [30 points]

Let $T(n)$ be the time complexity of an algorithm to solve a problem of size n. Assume $T(n)$ is $O(1)$ for any n less than 3. Solve the following recurrence relations for $T(n)$.

(a) [3 points] $T(n) = 8T\left(\frac{n}{2}\right) + n^2$

Solution: By case 1 of the Master Theorem, since $f(n) = n^2$ is polynomially less than n^3, $T(n) = \Theta(n^3)$.

(b) [3 points] $T(n) = 8T(\sqrt[4]{n}) + \log^4 n$.

Solution: We solve this one by changing the variables. Assume that n is a power of 2, and let $n = 2^m$, then

$$T(2^m) = 8T(2^{m/2}) + m^4.$$

If we define $S(m) = T(2^m)$, the recurrence becomes

$$S(m) = 8S(m/2) + m^4.$$

We can use the Master Theorem (case 3), since $\log_2 8 < 4$ and the regularity condition holds ($8(m/2)^2 \leq 2m^4$). Therefore, $S(m) = \Theta(m^4)$, and $T(2^m) = \Theta(m^4)$. Changing the variable back ($m = \log n$), we have $T(n) = \Theta(\log^4 n)$.

(c) [5 points] $T(n) = T(n-1) + 3n$.

Solution: The master theorem doesn't apply here. Instead, we can use the iteration method. We see that

$$\begin{aligned} T(n) &= T(n-1) + 3n \\ &= T(n-2) + 3(n-1) + 3n \\ &\cdots \\ &= n \cdot 3n - 3\sum_{i=0}^{n} i \\ &= 3n^2 - 3n(n+1)/2 \\ &= \Theta(n^2) \end{aligned}$$

(d) [5 points] $T(n) = T(n/7) + 2T(n/5) + O(n)$.

Solution: We can do this in two ways:

1. We can use intuition, given that $n/7 + 2n/5 < n$, and guess a solution of the form $T(n) \leq cn$ for some $c > 0$. We can prove this using strong induction on n. Assume $T(n) \leq cn$ for all $n < k$ for some k. We will show that $T(k) \leq ck$.

   $$T(k) \leq T(k/7) + 2T(k/5) + dk$$

   for some fixed $d > 0$. Now by our inductive hypothesis,

   $$\begin{aligned} T(k) &\leq \frac{1}{7}ck + \frac{2}{5}ck + dk \\ &= \left(\frac{19}{35}c + d\right)k \leq ck \end{aligned}$$

   whenever $c \geq \frac{35}{16}d$. In addition, for our base case $k = 1$ to hold, we pick $c = \max\left(\frac{35}{16}d, T(1)\right)$. This proves our claim for large enough c, by induction.
2. A more complicated calculation can be done with the help of a recursion tree. Again, let $d > 0$ represent the constant factor in the term $O(n)$. We use the following claim which proof we defer.

   Claim: The sum of the sizes of all sub-problems at the k-th level is at most $(19/35)^k n$

   Assuming the claim and by the linearity of the cost term (the cost of problems of size m is dm), the total cost of nodes at level k is at most $(19/35)^k dn$. The sum of the costs at all levels is $\sum(\frac{19}{35})^k dn$, which is a geometric sum converging to $\Theta(n)$.

   We now prove the claim by induction on k. This clearly holds for the root of the tree ($k = 0$). Let m_i for $i = 1, ..., K$, be the sub-problem sizes of nodes at level k and note that the actual value of K is not important. By the induction hypothesis, $\sum m_i \leq (19/35)^k n$. Each sub-problem of size m_i generates 3 sub-problems at level $k + 1$ of size $m_i/5$ and 2 sub-problems of size $m_i/7$. Therefore, the total size fo sub-problems at level $k + 1$ is

   $$\begin{aligned} \sum_i (m_i/7 + 2m_i/5) &= \sum_i (1/7 + 2/5)m_i \\ &= (19/35)\sum_i m_i \\ &\leq (19/35)^{k+1} n \end{aligned}$$

(e) [14 points] Consider the following functions. Within each group, sort the functions in asymptotically increasing order, showing strict orderings as necessary. For example, we may sort $n^3, n, 2n$ as $2n = O(n) = o(n^3)$.

1. [4 points] $\log n$, $\sqrt[5]{n}$, $\log_9 n$, $\log n^5$, $\log\log n^{20}$, $\log^9 n$.

   Solution: $\log\log n^{20} = o(\log n) = O(\log n^5) = O(\log_9 n) = o(\log^9 n) = o(\sqrt[5]{n})$
   Note that a change of base in the logarithm is equivalent to a constant factor, so they are asymptotically equal.
   A change of variables can be helpful here - if you're not sure whether $\log^9 n$ is asymptotically bigger or smaller than $O(\sqrt[5]{n})$, replacing n with 2^m and comparing m^9 with $2^{m/5}$ is easier. Note also that $\log n^5 = 5\log n$. Also note that $\log\log n^{20} = \log(20\log n) = \log\log n + O(1)$.

2. [5 points] n^7, $n\log n^2$, $n^2\log\log n$, 3^n, $\log(n!)$.

   Solution: $\log(n!) = O(n\log n^2) = o(n^2\log\log n) = o(n^7) = o(3^n)$.
   Here, we use Stirling's approximation for the factorial which gives $\log n! = O(n\log n)$.

3. [5 points] $(\log n)^n$, $(\log n)^{n-1}$, 5^n, $n^{\log n}$.

   Solution: $n^{\log n} = o(5^n) = o((\log n)^{n-1}) = o((\log n)^n)$. Note that $(\log n)^{n-1}$ and $(\log n)^n$ are asymptotically different because the constant is in the exponent. To compare 5^n and $(\log n)^{n-1}$, replace n with 2^m. Then, we see that 5^{2^m} is asymptotically smaller than $m^{(2^m-1)}$. To compare $n^{\log n}$ and 5^n, take the log of each to get $\log^2 n$ and $n\log 5$. Clearly, 5^n is asymptotically larger.

Problem 1-2. Probability practice: Dice Rolling [20 points]

A pair of dice is rolled until a sum of either 5 or 7 appears. Find the probability that a 5 occurs first.

(a) [2 points] Let E_n denote the event that a 5 occurs on the nth roll and no 5 or 7 occurs on any of the first $(n-1)$ rolls. Compute $P(E_n)$.

Solution: 5 occurs with probability $\frac{4}{36} = \frac{1}{9}$ and 7 with probability $\frac{6}{36}$ therefore neither occurs with probability

$$1 - \frac{4}{36} - \frac{6}{36} = \frac{13}{18}$$

This gives the equation

$$P(E_n) = (\frac{13}{18})^{n-1} \cdot \frac{1}{9}$$

(b) [8 points] Compute $\sum_{n=1}^{\infty} P(E_n)$ and argue that it is the desired probability. **HINT:** $\sum_{n=0}^{\infty} r^n = \frac{1}{1-r}$ for $|r| < 1$.

Solution: Since the events E_n are mutually disjoint and their union is precisely the event whose probability we seek we get the following:

$$P(5 \text{ before } 7) = \sum_{n=1}^{\infty} P(E_n) = \sum_{n=1}^{\infty} (\frac{13}{18})^{n-1} \cdot \frac{1}{9}$$

This can be solved using the hint in the question to be:

$$\frac{1}{1 - 13/18} \cdot \frac{1}{9} = \frac{2}{5}$$

(c) [10 points] Now, suppose that we roll a standard fair die 100 times. Let X be the sum of the numbers that appear over the 100 rolls. Find an upper bound for the following probability $P[|X - 350| \geq 50]$.

1. [2 points] Let X_i be the number on the face of the die for roll i. Compute the expected value of X_i.

   Solution: The expected value of X_i is given by:

   $$\mathbf{E}[X_i] = \sum_{j=1}^{6} j \cdot \mathbf{P}[X_i = j] = \sum_{j=1}^{6} j \cdot \frac{1}{6} = \frac{21}{6} = \frac{7}{2}$$

2. [2 points] Compute $\mathbf{E}[X]$ where X is the sum of the dice rolls.

   Solution: By linearity of expectations:

   $$\mathbf{E}[X] = \sum_{i=1}^{100} \mathbf{E}[X_i]$$

   Thus $\mathbf{E}[X] = 100(\frac{7}{2}) = 350$.

3. [5 points] Compute the variance of X where X is again the sum of the dice rolls.
   Hint: $\sum_{j=1}^{n} j^2 = \frac{n(n+1)(2n+1)}{6}$

Solution: Since the dice rolls are independent we have:

$$Var[X] = Var(\sum_i X_i) = \sum_{i=1}^{100} Var(X_i)$$

To compute the variance of a single dice roll, we use $Var(X_i) = \mathbf{E}[X_i^2] - \mathbf{E}[X_i^2]$ To find $\mathbf{E}[X_i^2]$ we use the following equation:

$$\mathbf{E}[X_i^2] = \sum_{j=1}^{6} j^2 \mathbf{P}[X_i = j] = \sum_{j=1}^{6} j^2(1/6) = \frac{1}{6} \cdot \frac{6 \cdot 7 \cdot 13}{6} = \frac{91}{6}$$

where the simplification comes from the hint. The variance of X_i is thus:

$$Var(X_i) = \mathbf{E}[X_i^2] - \mathbf{E}[X_i]^2 = \frac{91}{6} - (\frac{7}{2})^2 = \frac{35}{12}.$$

Thus the variance for X is equal to the following:

$$Var(X) = 100 \cdot \frac{35}{12} = \frac{875}{3}.$$

4. [1 points] Find an upper bound for $P[|X - 350| \geq 50]$.

   Solution: Using Chebyshev's inequality and the values we have solved for we get the following:

$$\mathbf{P}[|X - 350| \geq 50] \leq \frac{100 \cdot (35/12)}{50^2} = \frac{7}{60}$$

   Note on problem 1-2: The solution to the problem is essentially an exercise in applying technique. No real ideas or creativity are needed. Students must make sure that they understand the key steps.

Problem 1-3. Mostly Sorting [50 points] In this problem we consider the task of determining whether an array of numbers is sorted or not, without examining all elements in the array. In order to do this, your algorithm is not required to give the correct answer if the array is *mostly sorted* but not completely sorted.

Precisely, given a array A of n elements, we say that S is *mostly sorted* if it contains a sorted subsequence of length at least $\frac{9}{10}n$. The sorted subsequence is not necessarily contiguous. That is, you can remove some $k \leq \frac{1}{10}n$ elements from A such that the remaining elements are in sorted order.

If the input array is completely sorted, your algorithm must return "SORTED". If the input array is mostly sorted, but not completely sorted, your algorithm may output anything. Otherwise, if the input array is not mostly sorted, your algorithm must return "UNSORTED".

Assume that binary comparisons on any two elements can be done in constant time.

(a) [10 points] Prove that any deterministic algorithm satisfying the above must have a runtime that is $\Omega(n)$.

Solution: Suppose you have a deterministic algorithm that only examines $k < \frac{9}{10}n$ elements in the input. The remaining $n - k > \frac{1}{10}$ elements of the input determine whether it is sorted or not almost sorted. Thus the algorithm cannot be correct unless it examines at least $\frac{9}{10}n$ elements, which means the runtime of any correct algorithm is $\Omega(n)$.

A very useful tool for proving impossibility results is the use of a malicious adversary whose sole role in life is to fool the algorithm. We make use of such an adversary.

More formally, given an algorithm X that examines less than $\frac{9}{10}n$ elements of the input, we can construct inputs A and A' such that A is sorted and A' is not mostly sorted but X returns the same answer for both A and A'.

Define $A = [1, 2, \ldots, n-1, n]$. To construct A' we will run X on A and keep track of which indices in A it queries. Denote the sequence of queried indices by $a_1, a_2, \ldots, a_k$. Now, set $A'[a_i] = a_i$ for each $i = 1, \ldots, k$. For every other index $j \notin \{a_i\}$, set $A'[j] = 0$. That is, we fill in A' with the same values as A in the locations where the algorithm X actually checks. Note that A' is well-defined only because X is deterministic.

Now, since A and A' agree on all the indices that the algorithm X checks, X must return the same answer for A and A'. But since A is sorted while A' is not mostly sorted, X must be incorrect on one of A or A'.

(b) [10 points] Now let's add randomness. One property of a sorted array is that any consecutive three elements in the array are also sorted. As a first try, our algorithm will repeatedly sample consecutive triples of elements and check whether the triple is sorted. The algorithm will output 'SORTED' if and only if all sampled triples are sorted.

Let the number of sampled triples be k. Show that we must have k be $\Omega(n)$ for the probability of success of this algorithm to exceed $\frac{1}{2}$.

Hint: It suffices to demonstrate a single "adversarial" input.

Hint: $(1 - \frac{1}{n})^n \approx e^{-1}$. You may also find the inequality $1 - x \leq e^{-x}$ useful.

Solution: Consider the input $A = [n/2, n/2+1, \ldots, n-1, n, 1, 2, \ldots, n/2-1]$. If a subsequence of A contains an element in the first half of A as well as an

element in the second half of the A, it cannot be sorted. Thus A is not mostly sorted.

Now, for a sample to detect that A is not mostly sorted, it must pick one of the two triples including both n and 1 (in the middle of the array). We'll say that a sample succeeds if it does so. The chance of a single sample succeeding is $\frac{2}{n-2}$.

The algorithm will succeed if any sample suceeds and fail if all samples fail. With k independent samples, the probability of failure is $(1 - \frac{2}{n-2})^k$. We need to prove that

$$P[\text{failure}] = (1 - \frac{2}{n-2})^k < \frac{1}{2} \tag{1}$$

only holds if $k = \Omega(n)$.

Since $(1 - \frac{1}{n})^n \approx e^{-1}$, the failure probability is roughly $(e^{-1})^{2k/n}$ which will be close to 1 unless k is $\Omega(n)$.

More carefully, we first note that because $1 - \frac{2}{n-2} < 1$, $P[failure]$ is monotonically decreasing as k increases. Thus it suffices to show that for some constant $c > 0$, $k = cn$ samples results in $P[failure] > \frac{1}{2}$. Let's pick $c = \frac{1}{100}$ and rearrange the expression for $P[failure]$:

$$\begin{aligned}
(1 - \frac{2}{n-2})^{\frac{n}{100}} &= (1 - \frac{1}{\frac{n}{2}-1})^{\frac{n}{100}} \\
&= (1 - \frac{1}{\frac{n}{2}-1})^{(\frac{n}{2}-1)(\frac{\frac{n}{100}}{\frac{n}{2}-1})} \\
&= (1 - \frac{1}{\frac{n}{2}-1})^{(\frac{n}{2}-1)(\frac{1}{49})}
\end{aligned}$$

Asympotically, as n becomes large, this approaches $e^{\frac{1}{49}}$ which is close to 1 and certainly larger than $\frac{1}{2}$.

Alternatively, we can directly prove that (1) implies $k = \Omega(n)$. Taking the natural log of both sides, (1) is equivalent to

$$\begin{aligned}
k \ln(1 - \frac{2}{n-2}) &< \ln \frac{1}{2} \\
k &> \frac{\ln \frac{1}{2}}{\ln(1 - \frac{2}{n-2})}.
\end{aligned}$$

Now, we'll use the fact that $1 - e^{-x} \leq x \implies 1 - x \leq e^{-x} \implies \ln(1-x) \leq -x$ with $x = \frac{2}{n-2}$ to get

$$k > \frac{\ln \frac{1}{2}}{-\frac{2}{n-2}} = (n-2)\frac{\ln 2}{2} = \Omega(n).$$

(c) [10 points] Another property of a sorted array is that binary search can be used to find values in the array. Before we can use this property as the basis of an algorithm, we should investigate the behavior of binary search on an unsorted array.

For simplicity, assume that input arrays will not contain any repeated numbers.

Fixing an input array A of length n, we define binary search as:

BINARY-SEARCH$(A, x, \text{left}, \text{right})$
  if right $-$ left $== 1$
    return left
    else median $\leftarrow \lfloor(\text{left} + \text{right})/2\rfloor$
    if $x < A[\text{median}]$
      return BINARY-SEARCH$(A, x, \text{left}, \text{median})$
    else
      return BINARY-SEARCH$(A, x, \text{median}, \text{right})$

Prove that for any two distinct elements x_1 and x_2 in A, if
BINARY-SEARCH$(A, x_1, 0, n) <$ BINARY-SEARCH$(A, x_2, 0, n)$ then $x_1 < x_2$, *even if A is unsorted*.

Hint: Consider the point at which the execution of BINARY-SEARCH$(A, x_1, 0, n)$ diverges from that of BINARY-SEARCH$(A, x_2, 0, n)$.

Solution: Because the output of the binary search procedure is restricted to the range of indices specified as arguments, at the point where the executions diverge, the search for x_1 must go left while the search for x_2 goes right. This implies that x_1 is less than the element being considered at the branching point (A[median]) while x_2 is larger.

(d) [20 points] Design a randomized algorithm for our problem that succeeds with probability at least $\frac{3}{4}$. The runtime of your algorithm should be $O(\log n)$.

Hint: Use BINARY-SEARCH as a subroutine.

Solution: Our algorithm is the same as in part b), except each sample is to pick (independently and uniformly at random) an index i and to return

whether a binary search for $A[i]$ finds that element
(i.e. whether BINARY-SEARCH$(A, A[i], 0, n) = i$). That is, we sample k such indices and output 'SORTED' if and only if the binary search procedure correctly outputs the given index for every sample.

We show now that a constant number of samples suffices.

First, we claim that the subsequence consisting of those indices on which the subroutine succeeds forms a sorted subsequence of A. Consider any two indices i and j such that BINARY-SEARCH$(A, A[i], 0, n) = i$ and
BINARY-SEARCH$(A, A[j], 0, n) = j$. By part c), we know that if $i < j$, $A[i] < A[j]$ and likewise if $j < i$, $A[j] < A[i]$.

Now, if A is fully sorted our algorithm is always correct. If A is mostly sorted but not fully sorted, it doesn't matter what our algorithm outputs. Finally, if A is not mostly sorted, then the number of indices on which the subroutine succeeds is at most $\frac{9}{10}n$, by the above claim. For our algorithm to be incorrect in this case, we would need to sample from these indices every time, which happens with probability $(1 - \frac{1}{10})^k$. For this to be less than $\frac{1}{4}$, it suffices to pick $k = \log_{\frac{9}{10}} \frac{1}{4}$, which is $O(1)$. (Note that we could achieve any constant probability of success with $k = O(1)$)