Problem Set 2

Kevin Peng
Collaborators: Genghis Chau, Jodie Chen, Xubo Sun

This problem set is due **Tuesday, October 1** at **11:59PM**.

This solution template should be turned in through our submission site. [1]

For written questions, full credit will be given only to correct solutions that are described clearly *and concisely*.

---

[1]Register an account, if you haven't done so. Then go to Homework, Problem Set 2, and upload your files.

Kevin Peng

Problem 2-1. [20 points] **Building bridges**

(a) [10 points] Describe an efficient algorithm to link together the n blocks using the greedy strategy outlined above. Make sure to describe the data structures that you use as well as analyze the runtime of your algorithm. For full credit, your algorithm should run in $O(n\,log\,n)$ time.

Given the list of n steel blocks, we can run first build a min-heap in $O(n)$ time. Then, we can extract the smallest block from the min-heap twice, link the two blocks together, and insert the sum of those two blocks back into the min-heap. Each extraction (removing the element and preserving the min-heap property for the entire tree) takes $O(\log n)$ time and inserting an element into the min-heap also takes $O(\log n)$ time. We assume that the calculating the combine cost takes constant time. Each time we perform this operation, we reduce the size of our heap by 1. Since we start with n elements in the heap and terminate when there are none, we perform this operation $n-1$ times, so the total time for running this greedy algorithm is $O(n \log n)$. The result will be a single node, which contains the total cost of the algorithm.

(b) [10 points] Now suppose you know that the longest block has length 1000. How can you modify the algorithm from part (a) to run in $O(n)$ time? Describe your new algorithm and argue that it has $O(n)$ runtime.

We can first sort the array using a counting sort in $O(n+k)$ time. Given n inputs and the fact that no value is greater than 1000, we can let $k = 1000n$. Since $k = O(n)$, we are able to sort all of the elements in $O(n)$ time. Counting sort produces an array of length $1000n$. We then go through this array one index at a time, add the smallest values up, decrease their counters by 1, and increase the counter of the sum by 1. We repeat this until we reach the end of the array.

For example, if we had 4 blocks of lengths 200, 200, 400, 700, we would use counting sort to first create an array of size $4000 = 4 \times 1000$ (let's assume the array starts with index 1 for simplicity) and set $A[200] = 2, A[400] = 1, A[700] = 1$. Then we would add the two smallest elements, 200 and 200 to get 400. Thus, the algorithm will subtract 1 from $A[200]$ twice (since the two smallest values were both 200) and increase $A[400]$ by 1. Then, we repeat, getting $A[400] = 0$ and $A[800] = 1$. Finally, we have $A[700] = 0$, $A[800] = 0$, and $A[1500] = 1$. The algorithm will then proceed to scan to the end of the list and find out that there are no more values with positive counters, so it will return 1500.

Scanning through a list of $1000n$ elements takes $O(n)$ time. We also had to do $O(n)$ additions, counter increments, and counter decrements, so assuming that each of those single operations takes constant time, this part of the process also takes $O(n)$ time.So the total running time for this algorithm is $O(n)$.

Kevin Peng

Problem 2-2. [20 points] **Master the Master Theorem and Sorting Too**

The following questions test your ability to use and apply the Master Theorem and sorting algorithms you learned in class to various scenarios.

(a) [10 points] Use the Master Theorem to find the runtime of a recursive algorithm whose execution time is given by the formula:

$$T(n) = 2T(n-1) + log\, n.$$

Hint: The Master Theorem cannot be used on the given formula as it stands. Consider what would happen if you substitute one of $n = 2^m$, $n = log\, m$, or $n = m^2$ for n. Identify which substitution allows you to apply the Master Theorem, and use it to find the runtime of $T(n)$.

We substitute $n = \log m$ into this equation to get a form that we can solve using the master theorem.

$$T(\log m) = 2T(\log m - 1) + \log\log m$$

We define a new recurrence: $S(2^m) = T(m)$. Thus, we get

$$S(m) = 2S(\frac{m}{2}) + \log\log m$$

We can now solve this recurrence using the Master Theorem! Here, $b = 2$ and $a = 1$, so we have $f(m) = \log\log m$ and $m^{\log_2 2} = m$. By the first case of the Master Theorem, we have that $S(m) = \Theta(m)$. Thus, $\Theta(m) = S(m) = T(\log m) = T(n)$, so $T(n) = \Theta(2^n)$.

(b) [5 points] The merge sort you learned in class divides the input array into **two** equal sized arrays and runs the algorithm recursively on those two parts. Now, suppose that a new merge sort, the "tri-merge sort", divides the array into **three** equal sized arrays and runs "tri-merge sort" recursively on these three equal sized parts. (You can assume that merging the three arrays still takes $O(n)$.) Use the Master Theorem to derive the runtime of "tri-merge sort".

The recursive algorithm for tri-merge sort is

$$T(n) \leq 3T(\frac{n}{3}) + cn$$

Using the Master Theorem, we have that $f(n) = cn = \Theta(n^{\log_3 3})$. This is the second case of the Master Theorem. Thus, the runtime of this algorithm is $\Theta(n \log n)$.

(c) [5 points] Suppose you are given a list of N integers. All but one of the integers are sorted in numerical order. Identify a sorting algorithm from class which will sort this special case in $O(N)$ time and explain why this sorting algorithm achieves $O(N)$ runtime in this case.

We can go through the array and find the element that is misplaced by looking at each element's left and right neighbor. This takes $O(N)$ time. Once we find the element that is out of place, we can sort it by running insertion sort on that single element. The reason that it achieves $O(N)$ run time in this case as opposed to normal insertion sort is that it only compares 1 element with at most $N-1$ elements in the array. The normal insertion sort compares all $N-1$ elements with at most $N-1$ other elements in the array.Thus, the running time for this special case is $O(N)$.

Kevin Peng

Problem 2-3. [20 points] **Hacking the MTV VMA** As a summer intern for MTV, you've been given the task of managing the seating at the 2014 MTV Video Music Awards. The seats are numbered from 1 to N^3. You've been given a list of N intervals indicating seats that have already been reserved. For example, an interval of $[4, 20]$ indicates that seats 4 through 20 have been reserved. However, a bug in MTV's computer system has caused it to return the reserved intervals in random order and to return some intervals that overlap. It is your job to find the first seat in numerical order that has not been reserved. Write an algorithm that accepts the list of intervals and returns the location of the first unreserved seat, or $N^3 + 1$ if all seats are reserved.

For example, if you are given $N = 5$ and the intervals $([1, 5], [2, 9], [18, 25], [10, 15])$, your algorithm should find that seat 16 is the first seat that has not been reserved.

Describe your algorithm and analyze its runtime. For full credit, your algorithm should have a runtime of $O(N)$.

We can first sort the *intervals* using radix sort with $k = N$ on the intervals' first elements. Thus, we can sort the intervals in an array in $O(3(N + N)) = O(N)$ time. The 3 comes from the fact that there are at most 3 digits in any number from 1 to N^3 in base N (assume that 000 in base N corresponds to 1 instead of 0, 001 to 2, etc.).
Once we have a sorted list, we first create a counter k (initialized to 0), which will denote that all seats in $[1, k]$ that we know are reserved. After that, we can go through each interval and run the following algorithm. If $k + 1$ is smaller than the new interval's first index, we return $k + 1$ (so we terminate). Otherwise, we set k equal to $max(k, e)$, which e is the second element of the current interval. We continue this algorithm until we reach the end of the array (unless we already terminated), at which point we return $k + 1$. This part will also run in $O(N)$ time since we have to iterate through at most N intervals, spending constant time's worth of operations in each iteration.
We know that this algorithm is correct because of the invariant k, denoting that all seats in $[1, k]$ are reserved. Before we check the array, $k = 0$, so no seats are reserved, which is correct. To show maintenance, consider that seats $[1, k]$ are reserved. If $k + 1$ is smaller than the first index of the next interval, then $k + 1$ is not in that interval and thus not in any interval, since we have sorted the list by the first index. The loop immediately terminates here, so we have the correct answer, which is $k + 1$, the seat with lowest index that was not reserved. Otherwise, let us denote the first index of this next interval as l and second index as m. This indicates that seats l through m are reserved. If $m > k$, we will set $k = m$, which maintains the invariant since seats in $[k + 1, m]$ are reserved ($l \leq (k + 1)$ or else we would've stopped at the first case). If $m \leq k$, then we leave k as it is, which also maintains the invariant, since we did not find out that any extra seats were reserved. We will terminate either when we reached the first case of each iteration, or we reach the end of the array, in which k indicates that all seats from 1 to k are reserved, so $k + 1$ is the smallest index of a seat not reserved.

Kevin Peng

Problem 2-4. [45 points] **Cracking a firewall**

(a) [30 points]

Write your solutions in `pset2_solution_template.py`.

(b) [15 points] Suppose that instead of receiving a stream of packets you are batch processing all packets in the follow way: *first* all the control packets arrive, *then* all the regular packets arrive, *then* you process them all. Note that this means all control packets should be applied before any regular packets are checked. For this part, you may also assume that MAX_PORT $= n^6$.

Describe an efficient algorithm to process all process all packets and generate the three lists of regular packets (forwarded, quarantined, and dropped). Analyze its runtime. You may refer to your customized AVL-tree from part (a) in your solution. For full credit, your algorithm should have $O(k \log k + n)$ runtime.

First, we build an AVL tree with the k control packets in $O(k \log k)$ time. The purpose of this step is to reject invalid control packets. We can then take the minimum element of the AVL tree in $O(\log k)$ time and add it to an array k times. Thus, we get a sorted array of control packets in $O(k \log k)$ time. (Interval A is smaller than interval B if interval A's first element is smaller than interval B's first element or if interval A's first element is equal to interval B's first element and interval A's second element is smaller than interval B's second element). We can sort the list of regular packets in $O(6(n + n)) = O(n)$ time using radix sort with $k = n$ (which will give us that $d = 6$).

We refer to the pseudocode below as our algorithm for generating the three lists (we assume that we already have empty forward, quarantine, drop lists elsewhere; otherwise we could have these lists as return values). This part of the algorithm takes at most $O(n + k)$ time since in each iteration we increase the index of one of the pointers by 1. Thus, the total running time of the algorithm is $O(n + k + k \log k) = O(n + k \log k)$.

The pseudocode below is correct because in each iteration, we check if the received packet is in the control packet's interval. If it is, we put it in the correct list and move to the next received packet. If it doesn't, we check if the control packet's minimum is greater than the received packet. If it is, we know that the port in this received packet is not in any of the control packets since all the remaining control packets (the ones to the right) only contain ports larger than the minimum port of the current control packet. Otherwise, we have that the port in this received packet is greater than the maximum of the current control packet. Thus, we move to the next control packet since all the control packets are in sorted order and do not overlap.

```
def generatePacketLists(controlList, regularList):
    a = 0
    b = 0
```

```
while b < n:
  if a >= k:
    put in dropped list
    b++
  elif regularList[b] in controlList[a]:
    put in appropriate list (forward, quarantine)
    b++
  else:
    if controlList[a].min > regularList[b]:
      put in dropped list
      b++
    else:
      a++
```