Problem Set 2

Kevin Peng
Collaborators:

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 takes $O(\log n)$ time and inserting an element into the min-heap also takes $O(\log n)$ time. We assume that the combine cost takes constant time. We append the lengths of the blocks we combined as a tuple to an auxiliary array, which takes constant time as well. 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 times, so the total time for running this greedy algorithm is $O(n \log n)$.

(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. The output is an array of length $1000n$. We then go through the length $1000n$ element array one element at a time and add the smallest values up and increase the counter of the index corresponding to the sum by one. Since we have to do $O(n)$ additions, this algorithm will run in $O(n)$ time.

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.

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

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

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

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 $m^{\log_2 1} = 1$.

(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 $O(n \log n)$.

(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. 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 works is because insertion sort will compare it with at most n elements in the array, but unlike normal insertion sort, we only have to run it on one element. 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 use radix sort with $k = N$. Thus, we can sort the elements in the array in $O(3(N + N))$ time. After that, we can go through each interval and run the following algorithm. For the first interval, if 1 is not included, then we return 1. After that, we keep track of the maximum value included every time. If the minimum of the next interval is $> 1+$ this maximum value, we return $1+$ this maximum value. Otherwise, we set the new maximum value equal to the value of this upper bound for this new interval. We keep going on until we reach the end of the intervals.

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 $\text{MAX\_PORT} = n^6$.

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.

Write something here!