Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — September 27, 2016
Professors Erik Demaine, Debayan Gupta, and Ronitt Rubinfeld — Problem Set 2

Problem Set 2

All parts are due on October 13, 2016 at 11:59PM. Please download the .zip archive for this problem set. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Part A

Problem 2-1. [10 points] **Back-solving recurrences**

(a) [2 points] If $T(n) = \Theta(n^2 \log n)$ is a solution to $T(n) = aT(n/2) + \Theta(n^2)$, where a is a *positive integer*, find all possible values of a.

Solution: Since $f(n) = \Theta(n^2)$, we hope to be in Case 2 of the Master Theorem, so we solve the equation $n^{\log_2 a} = n^2$, which gives $a = 4$.

(b) [2 points] If $T(n) = \Theta(n^2)$ is a solution to $T(n) = aT(n/3) + \Theta(n)$, where a is a *positive integer*, find all possible values of a.

Solution: The desired runtime is polynomially larger than $f(n)$ so our only hope is $n^{log_3 a} = n^2$, which gives $a = 9$.

(c) [2 points] If $T(n) = \Theta(n^2)$ is a solution to $T(n) = 4T(n/b) + \Theta(n^2)$, where b is a *positive real number*, find all possible values of b.

Solution: $f(n)$ is our desired runtime, so $n^{log_b 4}$ should be polynomially less than n^2. $b > 2$ would satisfy this requirement.

(d) [2 points] If $T(n) = \Theta(n^{6.006})$ is a solution to $T(n) = 5T(n/b) + \Theta(n^5)$, where b is a *positive real number*, find all possible values of b.

Solution: $\Theta(n^{6.006})$ is polynomially larger than $f(n)$, so we must set $n^{log_b 5} = n^{6.006}$, which gives $b = 5^{\frac{1}{6.006}}$.

(e) [2 points] If $T(n) = \Theta(n^2)$ is a solution to $T(n) = 6T(n/6) + f(n)$, find one possible function f.

Solution: A possible function is $f(n) = n^2$.

Problem 2-2. [20 points] **Sorting a Rectangle**

In this problem, you are given a 2D array of integers A that has n rows and m columns. (Assume m is *much* (e.g., exponentially) smaller than n.) Array A has one special property: the integers in every row of A and every column of A are non-decreasing. More formally, $A[i, j] \leq A[i, j+1]$ for every $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, m-1$, and $A[i, j] \leq A[i+1, j]$ for every $i = 1, 2, \ldots, n-1$ and $j = 1, 2, \ldots, m$.

For example, one possible such 2D array is the following:

$$\begin{bmatrix} 1 & 2 & 4 & 8 \\ 3 & 5 & 5 & 10 \\ 4 & 6 & 19 & 30 \end{bmatrix}.$$

Describe and analyze an algorithm with running time $O(nm \lg m)$ that produces a sorted 1-D array of length nm that has exactly the same elements as A. For example, for the 2D array given above, the algorithm should return $[1, 2, 3, 4, 4, 5, 5, 6, 8, 10, 19, 30]$.

Solution: Note that it is possible to sort the matrix A by constructing a 1-D array of size nm that contains all the elements of A, followed by sorting the array with one the of the $O(N \log N)$ sorting algorithms we have seen so far. However, this algorithm would take $O(nm \lg(nm))$, which is not efficient enough.

In order to improve upon the $\Omega(N \log N)$ lower bound for comparison-based sorting we have seen in class, we have to take advantage of the structure of the matrix A. In particular, we have to use the fact that every row and every column of A is sorted. As we are going to see, it is possible to design an $O(nm \log m)$ algorithm by only using the fact that every column is sorted.

To that end, we can view A as a sequence of m sorted integer arrays, each of which contains n elements. When $m = 2$, we need to merge two sorted arrays of equal size. Note that this is precesily the same problem that we solve as a subroutine in Merge Sort. However, we need to generalize the algorithm so that it works when $m > 2$.

One way to solve the problem is by maintaining a list of m pointers, one for each of the m arrays, that points to the smallest element of the corresponding array that has not been merged yet. Then, we can find the minimum element among those m elements, add the corresponding element to the resulting array, and update the respective pointer to point to the next element of the array (which is now going to be the smallest one that has not been merged yet). However, such an algorithm would take $O(nm^2)$ because there are nm elements in total to be merged, and finding the minimum at each step takes $O(m)$.

Fortunately, if we look at the type of operations we are doing, we can see that we can use a heap to improve the running time to $O(nm \log m)$. Indeed, at every step of the algorithm, we can to maintain a heap with at most m elements, which are the smallest elements from each of the m arrays that have not been merged yet. Extracting and removing the minimum among those elements takes $O(\log m)$ time. Then, to obtain the heap for the next step, we only need to add at most one element to it, which also takes $O(\log m)$.

Finally, note that we can also use an AVL tree to implement the above algorithm in $O(nm \log m)$ time.

Problem 2-3. [25 points] **Sorting Venture Capital Offers for 6006LE**

The code you wrote for the hip new 6006LE search engine in the last problem set is attracting the interest of venture capitalists, who are lining up to bring you truckloads of venture capital dollars—literally!

You have a parking lot with parking spots numbered 1, 2, $\ldots$, n, to which n venture capitalists drive money-filled trucks of sizes a_1, a_2, $\ldots$, a_n (leaving no extra parking spaces). You'd like to sort the trucks by size (so you can work your way down the list of trucks later talking with the venture capitalists about their terms), but you can only see along a limited distance of the parking lot at any one time, so (1) you can only compare the sizes of two trucks at most k parking spots away from each other[1], and (2) you can only tell trucks at distance at most k from each other[1] to swap places.

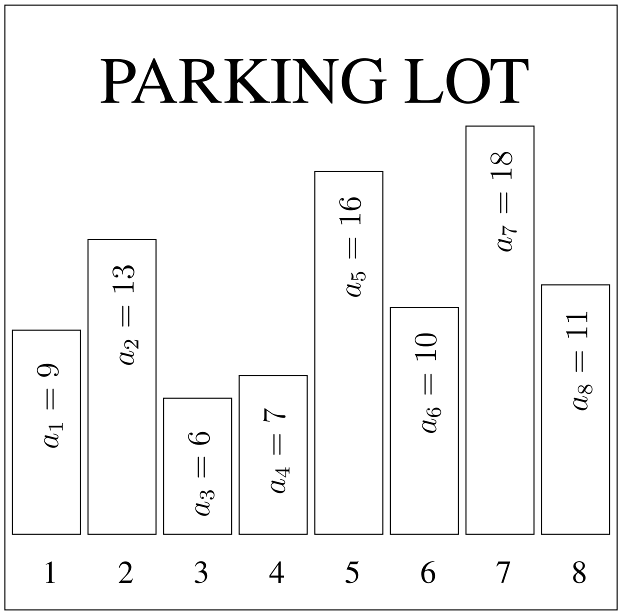

Figure 1: Bird's-eye view of a parking lot.

(a) [5 points] Design an algorithm that *switches* two arbitrary trucks (not necessarily at distance at most k) in $O(n/k)$ truck swaps (each of distance at most k), and returns any other trucks that were moved in the process to their original positions.

Solution: If the trucks to be swapped are at positions x and y with $y > x$, first swap the truck that was at position y down k spaces at a time until it's within k spaces of x: that is, swap trucks y and $y - k$, then $y - k$ and $y - 2k$, and so on until you swap trucks $y - (\lfloor \frac{y-x}{k} \rfloor - 1)k$ and $y - (\lfloor \frac{y-x}{k} \rfloor)k$. Then swap those two trucks (x and

[1]That is, only trucks in spots i and j where $|i - j| \leq k$

$y - (\lfloor \frac{y-x}{k} \rfloor)k)$, and undo all the original swaps. The number of swaps needed to get the trucks originally in positions x and y within distance k of each other is $O(n/k)$, the number of swaps needed to swap them is 1, and the number of swaps needed to return the truck from position x to y is $O(n/k)$, for a total of $O(n/k)$.

(b) [5 points] Design an algorithm that *compares* two arbitrary trucks (not necessarily at distance at most k) in $O(n/k)$ truck swaps (each of distance at most k), and returns any trucks that were moved in the process to their original positions.

Solution: If the trucks to be compared are at positions x and y with $y > x$, first swap y and $x + 1$ in time $O(n/k)$ by the previous part, then compare them, then swap $x + 1$ and y by the previous part.

(c) [15 points] Describe an algorithm that sorts the trucks in $O((n^2 \log n)/k)$ truck swaps. (*Hint:* modify an in-place $\Theta(n \log n)$ sorting algorithm, and possibly use the previous two parts.)

Solution: Use heapsort, which takes $O(n \log n)$ comparisons and $O(n \log n)$ swaps, but replace each swap with the swap from part a and each comparison with the comparison from part b, for a total of $O((n/k) * n \log n) = O((n^2 \log n)/k)$ truck swaps.

(d) [10 points] **(Extra credit)** Describe an algorithm that sorts the trucks in $O((n^2 \log k)/k)$ truck swaps.

Solution: Define a procedure called subregion-sort-1, where you sort $\lceil n/k \rceil$ subdivisions of the overall list by calling heapsort on k different local regions: parking spots 1 to k, $k + 1$ to $2k$, and so on. Define a procedure called subregion-sort-2 similarly, but shifted by $k/2$: first spots 1 to $k/2$, then $k/2 + 1$ to $3k/2$, and so on.

For the algorithm itself, first call subregion-sort-1, then subregion-sort-2, then subregion-sort-1 again, and so on, $4n/k$ times.

After i steps, the smallest $k/2$ elements are outside the last $ik/2$ spots or in place. After $i + 2$ steps, the next-smallest $k/2$ elements are outside the last $ik/2$ spots or in place, because by induction no elements smaller than them are in the swaps that swap things with the last $ik/2$. After $i + 2t$ steps, the smallest $kt/2$ elements are outside the last $ik/2$ spots or in place, by the same argument. After $2n/k + 2n/k$ steps, the smallest n elements are outside the last n spots or in place, that is, the list is sorted.

Part B

Problem 2-4. [45 points] **F1 Kart Racing**

Bowser is participating in the F1 Kart Racing Finals that take place on a circular track of integer length L. There are N competitors in total, indexed $0, 1, \ldots, N-1$; Bowser is index 0. At time $t = 0$ (the start of the race), competitor i starts at integer position p_i ($0 \leq p_i < L$) with a kart that has an integer velocity of $v_i \geq 0$. Assume that all competitors start from distinct starting positions (i.e., $p_i \neq p_j$ for all $i \neq j$), that all karts have distinct velocities (i.e., $v_i \neq v_j$ for all $i \neq j$), and that karts start at full speed (unchanging velocity v_i). Thus, at time t, kart i will be at position $(p_i + t \cdot v_i) \bmod L$ (because the track is circular so positions 0 and L are the same).

The race defines a ranking of competitors as follows. The first competitor that is passed by another competitor from behind is taken out of the competition. When only one competitor is left in the competition, they are deemed the winner, receive eternal glory, and get first position in the final ranking. A competitor s is taken out of the race by another competitor f if and only if $v_f > v_s$ and neither s nor f gets taken out of the race before f catches s; the time when this will happen is given by the following function:

$$losetime(f, s) = \begin{cases} \dfrac{p_s - p_f}{v_f - v_s}, & \text{if } p_s > p_f, \\[2ex] \dfrac{(L - p_f) + p_s}{v_f - v_s}, & \text{if } p_s < p_f. \end{cases}$$

The rank of a competitor i is defined to be 1 if they win the competition (and their finish time is $+\infty$), or $k + 1$ where k is the number of competitors whose finish time is strictly larger than i's finish time.

Note that this ranking can differ from the order of the karts' velocities, because of the differing starting positions of karts and the circular track which encourages "lapping" of other karts.

We are going to design and implement an $O(N \log N)$ algorithm that computes Bowser's final rank. (This way, we can bet all of our 6006LE venture capital and vastly multiply our resources!)

(a) [10 points] As a warmup, come up with an $O(N^2)$ algorithm that finds Bowser's final rank. (*Hint:* Can you find when competitor i leaves the race in $O(N)$ time?)

Solution: We will start with a claim.

Claim 1 *Competitor* i *leaves the race at time*

$$leavetime(i) = \min_{j \in \{1,2,\ldots,N : v_j > v_i\}} losetime(j, i).$$

Note that, competitor i can only be caught by competitors that are faster. Among the competitors that are faster than competitor i, the competitor that will catch i is the one that reaches i in the earliest possible time; this is precisely why we take the minimum in the above expression. Let us introduce some notation.

Definition 1 *Let $i^* \in \{1, 2, \ldots, N\}$ be some competitor that will catch competitor i (thus $i \neq i^*$) at time $leavetime(i)$. In other words*

$$i^* = \underset{j \in \{1,2,\ldots,N : v_j > v_i\}}{\arg\min} \; losetime(j, i).$$

It is tempting to think if the above claim holds. It is clear that competitor i will be part of the competition prior to $leavetime(i)$ because nobody could have caught him or her earlier in the competition. However, it may be the case that competitor i^* has already been eliminated prior to $leavetime(i)$ thus showing that the above claim does not hold. We are going to prove that this is not possible.

Lemma 2 *Competitor i^* will catch competitor i at time $leavetime(i)$.*

Proof. For the sake of contradiction, suppose that competitor i^* has been eliminated from the competition by some competitor k at some time t' earlier than $leavetime(i)$. Consider the race at time t'. It must be the case that $v_k > v_{i^*}$ for otherwise competitor k would not have been able to catch competitor i^* at all. Suppose that the distance between competitors i and i^* is d at the time when competitor k catches i^*. Competitor i^* would have needed additional $\frac{d}{v_{i^*} - v_i}$ time to catch competitor i, but competitor k only needs $\frac{d}{v_k - v_i}$ time to catch i which is strictly smaller than $\frac{d}{v_{i^*} - v_i}$. Thus competitor k would have caught competitor i before $leavetime(i)$. This is a contradiction, because, according to the definition, competitor i^* can catch i at the earliest possible time among all competitors.

Corollary 3 *We conclude that we $leavetime(i)$ is indeed the time when competitor i is going to leave the competition. This proves the above claim.*

We can compute $leavetime$ for a single competitor i in $O(N)$ because a single call to $losetime$ takes constant time, and we need at most $N - 1$ calls in total.

We can compute $leavetime(i)$ for all $i = 0, 1, \ldots, N - 1$ in $O(N^2)$ time. In order to find the rank of Bowser, we need to compare his leaving time with that of all other competitors, and apply defition of ranks given in the problem statement; this takes $O(N)$ time. Therefore, this algorithm takes $O(N^2)$ in total to compute the final rank of Bowser.

(b) [3 points] We will avoid using floating-point computations (such as noninteger division), which are inexact, and only use exact fractions via the class `Fraction`. In order to do that, we need to be able to compare two fractions. Fill in the necessary code in the `compare` method.

Solution: To compare two fractions, we can express $\frac{n_1}{d_1} < \frac{n_2}{d_2}$ as $n_1 d_2 < n_2 d_1$.

```python
def compare(a, b):
    """
    Return True if a is smaller than b, or False, otherwise.

    Arguments:
        a (Fraction): The first fraction.
        b (Fraction): The second fraction.

    Return:
        bool: A boolean that is True if a < b, or False, otherwise.
    """
    return a.num * b.den < a.den * b.num
```

(c) [2 points] Write the code for the function $losetime(f, s)$ defined above.

Solution: Note that we need to use the `Fraction` class to implement $losetime$ correctly.

```python
def losetime(p_s, v_s, p_f, v_f, L):
    """
    Return when (p_f, v_f) is going to take over (p_s, v_s).

    Return the time when the kart (p_s, v_s) is going to take over
    the kart (p_f, v_f) on a track of length L.

    Arguments:
        p_s (int): The starting position of the slower kart.
        v_s (int): The top velocity of the slower kart.
        p_f (int): The starting position of the faster kart.
        v_f (int): The top velocity of the faster kart.
        L (int): The length of the track.

    Preconditions:
         (1) 0 <= p_s, p_f < L;
         (2) v_s < v_f
         (3) p_f != p_s

    Return:
        Fraction: The time when the faster kart is going to
        take over the slower one.
    """
    if p_s > p_f:
        return Fraction(p_s - p_f, v_f - v_s)
    elif p_s < p_f:
        return Fraction((L - p_f) + p_s, v_f - v_s)
    else:
        raise Exception("There should not be two cars with the same
           speed.")
```

(d) [5 points] We need to maintain a data structure of the relative order of all competitors. This is to say that, for every competitor i, this data structure maintains the competitor behind i and the one in front of i. This data structure is implemented by two Python dictionaries `behind` and `ahead`. Write the method `remove(`i`)`, which removes competitor i from the data structure, and correctly maintains the `behind` and `ahead` links. This should work in $O(1)$ time.

Solution: To solve this part, we need to remove 1 element from both `behind` and `ahead`, as well as update one element from both. Therefore, we need to remove 2 elements and update 2 elements from Python dictionaries. This takes $O(1)$ in total.

```
def remove(i, ahead, behind):
    """
    Update the (ahead, behind) data structure by removing competitor
         i.

    Note that this method does not return anything.

    Arguments:
        i (int): The id of the competitor.
        ahead (dict): A dictionary where the competitor ahead of
                competitor j is given by ahead[j].
        behind (dict): A dictionary where the competitor behind
                competitor j is given by behind[j].

    Preconditions:
        (1) i is present in both dictionaries.
        (2) for every competitor i present in either ahead or
                behind, we have i = ahead[behind[i]],
                as well as i = behind[ahead[i]]

    Return: Nothing.
    """
    assert i in behind
    assert i in ahead
    behind_me = behind[i]
    ahead_of_me = ahead[i]
    del behind[i]
    del ahead[i]
    ahead[behind_me] = ahead_of_me
    behind[ahead_of_me] = behind_me
```

One way to find the rank of all competitors is by simulating the race, and removing the competitors that are passed by in the order those passing events happen in time. For each competitor still in the race, consider the event of their passing the competitor *immediately* ahead (we do not look at the cases when the competitor ahead is faster). If we look at all these events (of which there are at most N, one per competitor), then the one that happens earliest tells us the competitor that leaves the race first. Then, we can remove that competitor from consideration, and continue similarly.

(e) [5 points] Suppose that competitor Charlie passes competitor Bowser, competitor Alice is ahead of both of them, and competitor Deborah is behind both of them. What events might need to be added to or removed from consideration? Your answer may depend on the relative speeds v_a, v_b, v_c, and v_d of Alice, Bowser, Charlie, and Deborah.

Solution: Let $\mathcal{E}$ be our list of events for consideration, where each event is of the form (i, j) and corresponds to the event of competitor i catching competitor j.

Note that, when Charlie passes Bowser, we remove (Charlie, Bowser) from $\mathcal{E}$.

Suppose that $v_b > v_a$. Prior to Charlie catching Bowser, $\mathcal{E}$ contains (Bowser, Alice). However, since Charlie catches Bowser before Bowser has the chance to catch Alice, the event (Bowser, Alice) is no longer valid, and it needs to be removed from $\mathcal{E}$. This is Situation 1.

On the other hand, if $v_c > v_a$, we need to add (Charlie, Alice) to $\mathcal{E}$, because after Charlie catches Bowser, we should consider the possibility of Charlie catching Alice. Note that this event should be added regardless of the relative speeds of Bowser and Alice. This is Situation 2.

Finally, note that no events that involve Deborah need be added or removed.

Therefore, after competitor Charles passes competitor Bowser, we need to add $O(1)$ events to $\mathcal{E}$, and remove $O(1)$ events from $\mathcal{E}$ in order to obtain the list of events for considerations in the competition after eliminating Bowser.

(f) [20 points] In the previous part, you saw that at every step of the simulation, you need to find the event with the minimum time, remove $O(1)$ events, and add $O(1)$ events. For all of this, we can use the heap data structure. We will also need the data structure from part (d), but putting everything together results in an $O(N \log N)$ algorithm. Implement this algorithm in the `rank` method. (*Hint:* You may find the Python module `heapq` helpful.)

Solution: Combining the solutions to all previous parts lead to an $O(N \log N)$ solution. Note that in order to construct the `ahead` and `behind` data structure, we need to sort all karts according to their positions, but since sorting of N elements takes $O(N \log N)$, the overall runtime complexity is still $O(N \log N)$.

A possible $O(N \log N)$ implementation is given on the next two pages.

```
def rank(N, L, velocity, position):
    """
    Return the rank-of-0 (as defined in the problem statement).

    Arguments:
        N (int): The number of competitors.
        L (int): The length of the track.
        velocity (list[int]): The top velocities of all competitors,
                where the velocity of competitor i (0 <= i < N) is
                given by velocity[i].
        position (list[int]): The starting positions of all
                competitors, where the starting position of
                competitor i (0 <= i < N) is given by position[i].

    Preconditions:
        (1) len(velocity) = len(position) = N
        (2) all elements of velocity are distinct, and non-negative
        (3) all elements of position are distinct, and non-negative

    Return: The rank of competitor 0, which is a number
        between 1 and N, inclusive.
    """
    # Step 1: sort competitors by position, so that we can build
    # ahead-behind. This takes O(N log N) time.
    karts = sorted([(position[i], i) for i in range(N)])

    # Step 2: build the ahead-behind datastructure. Takes O(n) time.
    ahead, behind = {}, {}
    for i in range(N):
        ahead[karts[i][1]] = karts[(i + 1) % N][1]
        behind[karts[i][1]] = karts[(i - 1 + N) % N][1]

    event_q = []  # Create an empty min-heap.

    def add_event(i, j):
        """Add the event kart i takes over kart j to the heap."""
        assert velocity[i] > velocity[j]
        v_f, p_f = velocity[i], position[i]
        v_s, p_s = velocity[j], position[j]
        heapq.heappush(event_q, (losetime(p_s,v_s,p_f,v_f,L), i, j))

    # Step 3: initialize the priority queue with initial events.
    # Takes O(N log N).
    for i in range(N):
        if velocity[karts[i][1]] > velocity[karts[(i + 1) % N][1]]:
            add_event(karts[i][1], karts[(i + 1) % N][1])

    eliminated = set()  # We maintain a set of all eliminated
       competitors.
    my_finish_time = Fraction(2 * L, 1)
    # Step 4: do the simulation. Takes O(N log N)
    while len(event_q) > 0:
        time, fast, slow = heapq.heappop(event_q)
        if (fast in eliminated) or (slow in eliminated):
            # This event is no longer valid, because at least one
            # competitor is already eliminated. Ignore the event.
            # This is Situation 1 from part e).
            continue
        if slow == 0:
            my_finish_time = time  # Record our finishing time.
        if my_finish_time < time:
            # We are out of the game, and everybody still in will
            # be eliminated at time strictly larger than ours.
            break
        eliminated.add(slow)  # Eliminate slow.
        ahead_of_slow = ahead[slow]
        remove(slow, ahead, behind)  # Remove slow.
        if velocity[fast] > velocity[ahead_of_slow]:
            # We add the event of fast overtaking ahead_of_slow
            # to the heap. This is Situation 2 from part e).
            add_event(fast, ahead_of_slow)

    # Step 5: return the rank of 0. Takes O(1)
    return 1 if 0 not in eliminated else (N - len(eliminated)) + 1
```