CS 124 Data Structures and Algorithms — Spring 2015

Problem Set 4 solutions

Problem 1

There are n young wizards standing in a row at Hogwart's, sorted alphabetically by last name. The sorting hat would like to assign a house to each one: Hufflepuff, Gryffindor, Ravenclaw, or Slytherin. The hat's job is complicated by the fact that some children's parents absolutely hate certain houses, and if the hat assigned their child to such a house then the parents would complain to the headmaster, who in turn would burn the sorting hat to ashes. Furthermore the sorting hat does not like to put two alphabetically adjacent students into the same house. Given the sorted order of students, and constraints on certain students not being assigned to certain houses, is an assignment even possible?
Consider the following divide-and-conquer algorithm to answer the above question, where $S[i]$ is student i, and houses$[i]$ is the list of houses student i may be assigned to:

```
algorithm is_possible(houses):
  n = length of houses

  if n == 0:
    return true

  possibilities = houses[middle]

  # split array (excluding middle element)
  left = houses[0...middle-1]
  right = houses[middle+1...n-1]

  if length of possibilities >= 3:
    return is_possible(left) and is_possible(right)
  else:
    for each possibility in possibilities:
      # try assigning middle student to house 'possibility'
      a_left = 'left' with 'possibility' removed from houses[middle-1], if present
      a_right = 'right' with 'possibility' removed from houses[middle+1], if present
      if is_possible(a_left) and is_possible(a_right)
        return true

  return false
```

What's the running time of the above algorithm?

Solution If the base case doesn't apply to our input, then the algorithm may make recursive calls. If there are at least three housing possibilities for the middle wizard, the algorithm makes two recursive calls, one for each half of the array. If there are less than three, then the algorithm will try each possibility, making two recursive calls per possibility. In all, the algorithm makes the largest number of recursive calls at a particular level when there are 2 possibilities to try (4 recursive calls.)
To determine runtime, we use the Master Theorem to unwind the following recurrence:

$$T(n) \leq 4T(n/2) + O(1)$$

$$T(n) = O(n^2)$$

Problem 2

You are given an n-digit positive integer x written in base-2. Give an efficient algorithm to return its representation in base-10 and analyze its running time. Assume you have black-box access to an integer multiplication algorithm which can multiply two n-digit numbers in time $M(n)$ for some $M(n)$ which is $\Omega(n)$ and $O(n^2)$.

Solution Solution based on approach taken at https://cseweb.ucsd.edu/classes/wi14/cse202-a/ans414.pdf.
Consider the n-digit base-2 integer x as a string of base-2 digits, $x[0...n]$. Assume n is a power of 2. If not, we can pad the left side of the input integer with zeros. This will increase the number of digits by less than a factor of 2, which will not affect the big-O runtime.
We offer a recursive algorithm. In the base case, for an input of size 0, output 0.
Divide the integer into two halves, $x[0...\frac{n}{2}]$ and $x[\frac{n}{2}...n]$. Then, recursively convert each half into a base-10 integer. Call these (converted) base-10 integers y_l and y_r. Then, the desired integer can be found by computing $y_l * 2^{n-\frac{n}{2}} + y_r$. Note that, $p = 2^{n-\frac{n}{2}}$ must be in base-10 also. One way we could get the base-10 representation of p is by converting it recursively, too, along with the halves of the input integer. Then we make three recursive calls to our procedure, each with a base-2 integer which is roughly half the size of the problem input. After we obtain these base-10 integers, we have to perform an integer multiplication $(y_l \cdot p)$ and an integer addition $(y_l \cdot p) + (y_r)$. A recurrence that describes our runtime, then, is
$T(n) = 3T(\frac{n}{2}) + M(\frac{n}{2}) + O(n)$
where $O(n)$ is the time of the addition, and $M(n)$ is the time required to multiply two n-digit numbers.
Suppose multiplication of two n-digit numbers ($M(n)$) takes $O(n^k)$ time for $1 \leq k \leq 2$. By Master Theorem, if $\log_2 3 < k$, then the algorithm runtime is $O(n^k)$. If $\log_2 3 > k$, then the algorithm runtime is $n^{\log_2 3}$. And if $\log_2 3 = k$, then the algorithm runtime is $O(n^k \log n)$.

We can make a slight improvement to the algorithm. Before, we said that to obtain the base-10 representation of p, we'd call our function recursively. However, note that in our recursive call, we find the base-10 representation of $2^{n/4}$. So we can store the powers computed in the subcalls in a lookup table, and then compute the needed power (in this case, $2^{\frac{n}{2}}$) by squaring $2^{\frac{n}{4}}$. (We could also see if we've already computed $2^{\frac{n}{2}}$.) This squaring is just an integer multiplication, replacing the recursive call we originally made to do that last conversion. In all, our new recurrence becomes $T(n) = 2T(\frac{n}{2}) + 2M(\frac{n}{2}) + O(n)$. We can then do a case analysis on $M(n)$ to determine the algorithm's runtime. If $M(n)$ is $\Theta(n)$, then the algorithm takes $n \log n$ time. Otherwise the algorithm takes $M(n)$ time total.

Problem 3

Recall the following text search problem from class. There is an alphabet Σ and two strings $P \in \Sigma^m$, $T \in \Sigma^n$, $n \geq m$ (i.e. strings of length m, n, respectively, made up of characters in Σ). We would like to output a list of all indices i such that $T[i, i+1, \ldots, i+m-1] = P$, i.e. $t_{i+j-1} = p_j$ for $j = 1, \ldots, m$. In class, when $\Sigma = \{0, 1\}$ we showed how to use FFT to solve this problem in $O(n \log n)$ time (assuming a computer supporting infinite precision complex arithmetic). Making the same precision assumptions:

(a) Improve this time to $O(n \log m)$.

(b) Deal with the case when P (but not T) has "don't care" symbols *. A * symbol can match either a 0 or a 1.

(c) Show how to deal with the case $|\Sigma| > 2$, without don't care symbols. You can assume $\Sigma = \{1, 2, \ldots, |\Sigma|\}$.

(d) Give a solution for $|\Sigma| > 2$ **with** don't care symbols, which can match any character in $\Sigma = \{1, 2, \ldots, |\Sigma|\}$. An $O(n \log m)$ solution (which is possible) naturally implies you don't have to separately do parts (a) through (c).

Solution

Improving time to $O(n \log m)$.

We first pad the string of length n with arbitrary characters until its length is a multiple of m. Then we consider $\lceil \frac{n}{m} \rceil - 1$ substrings of T of length $2m$. These substrings are
$T_0 = T[0, 1, \ldots, 2m-1]$
$T_1 = T[m, m+1, \ldots, 3m-1]$
$\vdots$
$T_{\lceil \frac{n}{m} \rceil - 2} = T\left[m\left(\lceil \frac{n}{m} \rceil - 2\right), m\left(\lceil \frac{n}{m} \rceil - 2\right) + 1, \ldots, m\left(\lceil \frac{n}{m} \rceil\right) - 1\right]$
We then do shifted dot products between P and each T_i. Note to check whether P matches the substring of T starting at position j (0-indexed), it suffices to check whether P matches $T_{\lfloor j/m \rfloor}$ at position $j \mod m$.

Dealing with don't care symbols.

During the step where we map $1 \to 10$ and $0 \to 01$, we map $* \to 11$. Since T is not permitted to have don't care symbols, the only dot products we may see are given by the following table:

	0	1	*
0	1	0	1
1	0	1	1

As such, we get a result of 1 if and only if the two characters count as a match, and get a result of 0 if and only if the two characters don't match, so we can use the same approach as before.

Dealing with larger alphabets.

To compare the string P with the string $T[i, i+1, \ldots, i+m-1]$, we compute the value of

$$\sum_{j=0}^{m-1} (p_j - t_{i+j})^2 .$$

Since each term in the sum is strictly non-negative, if this evaluates to zero, each term in the sum must evaluate to zero, implying that the strings match. Similarly, if the two strings match, each term in the sum evaluates to zero, so the entire sum is equal to zero. Therefore, this sum is zero if and only if the two strings match. We rewrite this as

$$\sum_{j=0}^{m-1} p_j^2 - 2\sum_{j=0}^{m-1} p_j t_{i+j} + \sum_{j=0}^{m-1} t_{i+j}^2.$$

The first term can be computed once in time $O(m)$ and reused for all i (it does not depend on i). Obtaining the middle summation for all i can be obtained in $O(n \log m)$ time by doing shifted dot products on overlapping substrings of T of size $2m$ as in (a). By computing prefix sums $\tau(i) = \sum_{i=0}^{i} t_i^2$ (with $\tau(-1) = 0$), we can obtain the third summation in $O(1)$ time as $\tau(i+m-1) - \tau(i-1)$. The table of prefix sums can be computed in time $O(n)$ total with a single for loop, since $\tau(-1) = 0$ and $\tau(i) = t_i^2 + tau(i-1)$.

Larger alphabets and don't care symbols.

First, we iterate through the strings, replacing all don't care symbols with 0's. We modify the approach in the previous part, considering instead the sum

$$\sum_{j=0}^{m-1} (p_j - t_{i+j})^2 p_j t_{i+j}$$

to determine whether the strings P and $T[i, i+1, \ldots, i+m-1]$ match. Since every character in the alphabet is non-negative, each term in this sum is non-negative. Hence, if this sum is zero, then every term in the sum must be zero. If a given term is zero, then either $p_j = t_{i+j}$, $p_j = 0$, or $t_{i+j} = 0$, so either the characters match or at least one of them is a don't care symbol. Thus, the sum evaluates to zero precisely when the two strings match. On the other hand, if the two strings match, each term evaluates to zero, so the whole sum evaluates to zero. Therefore, this sum is 0 if and only if the two strings match. We expand this to

$$\sum_{j=0}^{m-1} p_j^3 t_{i+j} - 2\sum_{j=0}^{m-1} p_j^2 t_{i+j}^2 + \sum_{j=0}^{m-1} p_j t_{i+j}^3.$$

Define the vector $p^{(k)}$ to be such that $p_i^{(k)} = p_i^k$, and do similarly for $t^{(k)}$. The first summation can be computed by doing shifted dot products on $p^{(3)}$ and t on overlapping length-$2m$ strings as in (a); the time is $O(n \log m)$. The second and third summations are similar (between $p^{(2)}$ and $t^{(2)}$, and between p and $t^{(3)}$, respectively).

Problem 4

We construct an infinite sequence of arrays A_1, A_2, A_3, ... in the following recursive fashion. First, we specify that $A_1 = [1]$. For $k > 1$, we recursively define A_k to be two copies of A_{k-1} put together, with the number k inserted between the two lists.
To illustrate the above algorithm:

$$A_1 = [1]$$

$$A_2 = [1, 2, 1]$$

$$A_3 = [1, 2, 1, 3, 1, 2, 1]$$

$$A_4 = [1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1]$$

and so on.
Devise an efficient algorithm that accepts some k and two intervals $[a, b)$ and $[c, d)$, and determines the length of the longest subarray contained in both the arrays $A_k[a : b]$ and $A_k[c : d]$. An array C is a subarray of an array B if there exist i and j such that $C = B[i : j]$. As always, you must prove the algorithm's correctness and analyze its runtime.

Solution The idea of the algorithm is as follows - when comparing two intervals in A_k, there are three possibilities - either neither of the intervals contains the value k, exactly one contains the value k, or both contain the value k. In the first case, the problem easily reduced to a question about comparing two intervals in A_{k-1}. In the second case, we see that the maximum overlapping solution cannot contain the value k as k is in only one of the two intervals (without loss of generality let k be contained in $[a, b)$), so thus the problem is reduced to two questions about comparing two intervals in A_{k-1}, one with comparing $[c, d)$ against the portion of $[a, b)$ before k and the other comparing $[c, d)$ against the portion of $[a, b)$ after k. In the third case where both intersect k, we either include k in the optimal solution, or we don't

- the former case required no branching, while the latter case can give us up to four branches.

The key observation required is that if one interval is completely contained in the other, the optimal solution is simply the length of the shorter interval. Using this observation, we can see that we only ever make a constant number of branches.

Now we will write out our algorithm more clearly. First, we note that the length of A_k is $2^k - 1$, which can easily be seen through induction. In each case, if we ever get that interval 1 is contained inside interval 2 or vice-versa, we return the minimum of the two lengths.

Case 1: 2^{k-1} is not in $[a, b)$ or $[c, d)$ (i.e. the value k is in neither interval) - return the overlap for

$$[a \bmod 2^{k-1}, b \bmod 2^{k-1}) \text{ and } [c \bmod 2^{k-1}, d \bmod 2^{k-1}) \text{ in } A_{k-1}$$

Case 2: 2^{k-1} is contained in precisely one interval (without loss of generality $[a, b)$) - return the max of the overlap for

$$\begin{aligned} [a \bmod 2^{k-1}, 2^{k-1} - 1) \text{ and } [c \bmod 2^{k-1}, d \bmod 2^{k-1}) &\quad \text{in } A_{k-1} \\ [0, b \bmod 2^{k-1}) \text{ and } [c \bmod 2^{k-1}, d \bmod 2^{k-1}) &\quad \text{in } A_{k-1} \end{aligned}$$

Case 3: 2^{k-1} is contained in both intervals - return the max of the overlap for

$$\begin{aligned} [a \bmod 2^{k-1}, 2^{k-1} - 1) \text{ and } [c \bmod 2^{k-1}, 2^{k-1} - 1) &\quad \text{in } A_{k-1} \\ [a \bmod 2^{k-1}, 2^{k-1} - 1) \text{ and } [0, d \bmod 2^{k-1}) &\quad \text{in } A_{k-1} \\ [0, b \bmod 2^{k-1}) \text{ and } [c \bmod 2^{k-1}, 2^{k-1} - 1) &\quad \text{in } A_{k-1} \\ [0, b \bmod 2^{k-1}) \text{ and } [0, d \bmod 2^{k-1}) &\quad \text{in } A_{k-1} \end{aligned}$$

and the value

$$\min(b, d) - \max(a, c)$$

Now, we need to show that in fact we only branch a fixed number of times. To do this, we will first look at the case when we compare "prefixes" and "suffixes." Let a prefix be a string starting at position 0, and a suffix being a string ending at position $2^k - 1$. We note that if we are comparing two prefixes, we will immediately terminate as one of them must be contained in the other, and the same for if we are comparing two suffixes. Now, consider the case where we compare a prefix against a suffix. Consider the first time we enter a case that is not case 1. Suppose that only the prefix contains the k that we encounter - we note then that the entire suffix is contained in A_{k-1} which is completely contained in the prefix, and thus the optimal solution is the length of the suffix (we could hardcode this in to our

solution above, but all that matters is that we do not make more branches - we only continue computing one branch (the second half of the prefix against the suffix)). If both contain k, again only one branch is continued - the second half of the prefix agains the first half of the suffix (again noting that we could terminate here if we noted this case in the recurrence above, but it doesn't matter since we aren't making more branches).

Note then that there is only one time in which we are in case 2 or 3 and the first interval is not a prefix or suffix, and only one time in which we are in case 2 or 3 and the second interval is not a prefix or suffix. As each case only generates two branches (note that two of the branches in case 3 immediately terminate) thus we only branch at most 4 times in total. Thus, the runtime for the algorithm is big O of the depth of recursion, or $O(k)$.

Problem 5

Solve "Problem A - Zoo" on the programming server; see the "Problem Sets" part of the course web page for the link.

Solution
Note: Credit to http://codeforces.com/contest/459/problem/D for the original version of this problem.
Our goal in this problem is to count pairs where a certain condition is met. Before counting pairs, how can we quickly determine that a pair meets the condition?
We need to know, for the first exhibit in the pair, how many same-species animals are located west of that exhibit, and, for the second exhibit in the pair, how many same-species animals are located east of that exhibit. So we need to figure out the number of similar animals west and east of each animal. To compute this efficiently, we can make two passes through the array: an "easterly" pass and a "westerly" pass. Throughout the pass, we can keep track of the number of animals of each type that we have seen so far, using some sort of counter data structure. If the species ids were in a small range of numbers, a bucket sort-like approach to tracking counts would work: declare an array whose length is the range of the species IDs, and update counts appropriately. However, we are not as lucky: species IDs could be quite large, according to the problem description! In order to use such an array-based approach, we'll need to find some way to **map our species IDs down into a smaller range**. One way to do this: sort the original input array of exhibits by species ID. Then for each animal exhibit in the original array, replace its species ID with the index of the first appearance of that species ID in the sorted order. Note that these indices will be in the range $[0, N)$, reducing our range of species IDs down to the size of the input array. To find the index of this first appearance quickly, we can just use binary search on our sorted array. This precomputation takes time $\Theta(N \log N)$: we have to sort, and then do N binary searches. This will turn out to not affect our asymptotic runtime.
A hash map or dictionary will also suffice to count the animals. If you don't know what these are, that's fine, we haven't covered hashing at this point in the course yet.

On the easterly pass, we'll iterate through the array in forward, sequential order and record the number of similar animals to the west, for each exhibit, using the information in the counter data structure. Similarly, on the westerly pass, we'll iterate through the array in reverse order and record the number of similar animals to the east. From this, we'll end up with two arrays, `west[]` and `east[]`.

Now, we can quickly determine whether or not a pair meets a certain criteria. But how do we count all the pairs? One approach we could take is: enumerate all pairs, check whether the pair meets the condition, and increment a counter if it does. Can we do better than a brute-force consideration of all pairs, which has an $O(N^2)$ runtime?

We can, using a divide-and-conquer approach. Consider the array of exhibits, and imagine splitting it into a left and right half, divided by a middle element. Then, every pair of elements either

- lies entirely in the left half or
- lies entirely in the right half or
- has one element in the left half and one element in the right half

Let's start by thinking recursively, and build from there. Suppose we had a function that could count the number of pairs that lie entirely in a half of the array. Then we could call this function twice, once for each half, and count the intra-half pairs. Now, to complete the problem, we'd just need to find a way to count the pairs that cross between halves of the array. At first glance, this doesn't seem any easier than the original problem. After all, there are still $O(N^2)$ pairs that cross the array to consider. However, what if our recursive function also sorted the halves of the `west[]` and `east[]` arrays that were passed recursively? That is, in addition to obtaining the amount of intra-half pairs, we also obtained a sorted version of the subarray of the `west[]` and `east[]` arrays.

Then, we can count the inter-half pairs using a modification of the "merge" step of merge sort. Recall that in the "merge" step of merge sort, two sorted lists are merged into one larger, sorted list. At every point in the procedure, a pointer tracks some position in the left half, and a pointer tracks some position in the right half of the list. In a step of "merge", we compare the list values at these pointer locations, and this comparison informs our decision about how to order the output array. Here, though, we'll use the comparison to count pairs. In particular, if the value of the `west[]` array at the left pointer location is greater than the value of the `east[]` array at the right pointer location, then we know how many inter-half pairs the current left index participates in. It is the number of elements in the right array (before and including) the current right pointer's element. Clearly the current pair of elements referenced by the pointers is one pair that counts towards the total, but also, we can pair the current left element with any right element smaller than the current right element. Since the array halves are sorted, we can easily figure that out. To maintain the invariant that the `west[]` and `east[]` arrays are sorted by the routine, we can also carry out the normal array "merge" procedure on these arrays.

All inter-half pairs are accounted for: every one of these pairs must have an element in the left half of the array, and we count all of the pairs that all of the left array elements participate in. This procedure is linear in the length of the array - our two pointers each make one full pass through their halves of the array.
In all, we end up modifying merge sort to count the pairs. To arrive at a final number of pairs, we can combine the result of this algorithm with our counts of the intra-half pairs that we found recursively.
A recurrence describing the runtime of this algorithm is

$$T(N) = 2T(\frac{N}{2}) + O(N)$$

just like merge sort!
We also need to account for the initial preprocessing to determine the `west`[] and `east`[] arrays, but that's $O(N)$ - two linear-time passes through the array.
So the runtime of the improved algorithm is $O(N \log N)$. The space usage is $O(N)$.
Below, we show an example solution, including the $O(N^2)$ and $O(N \log N)$ implementations.
This solution was adapted from the solution at http://codeforces.com/contest/459/submission/7495239.

```
SLOW = False

from heapq import merge
from bisect import bisect_left

inf = float("inf")

def count_inversions(i, j, li, ri):

        if j - i < 2:
                return 0

        # find inversions contained within either half of the array (recursively)
        mid = i + (j - i)/2
        inversions = count_inversions(i, mid, li, ri) + count_inversions(mid, j, li, ri)

        # consider cross-half inversions
        p1, p2 = i, mid
        while p1 != mid or p2 != j:
                fli = li[p1] if p1 < mid else inf
                fri = ri[p2] if p2 < j else inf
                if fli <= fri:
                    p1 += 1
                else:
                    inversions += mid - p1
                    p2 += 1

        # merge sorted arrays
        for k, x in enumerate(merge(li[i:mid], li[mid:j])):
                li[i + k] = x
        for k, x in enumerate(merge(ri[i:mid], ri[mid:j])):
                ri[i + k] = x

        return inversions

def count_inversions_slow(li, ri):
        num_inv = 0
        for i, ai in enumerate(li):
                for aj in ri[i+1:]:
                        num_inv += 1 if aj < ai else 0
        return num_inv

def solve(N, A):

        li = [0] * N
        ri = [0] * N

        # universe reduction: map integers in array to range {0... N-1}
        sortA = sorted(A)
        reduced = map(lambda x: bisect_left(sortA, x), A)
```

```
        counts = [0] * N

        # compute f(0, i, a[i]) for all i
        for i, a in enumerate(reduced):
                counts[a] += 1
                li[i] = counts[a]

        counts = [0] * N

        # compute f(j, N - 1, a[j]) for all j
        for i, a in enumerate(reversed(reduced)):
                counts[a] += 1
                ri[N - i - 1] = counts[a]

        print li
        print ri
        return count_inversions_slow(li, ri) if SLOW else count_inversions(0, N, li, ri)


if __name__ == "__main__":
        N = int(raw_input())
        A = map(int, raw_input().split())
        print solve(N, A)
```