Homework 4

CS/ECE 374B

Due 8 p.m. on Tuesday, October 6

All of this has happened before and all this will happen again.

1. Solve the following recurrences. For parts (a) and (b), give an *exact* solution. For parts (c) and (d), give an asymptotic one. In both cases, justify your solution.

(2) (a) $A(n) = A(n-1) + 2n - 1; A(0) = 0$

(3) (b) $B(n) = B(n-1) + \binom{n}{2}; B(0) = 0$

(2) (c) $C(n) = C(n/2) + C(n/3) + C(n/6) + n$

(3) (d) $D(n) = D(n/2) + D(n/3) + D(n/6) + n^2$

2. In class, we discussed the recursive algorithm for the Towers of Hanoi problem.

```
def hanoi(ndisks, source, dest, tmp):
    """ Move 'ndisks' from the 'source' tower to the 'dest' tower,
    using the 'tmp' tower as temporary space """
    if ndisks > 0:
        # recursively move stack of ndisks-1 disks to tmp tower
        hanoi(ndisks-1, source, tmp, dest)
        # move one disk from source to destination
        moveone(source, dest)
        # recursively move stack of ndisks-1 disks to dest tower
        hanoi(ndisks-1, tmp, dest, source)
    else:
        pass # do nothing
```

In the following, assume that the towers are numbered 0, 1, 2 and the standard task is to move n disks from tower 0 to tower 1 (i.e., hanoi(n,0,1,2))

(5) (a) Suppose that moveone had a restriction that either the source or the destination must be tower 0. Modify the recursive algorithm to abide by this restriction. Analyze *exactly* how many calls to moveone are needed to move n disks in your solution.

(5) (b) Suppose instead that you are give another call, moveall that can move an entire stack of disks from one tower to another, but moveall can only be called to move disks *from* tower 2. I.e., you may call moveall(2,0) or moveall(2,1), using it with any other arguments will cause an error.

Modify the algorithm to take advantage of moveall. Calculate the exact number of calls to moveone and moveall your algorithm makes for n disks.

(3) 3. (a) Suppose you have a string of n Christmas lights, numbered $1, \ldots, n$ that are wired in series. One of the lights is broken and you want to find out which. You have a multimeter that you can use to test whether any section of the string works. I.e., test(i, j) returns True if lights i through j (inclusive) are all working, and False if one of them is broken. Design a recursive algorithm to identify the broken light (you should assume there is exactly one) and analyze its runtime. For full credit your algorithm should make a sublinear number of calls to test (i.e., $o(n)$).

(3) (b) Suppose now that up to k lights may be broken. Modify your algorithm to find all the broken lights. How big can k be before your algorithm is no longer faster than testing each light?

(4) (c) In cryptography, an RSA key is the product of two large primes, $n = pq$. Each key n_i should use its own, randomly generated primes p_i and q_i; however, due to flaws in random number generators occasionally two keys will share one or both factors.[1] For any two correctly generated keys, $\mathsf{gcd}(n_i, n_j) = 1$, but if keys share a factor then $\mathsf{gcd}(n_i, n_j) \neq 1$.

You are given a large collection of t keys, $n_1, \ldots, n_t$ and want to find out whether any of them share a factor. Since GCD takes time to compute, you can use a batch approach to speed up your computation. $\mathsf{batchgcd}(i, j, k, l)$ computes the GCD of two batches of keys:

$$\mathsf{batchgcd}(i, j, k, l) = \mathsf{gcd}\left(\prod_{m=i}^{j} n_m, \prod_{m=k}^{l} n_m\right)$$

If $\mathsf{batchgcd}(i, j, k, l) \neq 1$ then one of keys $n_i, \ldots, n_j$ shares a factor with one of the keys $n_k, \ldots, n_l$. (Note that you will want your two batches to be non-overlapping, since a key n_i always shares prime factors with itself. I.e., you should have $1 \leq i \leq j < k \leq l \leq t$.)

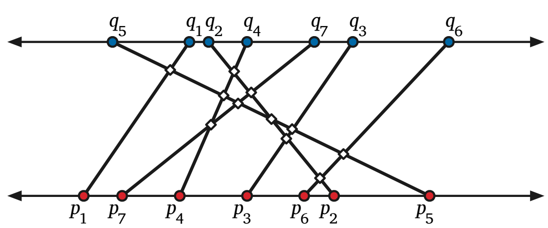
Design a recursive algorithm that finds a pair of keys with a shared factor in your collection of t keys and analyze its runtime. Your algorithm may assume there is exactly one such pair. For full credit your algorithm should make $o(t^2)$ calls to $\mathsf{batchgcd}$, which you can assume take constant time.

(0) (d) (Not to submit.) Can you modify the algorithm to find all pairs of keys with a shared factor without the assumption that there is exactly one?

(0) (e) (Not to submit.) Analyze the runtime of your algorithm if $\mathsf{batchgcd}$ takes logarithmic time in the size of the batches, i.e., $\Theta(\log(j - i) + \log(l - k))$?

Solved Problem

4. Suppose we are given two sets of n points, one set $\{p_1, p_2, \ldots, p_n\}$ on the line $y = 0$ and the other set $\{q_1, q_2, \ldots, q_n\}$ on the line $y = 1$. Consider the n line segments connecting each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect, in $O(n \log n)$ time. See the example below.



Seven segments with endpoints on parallel lines, with 11 intersecting pairs.

Your input consists of two arrays $P[1..n]$ and $Q[1..n]$ of x-coordinates; you may assume that all $2n$ of these numbers are distinct. No proof of correctness is necessary, but you should justify the running time.

Solution: We begin by sorting the array $P[1..n]$ and permuting the array $Q[1..n]$ to maintain correspondence between endpoints, in $O(n \log n)$ time. Then for any indices $i < j$, segments i and j intersect if and only if $Q[i] > Q[j]$. Thus, our goal is to compute the number of pairs of indices $i < j$ such that $Q[i] > Q[j]$. Such a pair is called an *inversion*.

We count the number of inversions in Q using the following extension of mergesort; as a side effect, this algorithm also sorts Q. If $n < 100$, we use brute force in $O(1)$ time. Otherwise:

- Recursively count inversions in (and sort) $Q[1..\lfloor n/2 \rfloor]$.

[1]See N. Heninger, Z. Durumeric, E. Wustrow, and J.A. Halderman, "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices," in *Proceedings of 21st USENIX Security Symposium*, 2012. https://factorable.net

- Recursively count inversions in (and sort) $Q[\lfloor n/2 \rfloor + 1 \mathinner{..} n]$.
- Count inversions $Q[i] > Q[j]$ where $i \le \lfloor n/2 \rfloor$ and $j > \lfloor n/2 \rfloor$ as follows:
  - Color the elements in the Left half $Q[1 \mathinner{..} n/2]$ bLue.
  - Color the elements in the Right half $Q[n/2+1 \mathinner{..} n]$ Red.
  - Merge $Q[1 \mathinner{..} n/2]$ and $Q[n/2+1 \mathinner{..} n]$, maintaining their colors.
  - For each blue element $Q[i]$, count the number of smaller red elements $Q[j]$.

The last substep can be performed in $O(n)$ time using a simple for-loop:

```
CountRedBlue(A[1..n]):
  count ← 0
  total ← 0
  for i ← 1 to n
      if A[i] is red
          count ← count + 1
      else
          total ← total + count
  return total
```

In fact, we can execute the third merge-and-count step directly by modifying the Merge algorithm, without any need for "colors". Here changes to the standard Merge algorithm are indicated in red.

```
MergeAndCount(A[1..n], m):
  i ← 1;  j ← m + 1;  count ← 0;  total ← 0
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i + 1;  total ← total + count
      else if i > m
          B[k] ← A[j];  j ← j + 1;  count ← count + 1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i + 1;  total ← total + count
      else
          B[k] ← A[j];  j ← j + 1;  count ← count + 1
  for k ← 1 to n
      A[k] ← B[k]
  return total
```

We can further optimize this algorithm by observing that *count* is always equal to $j - m - 1$. (Proof: Initially, $j = m + 1$ and $count = 0$, and we always increment j and *count* together.)

```
MergeAndCount2(A[1..n], m):
  i ← 1;  j ← m + 1;  total ← 0
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i + 1;  total ← total + j − m − 1
      else if i > m
          B[k] ← A[j];  j ← j + 1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i + 1;  total ← total + j − m − 1
      else
          B[k] ← A[j];  j ← j + 1
  for k ← 1 to n
      A[k] ← B[k]
  return total
```

The modified Merge algorithm still runs in $O(n)$ time, so the running time of the resulting modified mergesort still obeys the recurrence $T(n) = 2T(n/2) + O(n)$. We conclude that the overall running time is $O(n \log n)$, as required. ■

Rubric: 10 points = 2 for base case + 3 for divide (split and recurse) + 3 for conquer (merge and count) + 2 for time analysis. Max 3 points for a correct $O(n^2)$-time algorithm. This is neither the only way to correctly describe this algorithm nor the only correct $O(n \log n)$-time algorithm. No proof of correctness is required.