Problem Set 3

Reading: Chapters §8.1-8.3, 31.1-31.5, 31.7-31.8

There are **four** problems. Each problem is to be done on a **separate sheet** (or sheets) of paper. Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date, and the names of any students with whom you collaborated.

You will often be called upon to "give an algorithm" to solve a certain problem. Giving an algorithm entails:

1. A description of the algorithm in English and, if helpful, pseudocode.
2. A proof (or argument) of the correctness of the algorithm.
3. An analysis of the running time of the algorithm.

It is also suggested that you include at least one worked example or diagram to show more precisely how your algorithm works. Remember, your goal is to communicate. Graders will be instructed to take off points for convoluted and obtuse descriptions. If you cannot solve a problem, give a brief summary of any partial results.

Problem 3-1. Min and Max

Suppose we wish to find both the minimum and maximum values in an array of n distinct elements. To individually find either the minimum or maximum, there is clearly a $n - 1$ lower bound on comparisons, since we must compare every element at least once. We can save a comparison by first scanning for the minimum, removing it, then scanning for the maximum. This takes a total of $2n - 3$ comparisons.

Based on this observation, prove the following statement **true** or **false**:

"It takes at least $2n - c$ comparisons, for some constant c, to find both the minimum and the maximum in an array of n distinct elements."

Solution: False. Consider the following algorithm:

- Compare every pair of elements in $n/2$ comparisons.
- Put the $n/2$ "light" elements in a set L.
- Put the other $n/2$ "heavy" elements in a set H.
- Scan L and H for the minimum and maximum, respectively.

Since the minimum is less than all other elements, it will always be in L. Similarly, the maximum will always be in H. Exhaustively searching H and L will clearly find these elements. The total amount of comparisons will be $n/2$ to produce H and L, $n/2 - 1$ to scan H and $n/2 - 1$ to scan L. This is a total of $3n/2 - 2$ comparisons, clearly contradicting the above statement.

Problem 3-2. Monotone Priority Queues

A "monotone priority queue" (MPQ) is a data structure that supports the following operations:

- MIN(Q) - Returns the minimum element in Q. The minimum of a new, empty MPQ is initially $-\infty$. Otherwise, the minimum of an empty MPQ is the last element deleted.
- INSERT(Q, x) - Inserts x into Q given that $x \geq$MIN(Q). If $x <$MIN(Q), then the MPQ is not modified.
- DELETE-MIN(Q) - If Q is empty, returns the minimum. Otherwise, removes and returns the minimum from Q. If the queue is empty after the operation, the last deleted value remains the minimum. In other words, the minimum value is monotonically increasing and does not reset when the MPQ is empty.

For this problem, assume that x is an integer in the range $[0, k]$ for some fixed integer value k.

(a) Implement a monotone priority queue that takes $O(m \log m)$-time to perform m operations starting with an empty data structure.

Solution: Implement the MPQ using a simple Heap. The MPQ-Heap will test to ensure newly inserted elements are greater than or equal to the current minimum. Also, the MPQ-Heap must keep track of the value of the last deleted element if it becomes empty.

Each MIN operation takes $O(1)$. After m INSERT operations, a simple heap can have depth $O(\log m)$. This depth gives an upper-bound on the cost of each INSERT and DELETE-MIN. Therefore, the total amount of work over m operations is upper-bounded by $O(m \log m)$.

(b) Give an implementation of a monotone priority queue that takes $O(m + k)$ time to perform m total operations. Hint: Use an idea from COUNTING-SORT.

Solution:

1. Initialize an array A of size k. Let $min \leftarrow -\infty$.
2. MIN(Q) - Return min.
3. INSERT(Q, x) - If $(x \geq min)$, increment $A[x]$.
4. DELETE-MIN(Q) -
   - If $A[min] = 0$, return min.

- If $A[min] > 0$, return min. Decrement $A[min]$. While $(A[min] = 0) \wedge (min \leq k)$, increment min.

This uses an array to keep track of instances of values in $[0, k]$, like COUNTING-SORT. In addition, it uses a monotonically increasing pointer min to keep track of the minimum value in the array.

Clearly, MIN and INSERT take $O(1)$ time. Although DELETE-MIN runs in time $O(k)$, it can monotonically increase min a total of k times over any number of operations. Therefore, over m operations at most $O(m + k)$ work can be done.

(c) 6.046 student Ben K. Bitdiddle has invented a MPQ that operates on any *totally ordered* set, rather than just integers in the range $[0, k]$. A total ordering defines a $\leq$ relation for all pairs of elelments in a set.

Ben claims that his MPQ can perform m operations in $O(m \log \log m)$-time. Ben's classmate Alyssa P. Hacker quickly dismisses his claim as impossible. Explain who is correct and prove why.

Solution: Alyssa P. Hacker is correct. Given Ben's MPQ implementation, we could effectively sort any set of m totally ordered elements in $O(m \log \log m)$ time. Since the set is totally ordered, there must exist some minimum element which can be found in $O(m)$ time by scanning through the set. If the minimum is inserted into the MPQ first, then every other element can be inserted without any restriction.

Once the minimum is inserted, we can perform m INSERT operations followed by m DELETE-MIN operations to extract the set in sorted order. By Ben's claim, this would take $O(m \log \log m)$ time, which is less than the $\Omega(m \log m)$ lower bound for comparison based sorting.

Problem 3-3. Operations on Elliptic Curves

Throughout this problem we will be discussing operations on elliptic curve points. You do not need to know any specifics about elliptic curves or their operations. The basic operand will be points denoted by bold capital letters, for example **P**.

You are given a subroutine ADD which can add points, for example ADD(**P**,**Q**) = **P + Q**. You may assume that ADD runs in $O(n)$-time, where **P** has an n-bit representation.

Multiplying an elliptic curve point **P** by a scalar s is equivalent to adding **P** to itself $(s - 1)$ times. That is, 2**P** = **P + P**, 3**P** = **P + P + P**, etc. By this definition, (s**P** + t**P**) = $(s + t)$**P** = (t**P** + s**P**).

(a) Suppose you are given a k-bit integer s and a point **P** as inputs. Give a $\Theta(n2^k)$-time scalar multiplication algorithm that computes s**P** using only calls to ADD.

Solution:

```
SCALAR-MULTIPLY($s$, P):
1  Q $\leftarrow$ P
2  for $i \leftarrow 2$ to $s$:
3      Q $\leftarrow$ ADD(Q, P)
4  return Q
```

The point **P** is naively added to itself s times, clearly producing the correct result. This algorithm calls ADD $s < 2^k$ number of times, so runs in time $O(n2^k)$.

(b) Give an $\Theta(nk)$-time scalar multiplication algorithm.

Solution:

Let $s = (s_k s_{k-1} \dots s_1)$, i.e. s_i is the ith least significant bit of s. You may assume that $s_k = 1$

```
SCALAR-MULTIPLY($s$, P):
1  Q $\leftarrow$ P
2  for $i \leftarrow k - 1$ downto $1$:
3      Q $\leftarrow$ ADD(Q, Q)
4      if $(s_i = 1)$ Q $\leftarrow$ ADD(Q, P)
5  return Q
```

1. Pre-Condition: **Q**$=(s_k)$**P**
2. Inductive Hypothesis: Before the ith iteration, **Q**$=(s_k \dots s_{k-i+1})$**P**
3. Induction: When **Q** is added to itself, it becomes $(s_k \dots s_{i+1}0)$**P**. If $(s_i = 1)$, then **P** is added and **Q**$=(s_k \dots s_{i+1}1)$**P**. Thus, after the loop completes **Q**$=(s_k \dots s_{i+1}s_i)$**P**.
4. Post-Condition: The loop terminates when $i = 1$, so **Q**$=(s_k \dots s_{i+1}s_1)$**P**$=s$**P**.

In this algorithm, ADD is called at least k times and at most $2k$ times. Clearly it runs in $\Theta(nk)$-time.

(c) Ben K. Bitdiddle, notices that his solution to part (b) always makes between k and $2k$ calls to ADD. He thinks he can improve on this and writes a point doubling procedure DOUBLE that runs in $O(1)$ time. The output of DOUBLE(**P**) is 2**P**.

Rewrite your solution to part (b) using DOUBLE. What are the new upper and lower bounds on the runtime? What is the expected number of calls to ADD if s is chosen uniformly at random from $\{0, 1\}^k$?

Solution:

```
SCALAR-MULTIPLY($s$, **P**):
1  Q ← P
2  for $i \leftarrow k - 1$ downto $1$:
3      Q ← DOUBLE(Q)
4      if $(s_i = 1)$ Q ← ADD(Q,P)
5  return Q
```

Clearly this code is correct by part (b), since DOUBLE(**Q**) = ADD(**Q**,**Q**). Since it makes between 0 and k calls to ADD, this code runs in $O(nk)$ and $\Omega(k)$. If s is a random k-bit number, half of its bits are expected to be 1. Therefore, this algorithm will make an average of $k/2$ calls to ADD.

(d) Ben gets the idea to pre-compute the values **P**, 2**P**, 3**P**, . . ., $(2^d - 1)$**P** and store them in an array A such that $A[i] = i$**P**. Suppose you naively fill in the array A in $O(n2^d)$-time by repeated point addition. Give an $O(\frac{nk}{d})$-time scalar multiplication algorithm. You may assume that d divides $(k - 1)$ and may use both DOUBLE and ADD in your code.

Solution:

```
SCALAR-MULTIPLY($s$, **P**):
1  Q ← P
2  for $i \leftarrow k - 1$ downto $1$ in steps of $d$:
3      for $j \leftarrow 1$ to $d$:
4          Q ← DOUBLE(Q)  ▷ Shift Q by $d$ bits
5      $t \leftarrow (s_i \ldots s_{i-d+1})$   ▷ $t$ gets a value in $[0, 2^d - 1]$
6      if $(t > 0)$ Q ← ADD(Q,$A[t]$)
7  return Q
```

This code is essentially identical to part (b), except that it uses a “sliding window” of d-bits to look up a value in a pre-computed array. The correctness proof of (b) also holds in this example. In fact, part (b) can be thought of a “sliding window” with $(d = 1)$.

Comment: Since no relations were given between n and d, you could conceivably have a case where $d > n$, for example, $d = n^2$. Plugging this into the above $O(nk/d)$ runtime would give a nonsensical $O(k/n)$ runtime. We need to at least read the k-bit input, so this algorithm has a tight lower bound of $\Omega(k)$. Technically, we should have said $O(\lceil \frac{n}{d} \rceil k)$ or $O(k(1 + \frac{n}{d}))$, or explicitly stated that $n > k > d$. In practice it is usually the case that $n >> k$.

(e) Give a value of d such that the algorithm in part (d) runs in $o(nk)$. Include both the time it takes to fill A and compute s**P**, i.e. $O(n2^d + \frac{nk}{d})$.

Solution:
By taking the derivative of $n2^d + \frac{nk}{d}$, we see that we wish to minimize $d^2 2^d = k$. An approximate solution is $d = \log k - \log(\log k)$. Plugging this in, we get a solution that is asymptotically faster than $O(nk)$ ($d = \log\log k$ also works):

$$n2^{\log k - \log(\log k)} + \frac{nk}{\log k - \log(\log k)} = \frac{nk}{\log k} + \frac{nk}{\log k - \log(\log k)} = O\left(\frac{nk}{\log k - \log(\log k)}\right)$$

Problem 3-4. Man on the Moon

Alyssa (A) wishes to determine whether her $(n+1)$-bit string a is the same as Ben's (B) $(n+1)$-bit string b. Unfortunately, Ben lives on the moon and communication costs are very high. Ben devises a scheme to determine with high probability whether or not $a = b$, while minimizing communication. Let a_i and b_i denote the ith bits of a and b's respective representations:

1. A picks a prime p such that $n^2 < p < 2n^2$.
2. A defines a n-degree polynomial over $\mathbb{Z}_p$, denoted $a(x) = (\sum_{i=0}^{n} a_i x^i) \bmod p$.
3. A picks a random $x \in \mathbb{Z}_p$ and computes $a(x)$, and sends $a(x)$, x, and p to B.
4. B defines a n-degree polynomial over $\mathbb{Z}_p$, denoted $b(x) = (\sum_{i=0}^{n} b_i x^i) \bmod p$.
5. B computes $b(x)$ and accepts if $a(x) = b(x)$.

(a) Given that a n-degree polynomial can have at most n roots, if $a \neq b$ what is the maximum probability that Ben accepts?

Solution: If $a \neq b$, $a(x)$ can match $b(x)$ at most n points. Since there are at least n^2 points in $\mathbb{Z}_p$, there is at most a $n/n^2 = 1/n$ chance of x being a false match.

(b) Give an explicit upper bound (in terms of n) on the number of bits transmitted in this scheme. Do not give an asymptotic upper bound, but rather an actual function, e.g. $5n^2$ instead of $O(n^2)$.

Solution: Alice transmits p, $a(x)$ and x. We have that: $2\log n < \log p < 2\log n + 1$, $\log x < 2\log n + 1$ and $\log a(x) < 2\log n + 1$, so at most $6\log n + 3$ total bits are transmitted. (We will also accept $6\log n$.)

Alyssa suggests a second scheme:

1. Repeat k times:
   (a) A picks a prime p uniformly at random from the range $[1, W]$.

(b)A sends p and $(a \bmod p)$ to B.

(c)B rejects if $(b \bmod p) \neq (a \bmod p)$.

2.B accepts if $(b \bmod p) = (a \bmod p)$ for all steps.

(c) Assume that there are L primes less than W, that is $(p_1 < p_2 < \ldots < p_L < W)$, and that $(\Pi_{i=1}^{L} p_i) = P_L > 2^n$. If $a \neq b$, give an upper bound on the probability, in terms of k and L, that $(a \bmod p) = (b \bmod p)$ for all k rounds. You will need to use the Chinese Remainder Theorem.

Solution:

By the Chinese Remainder Theorem, every number $x \in [1, 2^n]$ will have a unique set of residues $\{x \bmod p_i\}_{i=1}^{L}$. If $a \neq b$, then there will be at least one prime p_j where $(a \bmod p_j) \neq (b \bmod p_j)$. If we pick a prime p_i uniformly from all primes less than W, then there is a least $1/L$ chance that we'll detect $a \neq b$.

Therefore, the probability that $(a \bmod p_i) = (b \bmod p_i)$ is at most $(1 - 1/L)$. Since p is chosen independently in each round, there is at most a $(1 - 1/L)^k$ chance of failing all k rounds.

Alternatively, if we assume Alyssa chooses k distinct primes, we can say that she has at least a k/L chance of choosing a bad prime, and thus at most $(1 - k/L)$ chance of failing.

(d) Using the the Prime number theorem in CLRS Section 31.8, upper bound the probability of failing all k rounds of the protocol in terms of k and W?

Solution: The Prime number theorem implies $L < 2W/(\log W)$, so $1/L > (\log W)/2W$ and $(1 - 1/L) < (1 - (\log W)/2W)$. Therefore there is at most a $(1 - (\log W)/2W)^k$ chance of failing all k rounds.

Alternatively, if Alyssa chooses k distinct primes, she'd have a $(1 - (k \log W)/2W)$ chance of failing all k rounds.