(1) Counting Significant Inversions. (10 points) Given a list of possibly non-unique numbers $x_1, \ldots, x_n$, we say that two members of this list x_i and x_j form an inversion if $i < j$ and yet $x_i > x_j$. So if the list is sorted in increasing order then it has no inversions. In contrast, if n different numbers are listed in decreasing order then the i-th number in an decreasing list will form an inversion with every number ahead of it, so the total number of inversions will be $0 + 1 + 2 + \cdots + (n-1) = n(n-1)/2$, i.e., all pairs will form inversions. Section 5.3 of the book offers an example of this problem (on page 222), and offers a divide and conquer algorithm running in $O(n \log n)$ time that can count the number of inversions in a given sequence of n numbers in Section 5.3. In this problem, we say that a pair x_i and x_j form a *significant* inversion if $i < j$ and yet $x_i > x_j + 1$. Design a $O(n \log n)$ algorithm that finds the number of significant inversions in a given sequence, and implement the algorithm in Java. The time limit of this problem will be 1 second.

Solution: Use Divide-and-Conquer. The two subproblems are the sequence $x_1, \ldots, x_k$ and $x_{k+1}, \ldots, x_n$ where $k = \lfloor n/2 \rfloor$. For both subproblems, sort and count number of significant inversions recursively. What remains is to merge the two lists and count the number of significant inversions with i in the first half and j in the second half. Recall that at this point both halves are sorted.

To do this we need to merge the two sorted lists like in merge sort. Let x_i be the head of the left list and let x_j be the head of the right list. Notice that $i < j$.

- If $x_i \leq x_j$, we add x_i to the tail of the final list.
- Otherwise, if $x_i > x_j$, we add x_j to the tail of the final list.
  - At the same time, find the first item x_k on the left list that satisfies $x_k > x_j + 1$, if there is any such element. If there is such an element, we have a significant inversion between x_k and x_j. In addition, since the left list is sorted, all the other elements in the left list are greater than or equal to $x_k > x_j + 1$, so they are also all significantly inverted with respect to x_j. Thus in this case, we increment the number of significant inversions by the number of elements after and including x_k of the left list when we add x_j to the tail of the list.
  - To be able to do this all in $O(n)$ time we need to keep an additional pointer for the last x_k we found as a significant inversion. Later heads of the left list can start searching for inversions from the previous position, as the left list is sorted.
- Once we complete merging, we add the number of significant inversions from the left and right list that we got from solving their subproblems.

Running time is $O(n \log n)$ as the merge-and-count of the two sorted list at the end is linear time, and for a list of length n satisfies the recurrence $T(n) = O(n) + 2T(n/2)$.

(2) Finding the best interval. (10 points) Suppose you have perfect foresight for the future prices of a stock, and you know the next n periods the stock price will be $x_1, x_2, \ldots, x_n$. You want to choose two dates $1 \leq i \leq j \leq n$ to make as much money buying a stock on day i and selling it on day j. So selecting a pair $i \leq j$ would make you $x_j - x_i$ in income. Unfortunately n is really huge and you are doing this on a device that is quite restricted in memory. We can find the best $i \leq j$ pair by trying all pairs in $O(n^2)$ time. In this problem we are asking you to give an $O(n \log n)$ algorithm to solve this problem but **you are only allowed to use up to $O(\log n)$ additional memory**. Note that this does not include the input array's $O(n)$ memory usage. You may assume n is a power of 2.

Hint: if you do multiple recursive calls these calls can reuse memory.

Solution Using Divide and Conquer: We can use divide and conquer: recursively find the best buy-sell pair in the first $n/2$ days, as well as the best buy-sell pair in the last $n/2$ days. The optimum

is one of these two or a third option, buy in the first half and sell during the last half. To accommodate this we need to also find the minimum of the first half, and the maximum value of the second half, and then take the maximum of the three options. Writing this as a code:

To compute the value of the best interval for segment $x_{i+1}, \ldots x_{i+j}$ while assuming j is a power of 2, we do the following

- If $j = 1$, then return best value of 0, and min = max = x_{i+j}.
- else
  - Recursively compute the best value A, as well as min and max value (called A_{min} and A_{max} for $x_{i+1}, \ldots x_{i+j/2}$
  - Recursively compute the best value B as well as min and max called B_{min} and B_{max} for $x_{i+j/2+1}, \ldots x_{i+j}$
  - Compute $C = \max_{j/2<k\leq j} x_{i+j} - \min_{1\leq k\leq j/2} x_{i+j}$
- return $\max(A, B, C), \min(A_{min}, B_{min}), \max(A_{max}, B_{max})$

If $T(n)$ is the running time on n long list, then T satisfies the recurrence $T(n) = 2T(n/2) + O(1)$, as all we have to do is divide into two lists, and compute a few max and min values. This recurrence solves to $O(n)$: to see why consider the recursion tree: this is a binary tree with $\log_2 n$ levels and n leaves. We are spending $O(1)$ time at each internal node of the tree, and there are $2n - 1$ nodes in total, so a total of $O(n)$ time.

Alternately we can compute the min and max values directly, not part of the recurrence, this will give us the recurrence of $T(n) = 2T(n/2) + O(n)$ which solves to $O(n \log n)$ due to the master theorem for $p = q$.

For analysing the memory use: we can use $M(n)$ the amount of additional memory used on a n long list. This function satisfies the recurrence of $M(n) = M(n/2) + O(1)$. This is true as computing the min and max only takes constant memory, we also need to remember A and B above, as well as $j/2, j/2 + 1$. Note that $M(n/2)$ doesn't need to be counted twice, as memory can be reused between the two recursive calls.

Direct Solution using only $O(n)$ time: It is also possibly to solve this problem directly using $O(n)$ time, though this solution is less useful in practicing divide and conquer: keep a single memory m going through the minimum value seen so far, and the best solution found so far for a profit p. So when reading x_i we will have $m = \min_{j\leq i} x_j$. If the solution involves selling on say i, then the best day to buy is the minimum price on a previous day, so the profit is then $p = x_i - m$. Update the solution and the achievable profit if this is higher than previous seen options.

(3) Perfect Hashing. (15 points) On Monday September 30 we will cover universal hashing: a randomized hash function is a function $h : U \to \mathbb{Z}_p$ selected at random, such that $\Pr[h[u] = i] = 1/p$ for all $u \in U$ and all $i \in \mathbb{Z}_p$, and also $\Pr[h[u] = h[v]] = 1/p$ for any two elements $u \neq v$. Here $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ is the set of integers mod p and U is our universe of elements we want to hash.

In class we considered storing a set $S \subset U$ of $n = |S|$ elements using this hash function (storing elements u in an array at position $h[u]$), and showed that if $n \leq p$ than the expected number of elements colliding with any element u is at most 1. This problem considers extensions of this idea.

(a) (5 points) Even with the random choice of a hash function h as we have done in class, with high probability there will be some collisions. Suppose $n = p$, and let n_i be the number of elements $u \in S$ that hash to i, that is $n_i = |\{u \in S \mid h(u) = i\}|$. Show that $\Pr[\sum_{i=0}^{p-1} n_i^2 > 4p] \leq 1/2$.
Hint: Consider a variable $X_{u,v}$ for any pair of elements $u, v \in S$ that is 1 if u and v collide (that is $h(u) = h(v)$). Note that $\sum_i n_i^2 = \sum_{u,v\in S} X_{u,v}$ assuming we have for any two elements $u \neq v$ we have two random variables $X_{u,v}$ and $X_{v,u}$ and we also have $X_{v,v}$ for any element $v \in S$. Do you see why? Include an explanation of this fact, if you use it.

Solution: First let us explain the claimed fact above. For each i, if there are n_i element colliding on hash value i then these n_i elements create n_i^2 pairs (u, v) with $X_{u,v} = 1$, as all pairs from the

set $h^{-1}(i) \cap S$ collide. Using this fact, we get $E[\sum_i n_i^2] = E[\sum_{(u,v)} X_{uv}]$.
Now we calculate the expectation on the right hand side. Using linearity of expectations, we see that $E[\sum_{(u,v)} X_{uv}] = \sum_{(u,v)} E[X_{uv}]$. Thus it suffices to calculate $E[X_{uv}]$ for each (u, v) and sum the results. For each (u, v) there are two case: either $u = v$ or $u \neq v$. Notice that $E[X_{(u,v)}] = \Pr[X_{(u,v)} = 1] = \Pr[h(u) = h(v)]$. When $u = v$, $h(u) = h(v)$ always, so $\Pr[h(u) = h(v)] = 1$. When $u \neq v$, $\Pr[h(u) = h(v)] = 1/p$ by property of universal hashing. In our sum over expectations, there are n terms where $u = v$ and $n^2 - n = n(n-1)$ terms where $u \neq v$, so we have

$$E[\sum_i n_i^2] = \sum_{(u,v)} E[X_{uv}] = n \cdot 1 + n(n-1) \cdot \frac{1}{p}.$$

Recall that we assumed $n = p$ in the problem statement, so

$$E[\sum_i n_i^2] = 2p - 1 < 2p.$$

By Markov's inequality we have that

$$\Pr[\sum_i n_i^2 > 4p] \leq E[\sum_i n_i^2]/4p < 2p/4p = 1/2.$$

(b) (5 points) One way to avoid collision is to use a $p \geq n^2$. Show that using universal hashing with such a large table, with probability at least $1/2$ there will be no collisions at all.

Solution: We want to compute $\Pr[\sum_{\{u,v\}} X_{u,v} = 0] = 1 - \Pr[\sum_{\{u,v\}} X_{u,v} \geq 1]$ where $u \neq v$. We can upper bound the second term on the right hand side by quantity by Markov's inequality which will translate to a lower bound for the left hand side. Markov's inequality requires us to compute $E[\sum_{\{u,v\}} X_{u,v}]$. By linearity of expectations, it suffices to compute $E[X_{u,v}]$ for each $\{u, v\}$. Notice that this sum is over sets of two distinct elments of S: $\{u, v\}$ where $u \neq v$, instead of pairs (u, v) as before. There are $n(n-1)$ pairs (u, v), but this double counts the total number of distinct pairs. Thus there are $n(n-1)/2$ sets of two distinct elements u, v in S. Once again, $E[X_{u,v}] = \Pr[X_{u,v} = 1] = 1/p$ since the the probability of distinct u, v colliding is $1/p$ by universal hashing property. Thus we have

$$E[\sum_{\{u,v\}} X_{u,v}] = \sum_{\{u,v\}} E[X_{u,v}] = n(n-1)/2 \cdot 1/p \leq \frac{n-1}{2n} < \frac{1}{2}$$

where we recall that we assumed $p \geq n^2$ so $1/p \leq 1/n^2$. By Markov's inequality, the probability of there having a collision is

$$\Pr[\sum_{\{u,v\}} X_{u,v} \geq 1] \leq E[\sum_{\{u,v\}} X_{u,v}]/1/ < \frac{1}{2}$$

Thus the probability that there are no collisions is greater than $1 - 1/2 = 1/2$.

(c) (5 points) Consider a dictionary that we will store a set of S elements doing a lot of lookups, but the set S will not change. For this case, it may be useful to design a hashing scheme that is perfect in the sense that there are really no collisions, and yet doesn't use as much space as the solution we have in part (b).
One idea is called 2-level hashing. Start with a random hash function with $p \geq |S|$, and repeatedly try selecting a function h until you find one that satisfies the inequality $\sum_i n_i^2 \leq 4p$. This will be our level 1 table.
Now take each position i with $n_i > 1$, and use a second level hash function, h_i that has a table size $p_i \geq n_i^2$ to hash these elements to a new table. For each i repeatedly select h_i until there are no collisions in this table. Now we have a two level table, where for elements u that don't collide we

store at $h[u]$ of the first table, for those that do collide, we set $i = h[u]$ and store u at a separate table T_i at location $h_i[u]$ of table T_i. The total space used is approximately $n + \sum_i n_i^2 \leq 5n$, linear in $|S|$. The most time consuming part of finding such perfect dual hashing scheme is the many computations of $h(u)$ and $h_i(u)$ for various proposed h functions and elements $u \in S$. What is the expected number of such computations needed to find a perfect hash function? (you may assume that $n \leq p \leq 2n$ in the 1st level hash table, and $n_i^2 \leq p_i \leq 2n_i^2$ for all i for the second level hash tables for need, and that finding such a prime p and p_i is an $O(1)$ lookup in a prime table.

Solution: Recall that in expectation we require $1/p$ tries before we get a success for independent tries that each succeed with probability p. From part a, we know that each try will fail with probability at most $1/2$ so each try will succeed with probability at least $1/2$. Thus the level 1 hashing will require at most 2 trials in expectation to find a level 1 hash function. Each try requires computing $h(u)$ for all $u \in S$, so in expectation this is $2|S| = 2n$ computations.
For the second level, for each i we succeed with probability at least $1/2$ using part b, so we need at most 2 tries in expectation to succeed. Each try requires n_i value computation (for i such that $n_i > 1$, so in total this is at most $2\sum_i n_i = 2n$. So the total evaluations of h is upper bounded by $4n = O(n)$ in expectation.