(1)

For a given sequence of numbers $p_0, p_1, \ldots, p_{n-1}$, we say that the sequence has equal gap triple, if there is a triple $i < j < k$ such that $p_j - p_i = p_k - p_j$. Consider the numbers $0, 1, 2, \ldots, n-1$. Give an algorithm that orders these numbers as $p_0, p_1, \ldots, p_{n-1}$, in a way that avoids an equal gap triple. Show that such an ordering always exists, and give an $O(n \log n)$ algorithm to find such an ordering. You may assume that n is a power of 2. (Note that $O(n)$ solution is also possible.)

Hint: it may be useful to notice that if p_i and p_k have different parity, they cannot be part of an equal gap triple with $i < j < k$.

Solution.

Using the hint we notice that putting all even numbers first, and then all odd numbers, we make sure there are no equal gap triples spanning the two halves. Now want to recursively solve the problem on the two halves. For the even numbers, $0, 2, 4, \ldots, n-2$ (recall n is a power of two), we can instead solve the problem with dividing each number by 2, to get the sequence $0, 1, \ldots, n/2-1$, and recursively solving the problem. Similarly for the even numbers, take same sort of $0, 1, \ldots, n/2-1$, and replace each number i with $2i+1$ to get a sequence of the odd numbers with no equal gap triple. Finally put even numbers first and then the odd numbers to get the resulting sort.

Correctness. We prove correctness by induction. Clearly true when $n = 2$. For larger n, first note that $p_0, p_1, \ldots, p_{n-1}$ has no equal gap triple, if and only if $2p_0, 2p_1, \ldots, 2p_{n-1}$ and also $2p_0+1, 2p_1+1, \ldots, 2p_{n-1}+1$ has no equal gap triple, so by the induction hypothesis, the first half or the second half doesn't contain any equal gap triple. The property noted in the hint shows that there are no equal gap triples spanning the two parts.

Running time. We spend $O(n)$ time writing out the sort at the end, plus one recursive call for a half size problem. So if $T(n)$ denotes the running time for numbers $0, 1, \ldots, n$ then we get the recurrence that $T(n) = cn + T(n/2)$ for some constant c. Summing over the recursive calls, we get that

$$T(n) = cn + cn/2 + cn/4 + \ldots = 2cn$$

Alternate view of this sort.

It may be nice to notice that the resulting sort is as follows. Think of each number as a string in binary on $\log_2 n$ digits. Write them in the usual order $0, 1, \ldots, n-1$, and then reserve the digits of each number (so for example, with $n = 8$, we start with $0, 1, 2, 3, 4, 5, 6, 7$, writing this all in binary we get $000, 001, 010, 011, 100, 101, 110, 111$. Reversing the digits of each number we get $000, 100, 010, 110, 001, 101, 011, 111$, which is $0, 4, 2, 6, 1, 5, 3, 7$, which is indeed the ordering we get from the divide and conquer solution above. We are not sure how to argue directly that this sort as no equal gap triple, but it follows from the proof above.

Another different solution.

We'll solve the problem for all sequences $L = (a, a+1, \ldots, a+2^k-1)$ recursively. Create two lists consisting of the first half and second half: $L_1 = (a, a+1, \ldots, a+2^{k-1}-1)$ and $L_2 = (a+2^{k-1}, a+2^{k-1}+1, \ldots, a+2^k-1)$. Run the algorithm recursively on L_1 and L_2, and let S_1 and S_2 be the

sequences returned. Now return the sequence S taking entries from the two list alternately: that is for any $0 \leq i < 2^k$, if $i = 2j$ then $S(i) = S_1(j)$, else if $i = 2j + 1$ then $S(i) = S_2(j)$. Handling the base case is the same as the algorithm above.

As usual, the grading rubric for different algorithms is different. The proof of correctness for this algorithm is significantly harder than the regular algorithms, so its a bigger fraction of the points.

Another view of this algorithm. Another solution, which is equivalent to this algorithm, divides the sequence into two smaller sequences the same way and solves each of the smaller sequences with recursion. However for merging the sequences, it selects evens from each part alternatively (first even number is from the first list, second even number is the first even number from the second list, third even number is the second even number from the first list etc.). It does the same thing for the odd numbers to get a list of the odd numbers. Now put the list of even numbers first and the list of the odd numbers after that.

Proof of Correctness. For any a such that $a \equiv 0 (\text{mod } 2^k)$, the output S of the algorithm on sequence of $L = [a, a + 1, \ldots, a + 2^i - 1]$, for any $i < k$ has the following property:

For any $j < i$, if you divide S into 2^j groups $S_0^j, S_1^j, \ldots, S_{2^j-1}^j$ of 2^{i-j} consecutive numbers in the list ($S = \text{Concat}(S_1^j, \ldots, S_{2^j-1}^j)$), then for any $x, y \in S_z^j$, $x \equiv y (\text{mod } 2^j)$. Note that this means for any $z \neq z'$, and any $x \in S_z^j$ and $y \in S_{z'}^j$, $x \not\equiv y (\text{mod } 2^j)$.

Proof. Use induction. For the base case (sequences of length 1,2, or 4) it is easy to check this property. Assume this is true for sequences of length 2^{i-1}. Now we show this is true for sequences of length 2^i. Note that since the dividing rule uses the $k - 1$th insignificant bit when the length of the sequence is 2^k, the merging rule is only based on the parity of indexes, and indexes in second half are equal to indexes of the first half in the first $i - 2$ insignificant bit, we can conclude that if we add 2^{i-1} to each number in the first half after using recursion, then it is equal to the output of recursion on the second half. When we are merging two consecutive sequences of length 2^{i-1}, by the merging rule of the algorithm, we are fixing this property for $j = i - 1$ (in this case we have 2^{i-1} groups of length 2, and the two numbers in a group are the same up to mod 2^{i-1} since the algorithm rearrange both of the smaller sequences the same way). For any other $j < i - 1$, by the induction hypothesis, the merging rule also assures that this property will not be violated, since for any group S_z^j of the merged sequence is created by merging S_z^j of the first sequence and S_z^j of the second sequence. Note that after merging the two smaller sequences, for any valid j and z, size of S_z^j of will be double the size of S_z^j of the smaller sequences by definition.

Now by using this property, we actually see that the algorithm created the same sequence as the previous recursive algorithm, so its has the no equal gaps property, as proved above. Alternately, we cab prove directly that the algorithm is correct using contradiction. Assume the output has 3 numbers p_i, p_j and p_k such that $i < j < k$ and $p_j = p_i + x$ and $p_k = p_i + 2x$ for some x. Let ℓ be the biggest number such that $p_i \equiv p_j \ (\text{mod } 2^\ell)$. So $x = p_j - p_i$ is also $x \equiv p_i \ (\text{mod } 2^\ell)$. This means that $p_k = p_i + 2x$, so $p_i \equiv p_k \ (\text{mod } 2^{\ell+1})$ while $p_i \not\equiv p_j (\text{mod } 2^{\ell+1})$. From the previous argument we know that if we divide the output of the algorithm into $2^{\ell+1}$ consecutive groups, p_k and p_i must be in the same group but p_j is not in this group. This means that p_j cannot be between p_k and p_i in the sequence, which is a contradiction.

(2)

In class we considered the coupon collector problem from Section 13.3 of the book. Recall that there are n types of coupons, and at each time step there you collect a coupon with each type equally likely. We have shown that the expected time that it takes to collect all n kinds is $nH(n) \approx n \log n$ (where the log here is base e), and that in a constant time more rounds you are very likely to have all the coupons. In this question we study what happens with not enough rounds.

a. After only n rounds, it is possible to have all the different coupons, but rather unlikely. What is the expected number of different coupons after n rounds? To answer this question you need to define X, which is the random variable of the number of different coupons you have, and the question asks you to compute $E(X)$. Show that $\lim E(X)/n = 1 - 1/e \approx 0.63$ as n goes to infinity. You may use the fact that $\lim(1 - 1/n)^n = 1/e$ as n goes to infinity.

Hint: It will be very helpful to define n random 0-1 variables, with $X_i = 1$ if you have the ith kind of coupon and 0 if you do not. Now note that we have $X = \sum_i X_i$, so you can use linearity of expectation.

Solution. As suggested by the hint, we can use X_i to denote a 0-1 variable, that is 1 if we got the coupon of type i and 0 otherwise. With n rounds of selecting a coupon, the probability that X_i is 1 is

$$Pr(X_i = 1) = 1 - Pr(X_i = 0) = 1 - (1 - \frac{1}{n})^n$$

Now the expected value $E(X_i) = Pr(X_i = 1$ by definition of expectation. As also suggested by the hint $X = \sum_i X_i$, so we get the following using the linearity of expectation

$$E(X) = \sum_i E(X_i) = \sum_i [1 - (1 - \frac{1}{n})^n] = n(1 - (1 - \frac{1}{n})^n)$$

Taking the limit, and using that $(1 - 1/n)^n = 1/e$ we get that

$$\lim_{n\to\infty} \frac{E(X)}{n} = \lim_{n\to\infty} 1 - (1 - \frac{1}{n})^n = 1 - 1/e$$

b. A common experience with such collection is that we have too many of one kind of coupon, while still missing others. To understand the likely extent of this phenomena, let Y_i be the number of coupons of type i you received after n rounds. What is $E(Y_i)$?

Solution. Here again we want to break Y_i into separate events. We can use Y_{ik} to be 1 if the k round we get a coupon of type i. This way $Pr(Y_{ik} = 1) = 1/n$, and so $E(Y_{ik}) = 1/n$. Now $Y_i = \sum_k Y_{ik}$, so we get that $E(Y_i) = \sum_k E(Y_{ik})$ by linearity of expectation, so $E(Y_i) = 1$ for all i.

c. To think about the experience mentioned above, we shouldn't focus on one kind of coupon, it is better to think about $\max_i Y_i$. We claim that $E(\max_i Y_i) \neq \max_i E(Y_i)$. Explain your answer using (a) and (b) above.

Note that the advanced section 13.9 of the book shows that in fact $E(\max_i Y_i) = \frac{\log n}{\log \log n}$, but we will not cover this proof in this class.

Solution. Using part (b) we get that $E(Y_i) = 1$ for all i, and so $\max_i E(Y_i)$. On the other hand $\max_i Y_i \geq 1$ every time, and $\max_i Y_i \geq 2$ whenever the first n coupons are not all different. So $E(\max_i Y_i)$ is the expectation of a variable that is always at least one, and some times more than 1 (actually often more than 1), so clearly $E(\max_i Y_i) > 1$.

(3)

Recall the problem on the prelim. Consider an n by m matrix of nonnegative rewards $v_{i,j}$. There is a robot starting at $(1, 1)$. Each time the robot can go one step to the right or one step down. While on the matrix cell (i, j) the robot collects the reward $v_{i,j}$. On the prelim, you designed an polynomial time algorithm to guide the robot from its starting position at $(1, 1)$ to the bottom right of the matrix (m, n)

maximizing the total reward it collects. Most of you offered a solution using dynamic programming that takes $O(mn)$ time, and uses $O(mn)$ space (see solutions). Use a combination of dynamic programming and divide and conquer to design an algorithm for guiding the robot from $(1,1)$ to (n,m) collecting the maximum amount of reward that takes $(m+n)$ space and $O(mn)$ time. You may assume that m is a power of 2.

Solution.

To solve the problem on the prelim we used dynamic programming to find the max value possible in the sub matrix of the top-left submatrix of columns $1,\ldots,i$ and rows $1,\ldots,j$, going from $(1,1)$ to (i,j). We will store this value as $M[i,j]$. Clearly, $M[1,j] = \sum_{k=1}^{j} v_{1,k} = v_{1,j} + M[1,j-1]$ and $M[i,1] = \sum_{k=1}^{i} v_{k,1} = v_{i,1} + M[i-1,1]$. Using this as base case, for $i,j > 1$ we use the recurrence:

$$M[i,j] = v_{i,j} + \max(M[i,j-1], M[i-1,j])$$

as we can reach the (i,j) corner either from the $(i,j-1)$ position or from the $(i-1,j)$ position. If we only wanted to get the max value, we can note that the ith row of $M[i,j]$ depends only on row $i-1$, so all other entries can be deleted, and we get the optimal value using only $O(n)$ space.

Alternately, we can also have different subproblems, where we dynamic programming to be the max value possible in the bottom-right submatrix of columns $i,\ldots,m$ and rows $j,\ldots,n$, going from (i,j) to (n,m). We will store this value as $\bar{M}[i,j]$. Clearly, $\bar{M}[m,j] = \sum_{k=j}^{n} v_{m,k} = v_{m,j} + M[m,j+1]$ and $\bar{M}[i,n] = \sum_{k=i}^{m} v_{k,n} = v_{i,n} + M[i+1,n]$. Using this as base case, for $i < m$ and $j < n$ we use the recurrence:

$$\bar{M}[i,j] = v_{i,j} + \max(\bar{M}[i,j+1], \bar{M}[i+1,j])$$

as we can go from the (i,j) corner either to the $(i,j+1)$ position or to the $(i+1,j)$ position.

Let $\ell = m/2$. As in the linear space Sequence Alignment algorithm, we will compute $M([i,j]$ for all $i = m,\ldots \ell+1$ keeping only the last computed row with $i = \ell+1$, and also compute $\bar{M}([i,j]$ for all $i = 1,\ldots \ell$ keeping only the last computed row with $i = \ell$.

Note that the optimal reward is

$$\max_j \bar{M}[\ell+1,j] + M[\ell,j]$$

This is true as $M[\ell,j]$ is the max reward from $(1,1)$ to (ℓ,j), and $\bar{M}[\ell+1,j]$ is the max reward from $(\ell+1,j)$ to (m,n). The optimal solution has to cross from row ℓ to row $\ell+1$ at some point, and the maximum above find the best possible point (ℓ,j) to pass through.

Let *FindMiddle*(m,n,V) denote the algorithm developed above. This algorithm uses $O(mn)$ time and $O(n)$ memory, and we got a position (ℓ,j) with $\ell \approx m/2$ that the optimal solution passes through.

Now we are ready to use divide an conquer to get the rest of the solution. For a matrix V we will use $V^{top}[i,j]$ to denote the submatrix of the top i rows and first j columns, and use $V^{bottom}[i,j]$ to denote the submatrix of the bottom rows starting with row i and the last columns starting with column j.

```
FindRewardPath(V,m,n)
if m = 1 or n = 1 output the unique path over the cells
else
    ℓ ← m/2
    Run FindMiddle(m, n, V) to get j
    FindRewardPath(V^top[ℓ, j], l, j)
    add down to the output path.  // for the step down from row ℓ to ℓ + 1
    FindRewardPath(V^bottom[ℓ + 1, j], m − l, n − j + 1)
```

We can prove correctness by induction. Clearly when $m = 1$ or $n = 1$, the path is unique and thus it gives the path with optimal reward. For larger m,n, we have found the best possible point (ℓ,j) to

pass through to row $\ell+1$, and the forward path from $(1,1)$ to (ℓ,j) as well as the backward path from (m,n) to $(\ell+1,j)$ are optimal by inductive hypothesis.

The memory used is $O(m+n)$, s we need $O(m+n)$ to output the solution. For the tables of the dynamic programming solutions, all we need is $O(n)$ and we can re-use the space, plus less than $O(m)$ space to maintain the stack of recursive calls.

To analyze the running time, let $T(m,n)$ denote the running time on an n by m matrix. On top level we spend cmn time for some constant c, and so we get the following recurrence

$$T(m,n) = cmn + T(\ell,j) + T(m-\ell, n-j+1)$$

as the column j is included in both subproblems. Consider the total time spent on the next level of recurrence. Note that $\ell = m-\ell = m/2$, so the time spent at the second level is $c(m/2)j+c(m/2)(n-j) = cm(n+1)/2$. Similarly, at the next level down we spend $cm(n+3)/4$ as both subproblems repeat a column, etc. Notice that a column will be only included in at most 2 subproblems at any level, so at the k level we get at most $cm(2n)/2^k$ time. This gives us a bound of at most:

$$T(m,n) = 2cmn + 2cmn/2 + 2cmn/4 + \ldots \leq 4cmn$$

Alternately, each column may at some point be where the down step occurs, and then it is part of both recursive calls, but each column is in at most two parts, so the sum of the n's of the subproblems at each level is at most $2n$, hence the total running time at level k is at most $c\frac{m}{2^{k-1}}2n$ summing to at most $4mn$ as claimed.

Alternate Solution.

An alternate way to write a recurrence using the same subproblems is to note that the optimal reward is

$$\max_j M[\ell,j] + M[\ell,j] - v_{\ell,j}$$

This is true, as there is some cell on row ℓ that the robot has to go through, and we need to subtract $v_{\ell,j}$ as that cell is counted in both subproblems. Lets call the code finding the j were max occurs in row ℓ as $FindCell(m,n,\ell,V)$ Now the recursive code is

```
FindRewardPath2(V,m,n)
if m ≤ 2 there are only ≤ n possible path, depending which point the path steps
to the second row.  Select the best of these paths and output it
else
    ℓ ← ⌈m/2⌉
    Run FindCell(m,n,ℓ,V) to get j
    FindRewardPath2(V^top[ℓ,j], ℓ, j)
    FindRewardPath2(V^bottom[ℓ,j], m−ℓ+1, n−j+1)
```

Note that in this version of the algorithm, we have to be careful with integer parts as in later calls for the algorithm the number of rows will not be a power of 2. Also, note that we need $m \leq 2$ as our base case, as with $m=2$ we have $\ell = 1$, and are looking for the position of the robot in row 1, which is cell $(1,1)$, so the next recursive call would be to the same problem.

We can prove correctness by induction as was done above. Clearly when $m \leq 2$ we find the path with optimal reward. For larger m, we have found the best possible cell (ℓ,j) to pass through, and ℓ as well as $m-\ell+1$ is less than m, so the forward path from $(1,1)$ to (ℓ,j) as well as the backward path from (m,n) to (ℓ,j) are optimal by inductive hypothesis.

The memory used is $O(m+n)$, s we need $O(m+n)$ to output the solution. For the tables of the dynamic programming solutions, all we need is $O(n)$ and we can re-use the space, plus less than $O(m)$ space to maintain the stack of recursive calls.

To analyze the running time, let $T'(m,n)$ denote the running time on an n by m matrix. On top level we spend cmn time for some constant c, and so we get the following recurrence

$$T'(m,n) = cmn + T'(\ell, j) + T'(m-\ell+1, n-j+1)$$

as the column j and row ℓ is included in both subproblems. Consider the total time spent on the next level of recurrence. Note that $\ell, (m-\ell+1) \leq m/2+1$, so the time spent at the second level is at most $c((m+/2+1)j + c(m/2+1)(n-j) = c(m/2+1)(n+1)$. Similarly, at the next level down we spend $c(m/4+1)(n+3)/4$ as both subproblems repeat a column, and the number of rows is at most $m/4+1$. Notice that a column will be only included in at most 2 subproblems at any level, so the total number of columns at any level of the recursion is at most $2n$. The number of rows in a subproblem at the kth level of recurrence is $m/2^k+1$. We can prove this by induction on k using the fact that m originally was a power of 2. So we get that the total running time spent at the kth level is at most $c(m/2^k+1)(2n)$ time. This gives us a bound of at most:

$$T'(m,n) = 2cmn + 2c(m/2+1) + 2cn(m/4+1) + \ldots \leq 4cmn + 2cn\log m = O(mn)$$

where the $2cn\log m$ accounts for the extra $+1$ in each term.

Reducing to Sequence Alignment.

Maybe the easiest way to solve this problem is to just reduce it to sequence alignment that we covered in class. Let the gap-penalty, $\delta = 1$, match-cost $\alpha_{xx} = 0$ for both characters $x = 0, 1$. And mismatch cost is infinite $\alpha_{01} = \alpha_{10} = +\infty$. Now we can only align identical characters. If no characters are aligned, the cost is $(n+m)\delta = n+m$, for the $m+n$ gap-costs. The cost of a general alignment is $k\delta$, if only k of the $(m+n)$ characters didn't align, so the remaining $(n+m-k)$ characters are matched, and aligned sequence form a superstring of length $k + (m+n-k)/2 = (m+n+k)/2$. So the alignment of minimum cost k is the same as the shortest superstring of length $(m+n+k)/2$ (as m and n are given, so in each case we are minimizing k.