Quiz 2 Solutions

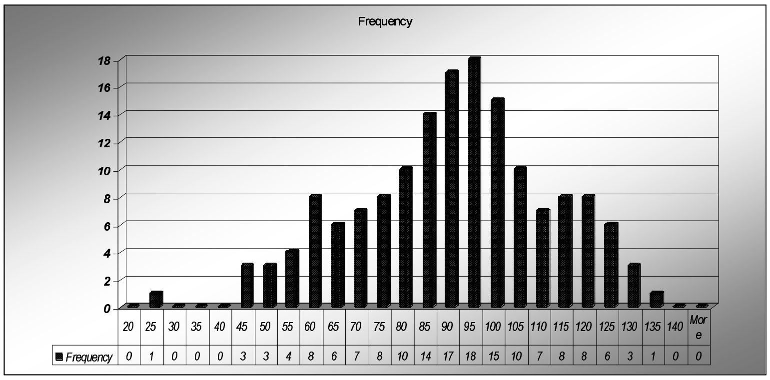

Figure 1: grade distribution

Problem 1. Analyzing auction sites

Professor Greenwood consults for Wall Street analysts who follow electronic auction sites, such as eBay and Yahoo!Auctions. The professor has developed a tool which can "walk" an auction site and produce a list of all products available for sale, their price, and the quantity of the product available. (Each product is uniquely identified by an integer key which is embedded in the HTML of the site but not displayed.) The professor runs the tool daily to produce this product information, which he emails to the analysts.

The analysts would like to understand the activity of an auction site better. Consequently, they have asked the professor to provide them daily with two additional lists:

List A: items that were sold between yesterday and today,

List B: items that have been newly posted on the site since yesterday.

Help the professor by designing an efficient algorithm to produce the two desired lists, including product quantities, given the product information for today and yesterday.

Solution: We will make the following assumption in the solutions - the professor will produce the new lists using only today's list and yesterday's list of product information. This means that it is not possible for the professor to tell if the difference in the quantities of a product in the two lists is the result of only additions of items of the product, only sales of the items of the product or is the result of instances of the same product being both added and sold between yesterday and today. The professor should make it clear to his clients that his lists merely give the differences in the quantities of products between the two days and do not give the actual number of items sold or added. Other reasonable assumptions are possible and may give slightly different solutions.

Assume that there are m items in yesterday's list Y and n items in today's list T. We will give two solutions that use $O(m + n)$ amount of space.

The first solution uses hash tables and has an average runtime of $O(m + n)$. Put all the items in yesterday's list Y into a hash table with $\Theta(m)$ slots. For each item in today's list T, check whether it exists in the hash table containing yesterday's items. If it does not exist, add the item together with its quantity to list B - this item is newly added. If the item exists, check whether the quantity has increased, decreased or remained the same. If it has increased, add the difference to list B under the assumption that the increase is caused by additions of products. If it has decreased, add the difference to list A under the assumption that the decrease is caused by items being sold. If it remained the same, do nothing.

Next, put all the items in today's list T into a hash table with $\Theta(n)$ slots. For each item in yesterday's list Y, check whether it exists in the hash table. If it does not exist, put the item together with its quantity into list A under the assumption that it has sold out between yesterday and today. This hash table can be stored to be reused instead of deleting it and recreating it again the following day.

Under the assumption of simple uniform hashing (or by using a universal hash function), insertions and lookups has average runtime of $O(1)$, giving a total runtime of $O(m + n)$. Some people used perfect hashing and claimed that this gives a worst case runtime of $O(m + n)$. This is not true if the perfect hash function has to be found each time. The technique in CLRS uses a randomized algorithm to find the perfect hash function - this has an average (not worst case) runtime of $\Theta(n)$, where n is the size of the key set.

The second solution sorts the two lists Y and T and uses a routine that is similar to the merge routine used in merge sort to create the lists A and B. The worst case runtime is $O(n \lg n + m \lg m)$ if mergesort or heapsort is used to sort the lists.

First Y and T are sorted. We then initialize i and j to 0. The following is repeated until one of the lists runs out of items.

- If the key of $Y[i]$ is the same as the key of $T[j]$, then the quantities are compared. If the quantity of $Y[i]$ is greater than the quantity of $T[j]$, the item is added to list A together with the difference in the quantities, under the assumption that the decrease is caused by the product being sold. If the quantity of $Y[i]$ is less than the quantity of $T[j]$, the item is added to list B together with the difference in quantities, under the assumption that the increase is caused by addition of the product. Both i and j are increased by 1.
- If the key of $Y[i]$ is less than the key of $T[j]$, this indicates that the item $Y[i]$ has been sold out because it cannot appear later in list T which is sorted. This item is added to the list A and i is increased by 1.
- If the key of $Y[i]$ is greater than the key of $T[j]$, this indicates that the item $T[j]$ is new as it cannot appear later in list Y which is sorted. This item is added to the list B and j is increased by 1.

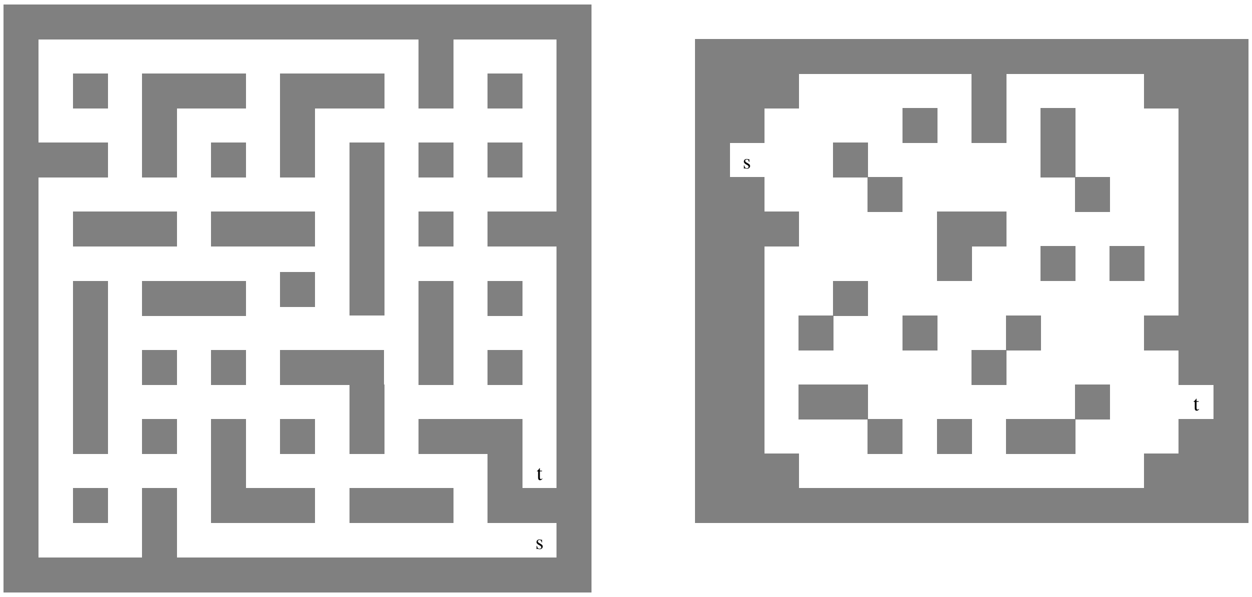

Figure 2: Two no-left-turn maps. Find a path from s to t with as few right turns as possible, and no left- or U-turns. You can play with these maps using a Java applet on `http://www.gjnem.demon.co.uk/noleft/ixnoleft.htm`.

If T runs out first, the rest of items in list Y are appended to list A as they do not appear in T and are considered sold out. If Y runs out first, the rest of the items in list T is appended to list B as they do not appear in Y and are considered as new items. The routine (excluding the sorting part) runs in time $O(m + n)$.

The sorted list can be used the as the next day's Y list, so only one list needs to be sorted each day. If the largest key size is known, radix sort can be used and will reduce the worst case runtime to $O(m + n)$, if $b = O(\lg z)$ where b is the number of bits required to represent the keys and z is the size of the list to be sorted.

Problem 2. No left turns

You've just stolen a brand-new BMVW, but the owner has locked the car with *The Spade*, a mechanical lock on the steering wheel, which prevents you from making left turns. In addition, to avoid car accidents and otherwise attracting attention, you aren't going to make any U-turns either. You want to reach your chop-shop[1] as quickly as possible, and since you can drive down straight streets very quickly, you only care about minimizing the number of (right) turns you make.

Fortunately, you have a map of the city, so you can plan out a route that uses no left turns and no U-turns. Even better, the map comes in electronic form as an $m \times n$ grid of cells, each marked as either ***empty*** (navigable) or ***blocked*** (unnavigable). At each cell you visit, you can continue in the direction you were going, or turn right. Two examples of no-left-turn maps are shown in Figure 2.

[1]A garage that illegally disassembles cars in order to sell the parts.

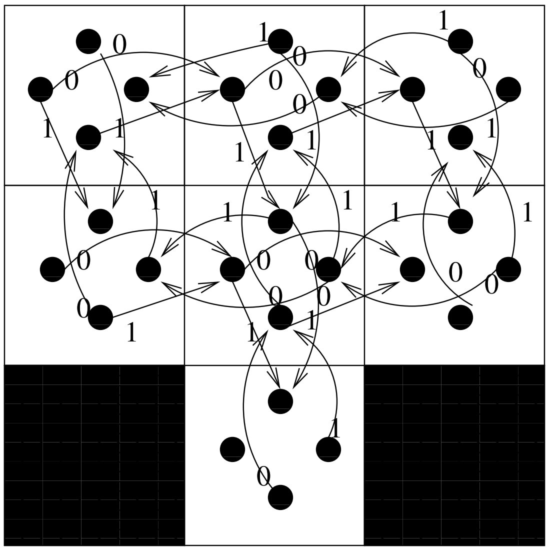

Figure 3: Graph Representation

Give an efficient algorithm to find a no-left-turn path through an $m \times n$ map using the minimum number of right turns. (*Hint:* For intuition, solve the left-hand map in Figure 2.)

Solution: The basic idea in solving this problem was to create a graph representation of the problem and then use a shortest paths algorithm to find the optimal no-left-turn path. Creating the graph typically took $O(nm)$ and the shortest path problem could be solved in $O(nm)$ as well. Acknowledgements to Lauren Bradford, Vida Ha, and Josh Yardley who expressed some of these ideas very clearly in their write ups.

One way to turn this problem into a graph problem is as in the figure. Our map of cells is converted into a weighted, directed graph. For each of the m*n cells of the map, we create 4 nodes. If the car enters a particular cell from the cell below it on the map it would enter the "south" node associated with that particular cell. After creating the nodes, we assign directed edges to them. Edges that correspond to a righthand turn have an edge weight of one and edges that correspond to driving straight have a weight of zero. Edges that would represent left hand turns or uturns are not included in the graph. An edge which goes from one orientation to another (example south to west) represents a right turn. It will take us $O(mn)$ time to create this graph representation.

Shortest Path Algorithms

Now that we have our graph representation we need to determine the path that minimizes right hand turns. With our edge weight representation finding the shortest path from source s to sink t should be effective. Many students observed that since we had positive weight edges, we could run Dijkstra's algorithm to calculate the shortest path in our graph. Since Dijkstra's algorithm runs in $0(V \lg V + E)$ time, this will give us a running time of $O(nm \lg mn)$. However, we can do better than this! One way to optimize this is to use 2 linked lists to optimize the min-priority-queue in Dijkstra's algorithm. The first list is of the elements reachable with weight 0 and the second is elements available with weight 1. Extract-min returns the head of the first list, unless it is null, in which case it returns the head of the second list. Since it only needs to maintain and look at these two lists, it works in constant time, making the algorithm run in $O(mn)$. Another optimization works similarly, using an auxiliary data structure to do a modified breadth first search to t. That is to say, you visit all the edges you can get to using the same number of right turns each time. The following is a breadth first search algorithm that uses a linked list with pointers to the beginning and end of the list.

In order to perform breadth first search you'll first pick off the first gray node in your queue. Then you'll want to look at the graph information and see if the cell in front and to the right are free. Check if the square to the front and to the right are empty. If so, then these cells, are the children of the node you're looking at. Depending on the children's color, you'll do one of three things.

1. If this node has no children, just color it black. No children means that the right and straight ahead of you are blocked cells, so this is a dead end.
2. If the children are all gray and black, this means that the children have been discovered by another path. Since this is a breadth first search by the number of right turn edges, we know that if the node has already been discovered, another path was able to reach it with the same or less number of right turns. Thus, this path cannot have a smaller number of right turns than that earlier path that discovered it. Just color this node black just like in case 1.
3. If either child is white, color this node black. If its right child is white, color that gray and add it to the end of the queue. If the straight ahead child is while, color it gray also, but add it to the beginning of the queue, since you did not need to turn right to get to that node. The cell reached by going straight has the same number of right turns as its parent cell, so both should be considered before the thing with more right-turn edges. By adding this cell to the beginning of the queue, you will consider this cell immediately next, which is the correct behavior since it has the same number of right turns as the one you checked.

Since the linked list can be maintained in constant time and breadth first search runs in $(V + E)$ our algorithm will run in $0(m, n)$.

Correctness
Since our graph represents driving straight with a zero weight edge and turning right with an edge weight of one, the shortest path from s to t will be the path with no right turns. Our shortest path algorithm cannot make any illegal turns because those edges do not exist. For the correctness of Dijkstra's algorithm, I refer you to CLRS.

The BFS algorithm presented above will visit every node accessible from s until it reaches t. Since it will always visit the nodes which it can go straight before it turns right, the path which reaches t first, will be the path with the least number of right turns.

Problem 3. Really short paths

If a digraph $G = (V, E)$ with edge-weight function $w : E \to \mathbb{R}$ contains a negative-weight cycle, some shortest paths may not exist. Assume that every vertex $v \in V$ is reachable from a given source vertex $s \in V$. Give an efficient algorithm that labels each $v \in V$ with the shortest-path distance from s to v or with $-\infty$ if no shortest path from s exists. Be sure to give a rigorous argument for the correctness of your algorithm.

Solution: In dealing with single source shortest-paths problem, some shortest paths may not exist if there is a negative-weight cycle. This is because it would then be possible to keep going round the cycle relaxing shortest path, thus making the concept of a shortest path ill-defined.

To deal with this problem, we will have to detect the negative-weight cycle and assign every vertex that is affected by the cycle with $-\infty$. Note that it's not just the vertices in the cycle that are affected. Those verticies outside the cycle, but which can be reached from the verticies in the cycle will also be affected, since on the way, we can go around the cycle numerous times to relax the shortest path.

Thus the algorithm is divided into two parts: first, we use a modified version of Bellman-Ford algorithm to assign those verticies in the negative weight cycle to $-\infty$. Then, from each of those vertices, we perform a Depth-First-Search, and assign all verticies reacheable from it with $-\infty$.

Specifically, we modify Bellman-Ford algorithm, as presented in CLRS on page 588, so that instead of returning false in line 7, v is assigned a value of $-\infty$, and added to a list which we call the Negative-Vertex-List. In line 8, instead of returning true, we perform a Depth-First-Search for the verticies in the Negative-Vertex-List. Every vertex reached in the Depth-First-Search is assigned a value of $-\infty$ after it is colored black.

Proof of Correctness

Lemma 3.1: The algorithm works correctly if there are no negative-weight cycles.

Proof: This follows directly from Lemma 24.2 and Theorem 24.4 of CLRS: The only changes we have made to the Bellman-Ford algorithm are in line 7 and line 8. But line 7 will never be reached if there is no negative-weight cycle, as described in the proof of theorem 24.4. Also, the modified line 8 refers only to vertices that were added in line 7, and hence will contain an empty list since line 7 is not executed and therefore no vertex is added to the Negative-Vertex-List. Thus, the algorithm works correctly, when there are no negative-weight cycles

Lemma 3.2: If there is a negative-weight cycle, the algorithm will detect it. Specifically, there exists an edge on each negative-weight cycle such that the test in line 6 will be true.

Proof: This follows from the second part of the proof of Theorem 24.4 in CLRS, reproduced in a slightly modified form below:

Suppose that graph G contains a negative-weight cycle that is reachable from the source s; let this cycle be $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$. Then,

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \ . \tag{1}$$

Assume for the purpose of contradiction that our algorithm does not find this negative-weight cycle i.e. line 7 is not executed. Thus, $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$. Summing the inequalities around cycle c gives us

$$\begin{aligned}\sum_{i=1}^{k} d[v_i] &\leq \sum_{i=1}^{k}(d[v_{i-1}] + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i) \ .\end{aligned}$$

Since $v_0 = v_k$, each vertex in c appears exactly once in each of the summations $\sum_{i=1}^{k} d[v_i]$ and $\sum_{i=1}^{k} d[v_{i-1}]$, and so

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}] \ .$$

Moreover, by Corollary 24.3, $d[v_i]$ is finite for $i = 1, 2, \ldots, k$. Thus,

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i) \ ,$$

which contradicts inequality (1). We conclude that our algorithm will detect a negative-weight cycle, if there is one.

Lemma 3.3: The algorithm will correctly assign all verticies reacheable from the negative-weight cycle with $-\infty$. (Remember that these verticies are precisely those which will not have shortest path, as explained in paragraph two of this solution)

Proof: Our line 7 has already assigned a vertex in the negative-weight cycle with $-\infty$. It follows from the definition of a Depth-First-Search that all the vertices reacheable from that vertex will also be explored and hence assigned $-\infty$.

Theorem: The algorithm works correctly to label each $v \in V$ with the shortest-path distance from s to v or with $-\infty$ if no shortest path from s exists.

Proof: This follows directly from the three lemmas that we have just proven.

Running Time: Clearly, the running time is the same as the unmodified Bellman-Ford algorithm: $O(VE)$. Depth-First-Search has a running time of $\Theta(V + E)$. The rest of the algorithm runs in the same time as the unmodified Bellman-Ford algorithm: $O(VE)$. Therefore, overall running time is $O(VE)$.

Problem 4. Grading

Professor Leemainerson has assembled a numerical score for each of the n students in his algorithms class based on final exam, quizzes (in-class and take-home), problem sets, and most importantly, class participation in lecture and recitation. The term scores form a list $\langle a_1, a_2, \ldots, a_n \rangle$

of real numbers. In order to assign a final grade of A, B, C, or F, the professor has devised the following scheme.

The professor first determines four ***archetypical*** scores x_A, x_B, x_C, and x_F. Then, he gives each student the grade corresponding to which archetypical score is closest to the student's term score, breaking ties in favor of the higher grade (because the professor is a nice guy).

To determine the four archetypical scores, the professor examines the distribution of term scores and finds the best piecewise constant approximation of the distribution. That is, he chooses the four archetypical scores to minimize the quantity

$$\sum_{i=1}^{n} \min_{k\in\{A,B,C,F\}} |a_i - x_k| \ ,$$

breaking ties by minimizing $x_A + x_B + x_C + x_F$.

Solve the professor's grading problem by giving an efficient algorithm to compute the final grades of his students from their term scores.

Solution: The professor's grading problem can be solved in a variety of different ways. We have seen correct approaches that take $O(n^5)$, $O(n^4)$, $O(n^3)$ and $O(n^2)$ time.

A crucial fact that several students didn't seem to know is that, given a sequence $a_1, a_2, \ldots, a_m$, the value X which minimizes $\sum_{i=1}^{m} |a_i - X|$ is the *median* of these m numbers and not their mean, as many students thought[2]. We can use a simple argument to prove this.

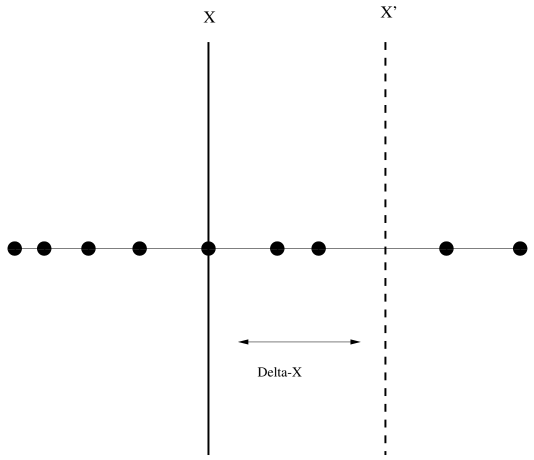

Figure 4: Why the median minimizes the sum

Proof Referring to Figure 4 , let X be the median and let $S = \sum_{i=1}^{m} |a_i - X|$. Suppose that there is another value X' such that $\sum_{i=1}^{m} |a_i - X'| < S$. We can write $X' = X + \Delta X$. Without

[2]The mean would minimize $\sum_{i=1}^{n}(a_i - X)^2$, commonly referred to as the "mean-square error"

loss of generality, say that X' lies to the right of X. We notice that when we change X by ΔX, the contribution of each of the m elements to the sum changes by an amount which is between ΔX and $-\Delta X$, depending on the element's position relative to the change. The contribution of each element on the left (including the point on which the median lies) increases by +ΔX. Since X is the median, there are $\lceil \frac{m}{2} \rceil$ elements on the left, and their total contribution to the sum increases by $\lceil \frac{m}{2} \rceil \cdot \Delta X$. The contribution to the sum of the remaining $\lfloor \frac{m}{2} \rfloor$ elements can decrease by *at most* $\Delta X \cdot \lfloor \frac{m}{2} \rfloor$, so the decrease cannot be bigger than the increase. Hence $\sum_{i=1}^{m} |a_i - X'|$ cannot be any smaller than S, which contradicts our original assumption that there is some other value $X' \neq X$ which (strictly) decreases the sum. Notice that if m is even, there will be two medians, both of which minimize the sum. $\square$

Going back to our original problem, we observe that, if the list of n grades is sorted, the optimal solution will look like Figure 5.

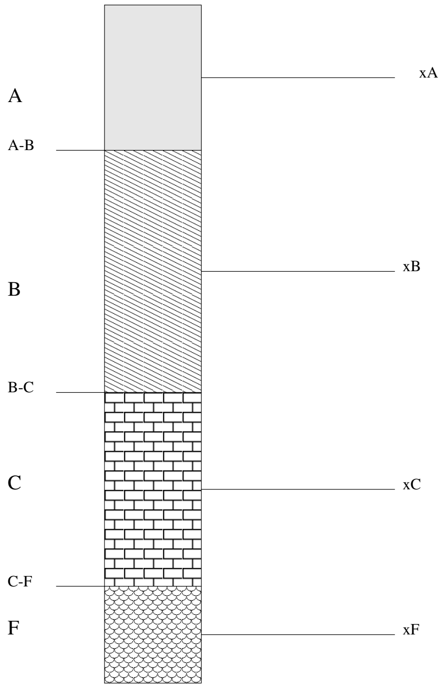

Figure 5: What the grade distribution may look like

We will now prove that in an optimal solution, the four archetypical scores x_A, x_B, x_C, x_F will be the *lower* medians[3] of the A range, the B range, the C range and the F range.

Proof Given the four ranges, let $\Sigma(A)$, $\Sigma(B)$, $\Sigma(C)$ and $\Sigma(F)$ be their contributions to the total sum that we want to minimize. Above, we have proved that if we let x_A, x_B, x_C and x_F be the me-

[3]the term "lower" refers to the smaller of the two medians, if there are two, as is the case when the range has an even number of elements

dians of the respective ranges, each of these four contributions will be minimized (independently), and so will their sum.

A slight complication that may arise, which very few students noticed, is the situation depicted in Figure 6 which zooms in on the A-range and the B-range. There, choosing the medians for the A range and the B range may well minimize the contribution of each of the two ranges to the sum, but we are violating the rule for choosing grades. Namely, the lowest score in the A range is closer to x_b than it is to x_a, so that score should have corresponded to a B.

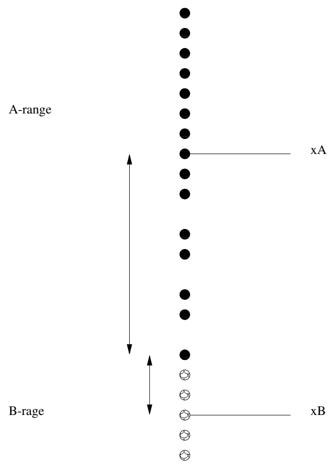

Figure 6: A complication that's impossible to happen in an optimal solution

However, such a grade distribution cannot be optimal. This is because if we keep x_a and x_b as they are, and move that problematic score to the B range, the contribution of that score to the sum will decrease while the contribution of all the other scores will remain the same, thereby giving us a lower total sum. This contradicts the assumption that what we have is an *optimal* solution. Therefore, we conclude that in any optimal solution, the archetypical scores will be the medians of the four ranges.

Now, in case there are *two* medians for a given range, they will both minimize the total sum. Since we are supposed to break such ties by minimizing $x_A + x_B + x_C + x_F$, we should choose the smaller of the two medians, namely the *lower* median. □

Given these facts, a straightforward algorithm to solve this problem is the following:

Try all $\binom{n}{4}$ combinations of x_A, x_B, x_C, x_F. For each choice, compute $\sum_{i=1}^{n} \min_{k \in \{A,B,C,F\}} |a_i - x_k|$ and pick the combination that leads to the minimum value. In case of ties, choose the combination that minimizes $x_A + x_B + x_C + x_F$. Once we find the best choice for x_A, x_B, x_C, x_F, we can compute the students' grades by finding the archetypical score which is closest to each score.

The correctness of the algorithm follows from the fact that we are comparing all the candidate solutions, and picking the one that minimizes the total sum. The bad thing is that our algorithm is terribly inefficient: we need $O(n^4)$ time to go over all the $\binom{n}{4}$ combinations of archetypical scores, and for each combination, we need another $O(n)$ time to compute $\sum_{i=1}^{n} \min_{k\in\{A,B,C,F\}} |a_i - x_k|$, for a total running time of $O(n^5)$. After we have determined the archetypical scores, we can compute the students' grades in $O(n)$ time since for each grade, we just perform 4 subtractions and pick the archetypical score which minimizes the absolute value of the difference.

We can do slightly better by observing that since the archetypical scores are the medians of the four ranges, we can instead try all possible combinations of ranges. The only thing we need to determine the ranges are the *cutoffs*. Since there are 4 ranges, there are 3 cutoffs (between A-B, B-C, C-F). Knowing the cutoffs allows us to determine the archetypical scores (medians) in $O(1)$ time[4], and proceed as before. However, there are now "only" $\binom{n}{3}$ combinations of cutoffs to try. For each, we can compute $\sum_{i=1}^{n} \min_{k\in\{A,B,C,F\}} |a_i - x_k|$ in $O(n)$ time, for a total running time of $O(n^4)$.

Sticking to our strategy of trying out all possible combinations of cutoffs, we can do even better if we observe that spending $O(n)$ time to compute the total sum every time is overkill. The contribution of grade G to the total sum is $\sum_{i=G_{low}}^{G_{high}} |a_i - x_G|$ and is uniquely determined by (G_{low}, G_{high}), the indexes of the lowest and highest scores in G. If we build a table with all $\binom{n}{2}$ possible ranges that grade G can have, we can use it to compute the total sum in $O(1)$ time for every combination of cutoffs by simply performing 4 table lookups. This would reduce the running time of our algorithm to $O(n^3)$, provided, of course, that we don't need more than $O(n^3)$ time to build that table. Indeed, a simple way to build the range-cost table in $O(n^3)$ time is to go over all $O(n^2)$ possible ranges, compute the median in $O(1)$ time, and then compute the contribution of that range to the sum in $O(n)$ time. Building the table this way requires $O(n^3)$ time, and allows us to try out all $\binom{n}{3}$ combinations of cutoffs in $O(n^3)$ time by doing 4 constant-time table lookups for each combination.

We can further improve the time it takes to build our lookup-table to $O(n^2)$ by observing that if we have computed Contrib(i, j), the contribution of range (i, j) to the total sum, then we can compute the Contrib$(i - 1, j + 1)$ in $O(1)$ time because the two ranges have the same median μ, and so the contribution of the second (larger) range is simply the contribution of the first one plus the two extreme values, namely

$$\text{Contrib}(i-1, j+1) = \text{Contrib}(i, j) + |a_{i-1} - \mu| + |a_{j+i} - \mu|$$

This way we can compute the contributions of all the ranges that have a *specific median* in $O(n)$ time. Since there are n possible medians, we can compute the contributions of all possible ranges in $O(n^2)$ time.

In order to reduce the running time of the entire algorithm even further, we can use the *dynamic programming* approach presented below.

[4]assuming that the scores are sorted. If they are not, sort them using MergeSort in $O(n \log n)$ time. This is too small to affect the total running time of any of the algorithms we present here

Let $C^4_{i,j}$ be the minimum total sum $\sum_{\ell=i}^{j} \min_{k\in\{G_1,G_2,G_3,G_4\}} |a_\ell - x_k|$, that is, the minimum total sum when we allocate four grades to a range (i, j). Similarly, let $C^3_{i,j}$ be the minimum total sum when we assign only three different grades to the range (i, j) and so on. Observe that $C^4_{1,n}$ is simply the minimum value of $\sum_{i=1}^{n} \min_{k\in\{A,B,C,F\}} |a_i - x_k|$ and that $C^1_{i,j}$ is just Contrib(i, j). Now, we can define $C^4_{1,n}$ as follows:

$$C^4_{1,n} = \min_{1\le k\le n} C^3_{1,k} + C^1_{k,n}$$

The value of k which minimizes the expression above is basically the A-B cutoff: indeed, in the optimal solution which makes use of 4 distinct grades, all the scores from 1 to k are assigned three different grades (B,C,F) in an optimal way, and the rest are assigned one grade (A). In general,

$$C^i_{1,j} = \min_{1\le k\le j} C^{i-1}_{1,k} + C^1_{k,j} \tag{2}$$

Thus, computing $C^i_{1,j}$ when $C^{i-1}_{1,k}$ and $C^1_{k,j}$ are available takes $O(n)$ time (we are going over $j \le n$ possible values of k and taking the minimum). Computing $C^i_{1,j}$ for *all* possible values of j thus takes $O(n^2)$ time.

We have seen above that we can compute all $C^1_{i,j}$ combinations in $O(n^2)$ time. Given these, we can compute and tabulate $C^2_{1,j}$ for all $1 \le j \le n$ in $O(n^2)$ time and similarly for $C^3_{1,j}$. As shown above, this will enable us to compute $C^4_{1,j}$ in a total of $O(n^2)$ time. By keeping track of the values of k that led to the minimum value each time in Equation 2, we can reconstruct the cutoffs for A,B,C,F. Knowing the cutoffs of the optimal solution, we can compute the archetypical scores by taking the lower median of each range, for the reasons explained above. Knowing the archetypical scores, we can assign grades in $O(n)$ time.

Problem 5. Holey files

The HOLIX operating system maintains files as consecutive sequences of characters. In addition to being able to append to a file and overwrite characters, HOLIX supports insertion of characters at arbitrary points in the file. To avoid shifting the whole file for an insertion at the beginning of the file, HOLIX allows ***holes*** to be stored in the file as a sequence of special ***hole characters***. An insertion can overwrite a hole character, but if the point of insertion occurs between two normal characters, then some characters must be shifted, but hopefully as few as possible. A read operation scans the file in the normal linear order, skipping over any holes it encounters.

HOLIX supports holey files through the abstraction of a ***cursor***. Each file has one cursor pointing to a location between two consecutive normal characters in the file. All operations take place at the location of the cursor, and the user is unaware of any holes in the file. More specifically, HOLIX supports the following API (application programming interface):

- NEW-FILE(): Creates an empty file and returns its cursor.

- MOVE-CURSOR(p, d): Moves cursor p forward in its file by d (normal) characters if d is positive, or backwards in its file by $-d$ characters if d is negative.
- READ(p): Returns the (normal) character immediately after p in p's file.
- INSERT(p, x): Inserts character x at p and moves p to immediately after x.

HOLIX's implementation of holey files treats each file as an array in which each array element is either a normal character or a hole character, and the two types of characters can easily be distinguished. The normal characters must remain in their proper order within the file, with hole characters interspersed. In order to support the movement of cursors, the first and last hole characters of a hole store the hole's length. (After all, the bits in the hole are otherwise unused.) Thus, skipping over a hole takes constant time.

Give an efficient algorithm for maintaining holey files. You should try to minimize both the time required for the operations and the total number of characters needed to support a file with n normal characters.

Solution: The best balance of space and speed of functions that we know of is the following method. With $\Theta(n)$ space, we can support the operations in the following times.

- NEW-FILE(): $\Theta(1)$
- MOVE-CURSOR(p, d): $\Theta(d)$
- READ(p): $\Theta(1)$
- INSERT(p, x): $\Theta(1)$

Our general strategy is to always keep only one group of holes all consecutive together, and maintain the invariant that p always points right before the leftmost hole of the group unless the array is full. Another strategy we use is an array that doubles in size (in amortized constant time) so that we can continually expand to handle any number of characters. This arrays running time and correctness was demonstrated in lecture.

Our procedures work in the following manner. HOLE-LENGTH returns to us the length of the hole that p is currently pointing to, and by our invariant, if a hole exists, p will be pointing to it. It will return 0 if no hole exists.

- NEW-FILE():
  - Allocate an array of size 2, all entries filled with holes and the pointer pointing to just before the first hole.
- MOVE-CURSOR(p, d):
  - If d is positive, we set p to be $p +$ HOLE-LENTH $+ d$ and shift the hole over d characters to the right by moving the d characters just to the right of the hole to before the hole.
  - If d is negative, we set p to be $p + d$ and again shift the hole left displacing only d characters.

    - Set the two holes at each end of the hole segment to display the proper hole length.
- READ(p):
    - Return the character at $p+$ HOLE-LENTH.
- INSERT(p,x):
    - If p points to a character, then the array is full so we use the constant amortized time array doubling algorithm and return the new array with all items before p in the first part of the array, followed by half an array of holes, followed by the items that were after p in the intial array. We set p to point to the beginning of the holes.
    - Insert the character x into the hole that p points to, and increment p, and decrement the hole size characters.

I argue that each of these procedures maintains our invariant and keeps the characters in order in the proposed running times.

NEW-FILE trivially maintains the invariant as it sets the cursor to point to the leftmost hole, runs in constant time and keeps all characters in order.

In MOVE-CURSOR(p,d) we take constant time to calculate the distance to move p and then take d time to shift the d displaced characters. The time to shift these characters is not dependent on the number of holes because holes do not need to be shifted as they are all identical. Furthermore it takes only constant time to fix the end hole lengths to their proper values. We do not change the order of any characters, and p continues to point to the beginning of the hole segment.

READ(p) also maintains our invariant because it does not change the state of the system, and returns the next real character after the hole as it should.

INSERT(p,x) also maintains the invariant. If the array is full then it will be doubled to a new array in constant amortized time, and the pointer position will be maintain. Then the character will simply be inserted into the first hole in constant time and p incremented so that it points to the beginning of the hole.

Clearly, this implementation requires at most $2*n$ space for any n characters in the system and so it uses space $\Theta(n)$.

Optimization We can optimize this algorithm further to not take as many actual operations although the asymptotic running time will be the same. We do this by not moving the hole every time a MOVE-CURSOR(p,d) is called, but instead wait until INSERT(p,x) is called to move the hole to where we need it. Since we can store up d potential for every time MOVE-CURSOR(p,d) is called, the worst we will have to move the hole to insert is as far as the cursor has moved, which is already stored potential. In some cases we may even move the hole less if the cursor moves in both directions and the final position is closer than other points it reached along its path and then we save operations.