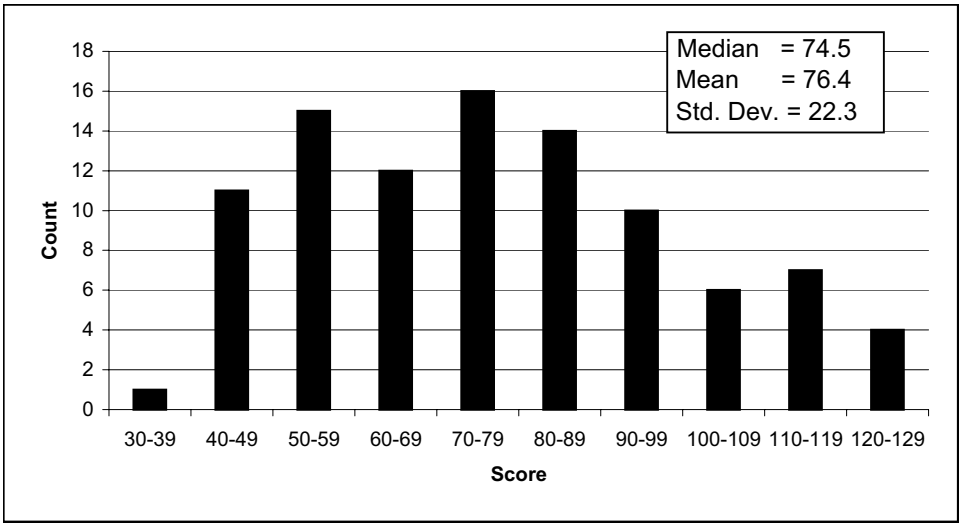
# Quiz 2 Solutions



**Problem 1. A Dynamic Set**

A dynamic set $Q$ contains items, where each item $x$ has an associated key *key*$[x]$. The dynamic set $Q$ supports the following operations:

- INSERT$(x, Q)$: Insert item $x$ into $Q$.
- $x \leftarrow$ EXTRACT-OLDEST$(Q)$: Remove and return the oldest item $x \in Q$. (The oldest item is the one inserted least recently.)
- $x \leftarrow$FIND-MAX$(Q)$: Return (but do not remove) the item $x \in Q$ for which *key*$[x]$ is maximal.

Design a data structure for $Q$ that can perform any sequence of $n$ operations efficiently.

**Solution:**

The goal of the problem is to come up with an efficient implementation of a *queue*, augmented with the additional FIND-MAX operation. Thus, it makes sense take a queue implementation[1] (e.g., a linked-list) and extend it so that it supports FIND-MAX as well.

For starters, we assume that all elements are distinct. This assumption is not crucial, but simplifies the description and the analysis of the algorithms. Note that, in any case, we can break ties by replacing each element $x$ by $(x, t_x)$, where $t_x$ is the time when $x$ was inserted, and using lexicographic ordering.

We will see that there are, essentially, three basic ways of supporting FIND-MAX:

1. Scan all elements in the list to find the maximum. This gives $\Theta(n^2)$ total worst-case running time. This solution was worth up to 8 points.

   A slightly better implementation maintains a pointer to the maximum element; when the element is deleted, the EXTRACT-OLDEST procedure searches for the new maximum. Although this implementation still has $\Theta(n^2)$ worst-case time bound, it gives a much better time bound in the *average case*[2]. Thus, this solution was worth up to 10 points. Also, this was by far the most popular solution.

2. Augment the queue with a search tree (description below). In this way, all three operations can be performed in $O(\log n)$ time. This gives $O(n \log n)$ total worst-case running time. This solution was worth up to 15 points.

3. Augment the queue with a queue-stack, that is, a hybrid of a queue and a stack. This is the best solution - it gives $O(n)$ total time in the worst-case (although individual operations could take $\Omega(n)$ time). Description below.

A remark: our "augmentation" is actually very non-invasive. That is, we do not modify the code or structures of the queue; instead, we add a new data structure that operates in parallel to the queue.

[1] One can also use a search tree with elements sorted by insertion time, but this is an overkill, since the elements are not deleted in a random order.

[2] E.g., when all insertions precede all deletions, and the order of deletions and insertions is random

This approach has an advantage of being simple and modular. E.g., there is no need to get into the guts of the queue code. However, modifying the queue code works fine as well.

**The queue with a tree.** Consider any implementation $L$ of a queue, with operations INSERT and EXTRACT-OLDEST. In addition, we will also maintain a search data structure $T$. The data structure should support three operations in $O(\log n)$ time: INSERT, DELETE and FIND-MAX. E.g., one can use a 2-3-tree, a Red-Black tree, or even a heap[3].

Each element $x$ that is present in $Q$ is maintained in both $L$ and $T$ (although space savings can be obtained by storing $x$ only once, and using pointers). Specifically, we implement insertions and extraction to $Q$ as follows.

```
Insert(Q,x):
  Insert(L,x)
  Insert(T,x)
End

Extract-Oldest(Q):
  x:=Extract-Oldest(L)
  Delete(T,x)
  Return x
End
```

Therefore, we can find the maximum as follows:

```
Find-Maximum(Q):
  Return Find-Max(T)
End
```

Clearly, all three operations run in $O(\log n)$ time. This gives $O(n \log n)$ total running time bound.

**The queue with a queue-stack.** We will use a queue $L$ and a queue-stack $L'$. The latter supports INSERT and EXTRACT-OLDEST (as in the queue), as well as EXTRACT-YOUNGEST (as in the stack). For simplicity, we also assume functions OLDEST and YOUNGEST, which return the same elements as EXTRACT-OLDEST and EXTRACT-YOUNGEST, but without removing them from $L'$. All operations are performed in constant time.

The queue $L$ contains *all* elements currently in $Q$. The queue-stack $L'$ contains only some of the elements of $Q$, selected in the following way. Let $a_1 \ldots a_l$ be the (current) elements of $Q$ (and $L$), the *oldest* element first. Let $b_1 \ldots b_k$ denote the elements of $L'$, the *oldest* element first. The choice of $b_i$'s is as follows. The element $b_1$ is the maximum element of $Q$. For each $i \geq 1$, let $j_i$ be the

[3]Although in this case one needs to argue how to delete an arbitrary element in the heap, since the data structure described in CLRS, p. 127, does not enable arbitrary deletions, at least as stated.

index of $b_i$ in $Q$, i.e., such that $a_{j_i} = b_j$. For $i > 1$, $b_i$ is defined as the maximum element among $a_{j_{i-1}} \dots a_l$.

In other words: as long as $b_1$ is in $Q$, it is the maximum element. However, when $b_1$ is deleted, then $b_2$ becomes the new maximum. And so on. Notice that the sequence $b_1, b_2 \dots$ is decreasing. We call it the *maximal sequence* of $a_1 \dots a_l$.

Clearly, given $L'$, we can answer FIND-MAX in constant time: simply return $b_1$, i.e., OLDEST($L'$). To maintain $L$ and $L'$ (that is, the sequences $a_1, \dots, a_l$ and $b_1, \dots, b_k$), we do the following.

```
Insert(Q,x):
  Insert(L,x)
  While key[Youngest(L')]<key[x] Extract-Youngest(L')
  Insert(L',x)
End

Extract-Oldest(Q):
  x:=Extract-Oldest(L)
  If x=Oldest(L') then Extract-Oldest(Q)
  Return x
End
```

The correctness of EXTRACT-OLDEST is immediate from the definition of the maximal sequence. The correctness of INSERT follows from the observation that the maximum sequence for $a_1 \dots a_l, x$ contains only the elements of the maximum sequence for for $a_1 \dots a_l$ which are $\geq x$.

To show that $n$ operations take $O(n)$ time, we first observe that both FIND-MAX and EXTRACT-OLDEST take $O(1)$ time in the worst-case. Moreover, the total cost of all INSERT operations is bounded from above by the total number of operations INSERT into $L'$ (which is at most $n$) plus the total number of operations EXTRACT-YOUNGEST from $L'$. But, the latter is at most the total number of insertions into $L'$, which again is at most $n$. Thus, the total time per $n$ operations is $O(n)$. Note that the same analysis can be done using the accounting or potential method.

**Problem 2. Great Minds Need Coffee**

It is a beautiful autumn morning, and Professor Indyk has decided to walk from his apartment to MIT where he will give a lecture for 6.046. To prepare for this trip, the professor has gone online and downloaded a map of the $n$ roads and $m$ intersections in the Boston area. For each road $e$ connecting two intersections, the map lists the length $w[e]$ of the road in meters. (Recall that in the Boston area, an arbitrarily large number of roads can meet at a single intersection, e.g., Davis Square.)

Like many great theorists, Professor Indyk cannot walk more than $1000$ meters without sitting down and assuaging his caffeine addiction. (This is especially true before 9:30 A.M.) Fortunately, the Moonbucks Coffee Company has recently opened a large number of coffee shops throughout

the Boston area. Let $B = \{b_1, b_2, \ldots, b_k\}$ be the set of $k$ intersections hosting Moonbucks coffee shops.

Thus, Professor Indyk is looking for a route from his apartment (located at one given intersection) to the Stata Center (located at another given intersection) in which he never travels more than $1000$ meters without passing through an intersection with a coffee shop. Help the professor get to lecture on time by designing an efficient algorithm to find the shortest acceptable route from his apartment to the Stata Center.

**Solution:**

**Summary:** We model the city of Cambridge as a graph, $G$, with the $m$ intersections as vertices, the $n$ roads as edges and weights on the edges representing the length of the road. We call an intersection "caffeinated" if Piotr is able to acquire coffee at the intersection. For the purposes of this problem, Cambridge has $k + 2$ caffeinated intersections (Note that coffee is available at each of the $k$ Moonbucks, as well as Piotr's apartment and the Stata center). Our goal is to find the shortest path from Piotr's apartment to the Stata center which never travels more than 1000 meters without passing through a caffeinated intersection. To do this, we create a new graph, $G'$, which has a vertex for each of the $k+2$ caffeinated. For each pair vertices $(i, j)$ in $G'$, if the shortest path between the corresponding vertices in $G$, denoted $d(i, j)$, is less than 1000 meters, then there is an edge in $G'$ between $(i, j)$ whose weight is $d(i, j)$. Since all street lengths in the city of Cambridge are non-negative, we construct the graph $G'$ by running $k+2$ executions of Djikstra's algorithm on $G$ (one execution starting at each of the $k+2$ caffeinated intersections). We then perform one more execution of Djikstra's algorithm on $G'$ to find the shortest path between Piotr's apartment and the Stata center in $G'$ (which we will show is the shortest acceptable path from Piotr's apartment).

**Algorithm:** We assume that the input to our algorithm is a graph with $m$ vertices and $n$ edges represented as adjacency lists. We also assume that the first $k$ vertices (numbered 1 through $k$) contain Moonbucks coffee shops and that vertex $k + 1$ is Piotr's apartment and vertex $k + 2$ is the Stata center. (If the vertices are not presented in this order, and we are merely given a list of vertices with coffee shops, then we can easily relabel the vertices in $O(n \log(k))$ time). We now run Djikstra's algorithm $k + 2$ times where the $i$th execution computes the single-source shortest paths from vertex $i$ to vertices $1, \ldots, k + 2$. We use a Fibonacci Heap to implement the priority queue used by Djikstra's algorithm, so the running time of each execution of Djikstra's algorithm is $O(m \log(m) + n)$. Therefore, the time for $k + 2$ executions of Djikstra's algorithm is $O(km \log(m) + kn)$. Now that we know the length of the shortest Path between all pairs of caffeinated vertices, we can construct the graph $G'$ as described in the summary above. Finally, we run Djikstra's algorithm one last time to find the shortest path from vertex $k + 1$ to vertex $k + 2$ in graph $G'$. Since graph $G'$ has that most $O(k^2)$ edges, this final step takes that most $O(k^2)$ time. Therefore, the total running time of our algorithm is $O(km \log(m) + kn + k^2)$. Since $k^2$ is $O(kn)$, this simplifies to $O(km \log(m) + kn)$.

**Correctness:** We show correctness we first argue that every path from Piotr's house (vertex $k + 1$) to the Stata center (vertex $k + 2$) in $G'$ corresponds to an acceptable path in $G$ (with the same length). We then argue that any shortest acceptable path from vertex $k + 1$ to vertex $k + 2$ in

$G$ corresponds to a path (of the same length) in $G'$. From this we conclude that by finding the shortest path from vertex $k+1$ to vertex $k+2$ in $G'$ our algorithm (1) Finds an acceptable path in the original graph $G$ and (2) Finds the shortest such path in $G$.

Consider any path $P'$ in $G'$. Each edge in this path corresponds to a path of length at most 1000 meters between caffeinated vertices in $G$. Therefore, by replacing each edge in $P'$ by the corresponding path in $G$ we can construct a path $P$ of the same length in $G$ which passes through a caffeinated vertex at least once every 1000 meters.

Now consider a shortest acceptable path $P$ in $G$ from vertex $k+1$ to $k+2$. Let $(k+1 = c_1, \ldots, c_s = k+2)$ be the caffeinated intersections that $P$ passes through (in the order that it passes through them). Note that for each $1 \leq i < s$, $P$ must travel on a shortest path from $c_i$ to $c_{i+1}$ (because every subpath of a shortest path must also be a shortest path). Additionally, for each $1 \leq i < s$, the subpath from $c_i$ to $c_{i+1}$ must be at most 1000 meters because $P$ is acceptable. Therefore, for each $1 \leq i < s$, the length of the subpath from $c_i$ to $c_{i+1}$ is the same as the weight on the edge from $c_i$ to $c_{i+1}$ in $G'$. Thus if the path $P'$ consisting of the vertices $(k+1 = c_1, \ldots, c_s = k+2)$ in $G'$ has the same length as the original path $G$.

**Grading notes:** Many students attempted to use a single execution of a greedy algorithm similar to Djikstra's algorithm find the shortest acceptable path from Piotr's apartment to the Stata center. Such algorithms were very efficient because they visited each vertex only once, but were generally incorrect. These solutions generally received around 8 points provided that they demonstrated a solid understanding of Djikstra's algorithm and traditional shortest path problems.

The principal problem with visiting each vertex only once is as follows. Assume there is a short Path from from the source to vertex $i$ that reaches vertex $i$ after having traveled 999 meters since the last coffee shop. Additionally assume that there is a longer path from the source to vertex $i$ that reaches vertex $i$ after having traveled only 100 meters since the last coffee shop. A Djikstra-like greedy algorithm will find the former path but taking the latter path may be necessary to reach the Stata center while satisfying the caffeine requirement. That is, you may want to take a longer path to vertex $i$ in order to arrive there with more caffeine in your blood stream. (Some students even observed that the optimal path might visit visit the same intersection twice. For instance, one might first walk to vertex $i$, then walk to nearby vertex $j$ to buy coffee, then back to vertex $i$ and onward to the Stata center. Clearly, any algorithm which visits each vertex only once will fail to find such a path).

Some students used Floyd-Warshall to construct $G'$ instead of using repeated executions of Djikstra's algorithm. However, since all edge weights in this problem are non-negative (Cambridge has no negative length streets) it is more efficient to use repeated applications of Djikstra's algorithm. Good solutions using Floyd-Warshall instead of Djikstra still generally received at least 20 points.

Some creative students observed that Cambridge (having relatively few overpasses) is nearly planar and hence graph $G$ is sparse. This allows for a tighter analysis of the above algorithm. If one additionally assumes that no two intersections are within 1 meter of each other then one can conclude that the number of intersections within 1000 meters of a given point is a (somewhat large) constant. This allows for a very (asymptotically) efficient algorithm. Finally, two students

observed that an asymptotically more efficient solution can be found if one assumes that all street lengths are an integer number of meters.

**Problem 3. Radio 107.9 FM**

The Eccentric Motors (EM) corporation is about to roll out their 2005 year car model when they discover that many of their automobiles have faulty radio tuners that cannot access the highest frequencies in the FM spectrum. In particular, these defective radios cannot receive radio stations broadcast at 107.9 FM. Replacing the defective radios would delay the roll out of the 2005 model, which would cost the company millions of dollars in lost sales. Fortunately, not every city has a 107.9 FM station. Therefore, EM has decided to send the defective cars only to those dealerships that are not within range of any 107.9 FM station.

Eccentric Motors has given you a list of their $n$ dealerships and has asked you to locate the dealerships that are out of range of all 107.9 FM stations. The list is organized so that dealership $i$ is $D_x[i]$ miles east and $D_y[i]$ miles north of St. Louis, Missouri. (Negative values of $D_x[i]$ and $D_y[i]$ denote miles west and south, respectively.) You have contacted the FCC, which has given you the location of the $k$ radio towers that broadcast at 107.9 FM. Tower $j$ is located $T_x[j]$ miles east and $T_y[j]$ miles north of St. Louis, Missouri, and it broadcasts at a signal strength that allows the signal to be received within a radius of $T_r[j]$ miles from the tower. FCC regulations guarantee that no point is able to receive signals from two different 107.9 FM broadcast towers.

Design an efficient algorithm to locate the EM dealerships that are out of range of all 107.9 FM broadcast towers.

**Solution:**

Since the problems involves points and disks (that is, dealerships and tower ranges), it is a computational geometry problem. The most natural geometric technique that can be used for this problem is, the *sweep-line approach*. This technique was introduced in lecture to find segment intersection using a line-sweep approach. This algorithm runs in $O((k+n)lg(k+n))$ time. In this method, the data is swept from left to right (West to East) by a vertical[4] line. The sweep line keeps a data structure that maintains some information about objects (i.e., disks) that intersect the line. When a sweep-line hits a point, the data structure should be able to answer (quickly) if this point belongs to one of the disks intersected by the sweep line, or not. This suffices, since if a point $(x, y)$ belongs to a disk $D$, then the sweep line positioned at $x$ must intersect $D$.

The details are as follows. First, we need to construct a trajectory, or schedule, for the sweep-line. That is, the sequence of important points the sweep-line encounters, sorted by the $x$-coordinate. The points are:

- The leftmost points of the disks, that is, the points $(T_x[j] - T_r[j], T_y[j])$
- The rightmost points of the disks, that is, the points $(T_x[j] + T_r[j], T_y[j])$

[4] A few people used a horizontal line moving top-down The main effect of this approach was confusing the graders; as such, this approach should be avoided as much as possible.

- The dealership positions $(D_x[i], D_y[i])$.

As in lecture and the book, we assume that all $x$-coordinates are different. In particular, this implies that the radiae of the towers are non-zero (and therefore, the leftmost and rightmost points of each disks do not coincide).

The data structure associated with the sweep-line is a search tree data structure $V$. The disks in the data structure are ordered by their $y$-coordinates (defined below). Again, we assume (for simplicity) that all such coordinates are distinct. The data structure supports insertions and deletion of elements, as well as search for a predecessor and successor of a given $y$. All three operations can be performed in $O(\log(n+k)$ time.

Denote the sweep-line by $L$. If $L$ is at position $x$, we denote it by $L(x)$. The data structure holds one $y$ coordinate per disk intersecting $L$. Specifically, it is the $y$ coordinate of the center of the disk. This is done because of the following nice observation. Consider any two disjoint disks $D$ and $D'$, with centers $(x, y)$ and $(x', y')$. Consider any "sweep-line position" $x$, such that $L$ intersects both $D$ and $D'$. Then, $y < y'$ if and only if all points in $L \cap D$ have $y$-coordinates lower than all points in $L \cap D'$.

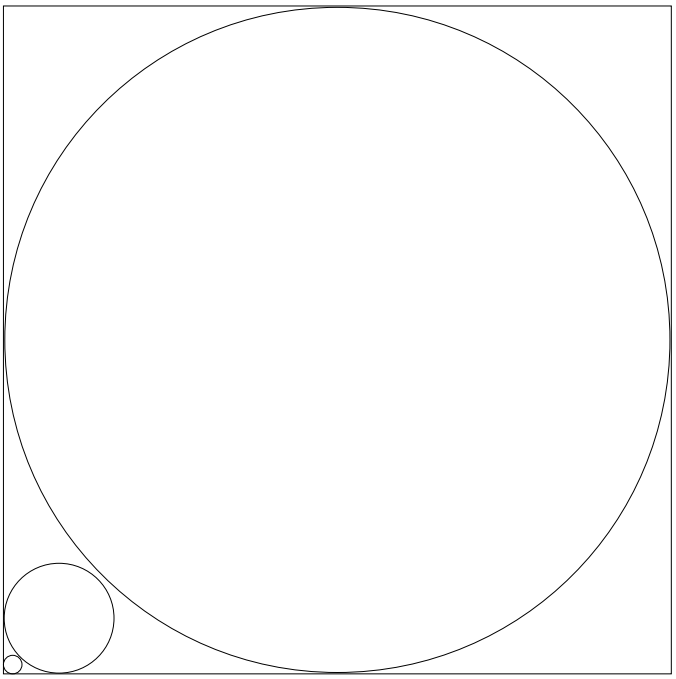
Given the above, each new point $p = (x, y)$ encountered by the sweep-line is handled as follows:

- If $p$ is the leftmost point of a disk $D$, add $D$ (with a key $y$) to $V$
- If $p$ is the rightmost point of a disk $D$, remove $D$ from $V$
- If $p$ is a dealership:
  - Find $D'$, centered at $(x', y')$, which is the predecessor of $y$ in $V$
  - Find $D''$, centered at $(x'', y'')$, which is the successor of $y$ in $V$.
  - Check if $p$ is contained in either $D'$ or $D''$. If yes, report it as "covered". If not, report it as "not covered".

**Correctness.** Clearly, any point reported as "covered" is indeed covered. We need to show the converse. That is, if a point $p = (x, y)$ is covered by $D' = (x', y')$, then $D'$ must be either a predecessor or a successor of $y$. We do it as follows. By symmetry, assume $y' < y$. For contradiction, assume there is another disk $D''$ with center $(x'', y'')$ where $y' < y'' < y$. Clearly, $D''$ does not contain $p$, since there can be only one such disk. Then, consider the sweep line $L = L(x)$ (at position $x$). From the earlier observation (applied to $D''$ and a point $p$), it follows that all points in $D'' \cap L(x)$ precede $p$ (w.r.t. $y$ coordinate). In the same way, we get that all points in $D' \cap L(x)$ precede all points in $D''(x) \cap L(x)$. But, this implies that all points in $D' \cap L(x)$ precede $p$ - a contradiction.

**Running time.** Sorting takes $O((n+k)\log(n+k))$ time. Processing each event takes $O(\log(n+k))$ time. Since there are $O(n+k)$ events, we get a total time of $O((n+k)\log(n+k))$.

**Variations.** Quite a few people gave the brute force algorithm checking every tower with every dealership with running time $O(n.k)$. A few people observed that the problem becomes easier, if the disks are replaced by (smallest) squares (or $t$-gons, for small $t$) containing the disks. In



particular, for the case of squares, the interval formed by intersecting $L(x)$ with a square is always the same, i.e., does not depend on $x$; thus, one can use interval trees to maintain the intersections. This introduces false "coverings" - a point can belong to the square but not to the disk. The solutions to this problem can be classified into the following categories:

- Claiming that a square (or a $t$-gon) is a "good enough" approximation of a disk. This might be true in some cases, but is not true in general. [5] In any case, the problem statement assumed exact solution.
- Whenever a square $S$ covers a point $p$, explicitly checking if $p$ is covered by the disk $D \subset S$. This gives a correct algorithm. However, its running time could be as big as $\Omega(n^2)$. This can happen since the fact that two disks do not intersect, does NOT imply that their squares do not intersect. See example in the figure **??**

**Other variations** included using grids, buckets or hashing. These approaches have the same problems as before: approximation or potentially large running time, which depends on the *spread* of the points - the ratio of largest to smallest distances. A few people gave an efficient, and exact divide and conquer algorithm.

### Problem 4. Isolating Kryptonite

Professor Luthor has obtained a meteorite fragment that contains high levels of a useful Kryptonite isotope. The fragment is a perfect cube $n$ centimeters on each side. The professor would like to

[5] However, this approach sometimes leads to very efficient *approximation* algorithms. E.g., see slide 6, Lecture 20 in `http://theory.csail.mit.edu/~indyk/6.838/ .`

distribute this fragment to $m$ secret laboratories for study in the hopes that the Kryptonite isotope can be synthesized in large quantities. The Kryptonite is not evenly distributed throughout the meteorite, however. The density of the Kryptonite at coordinate $(i, j, k)$ is given by $D(i, j, k)$, that is, $D(i, j, k)$ gives the amount of isotope in a cubic centimeter with origin $(i, j, k)$.

Professor Luthor has obtained access to a machine that will cut across the meteorite fragment at a given position, thereby dividing it into exactly two pieces. The machine only cuts at right angles (parallel to the $x$-, $y$-, or $z$-axis) and on centimeter boundaries. Thus, each cut results in two rectangular parallelepipeds with integral dimensions.

Design an efficient algorithm for dividing the meteorite into $m$ pieces so as to maximize the minimum amount of Kryptonite contained in any piece. (Partial credit will be given for solving the analogous two-dimensional problem.)

*Clarification:* The machine can only be used to cut a single piece into two smaller pieces. That is, after cutting the meteorite into two pieces, you must separate the pieces; you can then perform different cuts on each piece. Also, you cannot glue pieces back together.

**Solution:**

This problem can be solved using dynamic programming in $O(m^2n^7)$ time. The idea is to solve sub-problems of the following form: given a rectangular parallelepiped sub-section (or "piece") of the meteorite, how can it be divided into $k$ smaller pieces so as to maximize the minimum amount of Kryptonite in any piece? We demonstrate that the problem exhibits optimal substructure, allowing each of $O(n^6)$ sub-problems to be solved in $O(mn)$ time. We also give special attention to the base case ($k = 1$), yielding an $O(1)$ algorithm for calculating the amount of Kryptonite in a given piece as a function of the amounts in smaller pieces.

Many students developed a greedy algorithm for this problem. A greedy approach does not result in the optimal series of cuts; refer to the end of this discussion for counter-examples and grading guidelines.

*Terminology*

A ***piece*** $p = (x_1, y_1, z_1, x_2, y_2, z_2)$ is a rectangular parallelepiped sub-section of the meteorite with opposite corners at coordinates $(x_1, y_1, z_1)$ and $(x_2, y_2, z_2)$, where $0 \le x_1 < x_2 \le n$, $0 \le y_1 < y_2 \le n$, and $0 \le z_1 < z_2 \le n$. All coordinates are integers.

*Optimal Substructure*

Consider the problem of dividing a piece $p$ into $k$ smaller pieces. Because each cut of the machine divides a piece into exactly two smaller pieces, the optimal solution must start by cutting the piece $p$ into two pieces $p_1$ and $p_2$ (unless $k = 1$, in which case the problem is trivial). Consider that the optimal solution further divides $p_1$ into $j$ pieces and $p_2$ into $k - j$ pieces. Let $P$ denote the problem of dividing $p$ into $k$ pieces, $P_1$ denote the sub-problem of dividing $p_1$ into $j$ pieces, and $P_2$ denote the sub-problem of dividing $p_2$ into $k - j$ pieces.

The optimal substructure is as follows: the value of the overall solution for $P$ can never be improved by substituting a non-optimal solution for sub-problem $P_1$ or $P_2$ (or both). The proof is by contradiction. Suppose that the overall solution for $P$ did improve after changing the solution of $P_1$ from $\text{OPT}(P_1)$ (the optimal solution) to $\text{NON-OPT}(P_1)$ (a non-optimal solution). In other words, the minimum amount of Kryptonite over all pieces in $P$ has increased; since we did not change the solution of $P_2$, the minimum piece in $\text{NON-OPT}(P_1)$ must be larger than the minimum piece in $\text{OPT}(P_1)$. This contradicts the optimality of $\text{OPT}(P_1)$. The same argument applies to $P_2$. For the case in which both $P_1$ and $P_2$ are non-optimal, one can change the solution to $P_1$ and then the solution to $P_2$; neither change improves the overall solution.

Note that this argument is somewhat tricky because it is *not* the case that both sub-problems *must* be solved optimally in the overall optimal solution. In particular, the sub-problem that does not contain the piece with the minimum amount of Kryptonite can adopt a non-optimal solution, so long as the overall minimum is not affected. Nonetheless, the optimal substructure property shown above is enough to guarantee the correctness of our algorithm, as it shows that optimal solutions to sub-problems will always result in an optimal overall solution (i.e., one of the solutions that maximizes the minimum amount of Kryptonite in the pieces).

*Dynamic Programming Formulation*

The optimal substructure exposes the following sub-problems: for every piece $p$ in the meteorite, and for every $k \in \{1, 2, \ldots, m\}$, what is the best way to divide $p$ into $k$ pieces? The solution is comprised of optimal sub-solutions once we determine two things: 1) the best position to cut $p$ into two pieces, and 2) the best allocation of the $k$ desired pieces between the two sides of the cut. We determine these values by exhaustive search: for a given sub-problem, try every cut position and every allocation of pieces to the two sides, keeping track of the configuration that maximizes the minimum value of Kryptonite.

Let $A[p, k]$ denote the maximum value of the minimum-Kryptonite piece across all ways of dividing $p$ into $k$ pieces. Expanding $A[p, k] = A[x_1, y_1, z_1, x_2, y_2, z_2, k]$, we observe that the overall answer to the problem will be given by $A[0, 0, 0, n, n, n, m]$. Each sub-problem can be evaluated

as follows:

$$A[x_1, y_1, z_1, x_2, y_2, z_2, k] = \begin{cases} \text{if } k = 1: \quad \textsc{Sum-Kryptonite}(x_1, y_1, z_1, x_2, y_2, z_2) \\ \text{if } k > (x_2 - x_1)(y_2 - y_1)(z_2 - z_1): \quad -\infty \\ \text{otherwise:} \\ \quad \max_{j \in [1, k-1]} \begin{cases} \max_{c \in [x_1+1, x_2-1]} \min \begin{cases} A[x_1, y_1, z_1, c, y_2, z_2, j] \\ A[c, y_1, z_1, x_2, y_2, z_2, k-j] \end{cases} \\ \max_{c \in [y_1+1, y_2-1]} \min \begin{cases} A[x_1, y_1, z_1, x_2, c, z_2, j], \\ A[x_1, c, z_1, x_2, y_2, z_2, k-j] \end{cases} \\ \max_{c \in [z_1+1, z_2-1]} \min \begin{cases} A[x_1, y_1, z_1, x_2, y_2, c, j], \\ A[x_1, y_1, c, x_2, y_2, z_2, k-j] \end{cases} \end{cases} \end{cases}$$

The above equation has three cases. In the first case, $k = 1$ and we sum the Kryptonite contained in the piece; we use a special procedure to do the sum efficiently (see below). In the second case, it is impossible to obtain the desired number of pieces because the volume of the parallelepiped is too small. Thus, we return negative infinity to ensure that this configuration will never be selected as the optimal (largest) value.

In the third case, we evaluate all possible cuts that could be performed on the parallelepiped. The variable $j$ ranges over possible allocations of pieces between one side of the cut and the other. The variable $c$ ranges over possible locations of the cut on the $x$-, $y$-, and $z$-axis. For each cut, we evaluate the minimum amount of Kryptonite in the two resulting pieces. We select the value where this minimum is maximized.

The SUM-KRYPTONITE procedure calculates the total amount of Kryptonite contained in the parallelepiped. Instead of summing over all locations (which would require $O(n^3)$ time), it sums over two smaller parallelepipeds (which requires only $O(1)$ time if the sub-problems have already been solved).

SUM-KRYPTONITE$(x_1, y_1, z_1, x_2, y_2, z_2)$
1 **if** $x_1 \neq x_2 - 1$ **then return** $A[x_1, y_1, z_1, x_2 - 1, y_2, z_2, 1] + A[x_2, y_1, z_1, x_2, y_2, z_2, 1]$
2 **elseif** $y_1 \neq y_2 - 1$ **then return** $A[x_1, y_1, z_1, x_2, y_2 - 1, z_2, 1] + A[x_1, y_2, z_1, x_2, y_2, z_2, 1]$
3 **elseif** $z_1 \neq z_2 - 1$ **then return** $A[x_1, y_1, z_1, x_2, y_2, z_2 - 1, 1] + A[x_1, y_1, z_2, x_2, y_2, z_2, 1]$
4 **else return** $D[x_1, y_1, z_1]$

The SUM-KRYPTONITE procedure first tries to split the piece along the $x$-axis. However, if the piece is only 1 cm wide, then it tries to split along the $y$-axis, and subsequently the $z$-axis. If the piece is a single cubic centimeter, then it returns the density as given by $D$.

*Order of Evaluation*

A bottom-up dynamic programming algorithm solves sub-problems in a specific order to guarantee that solutions are ready before they are needed by other sub-problems. In our case, each sub-problem corresponds to an element of $A$, and we solve it by evaluating the equation for $A$ given previously. There are several legal orderings. We consider pieces in order of their size; for example, we first evaluate all $1 \times 1 \times 1$ pieces, then all $1 \times 1 \times 2$ pieces, then all $1 \times 1 \times 3$ pieces, and so on. Our toplevel algorithm is as follows:

FIND-CUTS$(n, m, D)$
1 Initialize all entries of $A$ to $-\infty$
2 **for** $\Delta x \leftarrow 1$ **to** $n$
3 **do for** $\Delta y \leftarrow 1$ **to** $n$
4 **do for** $\Delta z \leftarrow 1$ **to** $n$
5 **do for** $x \leftarrow 0$ **to** $n - \Delta x$
6 **do for** $y \leftarrow 0$ **to** $n - \Delta y$
7 **do for** $z \leftarrow 0$ **to** $n - \Delta z$
8 **do for** $k \leftarrow 1$ **to** $m$
9 **do** evaluate $A[x, y, z, x + \Delta x, y + \Delta y, z + \Delta z, k]$

This iteration order is correct because, in SUM-KRYPTONITE and the equation for $A$, a given piece is evaluated only in terms of strictly smaller pieces, i.e., in terms of pieces which have lesser or equal width, height, and depth. Because the loops on $\Delta x$, $\Delta y$, and $\Delta z$ correspond to the width, height, and depth, respectively, they impose a lexicographic ordering and guarantee that strictly smaller pieces are evaluated before strictly larger pieces.

The iteration order, in combination with the optimal substructure presented previously, forms the basis of the algorithm's correctness. Every piece is evaluated for the best cut and the global optimum is selected.

*Constructing the Cuts*

The value of the optimal solution is given by $A[0, 0, 0, n, n, n, m]$. To reconstruct the actual sequence of cuts corresponding to this solution, one can maintain an auxiliary array $B$ (with the

same dimensions as $A$) that tracks the values of $j$ and $c$ corresponding to the maximum for a given sub-problem. One can trace back through the $B$ matrix in the standard fashion to construct the cuts.

*Runtime Analysis*

There are $O(mn^6)$ distinct sub-problems, which is evident from the dimensions of $A$ (as well as the nested loops in FIND-CUTS). For each sub-problem, we have to evaluate an element of $A$. In the base case, this evaluation is $O(1)$, as SUM-KRYPTONITE performs a constant amount of work.

However, the general case requires more work. First, we vary the allocation of cuts between pieces from 1 to $m - 1$; this requires $O(m)$ iterations. On each iteration, we evaluate all possible cut positions on the $x$-, $y$-, and $z$-axis. Since there are three dimensions with at most $n$ positions on each, we consider $O(3n) = O(n)$ cuts per allocation of pieces. Thus, the total amount of time to evaluate a sub-problem is $O(mn)$.

There are $O(mn^6)$ sub-problems and each requires $O(mn)$ to solve, so the total runtime is $O(m^2n^7)$.

*A Faster Algorithm!*

A few students achieved a runtime of $O(m^2n^6 \lg n)$ by improving the search for the best cut position. They observed that the maximum value is a "mountain function" with respect to the position of the cut: if moving the cut in one direction worsens the score, then moving it further in that direction is fruitless, as the score will only continue to go downhill.

Thus, rather than evaluating $O(n)$ cut positions for a given sub-problem, one can do a binary search on intervals of cuts and reduce the runtime to $O(\lg n)$. Four points within each interval are sampled; if all scores are distinct, then three of these points (including an endpoint) must form a monotonic slope and the endpoint can be eliminated. The algorithm recurses on the interval spanned by the remaining three points, maintaining the invariant that the optimal score (the "peak" of the mountain) is always contained within the current interval. The runtime of the search is governed by the recurrence $T(n) = T(3n/4) + O(1)$; by the Master Theorem, $T(n) = O(\lg n)$.

However, it appears that this algorithm requires the scores to be distinct. For instance, if all scores except for one are equal, then the algorithm must resort to exhaustive search. Nonetheless, there are many cases in which this algorithm would perform better than the one detailed previously.

*Greedy Approaches*

More than one-quarter of the class took a greedy approach to this problem. A common solution was a top-down algorithm that finds the most Kryptonite-heavy piece and cuts it into two pieces so as to minimize the difference in Kryptonite between sides of the cut. While this algorithm is efficient and might serve as a good heuristic, it does not guarantee an optimal division of the meteorite. A counter-example is shown in Figure 1.

About 5 students suggested a bottom-up greedy algorithm, in which each cubic centimeter of the meteorite is initially considered as a distinct piece. At every step of the algorithm, the piece with the minimum amount of Kryptonite is fused with its lowest-Kryptonite neighbor (assuming some

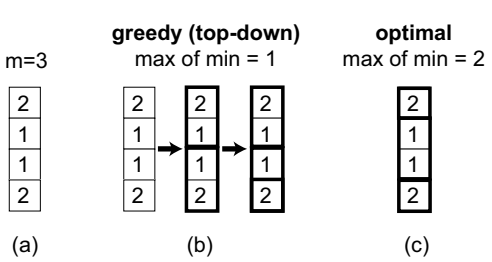



**Figure 1**: Counter-example for top-down greedy strategy. Given the input in (a) with m=3, the greedy algorithm produces the solution in (b) while the optimal solution is shown in (c).

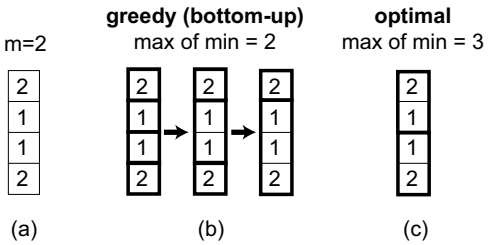



**Figure 2**: Counter-example for bottom-up greedy strategy. Given the input in (a) with m=2, the greedy algorithm produces the solution in (b) while the optimal solution is shown in (c).

geometrical constraints are satisfied). A counter-example illustrating the deficiency of this strategy appears in Figure 2.

Other variants on greedy algorithms included 1) searching for a cut where the amount of Kryptonite on a side can be expressed as $k/m$, where $k$ is an integer, 2) keeping the cut points constant but varying which piece is recursively cut, and 3) trying to shave off $1/m$ of the meteorite with each cut. All of these strategies have similar failure cases. In all, a greedy strategy fails because it does not consider all of the possibilities. A local choice might separate (resp. combine) two pieces that need to be combined (resp. separated) for a cascade of cuts to offer further benefits on the other side of the meteorite.

*Grading Notes*

There were several brute-force solutions that were correct but ran in exponential time. They received at most 7 points. A smaller number of solutions were on the right track algorithmically but the analysis showed exponential runtime. These received at most 12 points.

Many solutions offered a greedy algorithm, which received at most 10 points. Extra points were awarded for acknowledging that greedy is sub-optimal, or for using simple dynamic programming to keep track of partial sums. Solutions with incomplete or incorrect runtime analysis received lower scores.

Dynamic programming algorithms that were buggy or incomplete received at most 18 points. More than a quarter of the class designed the same algorithm as described here, and were generally awarded 19-25 points.

Several solutions neglected to factor in the cost of solving the sub-problems; this resulted in a score around 21 points. Also, many students used a brute-force algorithm to sum the total Kryptonite content in a given parallelepiped, thereby leading to an $O(n^9)$ runtime bottleneck that was usually unaccounted for. Solutions missing an analysis of the base case received at most 23 points.

**Problem 5. Finding Coolmail Users**

Macrohard Corporation has decided to start a free email service known as Coolmail. Coolmail users choose their favorite $k$-digit number $x$ as their user ID and get the email address $x$`@coolmail.com`. For instance, with $k = 9$ a user might chose 314159265 as her user ID and be given the address `314159265@coolmail.com`. Since many people find it hard to remember $k$-digit numbers, Coolmail provides the following helpful service. Whenever an email is sent to an address $y$`@coolmail.com`, where $y$ is not a valid user ID, Coolmail finds a nearest valid user ID $x$, that is, an $x$ that minimizes $|x-y|$, and sends the reply, "Sorry, but $y$`@coolmail.com` is not a valid address. Did you mean to type $x$`@coolmail.com`?"

It has come to the attention of the opportunistic Professor Ralsky that many Coolmail users are paying too much interest on their mortgages. He would therefore like to send every Coolmail user a message informing him or her that "You can ref1nance your m0rtgage at the amazing rate of only 3.49%!!" Unfortunately, the professor does not know the email addresses of all Coolmail users. Let $0 \leq x_1 < \cdots < x_n < 10^k$ be the user ID's of the $n$ Coolmail users, where $n$ is unknown to the professor. He has hired you as an MIT student intern to design an efficient algorithm to compute, with the help of Coolmail's automated reply service, the email addresses of all the Coolmail users. Minimize the worst-case number of emails required by your algorithm.

**Solution:**

We will assume that $n \geq 1$; the helpful email service doesn't make sense if there aren't any users. We also assume that helpful email responses from Coolmail arrive in a bounded amount of time; otherwise we'll never know if an email that we've sent was delivered successfully.

In the interest of clarity, we are not going to worry about constant factors in the number of emails sent. (We're even going to actually send more than one email to the same address.) The idea of our solution is to maintain a set of *non-empty regions* of identity space that are guaranteed to contain at least one user. Given a non-empty region, we'll query the region by sending an email to the median(s) to find a new user, and then update the set of regions on the basis of the query result. This will result in an algorithm that computes a list of all valid addresses in $\Theta(n)$ time, and sends $\Theta(n)$ emails.

Let $\text{NEAREST}(x)$ be a function that returns a nearest valid address to $x$, for a half-integral value $x$ (that is, $2x \in \mathbb{Z}$). If $x$ is an integer, then the function emails $x$, waits long enough for a helpful response to be returned, and then returns $x$ if the email was delivered successfully, and $y$ if the response "did you mean $y$@coolmail?" was received. (Note that if there is a tie—i.e., there are two values $\underline{y}$ and $\overline{y}$ that are equidistant from $x$—then $\text{NEAREST}(x)$ will break the tie arbitrarily.) If $x$ is non-integral, then the function returns whichever of $\text{NEAREST}(x-0.5)$ and $\text{NEAREST}(x+0.5)$ is closer to $x$, or one arbitrarily if there is a tie.

Let $\text{NONEMPTY?}(a, b)$ be a boolean function taking two values $a < b$ that returns true if and only if there is some valid address $x$ such that $a \leq x \leq b$. This function does the following: on input $a, b$, let $y := \text{NEAREST}((a+b)/2)$. Return true if and only if $y \in [a, b]$. Correctness of NONEMPTY? is immediate by inspection.

Finally, let PUSH-IF-NON-EMPTY$(a, b, S)$ be a function that, given two indices $a$ and $b$ and a stack $S$, pushes $[a, b]$ onto $S$ if NONEMPTY?$(a, b)$.

```
V := empty list.                              // V will hold all valid addresses.
R := empty stack; push (0,10^k-1) onto R.     // R holds non-empty regions.

while (R is non-empty)
{
  [a,b] := pop(R);
  median := (a+b)/2;
  valid := NEAREST(median);

  add valid to V.

  if valid < median
    push-if-non-empty(a,valid-1,R)
    push-if-non-empty(2*median-valid,b,R)
  elseif valid == median
    push-if-non-empty(a,median-1,R)
    push-if-non-empty(median+1,b,R)
  else (valid > median)
    push-if-non-empty(a,2*median-valid,R)
    push-if-non-empty(valid+1,b,R)
}
return V.
```

**Correctness.** We first note that, for a non-empty region $[a, b]$, the value of NEAREST$((a + b)/2)$ must be a valid email address in $[a, b]$. This ensures that every range $[a, b]$ that we attempt to push onto the stack has $a \leq b$.

We claim the following loop invariant: all valid addresses are contained either in $V$ or in a region in $R$, all regions in $V$ are disjoint from each other and from $V$, and no element appears twice in $V$. This invariant is satisfied at initialization by assumption. We now show that the invariant is maintained by an iteration of the loop. First note that in the case when $valid == median$, the invariant is trivially maintained. For the other cases, we prove maintenance for $valid < median$; the final case is analogous. Observe that in this case the effect on $R \cup V$ is (i) possibly to delete regions that are empty (according to PUSH-IF-NON-EMPTY, which does not falsify the invariant by the correctness of NONEMPTY?), and (ii) to delete the range $(valid + 1, 2 \cdot median - valid - 1)$. But (ii) cannot falsify the invariant either: since $valid$ is the closest valid address to $median$, there cannot be any valid addresses closer to $median$ than $valid$. Note that the resulting regions are disjoint subregions of $[a, b]$, and are also disjoint from $valid$. This fact implies that the invariant is maintained.

For termination, note that the number of addresses covered by $R$ is decreasing by at least one in every iteration. Thus the loop eventually terminates. When it does, $R$ is empty, and by the above loop invariant, we have that $V$ contains all valid email addresses. Furthermore, since we only add

an address returned by NEAREST(), the resulting $V$ contains only valid email addresses. Since all elements of $V$ are unique, the list $V$ contains exactly the valid addresses.

**Running time.** First, observe that NEAREST is the only place where our algorithm sends email.

Now, note than an iteration of the loop makes one call to NEAREST and two to PUSH-IF-NON-EMPTY (and thus two to NONEMPTY?, and thus two more to NEAREST). Thus each iteration of the loop causes three calls to NEAREST, each of which may result in two emails being sent. Thus there are at most six emails sent per loop iteration.

Since in each iteration, we find exactly one a new address to add to $V$, there are precisely $n$ iterations. Thus the total number of emails sent is $\Theta(n)$. Furthermore, since everything in the loop is a constant-time operation, the total running time is $\Theta(n)$ as well.

**Grading notes.** The general breakdown of grading was that about half of the credit was given for getting a correct algorithm that ran in $\Theta(n)$ time, and half of the credit was given for correctness proofs and running-time analysis.

A lot of people came up with the correct $\Theta(n)$-time algorithms (similar to the one above), but people had significantly more trouble with rigorous proofs of correctness and running time. For example, in the divide-and-conquer recursive version of the above algorithm, you make recursive calls on two subintervals of size at most half of the size of the original interval. Since $T(n) \leq 2T(n/2) + 1 = O(n)$, people said that they has an algorithm linear in the number of users. This is totally wrong: the true recurrence is in terms of the *size of the interval*, not in terms of the *number of users*. If done correctly, this analysis yields a $O(10^k)$-time running time.

Scores for the running-time analysis (6.5 points) range from zero for no attempt whatsoever, to about 2–3 points for a hand-wavy solution that asserted facts about why the algorithm was efficient but didn't prove any of them, through 6 points for a rigorous, correct proof.

Scores for correctness (6.5 points) were similar. Almost all submitted solutions required a rigorous inductive proof, and very few gave one. Significant credit (about four points, depending on the details provided) was given for people who argued for correctness without using induction.