Problem 1. Faulty Turbines

You live in Windyland where the winds are always blowing at 50 miles an hour. To harness the energy of this windy bonanza, Windyland has laid out \( N^2 \) windturbines on an \( N \times N \) grid – that is, for each \( 1 \leq i \leq N, 1 \leq j \leq N \), a turbine is placed at location \((i, j)\). Unfortunately, some \( m \) (very few compared to \( N \)) of these turbines are faulty. When the wind passes through at 50 miles per hour, a good turbine generates 1 mega-joule per second, while a faulty turbine generates only a \( \frac{1}{2} \) mega-joule per second. Windyland is trying to determine the location of the faulty turbines. To help this quest, they have a built-in test mechanism \( \text{TEST}(i, j) \) that tells them the total amount of energy generated by the turbines in the set \([1, \ldots, i] \times [1, \ldots, j]\), where they get to specify \( i, j \in [1, \ldots, N] \). However every \( \text{TEST} \) requires them to shut down the entire grid for an hour and is thus extremely expensive to run.

Give an algorithm that Windyland could use to find all the faulty turbines using as few \( \text{TESTs} \) as possible. Specify the running time of your algorithm as a function of \( m \) and \( N \). Your algorithm should be efficient when \( m \ll N \).

Solution: Executive Summary. Given an \( N \times N \) matrix of 0/1’s with \( m \) ones, we try to find the ones efficiently. To solve the problem, we use binary search (more abstractly, divide-and-conquer). We obtain an algorithm that does \( O(m \log \frac{n^2}{m}) \) tests and has the same running time.

Detailed solution. First we design a procedure \( \text{TEST-SUBRECT}(i_1, j_1, i_2, j_2) \), \( i_1 \leq i_2 \) and \( j_1 \leq j_2 \), that returns the amount of power generated by the turbines in the rectangle \( (i_1, j_1, i_2, j_2) \). This is done as follows:

\[
\text{TEST-SUBRECT}(i_1, j_1, i_2, j_2) = \text{TEST}(i_2, j_2) - \text{TEST}(i_1 - 1, j_2) - \text{TEST}(i_2, j_1 - 1) + \text{TEST}(i_1 - 1, j_1 - 1).
\]

(This is easy verifiable by a picture.)

Next imagine an abstract tree on the grid, which is in fact a quad-tree. The quad-tree is constructed as follows. Suppose, wlog, \( n \) is a power of 2. Divide the grid into 4 squares (of side-length \( n/2 \)). Then subdivide each square into another 4 squares (of side-length \( n/4 \)), and so forth. Now the quad-tree has as its root the entire grid, and it has 4 children, corresponding to the squares of side-length \( n/2 \). These squares also have 4 children each, each of side-length \( n/4 \), and so forth. At the leaf level, we have squares of side-length 1, i.e., they correspond to turbines.

The algorithm is now simple: design a procedure \( \text{SEARCH}(\text{node}) \) that returns all faulty turbines in the square corresponding to \( \text{node} \). For this, test each of the 4 subtrees of \( \text{node} \) if they have faulty turbines (using \( \text{TEST-SUBRECT} \)) and recurse into the subtrees that have.

To see the correctness and running time of this algorithm consider the following argument. Paint black all nodes that have faulty turbines inside. Then, the black nodes are exactly the union of the
paths from all faulty turbines to the root. There are thus $m \log n$ black nodes (the depth of the tree is $\log n$). We perform tests on the black nodes and their children only.

A more attentive counting of the black nodes actually gives a bound of $O(m \log \frac{n}{\sqrt{m}})$. The reason is that previously we overcounted the black nodes, especially near the root. The number of black nodes is maximized when all the first $\log_4 m$ level of the quad tree are black, and, in the rest, we have $m$ disjoint black paths. In this case the number of black nodes is $m + m(\log n - \log_4 m) = O(m \log \frac{n^2}{m})$.

Thus the running time is $O(m \log n^2/m)$. Consequently, we can at most the same number of tests.

One can prove that the above number of tests is optimal. An easy bound of $\Omega(\frac{m \log(n^2/m)}{\log m})$ can be obtained as follows. One can do the same lower bound for this problem as for the sorting: we draw a tree of queries, with each node having $d+1$ children (depending on the answer of faulty turbines, which is a number in $\{0, \ldots, d\}$). Then, there are $\left(\frac{n^2}{m}\right) = 2^{O(m \log n^2/m)}$ total leaves, and the lower bound is the log of that, divided by the log of the degree, $m + 1$. To obtain the optimal lower bound, partition the array into $n^2/m$ parts, each with 1 faulty turbine, and prove that to uncover the turbine in each part takes $\Omega(\log n^2/m)$, and that in total one needs $m$ times that (note that this is not an immediate corollary).

**Grading (out of 30 points).** Most students gave the right idea of divide-and-conquer by partitioning in subrectangles and recursing into non-empty ones. Another option was to find the columns with faulty turbines and to find faulty turbines inside the columns (both with binary search). We gave full credit for $O(m \log n)$ solutions.

Most students lost 7 points for not giving complete analysis of the running time. The following argument was insufficient: “to find one fault, we need $O(\log n)$ time (because of binary search); and so, for $m$ faults, we need $m$ times that”. This is not an immediate implication, and one needed to justify this step (e.g., see above). Also, many students lost 1-4 points for either not specifying the computational running time (besides the number of tests performed) or having a computationally inefficient algorithm. Note that this was the requirement for all problems (as stated in the preambule). Otherwise, points were deducted for imprecise description of the algorithm (e.g., for not showing how to compute the number of faults in a general subrectangle). Solutions with $\Omega(N)$ tests received at most 15 points.

**Problem 2. Tax Status**

You live in a community of $n$ people who have approached you for help with a tax question. They would like to find out how many of them will have to pay taxes, how many would receive tax credits (in this hypothetical example we assume that the IRS does give money to people with sufficiently low income), and how many will neither owe taxes nor receive any credit.

The amount that someone pay in taxes is a monotone non-decreasing function $f(x)$ of their income $x$ (so $f(x) > f(y)$ if $x > y$). The IRS has provided you with software to compute $f(x)$, but in
their characteristic style (showing lack of proper 6.046 training), this software is extremely slow and takes \(\sqrt{n}\) time to compute the \(f(x)\) on any single input \(x\).

You have available to you the incomes of all people in your community in the form of an unsorted array \(X[1..n]\), where \(X[i]\) is the income of the \(i\)th person. Give an efficient algorithm to compute an array \(Y[1..n]\) where \(Y[i]\) indicates the tax status of person \(i\), i.e., \(Y[i] = +\) if the \(i\)th person owes taxes (i.e., \(f(x) > 0\)), \(Y[i] = -\) if the \(i\)th person is owed money by the IRS, and \(Y[i] = 0\) if the \(i\)th person neither owes taxes, nor is owed money by the IRS. Analyze the running time of your algorithm as a function of \(n\).

You may assume all incomes are distinct.

**Solution: Executive Summary.** Search for the two “boundary” values of \(X\) in the community \(k, j\), where \(k\) is the index of the person who makes the most money but is still owed money by the IRS, and \(j\) is the index of the person who makes the most money of those that neither owe taxes nor are owed money by the IRS. Then a linear scan through \(X\) allows you to determine the values of \(Y\). In order to find the boundaries, find a query to \(X\), using the linear time median algorithm, which allows you to successively rule out half of the remaining elements from consideration as possible candidates for the boundary.

**Detailed Solution.**

In order to find the value of \(k\) (finding the value of \(j\) is analogous), the plan is to find a single query that allows you to rule out half of the indices from consideration. Find the median \(m\) of the \(X[i]\)'s using the linear time median algorithm. Query \(f\) on \(X[m]\). If \(X[m] \geq 0\), then construct a new list which contains only those people for which \(X[i] \leq X[m]\) (by the monotonicity of \(f\), the indices that have been thrown out must have \(X\) values which are \(\geq X[m] \geq 0\), and therefore cannot be the boundary), otherwise, construct a new list which contains only those indices for which \(X[i] > X[m]\) (again, by the monotonicity of \(f\), the indices that have been thrown out must have \(X\) values which are \(\leq X[m] < 0\) and therefore cannot be the boundary). Repeating this \(O(\log n)\) times brings you down to a list of constant size.

Once \(k, j\) have been found, scan through the indices of \(X\) and for each \(i\), set the value of \(Y[i]\) by comparing \(X[i]\) to \(X[k]\) and \(X[j]\). That is, set \(Y[i]\) to \(-\) if \(X[i]\) is smaller than or equal to \(X[k]\), set \(Y[i]\) to \(0\) if \(X[k] < X[i] \leq X[j]\), and set \(Y[i]\) to \(+\) otherwise.

**Time analysis:** Since half of the indices are ruled out from consideration after each query, \(O(\log n)\) queries to \(f\) will be made, taking \(O(\sqrt{n} \log n)\) time. The extra time to implement the search for each boundary value is given by \(T_b(n) = T(n/2) + c \cdot n\) which gives \(T_b(n) = O(n)\). Once the two boundary values are found, the scan to set the values of \(Y\) takes linear time. The total time is \(T(n) = O(\sqrt{n} \log n + 2T_b(n) + n) = O(n)\).

**Grading (out of 30 points).** Many sorted \(X\) and then did two binary searches for the boundaries. This requires \(\theta(n \log n)\) time, which is slower. Done correctly, this solution received 20 points.

Some used randomized selection. Depending on how well the analysis was done (which is harder than the deterministic case), up to 27 points were given for such a solution.
One common mistake was to give a recurrence for $T(n)$ in which the cost of the queries to $f$ was incorporated. The problem is that the cost of a query is $\sqrt{n_0}$ for $n_0$ the original number of people in the community, and does not decrease with the level. One point was taken off for this type of mistake.

Up to three points were taken off for mishandling the the case in which several people satisfy $f(X[i]) = 0$.

**Problem 3. Repair Work**

A hurricane just hit Cambridge and wiped out all of the roads. You need to repair the roads to connect up all the public buildings as quickly as possible. Every road connects two buildings, and the time to repair the road between buildings $i$ and $j$ is $t_{ij}$, where $t_{ij}$ is an integer between 1 and 10. Let $T_{\text{upandrunning}}$ be the minimum time you need to get enough roads built to connect all the public buildings (you can only work on one road at a time – no parallelism here!).

Give an efficient algorithm to compute $T_{\text{upandrunning}}$. Assume there are $n$ buildings and $m$ roads, and that your input comes in the form of an array of $n$ adjacency lists, where the $i$th list specifies all the roads incident to building $i$, the other endpoint for each road, and the time it takes to repair the road. Express the running time of your algorithm as a function of $n$ and $m$. (The more efficient your algorithm, the better your score.)

**Solution: Executive Summary:** The problem can be restated as finding the weight of a minimum spanning tree in the underlying graph. Using that all the edge weights are small integers, we modify Prim’s algorithm to run in time $O(m)$. This is achieved by using a different implementation of the Priority Queue inside that allows Extract-min and Decrease-Key operations in $O(1)$ time.

**Detailed Solution:** Let $G = (V, E)$ be the graph obtained by letting $V$ be the set of public buildings in Cambridge and $E$ be the set of roads. Assign to every road $ij \in E$ the weight $w(ij) = t_{ij}$. Then $T_{\text{upandrunning}}$ is simply the weight of a minimum spanning tree (MST) of $G$. We can find an MST of $G$ using either Prim or Kruskal’s algorithm. In the implementation seen in lecture, the running time of Prim is $O(nT_{\text{Extract-min}} + mT_{\text{Decrease-Key}})$, where $n = |V|$, $m = |E|$, and $T_{\text{Extract-min}}$ and $T_{\text{Decrease-Key}}$ are the time needed by the respective operation in the Priority Queue, which is used to keep track of the vertices not yet covered by the tree so far. Recall that the priority of a vertex in the Priority Queue is equal to the weight of the smallest edge connecting the vertex to any vertex in the tree constructed so far (or “$\infty$” if none exists).

In this problem, we are given an additional constraint that the edge weights are small integers. Using this, we can construct a Priority Queue where both extract-min and decrease-key operations take $O(1)$. There are many ways to do this. One way is to keep 2 arrays $A$ and $B$. $A$ is an array of length 11, where, for $1 \leq i \leq 10$, $A[i]$ contains a doubly linked list containing all the vertices with priority $i$, and $A[11]$ contains a doubly linked list with the vertices with infinite priority. $B$, on the other hand, is an array of size $n = |V|$ such that for every $v \in V$, $B[v]$ has a pointer to the node corresponding to the vertex $v$ in the linked list of $A$ (or a null pointer if $v$ is not in the priority queue anymore).
To extract the minimum element of the queue, we can do the following: Find the smallest \( i \) for which \( A[i] \) is not empty. Remove the first element in the linked list in \( A[i] \) (call this element \( v \)). Then, set \( B[v] \) to be a null pointer and output \( v \). Since \( A \) is of size 11, this takes \( O(1) \) time.

To change the priority of some vertex \( v \) from \( k' \) to \( k \) in the priority queue, we simply use \( B[v] \) to locate \( v \) in its corresponding doubly linked list. Then, we remove \( v \) from the list of \( A[k'] \) (this can be done in constant time by changing the pointer of the predecessor and successor of the node in the list). Then, we insert \( v \) to the beginning of the linked list in \( A[k] \) and modify \( B[v] \) to point the corresponding node associated to \( v \). This also takes \( O(1) \) time.

Thus, both \( T_{\text{Extract-min}} \) and \( T_{\text{DecreaseKey}} \) run in constant time, and therefore, Prim’s algorithm takes time \( O(nT_{\text{Extract-min}} + mT_{\text{DecreaseKey}}) = O(n + m) = O(m) \).

**Grading comments:** Any solution achieving this running time received 30 points. Many solutions were of this flavor with some mistakes. The most common mistake was to have only array \( A \) without \( B \) to help search in the priority queue. This kind of solution did not take into account the time needed to find an element in the priority queue \( A \) in order to decrease its key. This mistake was penalized with 8 points.

There were some alternative correct solutions. The most common one was an implementation of the previous priority queue \( A \) with arrays instead of linked list and an implementation of a priority queue without decrease-key method (Simply reinsert the elements to the queue with the new (smaller) priority, and modify the extract-min method to just check if the element had been extracted before or not. A careful analysis shows that the running time of Prim in this case is still \( O(m) \)). Also, there were some solutions that used a modification of Prim in which the priority queue holds edges instead of vertices.

Any solution that applied a direct implementation of Prim using binary heaps or a direct application of Kruskal (giving a running time of \( O(m \log n) \)) received 15 points. Better implementations that used Fibonacci Heaps (with running time \( O(m + n \log n) \)) or used Counting-Sort to sort the edges as the first step in Kruskal (giving a running time \( O(m \alpha(n)) \), where \( \alpha(\cdot) \) is the inverse of the Ackermann function) received up to 20 points.

**Problem 4. Brady Bunch Marriage**

In a small farming village there lives an old man with \( n \) sons, and on the farm next to his there lives an old woman with \( n \) daughters. In fact, they are the only two families remaining on the planet after the Bubonic plague wiped out everyone else in the world. Both the man and woman would like very much to have grandchildren in order to re-populate the Earth, but none of their children are yet married. Clearly, the only way for them to have grandchildren is to intermarry their children between their two families. But before they start making matches, the old man and woman agree on the following rule:

*If son \( A \) marries daughter \( B \), then no son younger than son \( A \) may marry a daughter older than daughter \( B \), and no son older than son \( A \) may marry a daughter younger than daughter \( B \).*
This rule prevents age-crossings in marriages between the two families, which the old man and woman are afraid may lead to infidelity, tearing the two families apart (and thus endangering the future of our entire species).

In addition, village records over the last 14 generations show that the number of children a couple has is affected by the couple’s height difference—the closer in height the husband and wife are, the more kids they can expect to have! If they are more than 12 inches apart, they will not have any children. Before he died of Bubonic plague, the village statistician found the exact formula for the expected number of children that a couple with a height difference of \( d \) inches (in absolute value), would have. This quantity, denoted by \( C(d) \), is given by the following formula:

\[
C(d) = \begin{cases} 
\frac{12-d}{2} & \text{if } d < 12 \\
0 & \text{otherwise}
\end{cases}
\]  

(1)

The old man and woman would like to maximize the expected number of grandchildren their children will give them. Your task is to give an efficient algorithm (in the number of children in each family, \( n \)), to help them find the best way to intermarry their children (where best is defined as yielding the highest expected number of grandchildren). Assume that the algorithm is given as input four arrays:

- \( A[1..n] \) where \( A(i) \) denotes the age of the \( i^{th} \) son.
- \( B[1..n] \) where \( B(i) \) denotes the age of the \( i^{th} \) daughter.
- \( G[1..n] \) where \( G(i) \) denotes the height of the \( i^{th} \) son.
- \( H[1..n] \) where \( H(i) \) denotes the height of the \( i^{th} \) daughter.

Note that it may be preferable to leave some children unmarried for the social good. Be sure to prove the correctness and running time of your algorithm—the fate of humanity rests in your hands!

**Solution: Executive Summary:** We will solve this using Dynamic Programming. To find the recursive substructure, consider the oldest son. Either he can marry the oldest daughter, or he can marry one of the \( n-1 \) younger daughters, or he can not marry anyone. Each of these cases reduces to a smaller subproblem, in a matrix of \( n \times n \) subproblems. Since each problem only depends on (at most) 3 subproblems, DP will find the optimal matching in \( \Theta(n^2) \) time.

**Detailed Solution:** At first, sort sons and daughters in increasing order of their ages. It takes \( \Theta(n \log n) \) time, and now we can assume \( A \) and \( B \) are sorted. \( T \) is a \( n \times n \) matrix, where the element \( T(i, j) \) represents the highest expected number of grandchildren by considering only the youngest \( i \) sons and the youngest \( j \) daughters. (Our goal is to compute \( T(n, n) \).) To find the recursive formula for \( T(i, j) \), take the \( i^{th} \) youngest son and consider 3 possible options he can choose when the youngest \( i \) sons and the youngest \( j \) daughters are intermarried. Either he can marry the \( j^{th} \) daughter, or he can marry one of \( j-1 \) younger daughters, or he cannot marry anyone. The best way for his choice can be decided by comparing these three cases can be expressed as the
following recursion:

\[
T(i, j) = \max \left\{ \frac{12 - |G(i) - H(j)|}{2} + T(i - 1, j - 1), \quad T(i, j - 1), \quad T(i - 1, j) \right\}.
\]

Using this recursion, the table \(T\) can be filled up starting from the base case \(T(i, 0) = T(0, i) = 0\), and \(T(n, n)\) is the highest expected number of grandchildren achievable. To find the explicit marriage strategy that gives \(T(n, n)\), we maintain another \(n \times n\) matrix \(S\). By assuming \(S(0, 0) = S(0, i) = \emptyset, S(i, j)\) can be computed together with \(T(i, j)\) as follows:

\[
S(i, j) = \begin{cases} 
(i, j) \cup S(i - 1, j - 1), & \text{if } T(i, j) = \frac{12 - |G(i) - H(j)|}{2} + T(i - 1, j - 1) \\
S(i, j - 1), & \text{if } T(i, j) = T(i, j - 1) \\
S(i - 1, j), & \text{if } T(i, j) = T(i - 1, j) 
\end{cases}.
\]

\(S(n, n)\) is the best way we are looking for, and it takes \(\Theta(n^2)\) time to fill up both \(S\) and \(T\) completely. Note that to fill up \(S\) in \(\Theta(n^2)\) time we must either build the lists up using pointers to smaller lists, rather than copying the sub-list into the bigger cell, or we must store directions in \(S\): “EAST”, “SOUTHEAST”, and “SOUTH” which tell us which path to take through \(S\) after we fill in the whole matrix—following these directions will allow us to read off the path (and thus the optimal marriages) in linear time.

The total running time is \(\Theta(n^2)\) because the sorting time \(\Theta(n \log n)\) at the initial stage is dominated by \(\Theta(n^2)\).

**Grading comments:** Many students designed Dynamic Programming with the \(n \times n\) table, but used a slow recursion which took \(\Theta(n)\) time to compute. It lead to the \(\Theta(n^3)\) running time, which got at most 23 points. Several solutions were naive or brute-force solutions, which got at most 7 points if their analysis was right. Any fundamentally incorrect algorithm got at most 10 points. (Some students used a greedy scheme, but it does not work.) Not writing down the recursion or making a small mistake in the recursion was a 3 to 5 point penalty. Incorrect analysis or no analysis at all took around 5 to 7 points off. Most implementations of the right \(\Theta(n^2)\) algorithm got 28 to 30 points depending on the clarity of the writeup.

**Problem 5. Dynamic Navigation**

Recall (from Lecture 9) the nightmare that Professor Rubinfeld faces when driving to work every morning. She can either take the path from her home \(X_0 \rightarrow X_1 \rightarrow X_2 \rightarrow \cdots \rightarrow X_n\) to work \(X_{n+1}\), or she can take the path from \(X_0 \rightarrow Y_1 \rightarrow Y_2 \rightarrow \cdots \rightarrow Y_n \rightarrow X_{n+1}\), or switch, as many times as she wants, from \(X_i \rightarrow Y_{i+1}\) or \(Y_i \rightarrow Y_{i+1}\). The delay in getting from \(X_{i-1} \rightarrow X_i\) is \(a_i\), while the delay in getting from \(Y_{i-1} \rightarrow Y_i\) is \(b_i\). The switching delay from \(X_i \rightarrow Y_{i+1}\) is \(\ell_i\) and the delay in switching from \(Y_i \rightarrow X_{i+1}\) is \(u_i\). For an illustration, see figure 1.

Her goal was to get to work as quickly as possible and this is still the case. But now she has a new feature to help her cope with the nightmare. A generous non-profit institution is monitoring
Figure 1: The routes from Professor Rubinfeld’s home, $X_0$, to her work, $X_{n+1}$.

all the streets and can provide live updates on the current delays. A typical update is of the form $\text{UPDATE}(U, W, i, v)$, where $U, W \in \{X, Y\}$ and the implication is that the associated street from $U_{i-1} \rightarrow W_i$ now has a delay of $v$. These updates are beamed directly to her laptop, possibly even as she drives. Professor Rubinfeld would like to be able to execute two kinds of queries: $\text{CURRENT-TRAVEL-TIME}$ which returns the current total travel time from $X_0$ to $X_{n+1}$; and $\text{NEXT}(U, i)$ where $U \in \{X, Y\}$ which should return the next destination she should drive to to get to $X_{n+1}$ by the shortest path, if she were currently at $U_i$.

Design a data structure to initialize the data structure and handle the updates and queries efficiently. (Your score will depend on the time complexity of the initialization, the updates, the queries, as well as the space complexity of the data structure. So specify all of these in your solution.)

The following example may be illustrative. Assume that initially all horizontal delays equal 1 (i.e., $a_i = b_i = 1$) and all switches cost 2 units of delay except for $l_1 = u_{n+1} = 1$, then the following table gives an example of a sequence of updates/queries and the desired responses. An illustration of the example appears in figure 2.

<table>
<thead>
<tr>
<th>Query/Update</th>
<th>Desired Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIALIZE($a_1, \ldots, a_{n+1}, b_2, \ldots, b_n, l_1, \ldots, l_n, u_2, \ldots, u_{n+1}$)</td>
<td>ACK</td>
</tr>
<tr>
<td>CURRENT-TRAVEL-TIME</td>
<td></td>
</tr>
<tr>
<td>NEXT($X, 2$)</td>
<td>$X_3$</td>
</tr>
<tr>
<td>UPDATE($X, X, 3, 5$)</td>
<td>ACK</td>
</tr>
<tr>
<td>CURRENT-TRAVEL-TIME</td>
<td></td>
</tr>
<tr>
<td>NEXT($X, 2$)</td>
<td>$X_3$</td>
</tr>
<tr>
<td>NEXT($X, 3$)</td>
<td></td>
</tr>
<tr>
<td>UPDATE($Y, Y, 3, 5$)</td>
<td>ACK</td>
</tr>
<tr>
<td>CURRENT-TRAVEL-TIME</td>
<td></td>
</tr>
<tr>
<td>NEXT($X, 2$)</td>
<td>$Y_3$</td>
</tr>
<tr>
<td>NEXT($Y, 2$)</td>
<td>$X_3$</td>
</tr>
</tbody>
</table>

**Solution:** Executive Summary: The problem asks to maintain shortest paths in a dynamic setting. The solution involves augmenting a data structure, a simple balanced binary tree, which maintains shortest path lengths of some, but not all, intervals from $X_i/Y_i$ to $X_j/Y_j$ so that changing
**Figure 2:** An example under the update shown in the table. The first picture shows the initial configuration, the second picture shows the configuration after the operation $\text{UPDATE}(X, X, 3, 5)$, and the third picture shows the configuration after the operation $\text{UPDATE}(Y, Y, 3, 5)$. For example, after we perform $\text{UPDATE}(X, X, 3, 5)$ (second picture), it is now faster to switch from $X_2$ to $Y_3$.

any one edge length changes the path length of at most $\log n$ of the intervals that we maintain. The resulting strategy is implemented below using $O(n)$ space, so that INITIALIZE takes $O(n)$ time, UPDATE and NEXT take $O(\log n)$ time while CURRENT-TRAVEL-TIME takes $O(1)$ time.

**Detailed Solution:** We maintain a nearly full balanced static binary search tree with $n + 1$ leaves. The $i$th leaf represents the $i$th transition point from $U_{i-1}$ to $W_i$ for $U, W \in \{X, Y\}$. The tree is keyed on the index $i$ and so every internal node represents a contiguous interval (or path) from $i$ to $j$, such that its children represent the intervals $i$ to $k$ and $k$ to $j$, where $k = (i + j)/2$. A node representing the interval $i$ to $j$ maintains the shortest path length from $U_{i-1}$ to $W_j$ for $U, W \in \{X, Y\}$. Note that this is information that can be computed locally given the information for a nodes children. This will allow us to modify the tree in $O(\log n)$ time during an update.

**INITIALIZE:** We build the tree bottom up. Every node $v$, representing say the interval $[i, j]$ has four fields, $v.XX$, $v.XY$, $v.YX$ and $v.YY$ representing the shortest paths from $X_i$ to $X_j$ etc. If the node $v$ has two children $v_1$ covering the interval $[i, k]$ and $v_2$ covering the interval $[k, j]$ then
the recurrence giving the four field values for $v$ is as follows, for $U, V \in \{X, Y\}$:

$$v.UV = \min\{v_1.UX + v_2.XV, v_1.UY + v_2.YV\}.$$ 

It is clear that this algorithm runs in $O(n)$ time.

**UPDATE($U, W, i, v$):** We modify the cost of the edge $U_{i-1} \rightarrow W_i$ to $v$, and then walk up the tree from the leaf $i - 1 \rightarrow i$ to the root, updating the information at all nodes using the recurrence above. Clearly this takes $O(\log n)$ time.

**CURRENT-TRAVEL-TIME:** Simply returns the value $r.XX$ where $r$ is the root node. This takes $O(1)$ time.

**NEXT($U, i$):** As a helper routine we first design an algorithm TIME-REMAINING($U, i$) which computes the length of the shortest path from $U_i$ to $X_{n+1}$. This algorithm walks up from the leaf for $i \rightarrow i + 1$ to the root. At a node $v_1$ representing the interval $[j, k]$ with $j \leq i < k$ it maintains the information for the shortest path length from $U_i$ to $X_k$ and $U_i$ to $Y_k$. It then uses the information from sibling $v_2$ of $v_1$ (if $v_1$ is the left child) to compute this information at the parent $v$ of $v_1$. When this algorithm reaches the root, it now has the shortest path length from $U_i$ to $X_{n+1}$.

Now using TIME-REMAINING it is easy to compute the next step from $U_i$. One simply has to go to the node $W_{i+1}$ for which the edge length from $U_i$ to $W_{i+1}$ equals TIME-REMAINING($U, i$) – TIME-REMAINING($W, i + 1$). Determining this requires computing TIME-REMAINING thrice, where each invocation takes $O(\log n)$ time. Thus NEXT takes $O(\log n)$ time.

**Grading comments:** Unfortunately, a large number of solutions simply recomputed all path lengths, either during UPDATE, or during the queries. Such solutions are roughly analogous to the use of arrays (sorted or unsorted) to maintain dynamic sets — either the update, or the query, takes linear time. Such solutions got somewhere between 5 to 10 points. Some solutions additionally pointed out the possibility of doing better using balanced trees, without being able to give the algorithm. For noticing the possibility that one can do better, the writeups got up to 5 more points. Most implementations of the above algorithm got 25-30 points depending on the clarity of the writeup.