Quiz 2 Solutions

<table>
<thead>
<tr>
<th>mean</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>median</td>
<td>50</td>
</tr>
<tr>
<td>std. dev</td>
<td>14</td>
</tr>
<tr>
<td>max</td>
<td>87</td>
</tr>
</tbody>
</table>

Problem 1. Smoothing Rough Data (29 points = 24 content + 5 clarity)
Professor Naidem wants to analyze the time-varying characteristics of traffic arriving at his web server. To do so, he logs all of the requests to his web server over an \( n \)-second period of time. He stores this data in a large array \( T[1 \ldots n] \), where \( T[i] \) contains the number of requests arriving to the web server during the \( i^{th} \) second of his test.

Unfortunately, the data he collects is too noisy to analyze well. In order to smooth out the data, he wants to use something called a **medianizer**. Given some odd integer \( k \), he wants to replace each array element with the median element of a length-\( k \) window centered at that element. More formally, if we let \( b = (k - 1)/2 \), then for any index \( i \in [1 + b, n - b] \), he replaces \( T[i] \) with the median of the subarray \( T[i - b \ldots i + b] \). This has the effect of smoothing out the data in the array (except for the first and last \( b \) elements).

(a) Describe an efficient algorithm which, given \( T[1 \ldots n] \) and \( k \), uses the techniques described above to smooth the data. The running time of your algorithm may depend on \( k \).

Solution. This subproblem was worth 8 points.

The idea will be that we maintain a binary search tree on the \( k \) elements in the current interval as we go through the data to be smoothed. In order to make life really easy for ourselves, we use an order statistic tree (described in section 14.1 of CLRS and in recitation, and based on a red-black tree) which maintains the size of the subtree rooted at node \( i \). We begin by building our order statistic tree in time \( O(k \log k) \). Then we can retrieve the median, delete the first element and add the next element all in \( O(\log k) \) time. Pseudocode is shown below:

```plaintext
MEDIAN-FILTER(T)
1 BUILD-OS-TREE(T[1 \ldots k]) % O(klog k)
2 for i ← 1 + b to n - b % executed n times
3 med ← OS-SELECT(\( k/2 \)) % O(\( \log k \))
4 S[i] ← med
5 OS-TREE-DELETE(T[i - b]) % O(\( \log k \))
6 OS-TREE-INSERT(T[i + b + 1]) % O(\( \log k \))
7 return S
```
As is clear from the pseudocode, the running time of this algorithm is $O(n \log k)$, provided our OS-Tree is balanced which is the case if we use a red-black tree.

Some people attempted to solve this by using binary search on a sorted array of size $k$, but inserting into a size $k$ array takes $O(k)$ time, so this was an $O(nk)$ solution.

(b) Show that the worst-case running time of any comparison based algorithm which solves part (a) is $\Omega(n \log k)$. For partial credit, show that the worst-case running time of any comparison based algorithm which solves part (a) is $\Omega(n \log n)$ if $k = \Omega(n)$ and $k \leq n/2$.

**Solution.** This subproblem was worth 6 points. Many people attempted a decision tree proof for this one. They argued that there were $k$ possible medians for each interval and thus $k^n$ leaves in the decision tree. This argument is flawed because the $n$ choices of medians are not independent. A more complex, but correct proof is presented below.

We begin by showing that the worst-case running time of any comparison based algorithm which solves part (a) is $\Omega(n \log n)$ if $n = 3m - 2$ and $k = 2m - 1$ for some $m$ (i.e. $k = \frac{2}{3}n + \frac{1}{3}$). We do this by reducing the problem to sorting. If we would like to sort an array $A$ of $m$ elements, we can create a new array $A'$ of $n = 3m - 2$ elements. This array contains $m - 1$ elements of value $-\infty$ (or some number guaranteed to be smaller than any element in the array) followed by the elements of $A$ followed by $m - 1$ elements of value $+\infty$ (or some value larger than any element in $A$). We then run a medianizer on $A'$ with $k = 2m - 1$. Elements $m$ through $2m$ of $A'$ will now be the elements of $A$ in sorted order (because of our choice of values $-\infty$ and $+\infty$ for the additional elements). Since we know that any comparison-based algorithm for sorting needs $\Omega(m \log m)$ time, this implies that any comparison-based medianizer will require $\Omega(n \log n)$ time if $k = \frac{2}{3}n + \frac{1}{3}$. If $k$ is smaller than $\frac{2}{3}n$, we can apply the same trick as above by letting $k = 2m - 1$ but adding more than $m - 1$ elements (of value $\pm \infty$) in front and at the end of the array. This shows a lower bound of $\Omega(k \log k) = \Omega(n \log n)$ whenever $k = \Theta(n)$ and $k \leq \frac{2n}{3}$.

However, this alone is not sufficient, since $k$ may not be $\Theta(n)$. Let $m$ be such that $k = 2m - 1$ and let $p = \frac{n}{3m-2} \sim \frac{2n}{3m}$ (assume for simplicity that $p$ is integral, if not, the construction can easily be modified). We consider the problem of sorting $p$ sets of $m$ elements each. We will show (i) that any comparison based algorithm requires $\Omega(pm \log m)$ time for this problem (using the decision tree method) and that (ii) any medianizer with parameters $n = (3m - 2)p$ and $k = 2m - 1$ can be used to sort these $p$ sets of $m$ elements. (i) and (ii) together then show that any comparison-based medianizer requires $\Omega(n \log k)$ time since otherwise we could sort $p$ sets of $m$ elements each in $o(pm \log m)$ time.

First we show (i). Since the $p$ arrays are completely independent, the number of possible outputs is $m^p$ (each sort can have $m!$ outputs and there are $p$ sorts). So the total number of possible leaves in our decision tree is at least $m^p$. Since a tree
has no more than 2^h leaves (where h is the height of the decision tree), we have m_w \leq 2^h. Taking logarithms, this gives us h = \Omega(pm \log m). So sorting p sets of m elements with a comparison-based algorithm requires \Omega(pm \log m) time.

Now let us show (ii). For each set of size m, as before, add m - 1 elements of value -\infty before and m - 1 elements of value +\infty after. Now concatenate all these arrays of size 3m - 2 into one long array of size n = (3m - 2)p. Now it is easy to see that a medians with parameters n and k applied to this new array easily gives the sorted order for each of these p sets of size b (the sorted arrays will be formed of T'[m + j(3m - 2).2m - 1 + j(3m - 2)] for j = 0, \cdots p - 1).

(c) Suppose that all of the elements in the T array are integers in the range \{0 \ldots M\}, where M = O(1). We wish to represent the elements of the T array in a data structure which behaves similarly to an array in that we can set the value of the i^{th} element of T, but in which we can also compute the median of length-k subarrays very efficiently. Design a data structure which supports the following two operations:

- \text{MODIFY}(i, v) : Set the i^{th} element of T to the value v.
- \text{MEDIAN}(i) : Return the median element of T[i - b, \ldots i + b] (assume that i \in [1 + b, n - b']).

Try to balance the running times of the two operations; try to make the worst of the two as efficient as possible.

\textbf{Solution.} This subproblem was worth 10 points.

To do this we create a data structure which given an interval [i - b, \ldots i + b] it will tell us the number of elements j in that interval. If we have that, then to find the median of an interval I we:

- retrieve the number \( n_j \) of elements \( j = 0 \ldots M \) in I,
- compute the median in constant time (since \( M = O(1) \)).

To get this structure we will build \( M + 1 \) binary trees, one for each value \( j \) an element can take. Each tree will consider the elements from 1 \ldots n as having value either 0, if not equal to \( j \), or 1 otherwise. We will build a complete binary tree on the indices of our array 1 \ldots n; the indices in our tree will be in order, i.e. like in a BST (the root will be indexed \((n+1)/2\), the left child \((n+1)/4, \ldots\)). For simplicity, let us assume that \( n \) is a power of 2 minus 1 (if not, for simplifying the exposition, we can simply increase n to the next power of 2 minus 1).

In each node \( x \) we maintain \( \text{count}(x) \) the number of 1’s in the corresponding subtree rooted at \( x \). The subtree for the root corresponds to the interval \([1 \cdots n]\), while the left (resp. right) child of the root corresponds to the interval \([1 \cdots n/2 - 1]\) (resp. \([n/2 + 1 \cdots n]\)), and so on. We denote the interval corresponding to node \( x \) by \([p(x), q(x)]\).

Now given an interval, say \([a, b]\), we can easily determine the number of 1 in it. This is done by the following recursive procedure by calling \text{SUM-INTERVAL}(a, b, root) (with \( root = (n+1)/2\):
**SUM-INTERVAL(a, b, x)**
1. if \(x\) is odd (i.e. \(x\) is a leaf) **return** \(\text{count}(x)\)
2. if \(a > b\) **return** 0
3. if \(b < x\) **return** SUM-INTERVAL(a, b, LEFT-CHILD(x))
4. if \(a > x\) **return** SUM-INTERVAL(a, b, RIGHT-CHILD(x))
5. if \(a \leq x \leq b\) **return** \(\text{count}(x) - \text{ADD-INTERVAL}(p(x), a - 1, \text{LEFT-CHILD}(x)) - \text{ADD-INTERVAL}(b + 1, q(x), \text{LEFT-CHILD}(x))\)

The procedure decomposes any interval into the addition and subtraction of at most \(O(\log n)\) intervals. The running time of SUM-INTERVAL is \(O(\log n)\).

To find the median from \(q\) to \(r\) we find the number of \(j\)’s in the interval for all \(j\) from 0 to \(M\). Since \(M\) is constant this is also \(O(\log n)\) time. Then add up our numbers as \(j\) increases from 0 to \(M\), and when we pass \(b + 1\) we have obtained our median.

So how can we maintain this structure over a modify? When we want to change an element in position \(i\) from \(C\) to \(D\), we begin by finding the \(C\) tree. We then subtract one from every interval in the path through the tree leading to \(i\). This is \(\log n\) since we have a perfectly balanced binary tree. We then add one to every interval in \(D\) on the path to \(i\). As a result, modify operations are also \(\log n\).

This can be further reduced to \(\log k\) time for both operations. One way to do this is to divide the array into (roughly) \(n/k\) arrays of size \(k\), and build \(M + 1\) trees in each such interval. Then, when interested in an interval of length \(k\), we may have to look at \(2(M + 1)\) trees (to compute the number \(n_j\) of elements of value \(j\) in the interval) since the interval can only intersect 2 of our consecutive subarrays of size \(k\). Since each tree has only \(k\) elements, operations on each tree take \(O(\log k)\), and both operations of our data structure take \(O(\log k)\).

It is useful to realize that we can trivially make either median or modify run in constant time by sacrificing the running time of the other one. Many people gave complex solutions to get Median in \(O(1)\) and Modify in \(O(k)\). However, it is important to realize that you can get an equally good solution (Modify in \(O(1)\) and Median is \(O(k)\)) by leaving the data structure in the array. We use linear time select on the \(k\) elements to find Median and Modify is trivial. Solutions that were slower than this earned few points.

**Problem 2. Videoconferencing (33 points = 28 content + 5 clarity)**

(a) (4 points)

First we prove that any path \(p'\) from \(s\) to \(t\) in \(G'\) is actually a minimum path of \(G\). The value \(\delta(t)\) is defined to be the minimum latency among all paths from \(s\) to \(t\), so we just have to show that \(\ell(p') = \delta(t)\).

Let \(p' = (s = v_0, v_1, \ldots, v_n = t)\). The latency of a path is the sum of the latencies of the edges on the path, so we have
\[
\ell(p') = \ell(v_0, v_1) + \ell(v_1, v_2) + \ldots + \ell(v_{n-1}, v_n).
\] (1)

Since every edge \((v_i, v_{i+1})\) on \(p'\) is in \(E'\), we have \(\delta(v_{i+1}) = \delta(v_i) + \ell(v_i, v_{i+1})\), so 
\[
\ell(v_i, v_{i+1}) = \delta(v_{i+1}) - \delta(v_i).
\]
Substituting this into equation (1) for every \(\ell(v_i, v_{i+1})\), we get:

\[
\ell(p') = (\delta(v_1) - \delta(v_0)) + (\delta(v_2) - \delta(v_1)) + \ldots + (\delta(v_{n-1}) - \delta(v_{n-2})) + (\delta(v_n) - \delta(v_{n-1}))
\]

This sum telescopes, and what remains is \(\ell(p') = \delta(v_n = t) - \delta(v_0 = s)\). Since \(\delta(s) = 0\), we get \(\ell(p') = \delta(t)\).

For the other direction, we must show that if \(p\) is a minimum path in \(G\), then all of the edges of \(p\) are in \(G'\). Let \(p^*\) be some minimum path \(p^* = (s = v_0, v_1, \ldots, v_n = t)\). Since \(p^*\) is a minimum path, we know that all subpaths of \(p^*\) are also minimum paths (since otherwise we could replace that subpath with a smaller path).

Let \(k\) be an arbitrary index in \(p^*, 0 \leq k \leq n - 1\). We know the subpath \((s = v_0, v_1, \ldots, v_k)\) is minimum, which means that \(\ell(s = v_0, v_1, \ldots, v_k) = \delta(v_k)\). This implies \(\ell(s = v_0, v_1, \ldots, v_{k+1}) = \delta(v_k) + \ell(v_k, v_{k+1})\). We also know that the subpath \((s = v_0, v_1, \ldots, v_k, v_{k+1})\) is minimum, and that means that \(\ell(s = v_0, v_1, \ldots, v_k, v_{k+1}) = \delta(v_{k+1})\), so we may conclude that \(\delta(k + 1) = \delta(v_{k+1})\). By the definition of \(E'\), \((v_k, v_{k+1})\) must be in \(E'\).

**Common errors.** There were a lot of “creative” proofs that didn’t quite prove much. Many people misunderstood the definition of \(E'\), and thus couldn’t complete the proof. Many others had some circular reasoning in their proofs.

(b) (8 points)

We will use a modified Dijkstra’s algorithm followed by a dynamic program to solve this problem. We use Dijkstra’s algorithm to build \(G'\), and the dynamic program to count all the minimum paths. Dijkstra’s algorithm can be used since all the edge weights are positive.

Consider Dijkstra’s algorithm (pg. 595 of the book). Theorem 24.6 (pg. 597) shows us that when the algorithm terminates, the labels \(d[v]\) of each vertex \(v\) in the graph are set to the actual minimum distance \(\delta(v)\) from \(s\) to \(v\). So, the algorithm is as follows: Run Dijkstra’s algorithm on \(G\), using \(s\) as the source, and the latencies \(\ell(e)\) as distances on the edges \(e\). When the algorithm terminates, each vertex \(v\) will be labeled with the value \(d[v] = \delta(v)\). Now for each edge \((u, v) \in E\) add \((u, v)\) to \(E'\) if \(d[u] + \ell(u, v) = d[v]\). Thus we have computed \(G' = (V, E')\).

Call \(D\) the running time of Dijkstra’s algorithm. The time to compute \(E'\) is then \(O(D + |E|)\). All implementations of Dijkstra’s algorithm take (at least) time \(\Omega(|E|)\), so this is dominated by the \(O(D)\) term.
By part (a), we know that $G'$ contains every minimum path from $s$ to $t$ in $E$, and that every minimum path of $E$ from $s$ to $t$ is a path in $G'$ from $s$ to $t$. Thus, all we have to do is count the number of paths from $s$ to $t$ in $G'$.

Note that $G'$ is acyclic, since all paths from $s$ to $t$ in $G'$ are minimum by (a), and any path that uses a cycle cannot be minimum (since cycles have total positive latencies). Since $G'$ is acyclic, we can topologically sort it. The counting algorithm begins by topologically sorting $G'$.

Let $N(v)$ be the number of paths in $G'$ from $s$ to $v$. We are interested in computing $N(t)$. The values $N(v)$ obey an optimal substructure based on the topological ordering. Let $\text{pred}(v)$ be the set of vertices $u$ where $(u, v) \in E'$. All paths from $s$ to $v$ must pass through a node in $\text{pred}(v)$ just before entering $v$. Furthermore, every path ending in a node $u \in \text{pred}(v)$ can be extended to $v$ via edge $(u, v)$. Therefore, we have

$$N(v) = \sum_{u \in \text{pred}(v)} N(u).$$

This relationship defines a recursive algorithm for computing $N(v)$. We construct a memoization table $N[1\ldots n]$ with one entry per node in $T$, and initialize all $N[v] = -1$, except $N[s] = 1$. Now run Count-Paths($t$), where Count-Paths is defined as:

\begin{algorithm}
\begin{algorithmic}
  \State \textbf{COUNT-PATHS}(v)
  \If{$N[v]$ does not equal $-1$}, \Return $N[v]$ (Memoization)
  \State $N[v] = \sum_{u \in \text{pred}(v)} \text{COUNT-PATHS}(u)$
  \Return $N[v]$
\end{algorithmic}
\end{algorithm}

We only need to compute $n$ entries, and $n < |V|$. The total computation to figure out the entries can be bounded by $O(|E|)$, since each edge is involved in at most one minimum computation. Thus the total running time of the COUNT-PATHS dynamic program is $O(|V| + |E|)$.

The total running time of the counting procedure is thus $O(D + |E| + |V|)$, which will be dominated by the $O(D)$ term. Page 599 of CLRS expounds upon the different implementations of Dijkstra’s algorithm, which can achieve running times of $O(|V|^2)$, $O(|E|\log |V|)$, or even $O(|V|\log |V| + |E|)$.

**Alternate Solution.** A lot of people just modified Dijkstra’s algorithm by augmenting each vertex with an extra field $N(v)$, and counting paths along the way. The only modification necessary (besides some initialization, where $N[s] = 1$) is in the RELAX procedure, where we say:
RELAX($u, v, w$)
1. if $d[v] > d[u] + w[u, v]$
2. then $d[v] \leftarrow d[u] + w[u, v]$
3. $\pi[v] \leftarrow u$
4. $N[v] \leftarrow N[u]$
5. if $d[v] = d[u] + w[u, v]$
6. then $N[v] \leftarrow N[v] + N[u]$

**Common errors.** Some people tried to enumerate all the paths, and just count them one at a time. This is inefficient, no matter how you enumerate them, since there could be an exponential number of them. Some people modified Dijkstra as above, but instead incremented by 1, instead of $N[u]$, when $d[v] = d[u] + w[u, v]$. This resulted in a mis-counting. Some did not compute $G'$ properly, or did not properly count the paths.

A common mistake was to search $G'$ via DFS or BFS, and try to count that way. That can be made to work, but you need to argue that $G'$ is acyclic, and that some memoization is employed in order to keep the search efficient. Some forgot these details.

(c) (8 points)

What we want to do here is find the minimum $s - t$ cut of $G'$. To do this, we first obtain $G'$ as we did in part (b) using Dijkstra’s algorithm. Then, we find a maximum flow from $s$ to $t$ in $G'$, where the capacity of each edge is set to 1. We set $x$ to be the value of the maximum flow (which is equal to the value of the min-cut by the max-flow min-cut theorem), and tell the professors “don’t worry, at least $x$ edges in $E$ must be disabled to block every minimum path from $s$ to $t$.”

The running time of this procedure is the running time of Dijkstra’s algorithm plus the running time of an algorithm for max flow.

Using Edmonds-Karp for max flow would take $O(|V|^2|E|)$, which would dominate the running time of even a naive implementation of Dijkstra’s algorithm. The best max flow algorithm contained in the book takes time $O(|V|^3)$, which still dominated the running time for Dijkstra’s algorithm. However, since our graph has unit capacities, any implementation of FORD-FULKERSON that saturates augmenting paths will take time $O(|E||f^*|)$ where $f^*$ denotes the maximum flow value. We know that $|f^*| \leq |V|$, since the min-cut has value at most $|V|$ (each edge entering $t$ has unit capacity, and there are at most $|V| - 1$ edges entering $t$). Therefore that running time is $O(|E||V| + |V| \log |V| + |E|)$, using the most efficient implementation of Dijkstra in the book, and this is dominated by the $O(|V||E|)$ term.

It remains to show why our statement is true, and why $x$ is the largest such value that would make our statement true. The $s-t$ min-cut value is the minimum number of edges necessary to remove from a graph to disconnect $s$ and $t$. Therefore one must remove $x$ edges from $G'$ to guarantee it is disconnected, and our statement is
true. Also, for any larger value \( y > x \), it is not true that one \textit{must} remove \( y \) edges to disconnect \( s \) from \( t \), since one \textit{can} remove only \( x \) edges, and disconnect the graph. Therefore \( x \) is the largest value that makes our statement true.

**Common Errors.** A lot of people didn’t realize that what we wanted was the min-cut of \( G' \). Some realized that we wanted the min-cut, but didn’t realize that one could invoke the min-cut max-flow theorem and just solve flow. A bunch of people gave alternative algorithms to find the “min-cut” (or whatever they characterized it as), but they usually fell short of being a correct min-cut algorithm.

Among the correct answers, only a few realized that the running time of the max-flow could be bounded by \( O(|E||V|) \).

(d) (8 points) This was a difficult problem, and only a handful of people received full credit. Here we present one solution, but there are many ways to do it.

A shortest (min-latency) path that is not minimum is a minimumish path. Since a path is minimum if and only if it is a path in \( G' \) (by part (a)), then what we want to find is a shortest path in \( G \) that uses at least one edge \textit{not} in \( E' \). So, if we figure out, for each edge \( (u, v) \) not in \( E' \), the latency of the shortest path using \( (u, v) \), and take the minimum over all \( (u, v) \) not in \( E' \), we have found the length of a minimumish path.

Let \( \gamma(v) \) be the length of the shortest path \textit{from} \( v \) \textit{to} \( t \). The shortest path from \( s \) to \( t \) that uses an edge \( (u, v) \) has latency \( \delta(u) + \ell(u, v) + \gamma(v) \), since the shortest path to from \( u \) to \( s \) has latency \( \delta(u) \), the edge has latency \( \ell(u, v) \), and the shortest path from \( v \) to \( t \) has latency \( \gamma(v) \).

So the algorithm is as follows:

- Compute \( \delta(v) \) for every vertex \( v \in V \) by running Dijkstra’s algorithm on \( G \).
- Compute \( E' \) as in part (b).
- Compute \( \gamma(v) \) for every vertex \( v \in V \) by running Dijkstra’s algorithm on \( G' \), using \( t \) as the source.
- For every \( (u, v) \in E \setminus E' \), compute the value \( \delta(u) + \ell(u, v) + \gamma(v) \). Set \( M \) equal to the minimum such value computed.

The running time of the algorithm is \( O(D) \) for the two runs of Dijkstra’s algorithm, and \( O(|E|) \) for the last step, for a total time of \( O(D) = O(|V| \log |V| + |E|) \), using the most efficient algorithm in the book.

**Other approaches.** A modification of Bellman-Ford can be made to work by adding an extra field \( M[v] \) at each node \( v \) (all initialized to \( \infty \)), and modifying the RELAX procedure as follows:
RELAX($u,v,w$)
1  if $d[v] > d[u] + w[u,v]$  
2      then $M[v] \leftarrow d[v]$  
3      $d[v] \leftarrow d[u] + w[u,v]$  
4      $\pi[v] \leftarrow u$  
6      then $M[v] \leftarrow d[u] + w[u,v]$  
7  if $M[v] > M[u] + w[u,v]$  
8      then $M[v] \leftarrow M[u] + w[u,v]$  

However, you need to reason about how many times all the edges must be relaxed in order to guarantee that $M[t]$ is set correctly. Imagine a minimumish path. Each vertex is visited in the path at most twice, since if it were visited three times, then the path would contain more than one cycle, we could remove one of them, and make a smaller non-minimum path (recall all strictly positive latency). So, $2|V|$ iterations of relaxing all the edges will suffice. Thus the running time is $O(|V||E|)$.

Another approach that we thought of, but no one in the class used, was to make a “two-layered” graph, that consists of two copies of $G$. To get from one copy to the other, you must use an edge in $E \setminus E'$. Then, you find a shortest path from $s$ in one layer to $t$ in the other layer. Ask the TAs for more details of this approach.

**Common Errors.** The most common error was to leave the page blank, which was not really an error in the graders’ opinions. Some tried to enumerate the paths and just look for the next-shortest latency. This is inefficient since there could be an exponential number of paths. A lot of people tried to modify Dijkstra’s algorithm by making a similar change to Relax as above. However, without a bit more work, this didn’t cover the case where the path contained a cycle.

**Problem 3. Allocating Bandwidth (19 points = 14 content + 5 clarity)**

We must find the solution that selects the best allocation of bandwidth between the type 1 (discrete) clients and the type 2 (fractional) clients. We partition the total available bandwidth between type 1 and type 2 clients and, within each partition, we optimally allocate the bandwidth among the clients. First, we notice this is just a combination of the fractional knapsack and 0-1 knapsack problems described on page 382 of CLRS. We solve the former using a greedy algorithm that takes and the latter using dynamic programming then combine the two for a solution that takes $\Theta(nW + W + m \log m)$ time.

First, let’s solve the subproblems.

**Allocating “discrete bandwidth” (4 points)**

Given an allocation of bandwidth $W$, a list of type 1 client requests $a_1,a_2,\ldots,a_n$, and pay scale $y_1,y_2,\ldots,y_n$, find the maximal income $I_M$ we can achieve fulfilling these requests in a 0-1 fashion. We use dynamic programming to solve this problem in $\Theta(nW)$ time. In this case
the subproblems are computing the maximal income available with less bandwidth and/or fewer clients.

Let \( I(i, w) \) be the maximum income we can generate using only \( w \) units of bandwidth and requests \( a_1, a_2, \ldots, a_i \). Our goal is to compute the maximal income available using all \( n \) clients and any number \( w \) of units of bandwidth for any integer \( w \) such that \( 0 \leq w \leq W \).

The relation of the optimal solution to the optimal solutions of the subproblems can be seen in our implementation of \( I(i, w) \) below:

\[
I(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
I(i - 1, w) & \text{if } i > 0 \land w < a_i \\
\max\{I(i - 1, w), I(i - 1, w - a_i) + y_i\} & \text{if } i > 0 \land w \geq a_i
\end{cases}
\]

If we have no clients, we get no income. The maximum income we can generate using only \( w \) units of bandwidth and requests from clients \( A_1, A_2, \ldots, A_i \) is the maximum of the maximum income we can generate clients \( A_1, A_2, \ldots, A_{i-1} \) and the sum of the income we get from client \( A_i \) and the maximum income we can generate using only \( w - a_i \) units of bandwidth (if nonnegative) and requests from \( A_1, A_2, \ldots, A_{i-1} \).

**Analysis**

Filling each entry in table \( I \) takes constant work. Notice that we can always compute an entry by looking up at most two entries in previous columns and rows. The table has \( \Theta(nW) \) entries, so computing the max income for discrete bandwidth takes time \( \Theta(nW) \).

**Allocating “fractional bandwidth” (4 points)**

Given an allocation of bandwidth \( w \), a list of type 2 client requests \( b_1, b_2, \ldots, b_m \), and pay scale \( z_1, z_2, \ldots, z_m \), find the maximal income \( I_M \) we can generate fulfilling these requests, fractionally. We use a greedy algorithm to allocate the bandwidth in \( \Theta(m \log m) \) time.

First, we compute the rate each client is willing to pay. That is the \$/unit of bandwidth \( r_j = z_i/b_j \). Then we sort the requests in order of non-increasing rate.

We select the first request \( b_j \) for client \( B_j \) willing to pay the maximum pay rate \( r_j \) and allocate \( \min\{w, b_j\} \) units of bandwidth yielding \( \alpha_j = \min\{w/b_j, 1\} \). We reduce \( w \) by the amount allocated, remove \( b_j \) from the set of requests, and repeat until either the available bandwidth or the set of requests has been exhausted.

**Correctness**

**Claim:** This greedy algorithm maximizes income.

**Proof:** (by contradiction)

Let \( G = g_1, g_2, \ldots, g_m \) be the allocations (\( \alpha \)) selected by the greedy algorithm generating income \( I_G \). Let \( M = m_1, m_2, \ldots, m_m \) be a maximum allocation allocating income \( I_m \). (Note, any \( g_j \) or \( m_j \) may be 0). WLOG assume that greedy processes the clients in the order \( 1, 2, \ldots, m \), i.e. we have that \( r_1 \geq r_2 \geq \cdots \geq r_m \). Observe that we can assume that \( \sum_{i=1}^m g_i = \sum_{i=1}^m m_i \) since otherwise we can increase the maximum allocation.

For contradiction assume \( I_M > I_G \).
Let $j$ be the position at which the two allocations first differ. We know that $g_j > m_j$ because the greedy algorithm allocates as much of the requested bandwidth as possible to each client. Furthermore, we know that $r_j \geq r_k$ for all $j \leq k$.

because the greedy algorithm always selects the request paying the maximum rate and all requests with rates strictly greater than $r(g_j)$ have already been used up in allocations $g_1 \cdots g_{j-1}$. Therefore we can create a new maximum allocation $M'$ (i.e. such that $I_M' \geq I_M$ since $r_j \geq r_k$ for all $j \leq k$) that increases $m_j$ to be equal to $g_j$ by removing bandwidth from some later allocation(s) $m_k$. (We know some later allocation exists because the greedy algorithm and the maximum allocation have allocated as much until $j - 1$ and also overall.) Thus, $M' = m_1, m_2, \ldots m_{j-1}, g_j, m_{j+1}', \ldots m_m'$. Notice that $M'$ has one more allocation in common with $G$ than $M$.

Now we can iterate, creating $M'$, $M''$, etc. until we have a maximal allocation that is exactly the same as $G$. That contradicts our assumptions that $I_M > I_G$, as needed.

**Analysis**

Sorting the requests takes time $\Theta(m \log m)$. We perform constant work for each allocation. At worst we have to allocate all $m$ requests. So the total worst-case running time is $\Theta(m + m \log m) = \Theta(m \log m)$.

**Putting it together (4 points)**

Let $D(w)$ be the maximal allocation of $w$ units of discrete bandwidth given $a_{1..n}$ and $y_{1..n}$ for any integer $w$ such that $0 \leq w \leq W$. Let $I_d(w)$ be the income derived from that allocation.

Let $C(W-w)$ be the maximal allocation of $W-w$ units of fractional bandwidth given $b_{1..m}$, and $z_{1..m}$ for any integer $w$ such that $0 \leq w \leq W$. Let $I_d(W-w)$ be the income derived from that allocation.

We want to find the correct amount of bandwidth to allocate to each kind of problem. That is, we want to find: $\max_{0 \leq w \leq W} \{I_d(w) + I_c(W-w)\}$. To find this maximum we calculate all $[W] + 1$ discrete allocations using the dynamic programming algorithm described above, all $[W] + 1$ fractional allocations using the greedy algorithm described above, sum the corresponding pairs, and select the maximal allocation.

**Analysis**

Notice that $I_d(w) = I(n, w)$. That is, the $W$ discrete allocations we are looking for is just the bottom row of $I$. Therefore we can compute all $W$ incomes in $\Theta(nW)$ time. We run the greedy algorithm $W$ times so our total running time is $\Theta(nW + Wm \log m)$ time.

**An optimization (2 points)**

We can improve on this solution by noticing that we don’t have to rerun the greedy solution from the beginning each time. In fact, the maximal fractional allocation that yields $I_c(W-w-1)$ is a prefix of $I_c(W-w)$. So we can compute the fractional allocations incrementally. More precisely, we modify the greedy algorithm in the following way. We let $w$ go from $[W]$ down to 0 (while computing $I_c(W-w)$) and $j$ go from 1 to $n$. As we decrement $w$ by 1, we
continue to allocate the additional unit of bandwidth to client $j$ until client $j$ is fully satisfied in which case we move to client $j+1$. With this, we can compute all the $I_c(W-v)$ in time $O(W + m)$, and we should not forget the $O(m \log m)$ time needed to sort the $r_j$’s. So the running time for the incremental greedy algorithm is $\Theta(m + W + m \log m) = \Theta(W + m \log m)$. So our entire optimized solution takes time $\Theta(nW + W + m \log m)$ time.

Common problems

Many students assumed either $W$ or the $b_j$’s were integers. These assumptions allowed some simpler solutions that were not given full credit.

One common but wrong solution was to just assume the the $b_j$’s were discrete and throw them in the dynamic programming table with the $a_j$’s. In addition to slowing down the dynamic programming (by making a bigger $(n + m) \times W$ table), this does not yield an optimal solution. Consider $W = 10$, $a_1 = 3$, $y_1 = 3$, $b_1 = 8$, and $z_1 = 8$. A pure discrete, dynamic solution yields an allocation of 8 units to $B_1$ and nothing to $A_1$ for an income of $8$. While the optimal solution is to allocate 3 units to $A_1$ and 7 units to $B_1$ for an income of $10$.

A variant on this solution was to break up the fractional requests into unit sized chunks. So the above example becomes $W = 10$, $a_1 = 3$, $y_1 = 3$, $b_1 = 1, \ldots, b_{18} = 1$, and $z_{18} = 1$. This solutions works (under the integer solution) but is slow. The table is now $W \times (n + \Sigma b_j)$. (You can cap $\Sigma b_j$ at $W$ if you are clever.)

Many students also failed to realize that we need to examine the entire bottom row of the dynamic programming table. Simply selecting $I(n,W)$ and then filling in the remaining bandwidth in a greedy fashion does not necessarily yield an optimal solution.

Finally, some students tried some variation of an all greedy solution. As mentioned in the book and recitation this does not necessarily yield an optimal solution in the discrete case. Some students realized this and tried to invent some sort of backing off search. Generally, such attempts collapsed under the weight of their own muddled explanation.

Problem 4. State-space exploration (19 points = 14 content + 5 clarity)

We need a data structure that supports the following operations:

- **MODIFY**$(i, v)$: set $S[i] = v$ ; $0 \leq v \leq max_v$.
- **STORE**: stores the current state in the dictionary
- **CHECK**: returns YES if $S$ has been earlier stored in the dictionary; otherwise, returns NO

We need it to support the above operation in $O(1)$ expected time. It turns out we can do this by using hashing techniques shown in class. But we will need three key ideas for actually doing this in $O(1)$ with the probabilities desired.
**Modify:**

**Idea 1:** (3 points) Use a universal hash to map the state $S[i]$ to a value that we use to index into $A[i]$.

We use a scheme shown in class, where:

$$H = \sum_{i=1}^{n} a_i S[i] \pmod{q}$$

$q$ is selected as a large prime number smaller than $n^c$. This can be computed by calling $P(c \log n)$. $a_i$ is selected as a random number in $[0, q-1]$. Computing all the $a_i$ and $q$ runs in time $O(n)$. Unfortunately, we were not allowed any extra time for preprocessing, so idea 2 will help us deal with this.

**Idea 2:** (7 points) We incrementally update the value of the hash. The basic idea is that we know the initial state is the all-zero state, which has a hash value of 0. At every **Modify** call, we change only ONE of the state values. Since only one of the $S[i]$ values changes at each point, we can take the previous $H$ computation, and fix it. The fix involves subtracting the old $a_i S[i]$ and adding a new $a_i v$, and then assigning $S[i] = v$. This is all done mod $p$ which can be computed in $O(1)$.

We still need to compute all the $a_i$ values and $q$. Well, computing $q$ takes $O(1)$, so we can just compute it with the first call to **Modify**. The only difficulty is in computing all $n$ $a_i$ values. Realize that initially, none of the $a_i$’s matter. After one modification, only ONE of the $a_i$’s is contribute to the new hash value (the one associated with the $S[i]$ that has changed). Each **Modify** operation can require at most one new $a_i$. Deciding on one $a_i$ value takes $O(1)$ time, so we construct the actual values of $a_i$ as we make state changes in new positions.

Clearly, the above takes $O(1)$ for all calls to **Modify**. It is correct because it’s simply using the universal hash functions we’ve been taught in class, but putting off some of the computations until they are needed.

**Store** and **Check**:

We can’t put the state in each address location (The state requires $n \log n$ space) and would take $O(n)$ time to copy. We could put a bit in there indicating this hash value is used, but then we can’t chain, so every collision would cause an error. Just by thinking about it in terms of the birthday paradox, we have $O(n^c)$ bins, and we’re going to worry about roughly $n^c$ possible values. Clearly the probability of a collision is greater than a half. The last idea resolves this dilemma.

**Idea 3:** (4 points) The trick is to use another hash to distinguish states. We will compute $H$ as before, but at the same time, also compute $H_2$. This $H_2$ will be another universal hash function similar to $H$, but with a larger output range. For $H_2$ we select a $q < n^{3c}$. $H_2$ is of size $O(3 \log n)$ bits, so we can store it in $A[]$. To compute both $H$ and $H_2$ takes twice as long as just $H$ (so we’re still in the $O(1)$ domain). Then, instead of storing a bit in $A[]$, we store $H_2$ (and use chaining for collisions).
Now when we want to check if we’ve seen a state before, we can check the hash position (and chain if necessary) and compare the second hash. We could make a mistake (if the value we’re checking shares both \( H \) and \( H_2 \) with some other value we’ve previously saved). We will argue that the probability of that is VERY low. Additionally, our running time is still expected \( O(1) \) for both STORE and CHECK since we’re using standard chaining\(^1\) as taught in class (with a load factor of at most 1/2).

To argue the probability bound, we’ll consider the case where only \( H_2 \) matters (and pretend that somehow \( H \) mapped everyone to the same place) and show that even under this much worse case, the probability of seeing no collisions after \( n^c \) operations is still \( > 1/2 \).

First of all, since \( H_2 \) is a universal hash function, the probability of a given pair of states hashing to the same value is \( n^{-3c} \). There are up to \( \binom{n^c}{2} \) possible pairs in our problem. Therefore, the probability of ANY two values colliding is bounded by the union bound, which implies

\[
\begin{align*}
&\leq \binom{n}{2} n^{-3c} \\
&\leq \frac{n^c(n^c - 1)}{2n^{3c}} \\
&\approx \frac{1}{n^c}
\end{align*}
\]

So, the probability of any collision for the above model is significantly less than 1/2. Which means the likelihood of us seeing absolutely no error over the entire \( n^c \) operations is \( > 1/2 \).

**Alternative (problematic) solutions.**

An alternative approach was to use hash functions as in the Karp-Rabin pattern matching algorithm, i.e., functions of the form

\[
h_Q(S) = S \mod Q
\]

where \( Q \) is a random prime, and \( S \) is interpreted as a number in an \( n \)-ary representation, i.e., \( S[0] + S[1]n + S[2]n^2 + \ldots \).

There are two problems with that approach, though:

- After modification of a state \( S \), the update of the hash value \( h_Q(S) \) can be done in \( O(1) \) time only after one precomputes the values of \( n^i \mod qQ \) for all \( i = 1 \ldots n \). This requires additional preprocessing stage.

- One needs \( \Omega(n^{c+1}) \)-size array in order to ensure low collision probability, since we need about \( n^{c+1} \) different primes.

---

\(^1\) Technically, the problem did not specify the option of creating new list elements to form a chain. However, this can be easily simulated using a part of the array \( A[] \).
Therefore, this method is inferior to the previous one. In addition, Karp-Rabin approach was not fully analyzed in class, so certain facts would have to be assumed in order to perform the analysis.

Other solutions involved using different universal hash functions, e.g.,

$$h(S) = ((aS + b) \mod P) \mod n^c$$

where $P$ is a large prime and $S$ is again interpreted as a number. This function, however, is difficult to update.

**Common errors.** Some people assumed that the above hash functions can be *evaluated* (not just updated) in constant time. This is very unrealistic, since the evaluation requires multiplying numbers which are $\Omega(n)$ ($\Theta(n \log n)$, to be precise) bits long. A standard assumption is that we can perform arithmetic operations in unit time on numbers which are $O(\log n)$ bits long. See e.g., lecture 19 or section 32.2 in CLRS for more information.