Practice Quiz 2 for Spring 2012

These problems are four of the five problems from the take-home exam given in spring 2011, along with the full instructions given with the exam so that you have a sense of what the take-home exam will be like.

Because of the order in which we're doing things this term, you should be able to work on **problems 1, 3, and 4 over spring break**. You may not have enough knowledge to do problem 2 until after we come back from break.

Guide to this quiz: For problems that ask you to design an efficient algorithm for a certain problem, your goal is to find the most asymptotically efficient algorithm possible. Generally, the faster your algorithm, the more points you receive. For two asymptotically equal bounds, worst-case bounds are better than expected or amortized bounds. The best solution will receive full points if well written, but ample partial credit will be given for any good solution, especially if it is well written. Bonus points may be awarded for exceptionally efficient or elegant solutions.

Plan your time wisely. Do not overwork, and get enough sleep. Your very first step should be to write up the most obvious algorithm for every problem, even if it is exponential time, and then work on improving your solutions, writing up each improved algorithm as you obtain it. In this way, at all times, you have a complete quiz that you could hand in.

Policy on academic honesty: The rules for this take-home quiz are like those for an in-class quiz, except that you may take the quiz home with you. As during an in-class quiz, you may not communicate with any person except members of the 6.046 staff about any aspect of the quiz, even if you have already handed in your quiz solutions. In addition, you may not discuss any aspect of the quiz with anyone except the course staff until you get them back graded.

This take-home quiz is "limited open book." You may use your course notes, the CLRS textbook, and any of the materials posted on the course web site, but *no other sources whatsoever may be consulted*. For example, you may not use notes or solutions to problem sets, exams, etc. from other times that this course or other related courses have been taught. In particular, you may not use any materials on the World-Wide Web. You probably will not find information in these other sources that will help directly with these problems, but you may not use them regardless.

If at any time you feel that you may have violated this policy, it is imperative that you contact the course staff immediately.

Write-ups: Each problem should be written up separately and submitted to the appropriate turn-in area on Stellar. The Stellar website will have a LaTeX template for each problem solution for you to use. You need not submit a LaTeX solution, although we would prefer it, but you must submit your solutions electronically.

Your write-up for a problem should start with a topic paragraph that provides an executive summary of your solution. This executive summary should describe the problem you are solving, the techniques you use to solve it, any important assumptions you make, and the asymptotic bounds on the running time your algorithm achieves, including whether they are worst-case, expected, or amortized.

Write your solutions cleanly and concisely to maximize the chance that we understand them. When describing an algorithm, give an English description of the main idea of the algorithm. Adopt suitable notation. Use pseudocode if necessary to clarify your solution. Give examples, draw figures, and state invariants. A long-winded description of an algorithm's execution should not replace a succinct description of the algorithm itself.

Provide short and convincing proofs for the correctness of your solutions. Do not regurgitate material presented in class. Cite algorithms and theorems from CLRS, lecture, and recitation to simplify your solutions. Do not waste effort proving facts that can simply be cited.

Be explicit about running time and algorithms. For example, don't just say that you sort n numbers, state that you are using merge sort, which sorts the n numbers in $O(n \lg n)$ time in the worst case. If the problem contains multiple variables, analyze your algorithm in terms of all the variables, to the extent possible.

Part of the goal of this quiz is to test your engineering common sense. If you find that a question is unclear or ambiguous, make reasonable assumptions in order to solve the problem. State clearly in your write-up what assumptions you have made. Be careful what you assume, however, because you will receive little credit if you make a strong assumption that renders a problem trivial.

Good Luck!

Problem 1. For Whom the Road Tolls. [24 points]

Eleanor Sevt is planning to drive from Boston to Los Angeles, but she has only limited funds to make the trip. Fortunately, she is using the MIT solar car, which requires no gas for the journey. Thus, her only costs will be the toll roads she takes on the way. She has acquired a digital model of the North American road network, which consists of a directed graph $G = (V, E)$, where V represents road intersections and E represents the roads themselves. For each edge $e \in E$ there is a length $\ell(e) \in \mathbb{R}$ in miles and a cost $c(e) \in \mathbb{Z}$ in cents for tolls (most of the costs are 0). Give an efficient algorithm to find the shortest path from Boston to L.A. that does not exceed x cents in tolls, where x is given as input.

Solution:

Executive Summary

We reduce the problem to finding a shortest path in a graph with lengths only (no costs), and solve the problem using Dijkstra's algorithm. The algorithm is run on a graph where the vertices also record the cost of reaching them. This graph is larger than the input graph by a factor of $\Theta(x)$.

We make the following assumptions: (1) The graph is sparse $|E| = O(|V|)$; (2) The budget is small, and in particular, $x \leq |V|$. Under these assumptions, the running time is $\Theta(Vx \lg V)$.

Algorithm

Given the graph $G = (V, E)$ with lengths $\ell(e)$ and costs $c(e)$, we run Dijkstra on a new graph with lengths only:

- The vertex set of the new graph is $V \times \{0, \ldots, x\}$, so a vertex is coupled with a cost of reaching it.
- The edge set of the new graph contains an edge $e' = (\langle u, c_1 \rangle, \langle v, c_2 \rangle)$ if $(u, v) \in E$ and $c_1 + c(u, v) = c_2$. The length of e' is $l(u, v)$.

The start vertex is $\langle s, 0 \rangle$ where $s \in V$ is the vertex that corresponds to Boston.

(Notice that lengths are positive, so one can invoke Dijkstra)

Denoting by $t \in V$ the vertex that corresponds to L.A., we then output the shortest path from $\langle s, 0 \rangle$ to $\langle t, c \rangle$ among all possible costs $c \in \{0, \ldots, x\}$ (outputting only the V vertices and not the coupled costs).

Correctness

We show that paths from s to t of cost $c \in \{0, \ldots, x\}$ in G are in one-to-one correspondence with paths from $\langle s, 0\rangle$ to $\langle t, c\rangle$ in the new graph, where corresponding paths have identical lengths. So, the shortest paths are in correspondence as well.

Lemma 1 *If $\langle s, 0\rangle \to \langle v_1, c_1\rangle \to \cdots \to \langle v_k, c_k\rangle$ is a path in the new graph, then $s = v_0 \to v_1 \to \cdots \to v_k$ is a path in G of cost c_k.*

Lemma 2 *If $s = v_0 \to v_1 \to \cdots \to v_k$ is a path in G of cost $c \in \{0, \ldots, x\}$, then for $c_0 = 0$, $c_i = c_{i-1} + c(v_{i-1}, v_i)$ we have that $\langle s, 0\rangle \to \langle v_1, c_1\rangle \to \cdots \to \langle v_k, c\rangle$ is a path in the new graph.*

(Proofs are by induction on k; they are omitted here, but should have appeared in your solution)

Running Time Analysis

Constructing the new graph takes time linear in its size $\Theta(Vx + Ex)$. Running Dijkstra using Fibonacci heaps takes time $\Theta(Vx \lg(Vx) + Ex)$. Finding the shortest path of cost at most x takes time $\Theta(x)$. This is a total of $\Theta(Vx \lg(Vx) + Ex)$.

Reasonable assumptions are that the graph is sparse, so $|E| = \Theta(V)$, and that the budget is small, and, in particular, $x \leq |V|$. The running time under these assumptions is $\Theta(Vx \lg V)$.

A Sketch of an Alternative Solution

Alternatively, one can use Dijkstra's algorithm on the original graph $x+1$ times to compute shortest paths from s to all vertices $v \in V$ of cost c for $c = 0, 1, \ldots, x$.

Denote the length of the shortest path of cost c from a vertex $u \in V$ to a vertex $v \in V$ by $\delta_c(u, v)$. In the c'th application of Dijkstra one computes $\delta_c(s, \cdot)$:

1. The initialization step sets $\delta_0(s, s) = 0$ and $\delta_0(s, v) = \infty$ for $v \neq s \in V$.
2. One invokes a modified Dijkstra on G with δ_c, updating only paths of cost c.
3. The computed shortest paths are updated in preparation for the next step, so for every edge $(u, v) \in E$ with $1 \leq c(u, v) \leq c + 1$, we have $\delta_{c+1}(v) \leq \delta_{c+1-c(u,v)}(u) + \ell(u, v)$.

Remarks

- A straightforward greedy strategy does not work. Consider the following scenario: there are two ways to get to a vertex $v \in V$: one has lower cost but is longer, while the other has higher cost but is shorter. It is not a-priori clear which is better. In general we are looking for the shortest path, but it might be better to take the longer cheaper road, if we are forced to take expensive roads later.
- If x can be exponentially large in n, the problem is NP-hard, so expect the running time for such large x to be exponential.

Solutions We Saw

Correct but less efficient algorithms:

- A Bellman-Ford type algorithm that computes the shortest path from s to v of length at most k and cost c for all $v \in V$, for all $k = 0, 1, \ldots, |V| - 1$, for all $c = 0, 1, \ldots, x$. This algorithm is correct but has worse running time $O(VEx)$.
- A Floyd-Warshall type algorithm, which is correct but even less efficient.

Incorrect Dijkstra+dynamic programming algorithms: A Dijkstra type algorithm where one stores per vertex the shortest distance for every possible cost $c = 0, 1, \ldots, x$, but each vertex $v \in V$ is "visited" only once (extracted from the queue, the edges adjacent to it are relaxed). This algorithm does not work.

Exponential-time algorithms:

- An algorithm that enumerates over all possible choices of toll roads. This is sub-exponential time when the number of toll roads is sub-linear. We do not consider reasonable the assumption that the number of toll roads is logarithmic or constant.
- A brute-force algorithm, enumerating all possible paths from s to t, running in exponential time.
- A 0-1 linear program. If formulated right, this gives an exponential time algorithm.

Incorrect Dijkstra algorithm: A Dijkstra algorithm that only dismisses paths of cost more than x.

Heuristics: These yield algorithms that do not work for many possible inputs:

- Iteratively remove the most expensive edge from the shortest path.
- Iteratively remove the edge with worst cost to distance ratio.
- Combine shortest path with cheapest path.

Others:

- A BFS/DFS type linear-time algorithms.
- Wrong formulation of linear program.

Problem 2. Rounding a Square Matrix. [24 points]

(**NOTE**: You may want to wait to work on this problem until after spring break.)

Consider an $n \times n$ matrix $A = (a_{ij})$, each of whose elements a_{ij} is a nonnegative real number, and suppose that each row and column of A sums to an integer value. We wish to replace each element a_{ij} with either $\lfloor a_{ij} \rfloor$ or $\lceil a_{ij} \rceil$ without disturbing the row and column sums. Here is an example:

$$\begin{pmatrix} 10.9 & 2.5 & 1.3 & 9.3 \\ 3.8 & 9.2 & 2.2 & 11.8 \\ 7.9 & 5.2 & 7.3 & 0.6 \\ 3.4 & 13.1 & 1.2 & 6.3 \end{pmatrix} \to \begin{pmatrix} 11 & 3 & 1 & 9 \\ 4 & 9 & 2 & 12 \\ 7 & 5 & 8 & 1 \\ 4 & 13 & 2 & 6 \end{pmatrix}$$

Give an efficient algorithm to determine whether such a rounding is possible, and if so, to produce the rounded matrix. Be sure to argue that your algorithm is correct.

Solution:

Executive Summary

Finding a feasible rounding can be reduced to finding a max flow in a flow network with nodes corresponding to rows and columns in A and capacities corresponding to row and sum columns. The total running time depends on the time to find a max flow. With a Ford-Fulkerson method, this is $O(n^4)$.

There is always a valid rounding that preserves the row and column sums. This is true regardless of A, as long as the initial row and column sums of A are integers.

Algorithm and Correctness

Denote the decimal part of A by $A' = (a_{ij} - \lfloor a_{ij} \rfloor)$, obtained by subtracting the floor of every element. Let r_i and c_j be the sum of row i and column j of A' respectively. Note that the identity $\sum_i r_i = \sum_j c_j$ always holds for any A because $\sum_i r_i$ and $\sum_j c_j$ are just two different ways of summing over all elements in A'.

Consider representing a rounding of A with a n-by-n binary matrix $B \in (b_{ij})$ where $b_{ij} = 1$ if a_{ij} is rounded up or $b_{ij} = 0$ if a_{ij} is rounded down or is an integer. If additionally the row and column sums of B are equal to those of A', i.e. $\sum_j b_{ij} = r_i$ and $\sum_i b_{ij} = c_j$, then B corresponds to a valid rounding of A. Thus, the problem of finding a valid rounding of A is equivalent to finding a B that satisfies the row and column constraints.

We can further reduce this problem to finding the max flow in the flow network $G = (V, E)$ constructed as follows

1. V is composed of
   - a source node s
   - a sink node t
   - a node $x_i \ \forall i = \{1, \ldots, n\}$
   - a node $y_j \ \forall j = \{1, \ldots, n\}$
2. Set the capacities $\forall i, j \in \{1, \ldots, n\}$
   - $c(s, x_i) = r_i$
   - $c(y_j, t) = c_j$
   - $c(x_i, y_j) = 1$
3. Set all other capacities to 0.

The x_i and y_j nodes represent the rows and columns of B, respectively. The capacities of the edges from s to x_i and the edges from t to y_j represent the row and column sum constraints of B, respectively. Every edge running from an x_i to an y_j has unit capacity. Note that G has $O(n)$ nodes and $O(n^2)$ edges of non-zero capacity.

G has a nice visual structure to it. The subgraph induced from taking the x_i nodes and the y_j nodes forms a bipartite graph where the nodes are likewise divided between the x_i nodes and y_j nodes.

Lemma 3 *There exists a valid rounding of A if and only if a max flow in G saturates the edges $(s, x_i) \ \forall i \in \{1, \ldots, n\}$.*

PROOF. Let B be the binary matrix representing a valid rounding of A. Recall B has the constraints $\sum_i b_{ij} = r_i$ and $\sum_j b_{ij} = c_j \ \forall i, j \in \{1, \ldots, n\}$. Define the flow f with values

$\forall i, j \in \{1, \ldots, n\}$

$$\begin{aligned} f(s, x_i) &= r_i \\ f(y_j, t) &= c_j \\ f(x_i, c_j) &= b_{ij} \end{aligned}$$

Infer the other values from skew symmetry. f satisfies flow conservation for every x_i node because $\sum_{v \in V} f(v, x_i) = f(s, x_i) - \sum_j f(x_i, c_j) = r_i - \sum_j b_{ij} = 0$, and similarly also for every y_j node. Clearly f satisfies the capacity constraints, so f is a valid flow in G. By construction, f saturates the edges (s, x_i) $\forall i$. Thus, $|f|$ equals the capacity of the cut $(s, V - s)$. By the min-cut max-flow theorem, f is a max flow.

For the converse, let f be a max flow in G that saturates the edges (s, x_i) $\forall i \in \{1, \ldots, n\}$. Because $\sum_i r_i = \sum_j c_j$ and $|f| = \sum_i f(s, x_i) = \sum_j f(y_j, t)$, f also saturates the edges (y_j, t) $\forall j \in \{1, \ldots, n\}$.

From f, construct a matrix B, where $b_{ij} = f(x_i, y_j)$ $\forall i, j$. Suppose f is found through the Ford-Fulkerson method. By the integrality theorem (c.f. recitation or CLRS 3rd ed., Theorem 26.10), all of the flow values of f are integer. (Note that non-Ford-Fulkerson methods do not necessarily produce integer max flows). Hence, $f(x_i, y_j) \in \{0, 1\}$ $\forall i, j$, and so B is a binary matrix. From flow conservation and saturation, $\forall i, j \in \{1, \ldots, n\}$

$$\begin{aligned} r_i = f(s, x_i) &= \sum_j f(x_i, y_j) = \sum_j b_{ij} \\ c_j = -f(y_j, t) &= \sum_j f(x_i, y_j) = \sum_i b_{ij} \end{aligned}$$

Thus, B corresponds to a valid rounding of A. Intuitively, the elements a_{ij} which are rounded up are precisely those that have positive flow in the edges (x_i, y_j). In this sense, the max flow f "marks" with positive flow the elements to round up and leaves "unmarked" with zero flow the elements to be rounded down.

$\square$

Lemma 4 *A max flow in G always saturates the edges* (s, x_i) $\forall i\{1, \ldots, n\}$.

PROOF. Define the flow f with values $\forall i, j \in \{1, \ldots, n\}$

$$\begin{aligned} f(s, x_i) &= r_i \\ f(t, y_j) &= c_j \\ f(x_j, y_j) &= a_{ij} - \lfloor a_{ij} \rfloor \end{aligned}$$

Infer the other flow values by skew symmetry. This flow saturates all of the edges coming out of the source and the edges into the sink. It also sets the flow of every edge (x_i, y_j) to the corresponding floating point element $a_{ij} - \lfloor a_{ij} \rfloor$ in A'. f satisfies the capacity constraints, and by definition of r_i and c_j, it is easy to see that f satisfies flow conservation. Thus, f is a valid flow.

$|f| = c(s, V - s)$, so f is a max flow.

(Side note: you may wonder why don't we just construct the max flow in this proof quickly in $O(n^2)$ time and then use it to deduce a valid rounding of A as we did in the previous lemma. This won't work because f here isn't necessarily integer valued, a requirement for the previous lemma.) □

The above lemma says we can always construct such a max flow, so it follows that there always exists a valid rounding of A.

Runtime

Constructing the flow network G takes $O(n^2)$. A Ford-Fulkerson method operates in $O(E|f^*|)$ time, where $|f^*|$ is the max flow. $|E| = O(n^2)$ and $|f^*| = O(\sum_i r_i) = O(n^2)$. So, the total time is $O(n^4)$.

An Edmonds-Karp analysis would yield $O(VE^2) = O(n^5)$ runtime. It is possible to achieve a better runtime using more sophisticated max flow algorithms we haven't learned in class, but we did not penalize students for not mentioning this.

Common pitfalls

1. A good number of students took a greedy approach to perform the rounding. In general these approaches do not work because a greedy algorithm rounds an element in A and ignores its effects on other elements. Moreover, a greedy approach may fail to find a valid rounding even if one exists. The solution, given above, indicates that such a rounding is *always possible*. A greedy approach, usually, doesn't give any guarantees on the existence of a solution.

   Consider a greedy strategy that starts out with a binary matrix B filled with zeros. The algorithm iteratively chooses an element b_{ij} to set to 1 such that the row and column sums of B are still less than or equal to the desired sums r_i and c_j, respectively. The algorithm keeps track of running totals of the row and column sums of B so that it knows which elements can be rounded up. If the running totals eventually equal r_i and c_j, then the algorithm has found a valid rounding, otherwise it will report that no rounding exists. We will show that this algorithm does not work on the matrix

$$A = \begin{pmatrix} 0.5 & 0.1 & 0.4 \\ 0.1 & 0.9 & 0.0 \\ 0.4 & 0.0 & 0.6 \end{pmatrix}$$

The algorithm initializes B as

$$B = \begin{array}{c|ccc} & 1 & 1 & 2 \\ \hline 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{array}$$

where the values on the borders represent the difference between and a row or column sum in A with the current sum (the running total) in the corresponding row or column in B. In other words, this is the number of elements the algorithm is still allowed to round up. Suppose that our greedy algorithm chooses to set all of the elements on the diagonal to be 1. We get

$$B = \begin{array}{c|ccc} & 0 & 0 & 1 \\ \hline 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{array}.$$

Convince yourself that given this selection, it is no longer possible to arrive at a valid rounding without backtracking. So, this greedy algorithm (modulo backtracking) would fail, even when there does indeed a valid rounding such as the one represented by

$$B = \begin{array}{c|ccc} & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{array}$$

In general, greedy approaches fail because they fix a rounding by examining a local set of elements. Usually, greedy algorithms neither backtrack nor modify the existing solution. This is what makes them simple and fast. However, we have seen that it is, often, required to backtrack and modify the current rounding if no further progress can be made. Besides that, analysing algorithms that extensively rely on backtracking is hard. More often than not, the bounds on backtracking algorithms, are powered by a strong theorem. Take for instance the max-flow. The Edmonds-karp algorithm finds augmenting paths over and over again. There is no guarantee that it will terminate in polynomial number of steps given that there could be an exponential number of augmenting paths. But, the Monotonicity Lemma bounds that number of iterations that Edmonds-Karp performs. One could find such theorems behind every max-flow algorithm. It is in this manner that the reduction to max flow transcends the limitations of a local approach.

Many greedy approaches purport an $O(n^2)$ running time. These approaches can be extended to solve for max-flow or maximum bipartite matching. However, it is well known that neither max-flow nor maximum bipartite matching take in greedy approaches. In other words, greedy approaches do not work for these problems.

2. Some students constructed a reduction to a flow network, very similar to one described above, and then found a maximum bipartite matching over it. It seems that some of you, may have,

are unable to distinguish (perhaps, unsurprisingly) between max-flow and maximum bipartite matching problem. A bipartite graph is a graph $G = (V, E)$ where V can be partitioned into A and B such that all the edges go from A to B. The bipartite matching problem asks for a selection of a subset of edges $E' \subset E$ from the bipartite graph such that $\forall v \in V$, there is at most one edge $e \in E'$ incident on v. This is different from the max-flow problem.

3. A few students attempted to formulate the row and column constraints as a linear program. However, the variables in these constraints have to be integer valued. This turns our linear program into an Integer Linear Program. As we have seen in class, solving Integer Linear Programs is NP-hard and an efficient solution is not yet known.

Problem 3. Hyperjumping to Cloud City. [24 points]

Having successfully evaded Darth Vader's fleet in the asteroid field near the planet Hoth, Han Solo and his crew need to make their way to Cloud City in the Bespin system to meet with Lando Calrissian. After a damage inspection, Chewbacca reports that the hyperdrive in the Millennium Falcon is partially disabled and can only make up to four jumps to hyperspace — a fifth jump would certainly prove disastrous.

Moreover, the control panel for entering jump coordinates is damaged, and the only hyperspace jumps that can be performed are from the database of "Recent Jumps," which contains a large number n of previous jumps. Each jump in the "Recent Jumps" database is an integer triple $(\Delta x, \Delta y, \Delta z)$, as specified using the Empire Coordinate System (ECS). A jump displaces the ship by that the specified number of light years in each dimension. That is, if the ship's current location is (x, y, z), its position after the jump is $(x + \Delta x, y + \Delta y, z + \Delta z)$.

Han Solo needs to know as quickly as possible if they can make it to Cloud City, and he needs your help to determine how. The Falcon is currently located at $(-21820, 27348, 36072)$ ECS. Find an efficient algorithm to determine if they can reach Bespin, which is located at $(-23252, 35712, 24387)$ ECS, in at most four of the n jumps from the "Recent Jumps" database, and if so, how.

Solution:

Executive Summary

If we add the jump $(0, 0, 0)$ to the hash table, then the task is to find a combination of 4 vectors in the database that add up to a specific vector. Observe that, this is exactly the same problem to find if for any of the 2-jump combinations there exists some other 2-jump combination such that the addition of both the pairs gives the desired vector. This problem can be done in $O(n^2)$ expected time by using hashing to perform the lookup. We first hash all the n^2 combinations of jumps and then search for the complementary jump in the hash table.

Algorithm

We use a hash table H supporting operations $\textsc{Insert}(k, v)$ which inserts value v at key k and $\textsc{Search}(k)$ which returns value v stored with key k or NIL if the no value is stored (see CLRS Chapter 11). Both operations can be performed in $O(1)$ expected time. We use a jump vector – a triple of three integers – as the key. To do so simply concatenate the binary representation of the integer coordinates. The stored values are pairs of such vectors, or 2-jumps.

Let ΔV denote the difference vector between Bespin and the current location, i.e.,

$$\Delta V = (-23252, 35712, 24387) - (-21820, 27348, 36072)$$

The algorithm inserts all 2-jumps into the hash table, and scans if the complementary 2-jump is already stored. While inserting them, if two different pairs of jumps combine to the same vector (i.e., the key is not unique), we keep the pair that is inserted first.

Assume RJ refers to the "Recent Jumps" database.

```
1   $H = \textsc{Create-Hash-Table}()$
2   $RJ' = RJ \cup (0,0,0)$
3   for $j_1 \in RJ'$
4       for $j_2 \in RJ'$
5           $H.\textsc{Insert}(j_1 + j_2, (j_1, j_2))$
6   // Search for complement
7   for $j_1 \in RJ'$
8       for $j_2 \in RJ'$
9           $x = H.\textsc{Search}(\Delta V - (j_1 + j_2))$
10          if $x \neq$ NIL
11              $(j_3, j_4) = x$
12              return $(j_1, j_2, j_3, j_4)$
13  return NIL // No solution
```

Correctness

If the algorithm returns (j_1, j_2, j_3, j_4) in line 12 then $j_3 + j_4 = \Delta V - (j_1 + j_2)$, hence $j_1 + j_2 + j_3 + j_4 = \Delta V$. If we ignore the $(0,0,0)$ jumps among those four, then we get a combination of up to 4 jumps from RJ that add up to ΔV.

On the other hand, if there exists a combination of 1, 2, 3, or 4 jumps in RJ that add up to ΔV then there exists a combination of exactly 4 jumps in RJ' that add up to ΔV. Denote them i_1, i_2, i_3, i_4. When $j_1 + j_2 = i_1 + i_2$, there must be a value stored at $i_3 + i_4 = \Delta V - j$ by then and therefore the algorithm will return in 12.

Running Time Analysis

We perform up to n^2 INSERT and SEARCH operations each taking $O(1)$ expected time. Thus the overall expected running time is $O(n^2)$.

Discussion

Using the $(0, 0, 0)$ vector is not necessary for correctness but simplifies the description and analysis of the algorithm dramatically so earned a bit of bonus.

Many solutions using hashing tried to "derandomize" and provide a worst-case bound instead of an expected time bound. Although using perfect hashing gives worst-case guarantees on the SEARCH operation, building a perfect hashing data structure for n^2 keys is a random process and takes $O(n^2)$ *expected* time. Also note that the fact that the algorithm tolerates key collisions (if two pairs of jumps add up to the same vector, we only need to keep one), the hash table implementation must still resolve collisions in the hash function.

Many students observed that the lookup of the complementary jump pair can be performed quickly if the list of jumps is sorted. First we define a total order of vectors: to compare two vectors compare them by the first coordinate and if equal use the next coordinate and so on. We build two lists: one of all 2-jump vectors and one of their complements to ΔV. We then sort both with the same comparison operator. We need to find two equal elements in the two lists. Students who used MERGE-SORT to perform the sort in $O(n^2 \lg n)$ time simply used binary search to find the complement of each of the n^2 pairs in $O(\lg n)$ time.

Many students seeked better performance by observing that both the sort and the search can be performed in $O(n^2)$ time. Given the two sorted lists, we can find the matching combination in $O(n^2)$ time (linear in size of the lists) by performing an operation similar to MERGE (see CLRS 2.3.1, p. 31) and returning whenever it finds two matching elements in the two lists. To obtain $O(n^2)$ time for sort, they used RADIX-SORT (see CLRS 8.3, p. 197), by assuming that the coordinates use a constant number of bits. Such solutions should include that factor in the overall running time, i.e., $O(dn^2)$.

A brute-force solution enumerates all 4-jumps and takes $O(n^4)$ time. Note that the desired combination of 4-jumps is not the shortest path to the destination.

Finally, a fraction of the students attempted to use linear programming, but in most cases missed the fact that this would require integer constraints and thereby, making the problem NP-complete.

Problem 4. The Price of a Favor [24 points]

Don Corleone is hearing requests for favors during his daughter Connie's wedding. You are one of the many waiting in line to ask a favor of the Don. It is customary that after requesting a favor, you offer the Don a monetary gift. The Don has a different price X for every favor he grants, which his consigliere (trusted adviser) knows implicitly.

It is highly impolite for you to ask outright, "What will the favor cost me?" to find out the Don's price X. Instead, you must place some cash in an envelope and hand it to the consigliere. The consigliere then inspects the envelope. If your offer meets or exceeds X, the consigliere nods and hands the envelope to the Don, who puts it in his pocket and then grants your favor. If the offer is less than X, however, the consigliere takes your offer and puts it in his own pocket and does not allow the Don to be insulted by your puny gift. In this case, your gift is lost to the consigliere, and you must try again with a bigger gift.

Give and analyze a competitive strategy that minimizes the amount of money you must pay to obtain your favor from the Don.

Solution:

Executive Summary

We assume that X is at least 1 unit. For otherwise, if X could be made arbitrarily small, the competitive ratio can be made as large as we wish. We also assume that the distribution of X is uniform over the entire domain. We, now, give a sketch of our strategy.

Our strategy is to pick an initial offer in the range $[1, e)$ randomly according to a probability distribution $p(x) = 1/x$. Each time the offer is rejected, we try again with an offer e times more than the last offer. The claim, we make, is that irrespective of what X is, the expected amount of money our we end up paying for a favor is at most eX. In other words, we achieve a competitive ratio of e.

Strategy

```
DON:
1  $r \in_R [1, e)$ // $r$ is chosen at random with $\Pr[r = x] = 1/x$
2  $\mathsf{O} = r$
3  while $\mathsf{O} < X$
4        $\mathsf{O} \geq e \cdot \mathsf{O}$
```

Correctness

We assume that it is possible to pick a number from the interval $[1, e]$ at random according to the probability distribution $\Pr[r = x] = 1/x$. Thus, given any O $\in_R [1, e)$. Each subsequent offer is a positive amount of money. Furthermore, the n^{th} offer will have a value of $r \cdot e^{n-1}$. Notice that the sequence is a monotonically increasing one. Thus, irrespective of X (X is some constant), $r \cdot e^{n-1} > X$ for sufficiently large n, so eventually one of the offers will be accepted.

Competitive Analysis

Lemma 5 DON *is e-competitive with OPT.*

Proof. DON generates a sequence of integers $\{x_i\}_{i\in\mathbb{Z}}$ during any execution. Specifically, the sequence generated by our scheme is $x_n = r \cdot e^{n-1}$, $r \in_R [1, e)$. Every offer that our scheme makes is e times the previous one. So, there is a unique c and n, $c \in [0, 1]$ and $n \in \mathbb{Z}$, such that $X = c \cdot e^n$. So, in our scheme if $r < c$, DON iterates for $n + 1$ rounds. If $r \geq c$, DON iterates n times. Let Y denote the total cost we end up paying Don. Then,

$$\begin{aligned}
\mathrm{E}[Y] &= \int_1^c 1/r\big(r \cdot \tfrac{e^{n+2}}{e-1}\big)dr + \int_c^e 1/r\big(r \cdot \tfrac{e^{n+1}}{e-1}\big)dr \\
&= (c-1) \cdot \tfrac{e^{n+2}}{e-1} + (e-c)\tfrac{e^{n+1}-1}{e-1} \\
&= \tfrac{1}{e-1}\big[c \cdot \big(e^{n+2} - e^{n+1}\big) + e \cdot \big(e^{n+1} - 1\big) - e^{n+2} + 1\big] \\
&= \tfrac{1}{e-1} \cdot \big[(e-1) \cdot (e^{n+1}) + (1-e)\big] \\
&= eX - 1 \\
&\leq eX
\end{aligned}$$

Thus, the total expected cost is no greater than eX, so DON is e-competitive with OPT. □

The analysis given above can be extended to r sampled from $[1, \alpha)$, for any constant α and thus, derive an expression for competitive ratio as a function of α. An interesting question would be to find an α which minimizes the competitive ratio. This is left as an exercise for the reader.

Alternate Strategies

There have been a variety of randomized schemes. The standard template seems to be choosing your initial offer uniformly at random and then generate a geometric series. A few students have chosen the i^{th} bid, uniformly at random from $[2^{i-1}, 2^i]$, to construct a 3-competitive scheme.

Another genre is to use only one bit of randomness. In the doubling strategy, we set the initial bid to 1 or $\sqrt{2}$ with equal probability. This yields a $(2 + \sqrt{2})$-competitive strategy.

A lot of students have come up with a deterministic strategy which doubles the offer, every-time it is rejected. This is 4-competitive. A bunch of them went on to generalize the scheme to any deterministic geometric sequence and have shown that doubling is, essentially, asymptotically optimal. A few of you have conjectured that 4 is optimal for deterministic strategies. As a matter of fact, it is true. With a little bit of effort, one may establish the lower bound. We leave it as an exercise.

Substantial partial credit was given for proposing a correct strategy and analysing the competitive ratio.

Common Pitfalls

A handful of students, wrongly, proved doubling strategy to be 2-competitive. A common error has been the failure to identify the worst case for their strategy. Some students considered arithmetic sequences. Even though, these sequences are correct strategies. These schemes have a competitive ratio linear in X – which is not good enough.