Introduction to Algorithms April 16, 2004
Massachusetts Institute of Technology 6.046J/18.410J
Professors Erik Demaine and Shafi Goldwasser Handout 21

Problem Set 6 Solutions

This problem set is due **in recitation** on **Friday, April 16**.

Reading: Chapter 15, Chapter 17, 16.1-16.3, 22.1-22.2, Chapter 23

There are **four** problems. Each problem is to be done on a **separate sheet** (or sheets) of paper. Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date, and the names of any students with whom you collaborated. As on previous assignments, "give an algorithm" entails providing a description, proof, and runtime analysis.

Problem 6-1. Danny's Daemon

Suppose there are m bins containing a total of n balls, where $m > n$. Initially, n of the bins contain one ball and the other $m - n$ bins are empty. Sitting on top of the bins is a daemon who rearranges the balls by a series of *moves*. Each move, the daemon will select a bin b_i containing k_i balls, and redistribute each ball to a unique bin. In other words, the bin b_i will lose all k_i balls and k_i other bins will each gain exactly one ball. We define the cost of this move to be k_i. The total number of balls in the system remains constant, i.e. $n = \sum k_i$.

(a) Define an infinite sequence of moves such that after some finite start-up period the cost of each move is $\Theta(\sqrt{n})$. Hint: Your solution may be exactly $\lfloor\sqrt{2n + 1/4} - 1/2\rfloor$.

Solution: The daemon can maintain a set of k bins of distinct heights 1 through k. On each move the daemon will empty the bin with k balls and redistribute $k - 1$ balls to each of the partially filled bins and place the remaining ball in an empty bin. The cost of this move will always be k. Since the system has n balls, we have the following inequality:

$$\begin{aligned} \sum_{i=1}^{k} i &\leq n \\ \frac{k(k+1)}{2} &\leq n \\ k^2 + k - 2n &\leq 0 \end{aligned}$$

Solving the inequality we have:

$$\frac{-1 - \sqrt{1 + 8n}}{2} \leq k \leq \frac{-1 + \sqrt{1 + 8n}}{2}$$

By setting k to the largest integer within the range, the cost of each move with this scheme is $\lfloor\sqrt{2n + 1/4} - 1/2\rfloor$.

(b) Prove that the amortized cost of a move is at most $\lfloor 2\sqrt{n} \rfloor$. Hint: Consider a potential function in which a bin with k_i balls contributes $\max\{0, k_i - t\}$, where t is a constant of your choice and is the same for all bins.

Solution: Let the potential of each bin be $\phi(k_i) = \max\{0, k_i - t\}$, for some t to be determined. It is clear that the potential of the system equals 0 initially and is always non-negative. Consider a move that redistribtes k_i balls to k_i bins, among which c of them already contains at least t balls. It follows that $ct \leq n$.

The cost of moving the bin is k_i. The change in potential is $c - \max\{0, k_i - t\}$. We consider the two cases:

1. $k_i \leq t$. The cost of moving is k_i and the change in potential is c. Therefore the amortized cost of the move is:

$$\begin{aligned} \hat{c}(m) &= c(m) + \Delta\phi \\ &= k_i + c \\ &\leq t + c \\ &\leq t + \frac{n}{t} \end{aligned}$$

2. $k_i > t$. The cost of moving is k_i and the change in potential is $c - (k_i - t)$. Therefore the amortized cost of the move is:

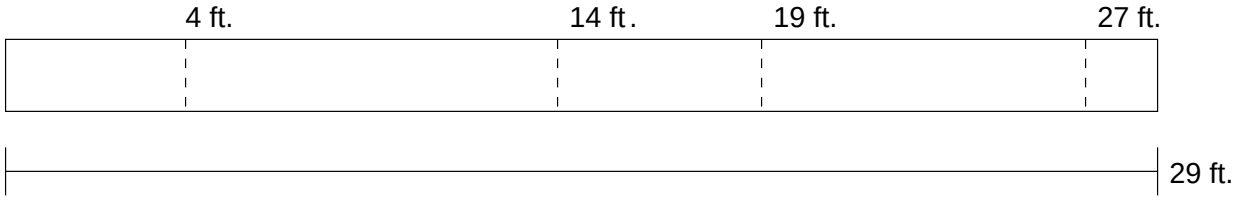
$$\begin{aligned} \hat{c}(m) &= c(m) + \Delta\phi \\ &= k_i + c - (k_i - t) \\ &\leq t + c \\ &\leq t + \frac{n}{t} \end{aligned}$$

By setting $t = \sqrt{n}$, we have $t + \frac{n}{t} = 2\sqrt{n}$. Therefore, the amortized cost is at most $\lfloor 2\sqrt{n} \rfloor$.

Problem 6-2. Cutting Wood

Your favorite sawmill charges by length to cut each board of lumber. For example, to make one cut anywhere on an 8 ft. board of lumber costs \$8. The cost of cutting a single board of wood into smaller boards will depend on the order of the cuts.

As input, you are given a board of length n marked with k locations to cut. The input to part (b) represents the following board:



(a) Give an algorithm that, given an input length n of wood and a set of k desired cut points along the wood, will produce a cutting order with minimal cost in $O(k^c)$ time, for some constant c.

Solution: For this problem, we let the values of the k cuts be $c[1 \ldots k]$. Let $c[0] = 0$ and $c[k+1] = n$. Then the board is partitioned into $k+1$ regions, where region r_i goes from $c[i-1]$ to $c[i]$ and has length $c[i] - c[i-1]$. The overlapping subproblems part is easy to see. The key to using dynamic programming is to notice that at any point in a solution, you have a set of boards, each of which is the union of *consecutive* subregions.

Let $A[i,j]$ be the cost of optimally cutting the region consisting of $r_i, r_{i+1}, \ldots, r_j$. Then we can recursively define $A[i,j]$ as follows:

$$A[i,j] = c[j] - c[i-1] + \min_{i \le k < j} \{A[i,k] + A[k+1,j]\}$$

The first term in this expression is the cost of making a cut in region $r_i, r_{i+1}, \ldots, r_j$, and the second is the cost of the recursive solution.

We use the following algorithm, which fills in the table, one diagonal at a time. We use the notational conventions mentioned above:

```
CUT(n, k, c[1, ... k])
c[0] ← 0
c[k + 1] ← n
for i = 1 to k + 1
    A[i, i] ← 0
for diff = 1 to k
    for i = 1 to k + 1 − diff
        j ← i + diff
        A[i, j] ← (c[j] − c[i − 1]) + min_{i≤k<j} {A[i, k] + A[k + 1, j]}
```

Each of $O(k^2)$ cells in the table is defined by $O(k)$ other cells, so the running time of this algorithm is $O(k^3)$.

(b) Suppose you have a 29 ft. board and you want to cut it at points 4, 14, 19, and 27 ft. from the left end. Use your solution from part (a) to determine the minimal cutting cost and illustrate the execution of your algorithm.

Solution:

	1	2	3	4	5
1	0	14	33	54	68
2	x	0	15	36	50
3	x	x	0	13	25
4	x	x	x	0	10
5	x	x	x	x	0

The cost of the optimal solution is \$68. The optimal cutting order is $14, 4, 19, 27$.

(c) Paul Bunyan suggests: Always cut a piece as close as possible to the center. Does this produce an optimal solution? Why or why not?

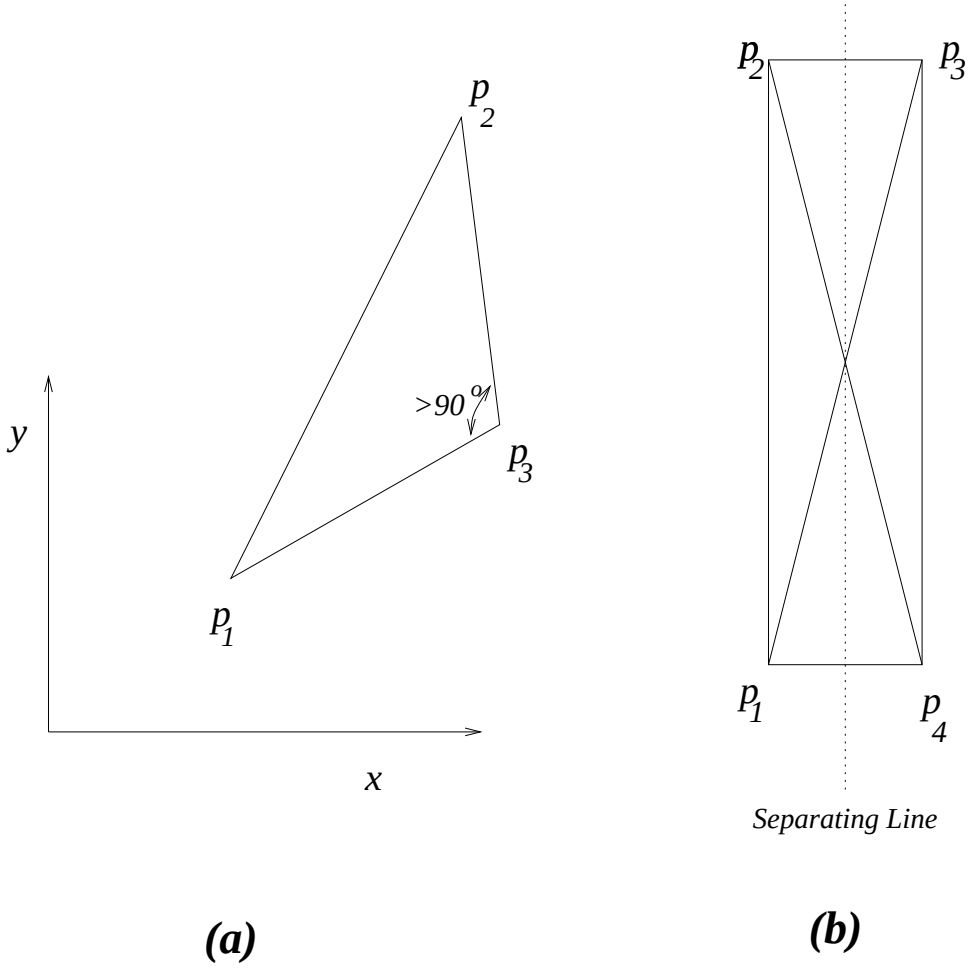
Solution: This does not work. Consider a 100 ft. board with cuts at $49, 50$ and 51. The Paul Bunyan heuristic says make cuts in the order $50, 49, 51$. This has cost 200. However if you cut in the order $49, 51, 50$, the cost is 153. In fact in this case, the heuristic yields the worst possible solution.

Problem 6-3. Minimum Spanning Tree in the Plane

Consider the problem of finding the minimum spanning tree connecting n distinct points in the plane, where the distance between two points is the ordinary Euclidean distance. In this problem we assume the distances between all pairs of points are distinct. For each of the following procedures, either argue that it constructs the minimum spanning tree of the n points or give a counterexample.

(a) Sort the points in order of their x-coordinate. (You may assume without loss of generality that all points have distinct x-coordinates; this can be achieved if necessary by rotating the axes slightly.) Let $(p_1, p_2, \ldots, p_n)$ be the sorted sequence of points. For each point p_i, $2 \leq i \leq n$, connect p_i with its closest neighbor among $p_1, \ldots, p_{i-1}$.

Solution: This procedure does not work. Notice that in the figure below we must use the line segment (p_1, p_2). However, using the segment (p_1, p_3) would yield a smaller spanning tree. Therefore, this procedure does not produce a MST.



(b) Draw an arbitrary straight line that separates the set of points into two parts of equal or nearly equal size (i.e. within one). (Assume that this line is chosen so it doesn't intersect any of the points.) Recursively find the minimum spanning tree of each part, then connect them with the minimum-length line segment connecting some point in one part with some point in the other (i.e. connect the two parts in the cheapest possible manner).

Solution: This procedure does not work. Notice that in the figure above if we follow the method suggested with the choice of line as in figure, then $(p_1, p_2), (p_3, p_4), (p_1, p_4)$ or $(p_1, p_2), (p_3, p_4), (p_2, p_3)$ would be the tree output by the algorithm. Clearly the tree $(p_1, p_4), (p_1, p_2), (p_2, p_3)$ is a MST.

(c) Begin with each point as an isolated tree with no line segments. Consider each segment e in *decreasing* order by length. If the segment e connects two distinct trees, add it to the set of segments in the current spanning forest, and merge the trees together. If the segment e connects two points x, y in the same tree of the forest, add this segment

to the spanning forest while removing the longest segment on the path between x and y in the tree.

Solution:

This procedure correctly outputs the MST. We argue that it works as follows:

First, the algorithm clearly gives some spanning tree. It starts with all the points in isolated "trees" and when each segment is added it maintains the "tree" property.

Proof by Contradiction:

Assume that the algorithm does not give an MST. Then there exists segment(s) in a MST that are not in the spanning tree of the algorithm (if the spanning tree contained every segment of an MST then to not be an MST it must contain more segments and would not be a tree). Let (u, v) be the smallest segment in the MST and not in the spanning tree of the algorithm.

Separate the MST graph about this segment so that we separate the points into two graphs, A and $\overline{A}$, and segment (u, v) is the only segment that crosses the cut (we can do this because the graph is a tree).

Now, I claim that segment (u, v) is light for this cut. (If it were not, then we could choose a smaller segment, and remove (u, v) and obtain a smaller MST).

Finally, consider the running of the algorithm presented. At some point, segment (u, v) is removed (since we are assuming the algorithm doesn't work). In order to remove (u, v) we must be adding an segment (x, y) and we must have found that (u, v) is the largest segment on the path from x to y. But this is impossible. We have two cases:

1. Assume that (x, y) does not cross the cut, $(A, \overline{A})$. Then for (u, v) to be on the path from x to y then there must exist another segment (c, d) on that path which crosses our cut, so (c, d) must be larger (we assume unique weights) and the algorithm would remove it and not (u, v). (Contradicting the removal of (u, v).)
2. Assume that (x, y) crosses the cut. Then (x, y) crosses the cut and (x, y) must be larger than (u, v) (since (u, v) is light for the cut). But this is impossible since we are examining segments in decreasing order.

Thus, our assumptions must be false. Therefore the algorithm produces an MST.

Problem 6-4. Shortest Paths

(a) We are given a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has an associated value $r(u, v)$, which is a real number in the range $0 \leq r(u, v) \leq 1$ that represents the reliability of a communication channel from vertex u to vertex v. We interpret $r(u, v)$ as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

Solution: To find the most reliable path between s and t, run Dijkstra's algorithm with edge weights $w(u, v) = -\log r(u, v)$ to find shortest paths from s in $O(E + V \log V)$ time. The most reliable path is the shortest path from s to t, and that path's reliability is the product of the reliabilities of its edges.

Here's why this method works. Because the probabilities are independent, the probability that a path will not fail is the product of the probabilities that its edges will not fail. We want to find a path from s to t such that $\prod_{(u,v)\in p} r(u, v)$ is maximized. This is equivalent to maximizing $\log(\prod_{(u,v)\in p} r(u, v)) = \sum_{(u,v)\in p} \log r(u, v)$, which is in turn equivalent to minimizing $\sum_{(u,v)\in p} -\log r(u, v)$. (Note: $r(u, v)$ can be 0, and $\log 0$ is undefined. So in this algorithm, define $\log 0 = -\infty$.) Thus if we assign weights $w(u, v) = -\log r(u, v)$, we have a shortest-path problem.

Since $\log 1 = 0$, $\log x < 0$ for $0 < x < 1$, and we have defined $\log 0 = -\infty$, all the weights w are nonnegative, and we can use Dijkstra's algorithm to find the shortest paths from s in $O(E + V \log V)$ time.

(b) Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \{0, 1, \ldots, W\}$ for some nonnegative integer W. Modify Dijkstra's algorithm to compute the shortest paths from a given source vertex s in $O(WV + E)$ time.

Solution: Observe that if a shortest-path estimate is not ∞, then it's at most $(|V| - 1)W$. Why? In order to have $d[v] < \infty$, we must have relaxed an edge (u, v) with $d[u] < \infty$. By induction, we can show that if we relax (u, v), then $d[v]$ is at most the number of edges on a path from s to v times the maximum edge weight. Since any acyclic path has at most $|V| - 1$ edges and the maximum edge weight is W, we see that $d[v] \le (|V| - 1)W$. Note also that $d[v]$ must also be an integer, unless it is ∞.

We also observe that in Dijkstra's algorithm, the values returned by the EXTRACT-MIN calls are monotonically increasing over time. Why? After we do our initial $|V|$ INSERT operations, we never do another. The only other way that a key value can change is by a DECREASE-KEY operation. Since edge weights are nonnegative, when we relax an edge (u, v), we have that $d[u] \le d[v]$. Since u is the minimum vertex that we just extracted, we know that any other vertex we extract later has a key value that is at least $d[u]$.

When keys are known to be integers in the range 0 to k and the key values extracted are monotonically increasing over time, we can use a Monotone Priority Queue (MPQ from Problem Set 3) to implement m INSERT, EXTRACT-MIN, and DECREASE-KEY operations in $O(m + k)$ time, with some minor modifications.

Here's how: We use an array, say $A[0\,..\,k]$, where $A[j]$ is a linked list of each element whose key is j. Think of $A[j]$ as a bucket for all elements with key j. We implement each bucket by a circular, doubly linked list with a sentinel, so that we can insert into or delete from each bucket in $O(1)$ time. We perform the MPQ operations as follows:

- INSERT: To insert an element with key j, just insert it into the linked list in $A[j]$. Time: $O(1)$ per INSERT.

- EXTRACT-MIN: We maintain an index *min* of the value of the smallest key extracted. Initially, *min* is 0. To find the smallest key, look in $A[\textit{min}]$ and, if this list is nonempty, use any element in it, removing the element from the list and returning it to the caller. Otherwise, we rely on the monotonicity property (and that there is no INCREASE-KEY operation) and increment *min* until we either find a list $A[\textit{min}]$ that is nonempty (using any element in $A[\textit{min}]$ as before) or we run off the end of the array A (in which case the MPQ is empty).
  Since there are at most m INSERT operations, there are at most m elements in the min-priority queue. We increment *min* at most k times, and we remove and return some element at most m times. Thus, the total time over all EXTRACT-MIN operations is $O(m + k)$.
- DECREASE-KEY: To decrease the key of an element from j to i, first check whether $i \leq j$, flagging an error if not. Otherwise, we remove the element from its list $A[j]$ in $O(1)$ time and insert it into the list $A[i]$ in $O(1)$ time. Time: $O(1)$ per DECREASE-KEY.

To apply a MPQ to Dijkstra's algorithm, we need to let $k = (|V| - 1)W$, and we also need a separate list for keys with value ∞. The number of operations m is $O(V + E)$ (since there are $|V|$ INSERT and $|V|$ EXTRACT-MIN operations and at most $|E|$ DECREASE-KEY operations), and so the total time is $O(V+E+VW) = O(VW+E)$.