Introduction to Algorithms November 15, 2002
Massachusetts Institute of Technology 6.046J/18.410J
Professors Erik Demaine and Shafi Goldwasser Handout 22

Problem Set 6 Solutions

(Exercises were not to be turned in, but we're providing the solutions for your own interest.)

Exercise 6-1. Let p, q be any pair of points from Q such that they are not both vertices of $CH(Q)$. Then without loss of generality, suppose p is not a vertex of $CH(Q)$. Then there are two cases: p is in the interior of $CH(Q)$, or p is on an edge of $CH(Q)$ but is not a vertex of it. In the first case, by drawing a ray from q to p we intersect the boundary of $CH(Q)$ at some point r, where $|q - r| > |q - p|$, and if r is not a vertex of $CH(Q)$ we are in the second case. In the second case, going along the edge in some direction takes us farther away from q, until we reach an endpoint of the edge (which is a vertex of $CH(Q)$). Therefore we have a point $r' \in Q$ that is a vertex of $CH(Q)$, and is farther from q than p is. Therefore p, q cannot be a pair of farthest points, and the proof is complete.

Exercise 6-2. Suppose $n \leq m$ (if not, then this solution will use columns instead of rows). We keep only rows i and $i + 1$ of the c table in memory at one time. Once row $i + 1$ has been filled in, we overwrite the data from row i when computing row $i + 2$. This is possible because elements of row $i+2$ only depend upon elements in row $i+1$ and the already-computed elements of row $i+2$. The space required is therefore $2n = 2 \cdot \min(m, n)$, plus $O(1)$ space for temporary variables.

To optimize even further, we can actually keep just one row R of the c table at a time. After computing row i, we will immediately overwrite it from left-to-right by row $i + 1$. Note that $c[i+1, j]$ depends only upon $c[i, j]$, $c[i, j-1]$, and $c[i+1, j-1]$. Suppose we are in the middle of filling in the array R with a new row of c. The invariant is: $R[0\,..\,j-1]$ contains $c[i+1, 0\,..\,j-1]$, while $R[j\,..\,n]$ contains $c[i, j\,..\,n]$, and furthermore, we've saved $c[i, j-1]$ in a temporary variable v. Then to compute $c[i+1, j]$, it is enough to look at $R[j] = c[i, j]$, $v = c[i, j-1]$, and $R[j-1] = c[i+1, j-1]$. Then we save the value $R[j]$ into v, and let $R[j]$ be the value of $c[i+1, j]$ we have just computed. Now the invariant is true for the next value of j, so we can iterate until R contains all of $c[i + 1, 0\,..\,n]$. The space requires is $2n = 2 \cdot \min(m, n)$, plus $O(1)$ space for temporary variables (including v).

Exercise 6-3. Actually, it wouldn't change the asymptotic running time at all. To compute $w[i, j]$ manually on line 8 would require $\Theta(j - i)$ additions, instead of $\Theta(1)$ as in the book's version. However, line 9 does a loop from i to j anyway, which takes $\Theta(j-i)$ time. Doing another $\Theta(j-i)$ work on line 8 would not affect the asymptotic running time of $\Theta(n^3)$.

Exercise 6-4. We use a greedy strategy: sort the points, and start from the smallest point x. Then do the following loop: add a unit-interval whose left endpoint is at x, then go over the points (starting at x) until a point $y > x + 1$ is found (or no more points remain). If such a y is found, let $x \leftarrow y$, and loop.

It is clear that the unit intervals cover all the points: during the loop, any points y' such that $x \leq y' \leq x + 1$ are covered by the interval that was added at the start of that loop iteration. The running time is $O(n \log n)$ because the algorithm sorts the points then makes one pass over them.

Now we just need to prove the greedy-choice property for this problem. That is, we must prove that, at any loop iteration, there is an optimal set of intervals for the points $P = \{z : zgeqx\}$ that includes the interval $[x, x+1]$. Given such an optimal set, it must include some interval I that covers x. Since no element in P is less than x, we can "shift I right" until its left endpoint equals x; add this shifted interval I' to the set and remove the original I. Now the set of intervals still covers P and is still optimal (because we haven't increased its size). But now $I' = [x, x+1]$, so indeed $[x, x+1]$ is included in some optimal set of intervals, as we sought to show. By induction, our algorithm yields an optimal solution.

Exercise 6-5. Suppose the characters are ordered such that $f(c_1) \geq f(c_2) \geq \cdots \geq f(c_n)$. We claim that there is some optimal code, having lengths d, where $d(c_1) \leq d(c_2) \leq \cdots \leq d(c_n)$. Take some optimal code for which there exist $i < j$ such that $d(c_i) > d(c_j)$. The length of the file is $\sum_{k=1}^{n} f(c_k)d(c_k)$. Now define a new code, having lengths d', where the encodings of c_i and c_j are swapped (relative to our initial code). Then $d'(c_i) = d(c_j) < d'(c_j) = d(c_i)$, and $d'(c_k) = d(c_k)$ for $k \neq i, j$. The difference between the lengths of the new and old files is:

$$\begin{aligned}
\sum_{k=1}^{n} f(c_k)(d'(c_k) - d(c_k)) &= f(c_i)(d'(c_i) - d(c_i)) + f(c_j)(d'(c_j) - d(c_j)) \\
&= (d(c_i) - d(c_j))(f(c_j) - f(c_i)) \\
&\leq 0
\end{aligned}$$

because $d(c_i) - d(c_j) > 0$ and $f(c_j) - f(c_i) \leq 0$. Then the new code is also optimal, because the file does not grow relative to the old code. Then by swapping the encodings of c_i, c_j for all such $i < j$ for which $d(c_i) > d(c_j)$ (i.e., sorting the encodings by length), we arrive at an optimal code whose codeword lengths are monotonically increasing.

Exercise 6-6. Suppose that (u, v) is in some minimum spanning tree T of a graph (V, E). Define a cut so that S contains all nodes reachable from u by edges in $T - \{(u, v)\}$, whereby $V - S$ contains all nodes reachable from v by edges in $T - \{(u, v)\}$. If (u, v) is not a light edge of the cut $(S, V - S)$, then there is some lighter edge (u', v') which crosses the cut. But then $T \cup \{(u', v')\} - \{(u, v)\}$ is a spanning tree, whose cost is smaller than T's. This contradicts the fact that T is a minimum spanning tree; therefore (u, v) is a light edge crossing $(S, V - S)$.

Exercise 6-7. When $|E| = \Theta(V)$, the running time of both algorithms is $\Theta(V \lg V)$. When $|E| = \Theta(V^2)$, the running time of the binary heap version is $\Theta(V^2 \lg V)$, while the running time of the Fibonacci heap version is $\Theta(V^2)$. For the Fibonacci heap implementation to be faster than the binary heap implementation, it is necessary (by the sparse-graph observation) and sufficient (by what we'll show below) for E to be $\omega(V)$. For if $E = \omega(V)$, there are two cases. First, if $E = O(V \lg V)$: the binary heap version runs in time $\omega(V \lg V)$, while the Fibonacci heap version runs in time $O(V \lg V)$, which is asymptotically better. Second, if $E = \omega(V \lg V) = \omega(V)$, then the binary heap version runs in time $\Theta(E \lg V)$ while the Fibonacci heap version runs in time $\Theta(E)$, which is a factor of $\lg V$ better.

Problem 6-1. Reducing space in van Emde Boas queues

(a) The space $S(u)$ occupied by the data structure is given by the recurrence

$$S(u) = (1 + \sqrt{u})S(\sqrt{u}) + O(\sqrt{u}) ,$$

because in each widget there are $\sqrt{u}$ recursive subwidgets, 1 recursive summary widget, and an array of size $O(\sqrt{u})$.

First we prove that $S(u) \le c_1 u - c_2$ by the substitution method. Assume by induction that $S(k) \le c_1 k - c_2$ for all $k < u$. Then

$$\begin{aligned} S(u) &\le (1 + \sqrt{u})(c_1\sqrt{u} - c_2) + O(\sqrt{u}) \\ &= c_1\sqrt{u} + c_1 u - c_2 - c_2\sqrt{u} + O(\sqrt{u}) \\ &= c_1 u - \Big((c_2 - c_1 - O(1))\sqrt{u} + c_2\Big) \\ &\le c_1 u - c_2 , \end{aligned}$$

provided that c_2 is chosen to be larger than c_1 plus the hidden constant in the $O(1)$ term. The constant c_1 must be chosen large enough to satisfy the base case.

Second we prove that $S(u) \ge cu$ by the substitution method. Assume by induction that $S(k) \ge ck$ for all $k < u$. Then

$$\begin{aligned} S(u) &\ge (1 + \sqrt{u})c\sqrt{u} + O(\sqrt{u}) \\ &= c\sqrt{u} + cu + O(\sqrt{u}) \\ &\ge cu . \end{aligned}$$

The constant c must be chosen small enough to satisfy the base case.

(b) The algorithm is similar to vEB-INSERT. One main change is that the two cases are distinguished based on testing whether a particular key is stored in the hash table *sub*$[W]$. A second main change is that when the key is not in the hash table, a new widget is created using CREATE-WIDGET. We summarize with the pseudocode:

```
MODIFIED-INSERT(x, W)
 1  if W = NIL
 2     then W ← CREATE-WIDGET(x)
 3     else if x < min[W]
 4             then exchange x ↔ min[W]
 5          if the hash table sub[W] has an entry for key high(x)
 6             then MODIFIED-INSERT(low(x), sub[W][high(x)])
 7             else W′ ← CREATE-WIDGET(x)
 8                  insert into hash table sub[W] the subwidget W′ with key high(x)
                                          ▷ Sets sub[W][high(x)] ← W′
 9                  MODIFIED-INSERT(high(x), summary[W])
10          if x > max[W]
11             then max[W] ← x
```

(c) The algorithm is identical to vEB-SUCCESSOR, except that references to *sub*$[W][i]$ translate into searches in the hash table *sub*$[W]$ for key i.

(d) Each recursive call used to perform $O(1)$ instructions, and now additionally performs $O(1)$ additional hash-table operations. Thus, under the assumption of simple uniform hashing, the total cost goes up by an expected constant factor from the normal van Emde Boas structure.

(e) (*Note:* This solution actually requires some techniques from amortized analysis and dynamic hash tables, which we have not yet studied. Therefore this part will essentially be ignored during grading. Sorry for the confusion.) We prove that each widget by itself (ignoring its subwidgets and summary widgets) takes $O(1)$ space: we store a widget only if its *min* field is occupied by an element. Using dynamic hashing, the hash table increases the space by a constant factor (amortizing over the constant cost of each subwidget). Thus the space is $O(n)$.

Problem 6-2. Pebbling a checkerboard

(a) There are 8 possible patterns: the empty pattern, the 4 patterns which each have exactly one pebble, and the 3 patterns that have exactly two pebbles (on the first and fourth squares, the first and third squares, and the second and fourth squares).

(b) For each pattern, there are a constant number of patterns that are compatible with it (for example, every pattern is compatible with the empty pattern). Define $c_j[i]$ to be the optimal value achievable by pebbling columns $1, \ldots, i$ such that the final column has pattern j. Then for any j, $c_j[i+1]$ is the value of the squares covered by j in column $i+1$, plus the maximum value of $c_{j'}[i]$, subject to j' being compatible with j. (This claim can be proven by a standard "cut-and-paste" argument: if not, then replace the first i columns with a higher-valued pebbling that still ends in pattern j', then pebble column $i+1$ with pattern j to get a higher-valued pebbling than the original.) The base cases are $c_j[0] = 0$ for all column patterns j. Furthermore, $c_j[i+1]$ can be computed in $O(1)$ time by examining the $O(1)$ values $c_{j'}[i]$ and adding the values of the $O(1)$ squares pebbled by j.

From here, the dynamic programming algorithm is clear: keep 8 separate arrays (one for each column pattern) of n elements. For $i = 1, \ldots, n$, compute $c_j[i]$ for each of the 8 values of j as described above. To reconstruct the actual pebbling, find the maximum value c from $c_j[n]$, and pebble the nth column according to some j such that $c = c_j[n]$. Then subtract the value of the pebbled squares from c, and search for c among $c_j[n-1]$, etc. The running time of this algorithm is $O(n)$ because filling each of the (constant number of) arrays takes $O(n)$ time, and backtracking takes $O(1)$ time per column.

Note that it is *not* sufficient to keep only the the largest value achievable by pebbling the first i columns (irrespective of the pattern in its final column). This is because it might be possible to get very high value from the $(i+1)$st column, but only by using a pattern incompatible with the best pebbling of the first i columns.

Problem 6-3. Monotonically increasing subsequences

(a) We claim:

$$c[i] = 1 + \max_{\substack{1 \le k < i \\ x_k \le x_i}} c[k]$$

where the max is taken to be 0 if no indices k meet the two conditions. To see that this is true, first note that $c[i]$ is at least the given quantity: we can always append x_i to a monotonically increasing subsequence that ends in x_k if $k < i$ and $x_k \le x_i$. Conversely, we prove that $c[i] - 1$ is at most the specified maximum: suppose we have some longest monotonically increasing subsequence that ends in x_i. If we remove x_i, then it now ends in some x_k where $k < i$ and $x_k \le x_i$, and is still monotonically increasing. Therefore its new length ($c[i] - 1$) is at most the specified maximum.

(b) Using the recurrence from the previous part, we construct the values $c[1], \ldots, c[n]$ in a bottom-up fashion using dynamic programming. For each $i = 1, \ldots, n$, compute $c[i]$ using the recurrence (by making a pass over the already-computed values in the array, for each i). Then make a pass over c to find a maximum $c[i]$. From there, walk backward over the array to find values $c[i]-1, c[i]-2, \ldots, 1$ where each corresponding element is no larger than the previous one, and output all of these elements in reverse order of their discovery. (It is also possible to remember each value of k that yielded each $c[i]$, and backtrack directly to output the subsequence.)

The running time of this algorithm is as follows: each $c[i]$ requires $O(n)$ time to compute, because it may have to check up to n values in $c[1, \ldots, i-1]$. The backtracking algorithm takes only $O(n)$ time, because it makes one pass over the c array. Therefore the total running time is $O(n^2)$.

(c) We describe a $O(\log n)$-time way to compute $\max c[k]$ subject to $k < i$ and $x_k \le x_i$. This is done using an augmented red-black tree on the elements $x_1, \ldots, x_{i-1}$. The additional information kept at each node x_j is the maximum value of c corresponding to each node in the subtree rooted at x_j. This can be maintained, by Theorem 14.1 in CLRS, without affecting the runtime of the red-black tree operations, because it is the maximum of the auxiliary data at x_j's children and the value $c[j]$.

In the expression $\max c[k]$, two constraints must be met on k: $k < i$ and $x_k \le x_i$. The first constraint is met because the red-black tree will contain $x_1, \ldots, x_{i-1}$ (see the full algorithm, described below). We now describe how to consider exactly those $x_k \le x_i$: search for the last occurrence of x_i in the red-black tree (so that all of its successors are strictly greater than x_i), then look at the path taken by the search. Consider all "right-going" nodes in the path at which the search goes right, and the left children of those right-going nodes. Take the maximum of the right-going nodes' c values, and their left children's auxiliary data, as $\max c[k]$. We have argued before (in Problem 5-3) that these nodes and subtrees contain exactly those elements which are at most x_i. Because the auxiliary data for a node contains the maximum value of c over all elements in its subtree, taking the maximum over all appropriate nodes and subtrees

gives us $\max c[k]$ subject to $x_k \leq x_i$. The number of data elements examined is $O(\log n)$, because it is at most twice the length of the search path.

The entire algorithm is now as follows: for each $i = 1, \ldots, n$, compute $\max c[k]$ as described above and store it in $c[i]$, then insert x_i into the red-black tree and maintain the appropriate auxiliary data. After the loop (when $c[1\,..\,n]$ is completely computed), generate a longest monotonically increasing subsequence by making one pass over X and c in the same way as in the previous part (it is also possible to keep "back-pointers" to the k that yields the maximum value of c, for additional efficiency). For running time, each loop iteration requires $O(\log n)$ time, for a total of $O(n \log n)$. Generating the subsequence takes only $O(n)$ time.

Problem 6-4. Suppose, for the purposes of motivating our solution, that locations are allowed to have a negative number of cars; when a person picks up a car from a location, the number of cars at that location decreases by one, even if it is zero or negative. Now we note the following property: if a location starts with $c+1$ cars, then *at all times* it has exactly one more car than if it had started with only c cars.

This motivates the following algorithm: for each location, set its initial number of cars to zero. Also keep an array of the current number of cars at each location (so each element is initially zero as well). Now sort the pick-ups and drop-offs by time; in case of ties, put drop-offs first. "Simulate the day" by iterating over the sorted list of events. At a drop-off event, increase the number of cars currently at the specified location by one. At a pick-up event, do the following: if the number of cars currently at the specified location is positive, simply decrease that number by one; otherwise it is zero, so increase both the initial number of cars and the number of cars currently at that location by one, then decrease the number of cars currently at the location by one as usual. When all the events have been iterated over, return the array containing the initial numbers of cars at each location.

The algorithm is correct because of the greedy-choice property: whenever we decide to increment the initial number of cars for a particular location ℓ from c to $c+1$, our motivating fact implies that no feasible solution (i.e., one with which location ℓ always has at least 0 cars) puts fewer than than $c+1$ cars at location ℓ. Therefore an optimal solution has at least $c+1$ cars at location ℓ, and incrementing is safe. Because we only increment when it is safe, our final solution is optimal. It is also feasible because we passed through the entire day without facing a deficit of cars at any location.

The running time of the algorithm is $O(n \log n)$ because the algorithm sorts the events by time.