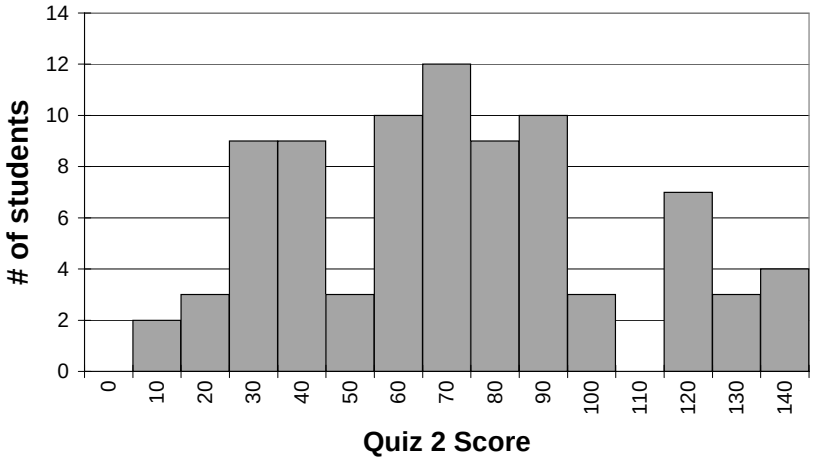
Introduction to Algorithms December 9, 2005
Massachusetts Institute of Technology 6.046J/18.410J
Professors Erik D. Demaine and Charles E. Leiserson Handout 33

Quiz 2 Solutions



Problem 1. Ups and downs

Moonlighting from his normal job at the National University of Technology, Professor Silvermeadow performs magic in nightclubs. The professor is developing the following card trick. A deck of n cards, labeled $1, 2, \ldots, n$, is arranged face up on a table. An audience member calls out a range $[i, j]$, and the professor flips over every card k such that $i \leq k \leq j$. This action is repeated many times, and during the sequence of actions, audience members also query the professor about whether particular cards are face up or face down. The trick is that there are no actual cards: the professor performs these manipulations in his head, and n is huge.

Unbeknownst to the audience, the professor uses a computational device to perform the manipulations, but the current implementation is too slow to work in real time. Help the professor by designing an efficient data structure that supports the following operations on n cards:

- FLIP(i, j): Flip over every card in the interval $[i, j]$.
- IS-FACE-UP(i): Return TRUE if card i is face up and FALSE if card i is face down.

Solution: Let F be the number of FLIP operations requested by the audience. Notice that performing a single flip FLIP(i, j) is equivalent to performing two flips, FLIP(i, ∞) and FLIP$(j + 1, \infty)$.

One fast solution is to use a dynamic order-statistic tree, where an element with key i represents a flip of the interval $[i, \infty)$. For both $x = i$ and $x = j + 1$, FLIP inserts x into the tree T if $x \notin T$, and deletes x from T if $x \in T$. IS-FACE-UP is implemented by returning true if the number of elements in the tree less than i (call it z) is even, and false if z is odd. One way to compute z is to insert i into T, set z to be one less than the rank of i, and then delete i from T. This data structure requires $O(\min\{F, n\})$ space and supports FLIP and IS-FACE-UP operations each in $O(\lg(\min\{F, n\}))$ time. A completely correct solution of this type received full credit.

The following list describes other types of solutions that students submitted and the approximate number of points awarded to each.

1. Another solution is to create a static 1-d range tree T containing the elements from 1 to n as its keys. Each node x in the tree stores a bit that corresponds to a flip of the interval of all elements in the subtree rooted at x. FLIP(i, j) performs a range query on $[i, j]$, and flips the stored bit for the $O(\lg n)$ disjoint subtrees that are found by the query. IS-FACE-UP(i) walks down the range tree to the node for i and returns true if the sum of the bits of nodes along the path from the root to i is even. This data structure uses $O(n)$ space and supports both operations in $O(\lg n)$ time. This type of solution received about 20 points.
2. Other students assumed that $F \ll n$, and stored each of the flip intervals $[i, j]$ in an interval tree. The IS-FACE-UP(i) procedure simply queries the interval tree and returns true if the number of intervals that overlap i is even. If the tree stores overlapping flip intervals, then FLIP and IS-FACE-UP run in $O(\lg F)$ time and $O(F \lg F)$ time, respectively. If we do not allow overlapping intervals in the tree, then FLIP runs in $O(F \lg F)$ time but IS-FACE-UP runs in $O(\lg F)$ time. This type of solution received about 13 points.
3. A simple slow solution is to use a bit-vector of length n to record which bits are flipped. This solution supports FLIP and IS-FACE-UP in $O(j - i)$ and $O(1)$ time, respectively. Similarly, another simple solution that stores all F flip-intervals in a linked list supports FLIP and IS-FACE-UP in $O(1)$ and $O(F)$ time, respectively. Either of these solutions received about 8 points.
4. Any solution that ran slower than the simple slow solution (for example, $O(\lg n + j - i)$ for FLIP) received about 5 points.

Problem 2. The Data Center

The world-famous architect Gary O'Frank has been commissioned to design a new building, called the Data Center. Gary wants his top architectural protégé to design a scale model of the Data Center using precision-cut sticks, but he wants to preclude the model from inadvertently containing any right angles. Gary fabricates a set of n sticks, labeled $1, 2, \ldots, n$, where stick i has length x_i. Before giving the sticks to the protégé, he shows them to you and asks you whether it is possible to create a right triangle using any three of the sticks. Give an efficient algorithm for determining whether there exist three sticks a, b, and c such that the triangle formed from them — having sides of lengths x_a, x_b, and x_c — is a right triangle (that is, $x_a^2 + x_b^2 = x_c^2$).

Solution: Let $X[1..n]$ be the array with stick sizes. We are asked to determine whether there exist distinct indices i, j, and k such that $X[i]^2 + X[j]^2 = X[k]^2$.

As an initial step we sort $X[1..n]$ in ascending order. This can be done in worst-case $O(n \lg n)$ time by for example heapsort. As soon as the array is sorted we first observe that it is sufficient to check $X[i]^2 + X[j]^2 = X[k]^2$ for indices i, j, and k with $i < j < k$.

In order to obtain a worst-case $O(n^2)$ algorithm, we increase k from 1 to n in an outerloop and we search in linear time in an innerloop for indices i and j, $i < j < k$, with $X[i]^2 + X[j]^2 = X[k]^2$.

```
 1  Sort X[1..n] in ascending order
 2  k ← 1
 3  while k ≤ n
 4       do  i, j ← 1, k − 1
 5           while i < j
 6                 if X[i]^2 + X[j]^2 = X[k]^2 then return true
 7                 if X[i]^2 + X[j]^2 < X[k]^2
 8                   then i ← i + 1
 9                   else  j ← j − 1
10  return false
```

We will prove the invariant:

$$[1 \leq i' < j' \leq k-1 \text{ and } X[i']^2 + X[j']^2 = X[k]^2] \text{ implies } [i \leq i' < j' \leq j].$$

Initially, $i = 1$ and $j = k - 1$ and the invariant holds. For the inductive step, assume that the invariant holds when we enter the innerloop. Notice that $X[1..n]$ is sorted in increasing order. If $X[i]^2 + X[j]^2 < X[k]^2$, then, for $i < j' \leq j$,

$$X[i]^2 + X[j']^2 \leq X[i]^2 + X[j]^2 < X[k]^2$$

and the invariant holds for $i \leftarrow i + 1$. If $X[i]^2 + X[j]^2 > X[k]^2$, then, for $i \leq i' < j$,

$$X[i']^2 + X[j]^2 \geq X[i]^2 + X[j]^2 > X[k]^2$$

and the invariant holds for $j \leftarrow j - 1$.

If the innerloop finishes, then $i = j$ and the invariant shows that there are no indices i' and j', $1 \leq i' < j' \leq k - 1$, such that $X[i']^2 + X[j']^2 = X[k]^2$. This proves the correctness of the algorithm. The innerloop has worst-case $O(n)$ running time, hence, together with the outerloop the algorithm runs in worst-case $O(n^2)$ time.

Problem 3. Nonnegativizing a matrix by pivoting

A matrix $M[1\,..\,n, 1\,..\,n]$ contains entries drawn from $\mathbb{R} \cup \{\infty\}$. Each row contains at most 10 finite values, some of which may be negative. The goal of the problem is to transform M so that every entry is nonnegative by using only ***pivot*** operations:

```
PIVOT(M, i, x)
1  for j ← 1 to n
2      do M[i, j] ← M[i, j] + x
3         M[j, i] ← M[j, i] − x
```

Give an efficient algorithm to determine whether there exists a sequence of pivot operations with various values for i and x such that, at the end of the sequence, $M[i, j] \geq 0$ for all $i, j = 1, 2, \ldots, n$.

Solution:

Suppose that we have a sequence $\langle \text{PIVOT}(M, i_1, x_1), \text{PIVOT}(M, i_2, x_2), \ldots, \text{PIVOT}(M, i_k, x_k)\rangle$ of pivots. Since each pivot operation simply adds to columns and subtracts from rows, by the associativity and commutativity of addition, the order of the pivot operations is irrelevant. Moreover, since $\text{PIVOT}(M, i_1, x_1)$ followed by $\text{PIVOT}(M, i_2, x_2)$ is equivalent to $\text{PIVOT}(M, i_1, x_1 + x_2)$, there is no need to pivot more than once on any index. Thus, any sequence of pivots is equivalent to a sequence of exactly n pivots $\langle \text{PIVOT}(M, 1, x_1), \text{PIVOT}(M, 2, x_2), \ldots, \text{PIVOT}(M, n, x_n)\rangle$, where some of the x_i may be 0.

The only pivots that affect a matrix entry $M[i, j]$ are $\text{PIVOT}(M, i, x_i)$ and $\text{PIVOT}(M, j, x_j)$. Thus, after the entire sequence of pivots has been performed, the resulting matrix M' has entries $M'[i, j] = M[i, j] + x_i - x_j$. We want every entry $M'[i, j]$ to be nonnegative, which is to say that $M[i, j] + x_i - x_j \geq 0$, or $x_j - x_i \leq M[i, j]$. Thus, we only need to solve a set of difference constraints. We can ignore the difference constraints where $M[i, j] = \infty$, which leaves at most $10n$ difference contraints in n variables. It takes us $O(n^2)$ time to find the at-most $10n$ finite entries, and we can use the Bellman-Ford algorithm [CLRS, Section 24.4] to solve them in $O(n \cdot (10n)) = O(n^2)$ time, for a total of $O(n^2)$ time.

Problem 4. Augmenting the Queueinator™

By applying his research in warm fission, Professor Uriah's company is now manufacturing and selling the Queueinator™, a priority-queue hardware device which can be connected to an ordinary computer and which effectively supports the priority-queue operations INSERT and EXTRACT-MIN in $O(1)$ time per operation. The professor's company has a customer, however, who actually needs a "double-ended" priority queue that supports not only the operations INSERT and EXTRACT-MIN, but also EXTRACT-MAX. Redesigning the Queueinator™ hardware to support the extra operation will take the professor's company a year of development. Help the professor by designing an efficient double-ended priority queue using software and one or more Queueinator™ devices.

Solution:

Keep two Queueinators, MIN and MAX, where MAX has the keys negated, of roughly equal size and keep an element mid which is the smallest element in MAX. While inserting an element with key k, compare it to mid and insert it in MIN if $k < mid$ and into MAX otherwise. For EXTRACT-MIN, extract from MIN and for EXTRACT-MAX, extract from MAX. If one empties, extract all the elements from the other into a sorted array. Split the array into half and insert the smaller half into MIN and the larger half into MAX. This solution provides $O(1)$ amortized cost for all operations.

Proof of Correctness: The invariant is that all the elements in MIN are smaller than mid and all the elements in MAX are larger than or equal to mid. The other invariant is that MIN and MAX together contain all the elements that have been inserted and not yet been extracted. The invariants are maintained during all operations. Therefore, extractions return the correct results.

Running time analysis: The potential function is $\Phi(i) = 2\,||MIN| - |MAX||$, where $|MIN|$ and $|MAX|$ are the number of elements in the MIN and MAX queueinators respectively. Both MIN and MAX are empty initially, and the potential is 0. The potential obviously never becomes negative. Let us look at the amortized costs.

1. INSERT Inserts into MIN or MAX. The real cost is 1 and the potential either increases by 2 (if you inserted into the bigger queue) or decreases by 1 (if you insert into the smaller queue). Thus the cost is always $O(1)$.
2. EXTRACT-MIN If the MIN is non-empty, then the real cost is 1 and the potential either increases or decreases by 2. If MIN is empty, then you have to rebalance the queues. The potential before the rebalance is equal to $2 *$ the number of elements in MAX, and the potential after the rebalance is either 0 (or 2 if the number of elements is odd, but that doesn't change much). The real cost of rebalancing is $2 *$ the number of elements in MAX, since all the elements are extracted and then inserted again. Thus the change in potential pays for the rebalancing, and the amortized cost is $O(1)$.
3. EXTRACT-MAX It is analogous to EXTRACT-MIN.

Therefore the amortized cost of the operations is $0(1)$.

There were other good solutions, some of them are given below

1. Use 4 queueinators where 2 of them to keep track of the deleted elements. Those solutions lost a couple of points due to the use of the extra hardware.
2. Use extra fields in the items to keep track of deletions. These also lost a couple of points due to extra assumptions that you can change the elements.
3. Use hash tables to keep track of deleted elements. This only provides expected $O(1)$ amortized time, and assumes that the keys are integers in a range. Therefore, they lost about 5 points.
4. Use direct access tables to keep track of deleted elements, lost about 6-7 points due to the extra space.

The analysis was very important for the solutions and the solutions that had incomplete or incorrect running time analysis lost points for that. In addition, a convincing correctness argument was required for full credit. People who gave the correct algorithm but did not claim or prove amortized bounds lost many points.

Problem 5. Spam distribution

Professor Hormel is designing a spam distribution network. The network is represented by a rooted tree $T = (V, E)$ with root $r \in V$ and nonnegative edge-weight function $w : E \to \mathbb{R}$. Each vertex $v \in V$ represents a server with one million email addresses, and each edge $e \in E$ represents a communication channel that costs $w(e)$ dollars to purchase. A server $v \in V$ receives spam precisely if the entire path from the root r to v is purchased. The professor wants to send spam from the root r to $k \leq |V|$ servers (including the root) by spending as little money as possible. Help the professor by designing an algorithm that finds a minimum-weight connected subtree of T with k vertices including the root. (For partial credit, solve the problem when each vertex $v \in V$ has at most 2 children in T.)

Solution:

Executive Overview. The solution to this problem is based on dynamic programming. We define $k|V|$ subproblems, one for each combination of a vertex $v \in V$ and a number $k' \leq k$. Each subproblem is of the form "Find the minimum-weight connected subtree of rooted at v with k' vertices." Each subproblem can be solved in $O(k)$ time, resulting in a running time of $O(k^2|V|)$.

Notation. Throughout this solution, we use the following notation:

- MWCT(v,i): a minimum-weight connected subtree of i nodes rooted at v
- *child*(v,i): child i of node v
- $w(v, i)$: the weight of the edge from v to child i
- $W(S)$: the sum of the weights of all the edges in tree S
- *deg*(v): the number of children of node v

Optimal Substructure. This problem exhibits both optimal substructure and overlapping subproblems. For intuition, consider a binary tree T. Let v be a node in the tree, and consider a MWCT of T rooted at v with ℓ nodes. If T_1 is the left subtree of v and has ℓ_1 nodes, then T_1 is a MWCT rooted at v's left child with ℓ_1 nodes. Similarly, if T_2 is the right subtree of v and has ℓ_2 nodes, then T_2 is a MWCT rooted at v's right child with ℓ_2 nodes. The proof follows by a cut-and-paste argument. Assume that T_1 is *not* a MWCT rooted at v's left child. Then there is another tree, S, rooted at v's left child with ℓ_1 nodes and with less weight than T_1. We can then reduce the weight of the tree rooted at v by substituting S for tree T_1, contradicting our assumption that the tree rooted at v is a minimum-weight connected subtree with k nodes.

We can now observe how to use the overlapping subproblems. Once we have calculated the MWCT(v, ℓ) for all $v \in V$ and all $\ell \leq k'$, we can determine the MWCT$(v, k' + 1)$. In particular, we choose subtrees rooted at v's children that contain k' nodes in total. That is, if there are ℓ nodes in the left subtree of v, then there must be $k' - \ell$ nodes in the right subtree. We choose ℓ to

split the tree between v's left and right children to minimize the total weight. We now proceed to generalize this to arbitrary degree trees, and explain the algorithm in more detail.

Algorithm Description. Let $T[v]$ be the subtree of T rooted at v. Let $T[v, c]$ be the subtree of T rooted at v consisting of v and the trees rooted at children $1 \ldots c$. More formally, $T[v, c] = v \cup \{T[w] : w = \textit{child}v, i, 1 \leq i \leq c\}$.

We create two $|V| \times k \times |V|$ arrays: $C[1 \ldots |V|, 1 \ldots k, 1 \ldots |V|]$ and $B[1 \ldots |V|, 1 \ldots k], 1 \ldots |V|$. $C[v, k', c]$ holds the cost of the minimum-weight connected subtree of $T[v, c]$ with k' nodes. The array B is used to reconstruct the tree after the dynamic program has terminated. $B[v, k', c]$ holds the number of children in the subtree of the MWCT of $T[v, c]$ with k' nodes rooted at child c.

Notice that the resulting arrays are three-dimensional tables of size $k|V|^2$. Fortunately, we will only have to fill in $k|V|$ of the entries, leaving the rest empty/uninitialized, as we will see when the algorithm is analyzed. (For simplicity, we assume that arbitrary-sized memory blocks can be allocated in $O(1)$ time. If memory allocation is more costly, we can reduce the memory usage to $O(k|V|)$.)

The main procedure to calculate the MWCT of V with k nodes is as follows:

```
MWCT(V, k)
 1  for all v ∈ V
 2      do  for c = 1 to deg(v)
 3              do  C[v, 1, c] ← 0
 4                  B[v, 1, c] ← 0
 5  for ℓ = 2 to k
 6      do  for all v ∈ V
 7              do  w ← child(v, 1)
 8                  C[v, ℓ, 1] ← w(v, w) + C[w, ℓ − 1, deg(w)]
 9                  B[v, ℓ, 1] ← ℓ − 1
10                  for c = 2 to deg(v)
11                      do  i ← FIND-NUM-CHILDREN(v, ℓ, c)
12                          B[v, ℓ, c] ← i
13                          if i = 0 then C[v, ℓ, c] ← C[v, ℓ, c − 1]
14                          if i = ℓ then C[v, ℓ, c] ← w(v, w) + C[w, ℓ − 1, deg(w)]
15                          if 0 < i < ℓ then C[v, ℓ, c] ← C[v, ℓ − i] + w(v, w) + C[w, i, deg(w)]
```

We begin in lines 1–4 by initializing $C[v, 1, c] = 0$ for all $v \in V$ and all $c \leq |V|$. These are the smallest subproblems considered: the one node minimum-weight connected subtree rooted at v consists simply of v itself, and hence has cost zero.

We proceed with three nested loops to fill in the arrays. In the outer loop (line 5) , we iterate ℓ from 2 to k. In the second loop (line 6), we iterate v over all nodes in V. In the inner loop (line 10), we iterate c from 2 to $\textit{deg}(v)$.

Within the loop, we are examining the subtree $T[v, c]$ and the node $w = \textit{child}(v, c)$. Let S be the MWCT of $T[v, c]$ containing ℓ nodes. We want to calculate $C[v, \ell, c]$, the cost of S.

First consider the case where $c = 1$. In this case, all the nodes in S are in the subtree rooted at w. Therefore, S consists of node v and the MWCT of w with $\ell - 1$ nodes.

Now consider the case where $c \geq 2$. Notice that some of the nodes in S may be in subtrees rooted at the first $c - 1$ children of v, and some may be in the subtree rooted at w. The function FIND-NUM-CHILDREN(v, ℓ, c) calculates the number of nodes in S that are in the subtree rooted at w; the remaining $\ell - i - 1$ are in the subtrees rooted at the first $c - 1$ children of v.

It is then easy to determine the cost of S. If no nodes in S are in the subtree rooted at w, then S is just the MWCT of $T[v, c-1]$ with ℓ nodes (line 11). If all the nodes in S are in the subtree rooted at w, then S consists of node v along with the MWCT of w with $\ell - 1$ nodes (line 12). Finally, if i is between 0 and ℓ, then S consists of the MWCT of $T[v, c-1]$ with $\ell - i$ nodes, plus the MWCT of w with i nodes (line 13).

It remains to discuss the FIND-NUM-CHILDREN procedure:

```
FIND-NUM-CHILDREN(v, ℓ, c)
 1  w ← child(v, c)
 2  min-weight ← C[v, ℓ, c − 1]
 3  num ← 0
 4  for i = 1 to ℓ − 2
 5       do wt ← C[v, ℓ − i, c − 1] + w(v, c) + C[w, i, deg(w)]
 6          if wt < min-weight
 7            then min-weight ← wt
 8                 num ← i
 9  if min-weight > w(v, w) + C[w, ℓ − 1, deg(w)]
10    then num ← ℓ
11  return num
```

The FIND-NUM-CHILDREN procedure compares all possible ways of dividing ℓ nodes between $T[v, c-1]$ and the subtree rooted at $w = \textit{child}(v, c)$. First, it considers the case where all ℓ children are in $T[v, c - 1]$ and zero children are in the tree rooted at w (line 2–3). Next, it iterates through the loop $\ell - 2$ times, placing $\ell - i$ nodes in $T[v, c-1]$ and i nodes in the subtree rooted at w (lines 4–8). Finally, it considers the case where $\ell - 1$ children are in the subtree rooted at w (lines 9–10). It then returns the choice which minimizes the weight.

The final procedure in the algorithms prints out the tree, performing a depth-first traversal of the resulting tree:

```
PRINT-TREE(v, k)
1  PRINT(v)
2  ℓ ← k
3  for c = deg(v) downto 1
4       do  if ℓ > 0 and B[v, ℓ, c] ≠ 0
5             then w ← child(v, c)
6                  PRINT-TREE(v, B[v, ℓ, c])
7                  ℓ ← ℓ − B[v, ℓ, c]
```

The PRINT-TREE procedure starts with the last child of v, and examines $B[v, \ell, c]$ to determine the number of nodes in the subtree root at child c. It then recurses into that subtree, and updates the number of nodes remaining in the MWCT.

Running Time. First, examine lines 1–4 of MWCT: the two loops iterate over every edge of every node, resulting on $O(|E|) = O(|V|)$ operations. Next, the for-loop on line 5 repeats $O(k)$ time. Now we examine lines 6–15. The innermost loop executes once for every edge of every node, i.e., $O(|E|) = O(|V|)$ times. Each iteration of the loop calls FIND-NUM-CHILDREN(v, ℓ, c) which takes $O(\ell) = O(k)$ time. Therefore each iteration of lines 6–15 costs $O(k|V|)$. Therefore the entire cost of MWCT is $O(k^2|V|)$. Finally, printing the output costs $O(|V|)$.

Correctness. The correctness follows by induction on k using a cut-and-paste argument. The loop invariant maintained by the innermost loop is that for all $c' \leq c$, $C[v, \ell, c']$ is equal to the cost of the MWCT of $T[v, c']$. The loop invariant of the outer loop is that for all v', for all $\ell' \leq \ell$, for all $c' \leq \textit{deg}(v)$, $C[v, \ell', c']$ is equal to the cost of the MWCT of $T[v', c']$ with ℓ' nodes. These two invariants need to be proved together by induction. (We omit mention of the B array here, though its correctness follows by the same argument as follows.)

Consider immediately after the innermost loop has terminated for some v, ℓ, and c. We argue that $C[v, \ell, c]$ is the cost of the MWCT of $T[v, c]$. Assume not. Then there is some other subtree S of $T[v, c]$ with ℓ nodes and cost $< C[v, \ell, c]$. Consider the subtree of S rooted at child c of v with i nodes. (S may be the empty tree.) We know that the cost of this subtree must be at least $x = C[\textit{child}(v, c), i, \textit{deg}(\textit{child}(v, c))]$, since by induction x is the MWCT of child c of v with i nodes. Similarly, consider the subtree of S consisting of all the nodes in $T[v, c-1]$. Again, the cost of this subtree must be at least $y = C[v, \ell - i, c-1]$, since by induction y is the MWCT of $T[v, c-1]$ with $\ell - i$ nodes. But FIND-MIN-CHILDREN ensures that the $C[v, \ell, c]$ is no more than $x + y + w(v, c)$, contradicting our assumption. Finally, the inductive step of the outer loop invariant follows from the inner loop invariant, concluding the proof.

Problem 6. A tomato a day

Professor Kerry loves tomatoes! The professor eats one tomato every day, because she is obsessed with the health benefits of the potent antioxidant lycopene and because she just happens to like them very much, thank you. The price of tomatoes rises and falls during the year, and when the price of tomatoes is low, the professor would naturally like to buy as many tomatoes as she can. Because tomatoes have a shelf-life of only d days, however, she must eat a tomato bought on day i on some day j in the range $i \leq j < i + d$, or else the tomato will spoil and be wasted. Thus, although the professor can buy as many tomatoes as she wants on any given day, because she consumes only one tomato per day, she must be circumspect about purchasing too many, even if the price is low.

The professor's obsession has led her to worry about whether she is spending too much money on tomatoes. She has obtained historical pricing data for n days, and she knows how much she actually spent on those days. The historical data consists of an array $C[1\,..\,n]$, where $C[i]$ is the price of a tomato on day i. She would like to analyze the historical data to determine what is the minimum amount she could possibly have spent in order to satisfy her tomato-a-day habit, and then she will compare that value to what she actually spent.

Give an efficient algorithm to determine the optimal offline (20/20 hindsight) purchasing strategy on the historical data. Given d, n, and $C[1\,..\,n]$, your algorithm should output $B[1\,..\,n]$, where $B[i]$ is the number of tomatoes to buy on day i.

Solution:

You can solve this problem in $\Theta(n)$ time, which is the fastest possible asymptotic solution because it requires $\Omega(n)$ time to scan the cost array. For every day i, you must buy the tomato for that day in the window $W(i) \equiv [i - d + 1, i] \cap [1, n]$. Note that the greedy choice works: in the optimal solution, you should buy the tomato for day i on the minimum cost day T_i in $W(i)$. Once you calculate T_i for all i, you can easily compute B in $\Theta(n)$ time.

You can compute all T_i in $O(n)$ time by computing the sliding window minimum T_i for each i in amortized $O(1)$ time. Use a double-ended queue (deque) of days $Q = \langle q_1, q_2, \ldots, q_m \rangle$ such that day q_1 is the minimum cost day in $W(i)$ and each entry $q_{k>1}$ is the minimum cost day of the days in $W(i)$ following q_{k-1}. Then $q_1 = T_i$ and this invariant can be maintained in amortized $O(1)$ time using the following algorithm:

```
    BUYTOMATOES(d, C[1..n])
 1  Q ← ∅
 2  B[1..n] ← 0
 3  for i ← 1 to n
 4       do  if not EMPTY(Q) and HEAD(Q) ≤ i − d
 5             then POP-FRONT(Q)          ▷ Remove old day
 6           while not EMPTY and C[i] ≤ C[TAIL(Q)]
 7              do  POP-BACK(Q)           ▷ Remove non-minimum-cost days
 8           PUSH-BACK(Q, i)              ▷ Add current day
 9           B[HEAD(Q)] ← B[HEAD(Q)] + 1
10  return B
```

Each day i is added once to Q and removed at most once from Q, so the total number of deque operations is $O(n)$. Each deque operation can be done in $O(1)$ time. The loop in the algorithm executes for n iterations, so the overall running time is $\Theta(n)$. The deque requires $O(\min(n, d))$ space since there can be at most one element for each of the past d days. The return array B requires $\Theta(n)$ space, so the overall space used is $\Theta(n)$.

There are other ways to compute T_i in amortized $O(1)$ time that also received full credit. Slower solutions that received partial credit include solutions that compute T_i in $O(\log d)$ or $O(d)$ time, solutions that compute B directly by looking ahead up to d days at each step, dynamic programming solutions, and solutions that sort the tomato prices and maximize the number of tomatoes purchased at lower prices.