Introduction to Algorithms October 24, 2004
Massachusetts Institute of Technology 6.046J/18.410J
Professors Piotr Indyk and Charles E. Leiserson Handout 18

Problem Set 4 Solutions

Reading: Chapters 17, 21.1–21.3

Both exercises and problems should be solved, but *only the problems* should be turned in. Exercises are intended to help you master the course material. Even though you should not turn in the exercise solutions, you are responsible for material covered in the exercises.

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.

Three-hole punch your paper on submissions.

You will often be called upon to "give an algorithm" to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of the essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudo-code.
2. At least one worked example or diagram to show more precisely how your algorithm works.
3. A proof (or indication) of the correctness of the algorithm.
4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Full credit will be given only to correct algorithms which are *which are described clearly*. Convoluted and obtuse descriptions will receive low marks.

Exercise 4-1. The Ski Rental Problem

A father decides to start taking his young daughter to go skiing once a week. The daughter may lose interest in the enterprise of skiing at any moment, so the kth week of skiing may be the last, for *any* k. Note that k is *unknown*.

The father now has to decide how to procure skis for his daughter for every weekly session (until she quits). One can *buy* skis at a one-time cost of B dollars, or *rent* skis at a weekly cost of R dollars. (Note that one can buy skis at any time—e.g., rent for two weeks, then buy.)

Give a 2-competitive algorithm for this problem—that is, give an online algorithm that incurs a total cost of at most twice the offline optimal (i.e., the optimal scheme if k is known).

Problem 4-1. Queues as Stacks

Suppose we had code lying around that implemented a stack, and we now wanted to implement a queue. One way to do this is to use two stacks S_1 and S_2. To insert into our queue, we push into stack S_1. To remove from our queue we first check if S_2 is empty, and if so, we "dump" S_1 into S_2 (that is, we pop each element from S_1 and push it immediately onto S_2). Then we pop from S_2.

For instance, if we execute INSERT(a), INSERT(b), DELETE(), the results are:

	$S_1 =$`[]`	$S_2 =$`[]`	
INSERT(a)	$S_1 =$`[a]`	$S_2 =$`[]`	
INSERT(b)	$S_1 =$`[b a]`	$S_2 =$`[]`	
DELETE()	$S_1 =$`[]`	$S_2 =$`[a b]`	"dump"
	$S_1 =$`[]`	$S_2 =$`[b]`	"pop" (returns a)

Suppose each push and pop costs 1 unit of work, so that performing a dump when S_1 has n elements costs $2n$ units (since we do n pushes and n pops).

(a) Suppose that (starting from an empty queue) we do 3 insertions, then 2 removals, then 3 more insertions, and then 2 more removals. What is the total cost of these 10 operations, and how many elements are in each stack at the end?

Solution: The total work is $3 + (6 + 2) + 3 + (1 + 6 + 1) = 22$. At the end, S_1 has 0 elements, and S_2 has 2.

(b) If a total of n insertions and n removals are done in some order, how large might the running time of one of the operations be (give an exact, non-asymptotic answer)? Give a sequence of operations that induces this behavior, and indicate which operation has the running time you specified.

Solution: An insertion always takes 1 unit, so our worst-case cost must be caused by a removal. No more that n elements can ever be in S_1, and no fewer than 0 elements can be in S_2. Therefore the worst-case cost is $2n + 1$: $2n$ units to dump, and one extra to pop from S_2. This bound is tight, as seen by the following sequence: perform n insertions, then n removals. The first removal will cause a dump of n elements plus a pop, for $2n + 1$ work.

(c) Suppose we perform an arbitrary sequence of insertions and removals, starting from an empty queue. What is the amortized cost of each operation? Give as tight (i.e., non-asymptotic) of an upper bound as you can. Use the accounting method to prove your answer. That is, charge \$$x$ for insertion and \$$y$ for deletion. What are x and y? Prove your answer.

Solution: The tightest amortized upper bounds are 3 units per insertion, and 1 unit per removal. We will prove this 2 ways (using the accounting and potential methods; the aggregate method seems too weak to employ elegantly in this case). (We would also accept valid proofs of 4 units per insertion and 0 per removal, although this answer is looser than the one we give here.)

Here is an analysis using the accounting method: with every insertion we pay \$3: \$1 is used to push onto S_1, and the remaining \$2 remain attached to the element just inserted. Therefore every element in S_1 has \$2 attached to it. With every removal we pay \$1, which will (eventually) be used to pop the desired element off of S_2. Before that, however, we may need to dump S_1 into S_2; this involves popping each element off of S_1 and pushing it onto S_2. We can pay for these pairs of operations with the \$2 attached to each element in S_1.

(d) Now we'll analyze the structure using the potential method. For a queue Q implemented as stacks S_1 and S_2, consider the potential function

$$\Phi(Q) = \text{number of elements in stack } S_1.$$

Use this potential function to analyze the amortized cost of insert and delete operations.

Solution: Let $|S_1^i|$ denote the number of elements in S_1 after the ith operation. Then the potential function Φ on our structure Q_i (the state of the queue after the ith operation) is defined to be $\Phi(Q_i) = 2|S_1^i|$. Note that $|S_1^i| \geq 0$ at all times, so $\Phi(Q_i) \geq 0$. Also, $|S_1^0| = 0$ initially, so $\Phi(Q_0) = 0$ as desired.
Now we compute the amortized costs: for an insertion, we have $S_1^{i+1} = S_1^i + 1$, and the actual cost $c_i = 1$, so

$$\hat{c}_i = c_i + \Phi(Q_{i+1}) - \Phi(Q_i) = 1 + 2(S_1^i + 1) - 2(S_1^i) = 3.$$

For a removal, we have two cases. First, when there is no dump from S_1 to S_2, the actual cost is 1, and $S_1^{i+1} = S_1^i$. Therefore $\hat{c}_i = 1$. When there is a dump, the actual cost is $2|S_1^i| + 1$, and we have $S_1^{i+1} = 0$. Therefore we get

$$\hat{c}_i = (2|S_1^i| + 1) + 0 - 2|S_1^i| = 1$$

as desired.

Problem 4-2. David Digs Donuts

Your TA David has two loves in life: (1) roaming around Massachusetts on his forest-green Cannondale R300 road bike, and (2) eating Boston Kreme donuts. One Sunday afternoon, he is biking along Main Street in Acton, and suddenly turns the corner onto Mass Ave. (Yes, that Mass Ave.) His growling stomach announces that it is time for a donut. Because Mass Ave has so many donut shops along it, David decides to find a shop somewhere along that street. He faces two obstacles in his quest to satisfy his hunger: first, he does not know whether the nearest donut shop is to his left or to his right (or how far away the nearest shop is); and second, when he goes riding his contact lenses dry out dramatically, blurring his vision, and he can't see a donut shop until he is directly in front of it.

You may assume that all donut shops are at an integral distance (in feet) from the starting location.

(a) Give an efficient (deterministic) algorithm for David to locate a donut shop on Mass Ave as quickly as possible. Your algorithm will be online in the sense that the location of the nearest donut shop is unknown until you actually find the shop. The algorithm should be $O(1)$-competitive: if the nearest donut shop is distance d away from David's starting point, the total distance that David has to bike before he gets his donut should be $O(d)$. (The optimal offline algorithm would require David to bike only distance d.)

Solution: WLOG, let's call the two directions of Mass Ave "east" and "west."

```
1. Check for a shop at the origin.
2. $i := 0$.
3. $direction := east$;
4. Repeat the following until a donut is found:
   (a) Bike $2^i$ units in direction $direction$. If you pass a donut shop, stop and eat.
   (b) Bike $2^i$ units back to the origin.
   (c) $i := i + 1$.
   (d) $direction := -direction$.
```

Notice that you are back at the origin after every iteration of the loop.

Suppose that the nearest donut shop is d feet away from the origin. Let k be such that $2^k < d \leq 2^{k+1}$. Observe that in the ith iteration of the loop, we explore the stretch of road between the origin and $\pm 2^i$, so after $k+2$ iterations we are guaranteed to have found the shop.

The total distance that we travel in the ith iteration is $2 \cdot 2^i$, so the total distance traveled in these $k+2$ iterations is $\sum_{i=0}^{k+2} 2^{i+1} \leq 2 \cdot 2^{k+3} = 16 \cdot 2^k < 16d$. Thus the algorithm is 16-competitive.

(b) Optimize the competitive ratio for your algorithm—that is, minimize the constant hidden by the $O(\cdot)$ in the competitive ratio.

Solution: The only tweak to the above is to more tightly analyze the last iteration. In the $(k+2)$nd iteration, we are done after we travel distance d, since we were at the origin and had to travel only distance d. Thus the total distance we travel is $\sum_{i=0}^{k+1} 2^{i+1} + d \leq 8 \cdot 2^k + d \leq 9d$. (One can also try to optimize the base of the exponential search, but it turns out that two is optimal.)

(c) Suppose you flip a coin to decide whether to start moving to the left or to the right initially. Show that incorporating this step into your algorithm results in an improvement to the expected competitive ratio.

Solution: Then, using the above notation, with probability $1/2$ we will find the shop in the $(k+1)$st iteration, and with probability $1/2$ we will find it in the $(k+2)$nd iteration. Thus the expected travel distance is

$$(\sum_{i=0}^{k+1} 2^k + d)/2 + (\sum_{i=0}^{k+2} 2^k + d)/2 \leq (5d + 9d)/2 = 7d.$$