Massachusetts Institute of Technology
6.854J/18.415J: Advanced Algorithms
David Karger

Handout 23
Wednesday, November 14, 2012

Problem Set 10 Solutions

Problem 1. **(a)** Suppose we have a sequence with at most k distinct items. Suppose for the sake of contradiction that there were more than k page faults. Then by Pigeonhole, some item i gets faulted on twice. We show that in either case (LRU or FIFO), we get a contradiction:

- *LRU*
  If i is faulted on twice, then it must be used at some point t_1 in the sequence, then get evicted later and then used again at t_2. When it is used at t_1, is will be the most; then by the time we reach t_2, it cannot drop below the k^{th} most recently used. Therefore it could not have been evicted.
- *FIFO*
  Choose i such that it is faulted on twice, and both of these faults are among the first $k+1$ faults that occur in the sequence of pages. If i is faulted on twice, then, once again, it must be used at some point t_1 in the sequence, get evicted, and then used again at t_2. But, when it is faulted on at t_1 it is the "last one in". By the FIFO rule, i will be evicted on the k^{th} fault afterwards. But there are not k faults between the two faults of i which we are considering, a contradiction.

(b) We show that any conservative algorithm is k-competitive. Let $\mathcal{A}$ be a conservative online algorithm, and let $\mathcal{OPT}$ be the optimal algorithm for the given page sequence.

Assume that $\mathcal{OPT}$ begins by faulting. (We can make this assumption in a few ways. First of all, we might assume that the cache starts out empty, in which case any algorithm has to fault at the beginning. Alternatively, we could assume that the cache starts in an arbitrary state. Then, if $\mathcal{OPT}$ avoids faulting for some number of page requests, then all of those pages must have been in the cache to begin with. But then, $\mathcal{A}$ would not fault on any of them either. Thus, we may discount these beginning page requests for our analysis.)

Divide the sequence of page requests into "phases," defined as follows. The i^{th} phase P_i begins at the page which is the i^{th} fault that $\mathcal{OPT}$ makes, and ends right before the $(i+1)^{\text{th}}$ fault of $\mathcal{OPT}$ (or goes until the end of the sequence of requests, if $\mathcal{OPT}$ only makes i faults).

Suppose that $\mathcal{OPT}$ makes m faults; then there are m phases. Consider any particular phase. We claim that it has at most k distinct pages. Indeed, if it had $k+1$ or more distinct pages, then $\mathcal{OPT}$ would not be able to service all of these requests without making another fault.

Now, since $\mathcal{A}$ is conservative, by part (a) we get that it makes at most k page faults on any particular phase. Thus it makes at most mk page faults total. Therefore, $\mathcal{A}$ is k-competitive.

Problem 2. **(a)** Let our potential function be

$$\phi = kM_{min} + \sum_{DC} = kM_{min} + S,$$

where $S = \sum_{DC} = \sum_{i<j} d(s_i, s_j)$. Break up the DC-TREE algorithm into phases, each composed of several steps. Let a phase be the time it takes for a server to get to the place where the request occurred, and let a step be the interval where both OPT and DC-TREE gets to move each server exactly one unit. By comparing the cost of OPT to DC-TREE step by step, we show that DC-TREE is k-competitive.

First, let OPT move first. OPT never moves more than one server. If OPT moves a server s_i by one unit, it doesn't change $\sum_{DC}$. In the worst case, it increases M_{min} by one unit because s_i could get farther away from its match. Therefore $\Delta\phi \leq k$.

Now consider the corresponding step of DC-TREE. Suppose in this step, α servers move towards the request.

- If α servers are moving, $k - \alpha$ servers are blocked. Look at any server t that is blocked by some server s_i. The $\alpha - 1$ servers $s_j \neq s_i$ that move in this step must get one unit closer to t. The server s_i is getting one unit farther from t. The change in $\sum_{DC}$ due to changes in distance between a moving server and a non-moving server is $-(k-\alpha)(\alpha-2)$.
- Each of the moving servers s_i are not yet at x, but moving towards it. Therefore, they all get two units closer to each other. This changes $\sum_{DC}$ by $-2\binom{\alpha}{2} = -\alpha(\alpha - 1)$.
- If α servers move, all but one of them could be moving away from its optimal matching. One moving server must be getting closer to its optimum: the one server that was matched with the server moved by OPT, because they are both moving towards the request x. This means M_{min} increases by at most $\alpha - 2$.

On a given step, when OPT moves,

$$\Delta\phi_{OPT} \leq k.$$

On that same step, if DC-TREE moves α servers, the total change in potential is

$$\begin{aligned} \Delta\phi_{DC} &\leq k(\alpha-2) - (k-\alpha)(\alpha-2) - \alpha(\alpha-1) \\ &= \alpha(\alpha-2) - \alpha(\alpha-1) \\ &= -\alpha \end{aligned}$$

By our accounting, when OPT moves (one step), it increases the potential by at most k. When DC-TREE moves a distance α (on one step), it decreases the potential by at most α. As the problem we did in class, we find that the total decrease in potential is less than the total increase ($\alpha \leq k$). This means DC-TREE's work is less than or equal to k times OPT's work. Therefore, DC-TREE is k-competitive.

(b) If there are n pages in memory, consider the simple tree that has one root node connected to n nodes. We start the k servers on k of the leaf nodes. The position of each server tells us the initial pages that are in memory. A request for a page causes (at least) one of the k servers to move from the page it is on to the new page. If a server moves off a leaf node, but onto the root node, we interpret this as still keeping whatever page corresponded to the leaf node we were on before in memory. In other words, a server being at the root means the last page that server visited is still in memory. We only bring in a new page when we move the server onto a leaf node (different than before).

(c) If we start the k servers on k different pages, and applying the DC-TREE algorithm, then we actually get a version of the marking algorithm for paging. A marked page corresponds to a page with a server at the leaf node for that page, and an unmarked page corresponds to having the server at the root node.

If all pages are marked, then on a fault, we unmark all the pages because DC-TREE will move all servers to the root. Then, (assuming DC-TREE picks one random server when there are multiple servers at the exact same place), one page gets evicted when we move a server onto the leaf. In general, if there are servers at the root, (i.e. unmarked pages), then DC-TREE will move only one of those servers. This corresponds to only kicking out an unmarked page.

Problem 3. **(a)** Let A be any deterministic algorithm. The execution of A proceeds in rounds: in the ith round the algorithm picks a date d_i and learns its ordering with respect to the previous $i-1$ dates. At this point, it chooses to select d_i or continue execution.

We construct an adversary that gives the algorithm a date d in each round with the property that the relative rank of the dates decreases in each round. Suppose that the algorithm accepts at step i; then by construction d_i is of worse rank than the previous $i-1$ dates. Furthermore, since A has not been able to examine the last $k-i$ dates, the adversary can specify these to all be of greater absolute rank than the i dates A has examined.

Note that this lower bound strategy cannot work against a randomized algorithm – this is because the adversary cannot see the coin flips the randomized algorithm makes. Therefore, the randomized algorithm could choose a random date; the adversary must assign an absolute rank to each date without knowledge of the algorithm's random choices.

(b) The algorithm is as follows: we pick a random permutation π of the dates: we examine the first $k/2$ dates and never accept, and let d_i be the date with the highest relative rank. We then consider the next $k/2$ dates in order, and accept the first date that is better than d_i. In the case that this never occurs, we accept the last date.

We will accept the date with best absolute rank if the following occurs:

- The date with second highest absolute rank appears in the first $k/2$ dates.
- The date with highest absolute rank appears in the second $k/2$ dates.

The probability that this occurs is $1/4$, so we have our desired result.

(c) Here is a randomized algorithm that achieves a date with expected absolute rank of $O(\log k)$.

We first select a random permutation π of the dates. Let S be the set of the first $k/2$ dates, and let T be the set of the last $k/2$ dates. We will sample from S which gives a relative ranking $1 \ldots k/2$ on the first $k/2$ dates.

Now we process each of the dates in T in the order specified by the permutation π. The relative ranking on the first $k/2$ dates induces a relative ranking on any date in T. The algorithm is to accept the first date in T with relative rank at most $\log k$; if this does not occur, accept the last date.

Now we analyze the algorithm to determine its expected absolute rank. The probability that we accept the last date is the probability that we do not see any date in T with relative ranking at most $\log k$. For this to occur, it must be the case that dates with absolute rank $1 \ldots \log k$ are placed in S. The probability of this occurring is:

$$\left(\frac{1}{2}\right)^{\log k} = \frac{1}{k}$$

In the worst case, the absolute rank of the last date is k, so the contribution to the expected absolute rank is 1.

If we do not accept the last date, then we accept the first date d in T with relative rank at most $\log k$. This occurs with probability $1 - \frac{1}{k}$. We want to determine the absolute rank of d.

If d is ranked above a date in S with absolute rank $C \log k$, then d has absolute rank at most $C \log k$. This occurs if S contains at least $\log k$ many dates with rank at most $C \log k$. Note that our permutation was random; therefore, with equal probability a date is placed in S and T.

For the elements with absolute rank $1 \ldots C \log k$, we specify a random variable X_i that is 1 if the element is placed in S, and 0 otherwise. We wish to know the probability that less than $\log k$ of $X_1 \ldots X_{C \log k}$ are put in S. Appealing to the Chernoff bound, we have:

$$\Pr \sum X_i < \log k \leq \frac{1}{k^c}$$

Therefore, the probability that S contains less than $\log k$ many dates with rank at most $C \log k$ is less than $\frac{1}{k^c}$, (for some constant c). In this case, let us just assume the absolute rank we achieve is k, the worst possible. Therefore, the expected absolute rank is:

$$\frac{1}{k} \cdot k + \left(1 - \frac{1}{k}\right) \frac{1}{k^c} \cdot k + \left(1 - \frac{1}{k}\right) \left(1 - \frac{1}{k^c}\right) \cdot C \log k = O(\log k)$$

To achieve a constant expected absolute rank, we must work a bit harder; most students who got this used the analysis from the following paper, by Rudolf Fleischer:

`http://www.cs.fudan.edu.cn/theory/~rudolf/Courses/Online00/Resources/Slides/sec.ps`

Problem 4. **(a)** Suppose the online algorithm makes a mistake, and let the total weight be w. This implies that the set of faculty S with weight at least $\frac{1}{2}w$ answered incorrect. We halve the weight of each faculty member in S. The new weight is thus at most

$$\frac{1}{4}w + \frac{1}{2}w = \frac{3}{4}w.$$

To upper bound the total weight of the faculty, consider a sequence of k questions in which the online algorihtm makes a mistake l times. To result in the greatest weight, each time the online algorithm is correct, no faculty is wrong; each time the online algorithm is wrong, the weight decreases by at most a factor of $\frac{4}{3}$. Therefore, we would have that an upper bound on the weight of the faculty is

$$\left(\frac{3}{4}\right)^l n$$

since n is the weight at the start of the algorithm.

(b) The wisest faculty member is wrong m times. His weight is at least $\frac{1}{2^m}$. This is thus a lower bound on the weight of the total faculty.

(c) We have the following bounds on the final weight of the faculty:

$$\frac{1}{2^m} \leq w \leq \left(\frac{3}{4}\right)^l n$$

Solving for l, we get

$$l \leq \frac{1}{\log \frac{4}{3}}(m + \log n) \leq 2.41(m + \log n)$$

(d) Independent of whether or not we guess the correct answer, we decrease the weight of the faculty with the wrong answer. Therefore, following the analysis in part (a), we have the new weight is:

$$\beta F w + (1 - F)w = (1 - (1 - \beta)F)w$$

so that the expected multiplicative change in total weight is:

$$1 - (1 - \beta)F$$

(e) As noted above, the value of the weights are not dependent on whether or not we guess the correct answer. Therefore, we can apply part (d) to get the following upper bound on the weight after n questions have been asked:

$$(1 - (1 - \beta)F_n)\,(1 - (1 - \beta)F_{n-1}) \ldots (1 - (1 - \beta)F_1)\,n$$

where F_i is the fraction of answer correct on the Now, applying the inequality

$$1 - (1 - \beta)F_i \leq e^{-(1-\beta)F_i}$$

we get the following upper bound on the total weight:

$$w \leq e^{-(1-\beta)\sum F_i} n$$

For a lower bound on the total weight, again consider the wisest faculty member who makes m mistakes. His weight is at least β^m, and this is a lower bound on the total weight. So we have the following bounds on the final weight w:

$$\beta^m \leq w \leq w \leq e^{-(1-\beta)\sum F_i} n$$

The probability of getting an incorrect answer in the ith stage is just F_i. Therefore, $\sum F_i$ is just the expected number of incorrect answers.

Solving for $\sum F_i$, we get:

$$\sum F_i \leq \frac{m \ln \frac{1}{\beta} + \ln n}{1-\beta}$$

(f) By setting

$$\beta = \frac{\sqrt{m}}{\sqrt{m} + \sqrt{\ln n}}$$

we get that:

$$\sum F_i \leq m + \ln n + 2\sqrt{m \ln n}$$

which is the desired result. β can be found by minimizing the bound found in (e), and simplyfing $\ln \frac{1}{\beta}/(1-\beta)$ by applying Taylor series expansion.

Another clever way[1] of finding β involves defining a new variable $\gamma > 0$ such that $1/\beta = 1 + \gamma$ and simplifying the expression in terms of γ. In particular, we note that $\beta = \frac{1}{1+\gamma}$, $1 - \beta = \frac{\gamma}{1+\gamma}$, and $\ln(1+\gamma) \leq \gamma$, yielding us the following inequality.

$$\begin{aligned} \frac{m \ln(1/\beta) + \ln n}{1-\beta} &= \left(\frac{1+\gamma}{\gamma}\right)(m \ln(1+\gamma) + \ln n) \\ &\leq (1+\gamma)\left(m + \frac{\ln n}{\gamma}\right) \\ &= m + \ln n + \left(m\gamma + \frac{\ln n}{\gamma}\right) \end{aligned}$$

Balancing parameters gives us $\gamma = \sqrt{\frac{\ln n}{m}}$, simplifying the upper bound to $m + \ln n + O(\sqrt{m \ln n})$ as desired.

Problem 5. **(a)** Given a set $S = \{r_1 \ldots r_n\}$ of axis-parallel rectangles in the plane, we map each rectangle r_i to a point $p(r_i) \in \mathbb{R}^4$. Given a query rectangle q in the plane, we map the query point to a rectangle $r(q) \in \mathbb{R}^4$. The key property that we satisfy is that:

$$r_i \in q \iff p(r_i) \in r(q).$$

[1]Found by Thomas Mildorf

If we can construct maps p, r, then it follows that we can answer queries in $O(\log^4 n + k)$ time with a data structure that uses $O(\log^3 n)$ space.

Define the maps p, q as follows:

$$\begin{aligned} p &: [x, x'] \times [y, y'] \to (x, x', y, y') \\ r &: [w, w'] \times [z, z'] \to [w, w'] \times [w, w'] \times [z, z'] \times [z, z'] \end{aligned}$$

It is easy to check that:

$$[x, x'] \times [y, y'] \in [w, w'] \times [z, z'] \iff (x, x', y, y') \in [w, w'] \times [w, w'] \times [z, z'] \times [z, z'].$$

(b) For a polygon $Q \in \mathbb{R}^2$ defined by points $\{(x_i, y_i)\}$, let

$$\begin{aligned} x &= \min_i x_i \\ x' &= \max_i x_i \\ y &= \min_i y_i \\ y' &= \max_i y_i \end{aligned}$$

Consider the polygon's bounding box, or the rectangle $R = [x, x'] \times [y, y']$, and let $q = [w, w'] \times [z, z']$ be a query rectangle. It should be clear that:

$$Q \in q \iff R \in q.$$

We have reduced the problem to determining if a rectangle is completely contained in a query rectangle, the problem we solved in part (a).