Midterm Examination I

February 04, 2015

Name: _____
LASTNAME, Firstname(s)

Student Number: _____

Duration: 90 minutes
Aids allowed: one double-sided, 8.5" $\times$ 11" sheet of notes

Keep the exam booklet closed until the beginning of the examination. Make sure that your booklet has 6 pages (including this one). Write your answers in the spaces provided. *Please write legibly*.

Take your time. Read each question carefully and try to avoid making problems more difficult than they really are.
Good luck!

1	2	3	4	total
/20	/20	/20	/25	/85

1. (5+ 7 + 8 marks)

In Assignment 1, Question 3, you discovered that the *minimum-gap problem* (finding the closest pair in a given collection S of n real numbers) is *linearly reducible to sorting*: given *any* algorithm that sorts S in $f(n)$ time, we can construct an algorithm for the minimum-gap problem that runs in $O(f(n))$ time (assuming $f(n) = \Omega(n)$).

(a) Briefly describe this reduction.

If the set S has been sorted, in $f(n)$ time, then the minimum gap can be found in an additional $\Theta(n)$ time by scanning the elements in S, in increasing order, maintaining the smallest gap between successive elements. (Here we are using the fact that the closest pair must be successive elements in the sorted order.) Thus the total cost of solving the minimum-gap problem is $O(n + f(n))$.

We subsequently saw that the minimum-gap problem (renamed as the *closest-pair problem*) has an $O(n)$ expected time solution (even in higher dimensions).

(b) Briefly describe the role of hashing in this solution.

Hashing was used to reduce the space cost that would be required if the two (or higher) dimensional space of buckets (into which elements are mapped) were implemented (as initially described) with a direct access table. The size of the bucket space depends inversely on the smallest separation between distinct elements, so without hashing we have no upper bound available on the amount of space needed.

Suppose that we simply want to check whether or not the minimum gap in S exceeds some specified value $\Delta > 0$.

(c) Describe an algorithm that solves this gap-checking problem in $O(n)$-time *in the worst case*. (Hint: (i) there is not enough time to sort S; (ii) there is no loss of generality in assuming that $0 \leq \min(S) \leq \max(S) \leq 1$, (iii) you are free to use a lot of space (assuming you know how to initialize it quickly).)

The idea here is to scale the numbers in S (as well as Δ) so that $0 \leq \min(S) \leq \max(S) \leq 1$. Now it suffices to divide the interval $[0, 1]$ into buckets of size Δ, and assign the numbers (in any order) to buckets (by taking the quotient on division by Δ). If number x is assigned to bucket b then it suffices to check buckets $b-1$, b, and $b+1$ to see if there are any (previously bucketed) numbers that are within distance Δ of x. The bucket space has size proportional to $1/\Delta$, which could be very large. However, using the idea for implicit initialization of direct access tables, discussed in class and homework, the total cost of table accesses is $O(|S|)$.

2. (7 + 8 + 5)

We want to represent a set S (a static dictionary) consisting of n keys as a simple list and perform MEMBER queries by comparing with each element in turn (stopping with a successful comparison). For each element x_i of S we know its probability p_i of being queried. Our objective is to find an ordering of the elements so that the corresponding list (the *optimal list*) has the *minimum expected cost* for successful MEMBER queries (among all possible lists).

(a) Give an expression for the expected cost for successful MEMBER queries, if the elements appear in the order $x_1, x_2, \ldots, x_n$.

Since element x_i appears in position i on the list, and has probability p_i of being accessed, the expected access cost is $\sum_{i=1}^{n} i \cdot p_i$.

(b) Show that the element x_i with maximum p_i appears in the first position of the optimal list. (Hint: consider how an interchange of consecutive elements changes the expected cost.)

Suppose that the element x_i, with maximum p_i appears in position $j > 1$ of a list, and suppose that element x_k appears in position $j - 1$. Then, if we interchange the positions of elements x_i and x_k, the expected access cost (from part (a)) changes by $p_k - p_i$ (since the position of x_k increases by one, the position of x_i decreases by one, and all other elements stay in their same position. But, by assumption, $p_k < p_i$, so the expected access cost decreases by this interchange. It follows that the original list was not optimal, and hence any optimal ordering must have x_i in the first position.

(c) Suppose that we choose instead to represent S as a binary search tree. Is it the case that the tree T that minimizes the expected cost for successful MEMBER queries must have the element x_i with maximum p_i at its root? Explain your answer.

As we noted in class, the optimal binary search tree does not necessarily have the element x_i with maximum p_i at its root. The basic idea is that the element x_i with maximum p_i may not serve as a good "splitter" of the entire set of keys. So, for example, if the smallest key x_1 has probability $1/n - \epsilon$ and all the rest have probability $1/n + \epsilon/(n-1)$, then the expected cost, when ϵ is small, of the optimal tree with x_1 at the root is about one more than the cost of the perfectly height balanced binary search tree.

3. (10+10 marks)

The procedure `RandomlyPermute` takes an array of real numbers as input and returns a random permutation of the array, where all permutations are equally likely. Consider the following program fragment, where *expensive case* and *cheap case* stand for unspecified fragments with the specified costs.

$A[1:n] \leftarrow$ `RandomlyPermute`$(A[1:n])$
$max \leftarrow A[1]$
for $i \leftarrow 2$ **to** n **do**
  if $A[i] > max$
    then do
      expensive case: some work of cost $\Theta(i)$
      $max \leftarrow A[i]$
    else do
      cheap case: some work of cost $\Theta(1)$

(a) What is the (asymptotic) *expected* cost of executing this fragment? Explain your analysis.

Following the analysis of the randomized incremental algorithm for the hiring problem (or the Golin et al. algorithm for the closest-pair problem, which is a special case of this abstract procedure), we note that the probability that $A[i]$ leads to an update of max, which is just the probability that $A[i]$ is larger than $A[j]$, for all $j < i$, is at most $1/i$. Thus the expected cost of executing the ith step of the **for** *loop of the fragment above is $O(i \cdot i/i + 1 \cdot (1 - i/i))$, which is $\Theta(1)$. It follows that the expected total cost is $\Theta(n)$.*

Recall the *un-dominated points problem* from Assignment 3: **Input**: a set S of n points in the plane, whose x and y coordinates are specified by two arrays $X[1:n]$ and $Y[1:n]$ of real numbers (i.e. the i-th point is $(X[i], Y[i])$). **Output**: The sequence consisting of all of the un-dominated points in the input set, ordered by x-coordinate. (Recall that point (x, y) is *dominated* by point (p, q), if $x < p$ and $y \leq q$, or $x = p$ and $y < q$.)

(b) Suppose the points in S are chosen uniformly at random from the unit square $[0, 1]^2$ (i.e. all coordinates are independent random numbers in $[0, 1]$). Show that the expected number of un-dominated points in S is $O(\lg |S|)$. (Hint: imagine an algorithm that encounters the points in order of decreasing x-coordinate. Since the y-coordinates are random, what is the probability the the i-th point encountered is un-dominated?)

If the points are sorted, as suggested in the hint, and accessed in order of decreasing x-coordinate, then each successive un-dominated point encountered must have a y-coordinate larger than all previously accessed points. But, because the set of y-coordinates is randomly chosen, it follows that the i-th point accessed has a probability of at most $1/i$ of being un-dominated (by the same analysis as in part (a)). Thus, the expected number of un-dominated points is at most $\sum_i 1/i$, which is $\Theta(\ln n)$.

4. (8 + 7 + 10) marks

We saw in class that the move-to-front heuristic, for adaptive list-structured dictionaries, can be used in data compression. The basic idea is that we maintain a list L of all of the characters in a given string σ of text, ordered according to the move-to-front strategy (i.e. as we encounter the next character in σ we move it to the front of L). We encode each successive character by the (integer) position it occupies in L (before it is moved). To make this encoding efficient, we need to encode *small* integers (the position of frequently occurring characters) with small codes. For this we can do something reminiscent of our scheme for exploiting locality of search in sorted arrays....

Suppose that a is a positive integer. The following algorithm outputs a binary sequence that we will interpret as the *encoding* of a.

```
                                  ▷ phase 1
r ← 1
while a ≥ 2r do                   ▷ invariant: a ≥ r
    output 1
    r ← r + r
output 0
                                  ▷ assertion: r = 2^⌊lg a⌋

                                  ▷ phase 2
low ← r;  high ← 2r;  gap ← r
while gap > 1 do                  ▷ invariant: low ≤ a < high = low + gap
    mid ← low + gap/2
    if a < mid
        then do
            output 0
            high ← mid
        else do
            output 1
            low ← mid
    gap ← gap/2
```

(a) Prove that he first phase of the algorithm outputs a sequence of exactly $\lfloor \lg a \rfloor$ 1's, followed by a zero.

It is clear from the structure of the pseudocode that the first phase outputs a sequence of 1*'s followed by a* 0*. To determine the length of the sequence of* 1*'s, we observe that each time a* 1 *is output the value of* r *doubles. Since* r *starts with value* 1 *and ends as soon as* $2r > a$*, it follows that* r *ends with value* $2^{\lfloor \lg a \rfloor}$ *(see the attached assertion), and hence exactly* $\lfloor \lg a \rfloor$ 1*'s are output.*

(b) The second phase of the algorithm performs a kind of binary search. How many bits are output in this phase, expressed as a function of a? Explain your answer.

Again, the answer is $\lfloor \lg a \rfloor$*. In this case we observe that each time a bit (either* 0 *or* 1*) is output the value of* gap*, initially* $2^{\lfloor \lg a \rfloor}$ *according to the assertion, is decreased by a factor of two. Since the phase ends when* gap *has been reduced to* 1*, it follows that exactly* $\lfloor \lg a \rfloor$ *bits are output.*

(c) Consider any coding scheme that outputs unique binary codes for all of the integers in some range $[1 : a]$. Argue that at least half of the numbers must have codes of length at least $\lfloor \lg a \rfloor - 1$.

The number of binary strings of length t is exactly 2^t. Thus the number of binary strings of length at most ℓ is $\sum_{t=0}^{ell} 2^t = 2^{\ell+1} - 1$. (Here we are even treating the empty string as a possible code.) Suppose now that more than half of the numbers in the range $[1 : a]$ have codes of length at most $\lfloor \lg a \rfloor - 2$. Then, since there are $2^{\lfloor \lg a \rfloor - 1} - 1 \leq a/2 - 1$ such codes, we have a contradiction, from which it follows that at least half of the numbers must have codes of length at least $\lfloor \lg a \rfloor - 1$.

(d) [BONUS] Since the first phase of the algorithm outputs a sequence of $\lfloor \lg a \rfloor$ 1's, followed by a zero, we could view this as the unary encoding of the number $\lfloor \lg a \rfloor$. If we applied the same idea used in this algorithm to create a more compact encoding of $\lfloor \lg a \rfloor$, what would the total length of the encoding of a become?

We can apply the idea of the full scheme to produce a smaller encoding of the sequence of 1's produced in the first phase. Thus the first-phase code is reduced from $\lfloor \lg a \rfloor$ to $\lfloor \lg \lfloor \lg a \rfloor \rfloor$. The length of the second-phase code remains unchanged.