Illinois Institute of Technology
Department of Computer Science

Solutions to Second Examination

CS 430 Introduction to Algorithms
Spring, 2012

11:25am–12:40pm, Wednesday, March 14, 2012
104 Stuart Building

Exam Statistics

52 students took the exam. The range of scores was 0–78, with a mean of 47.04, a median of 50, and a standard deviation of 16.14. Very roughly speaking, if I had to assign final grades on the basis of this exam only, 60 and above would be an A (9), 50–59 a B (18), 31–49 a C (16), 20–30 a D (5), below 20 an E (4).

Problem Solutions

1. We saw in class (notes of February 1), that the external path length can equivalently be defined recursively as

$$\begin{aligned} \mathit{EPL}(\Box) &= 0 \\ \mathit{EPL}(x) &= \mathit{EPL}(x.\mathit{left}) + \mathit{EPL}(x.\mathit{right}) + n + 1 \end{aligned}$$

   where $x.\mathit{left}$ and $x.\mathit{right}$ are the left and right subtrees, respectively, of x, and n is the number of internal nodes in the subtree rooted at x. Thus $\mathit{EPL}(x)$ does not satisfy the hypothesis of Theorem 14.1 (page 346 in CLRS), namely that $\mathit{EPL}(x)$ depend only on $\mathit{EPL}(x.\mathit{left})$ and $\mathit{EPL}(x.\mathit{right})$; hence we cannot conclude that it can be maintained in a red-black tree. But Theorem 14.1 only gives us *sufficient conditions* for maintainence, not *necessary conditions*.

   If we knew the number of internal nodes in the subtree rooted at x, then $\mathit{EPL}(x)$ would satisfy the hypothesis of Theorem 14.1. So, denote by $\mathit{size}(x)$ the number of internal nodes in the subtree rooted at x. $\mathit{size}(x)$ can be written recursively as

$$\begin{aligned} \mathit{size}(\Box) &= 0 \\ \mathit{size}(x) &= \mathit{size}(x.\mathit{left}) + \mathit{size}(x.\mathit{right}) + 1, \end{aligned}$$

   and hence $\mathit{size}(x)$ satisfies the hypothesis of Theorem 14.1 and can be maintained in a red-black tree. Thus by maintaining $\mathit{size}(x)$ we can also maintain $\mathit{EPL}(x)$ in a red-black tree.

2. This problem turned out to be more intricate than I intended!

   The hint for part (a) should be modified to read "Let L_j be the length of the longest increasing subsequence in $a_1, a_2, \ldots, a_j$, let A_j be index of the smallest possible largest element in that increasing subsequence, and let B_j be index of the second largest element in that increasing subsequence. Express L_j recursively. You may assume a dummy element $a_0 = -\infty$."

(a) Using the Principle of Optimality, we can express L_j recursively as

$$L_j = \begin{cases} 1 & \text{if } j = 1, \\ \max\{L_{j-1}, \max\limits_{\substack{1 \le i < j \\ a_{A_i} < a_j}} \{1 + L_i\}\} & \text{otherwise.} \end{cases}$$

Of course the values of A_j and B_j must be kept maintained in this recursive definition (that is, which of the various possibilities in the max was maximum with the lowest value of A_j and what the corresponnding value of B_j is).

Ignoring the hint, we might be tempted to express the recurrence looking at all increasing subsequences that have a_i as a member, recursively obtain longest increasing subsequences on the left and on the right, and the choose the combination with the overall maximum length. Writing the recurrence (and hence recursive code) is tricky because the element we insist on, a_i, must be larger than the largest element of the increasing subsequence on the left and smaller than the least element of the increasing subsequence on the right. This approach would mean adding two parameters to the recurrence so that it works out properly—a value that is an upper bound for a possible left increasing sequence and a a value that is a lower bound for a possible right increasing sequence.

(b) Let c_j be the rate of growth of the cost of evaluating L_j using the recurrence in (a). We have

$$c_j = \begin{cases} 1 & \text{if } j = 1, \\ 1 + \sum\limits_{1 \le i < j} c_i & \text{otherwise.} \end{cases}$$

Looking at the difference between successive values (as we did in the Quicksort recurrence to eliminate the summation), we find that $c_j - c_{j-1} = c_{j-1}$ implying $c_j = 2c_{j-1}$ and hence $c_j = 2^{j-1}$; we could also guess at this solution by computing the first few values and then verifying the guess by induction. Thus computing L_n would cost $\Theta(2^n)$.

(c) Following the (modified) hint, to memoize (a) we use an array $L[j]$ to tell us the length of the LIS among $a_1, a_2, \ldots, a_j$, an array $A[j]$ to tell us the index of the smallest possible largest (last) element of the LIS among $a_1, a_2, \ldots, a_j$, and an array $B[j]$ to tell us the index of the smallest possible second largest element of the LIS among $a_1, a_2, \ldots, a_j$. Define a dummy element $a_0 = -\infty$; then $L[1] = 1$, $A[1] = 1$, and $B[1] = 0$. We have:

$a_0 = -\infty$
$L[1] = 1$
$A[1] = 1$
$B[1] = 0$
for $j = 2$ **to** n **do**
    $L[j] \leftarrow L[j-1]$

```
    A[j] ← A[j − 1]
    B[j] ← B[j − 1]
    for i = 1 to j − 1 do
      if L[j] = L[i] and a_{A[j]} > a_{A[i]} > a_{B[j]} then
        // a_{A[i]} gives us a lower largest value for L[j]
        A[j] ← A[i]
      else if L[j] = L[i] and a_{A[j]} > a_j > a_{B[j]} then
        // a_j gives us a lower largest value for L[j]
        A[j] ← j
      else if L[j] ≤ L[i] and a_j > a_{A[i]} then
        // we can add a_j to end of L[i]
        L[j] ← L[i] + 1
        B[j] ← A[j]
        A[j] ← j
      end if
    end for
  end for
```

The $A[j]$ and $B[j]$ values allow us to recover the increasing subsequence of length $L[j]$ among $a_1, a_2, \ldots, a_j$:

```
procedure WriteLIS(i)
if i > 0 then
  if A[i] = i then
    WriteLIS(B[i])
    SystemPrint(a_i)
  else
    WriteLIS(A[i])
  end if
end if
```

(d) The cost of this double-nested loop computation is $\sum_{1<j\leq n}\sum_{1\leq i<j} 1 = \sum_{1<j\leq n}(j-1) = n(n-1)/2 = \Theta(n^2)$.

3. Let n be the number of elements on the stack before an operation. The calculations are nearly identical to those done in chapter 17 of CLRS (lecture notes of February 27):

$$
\begin{aligned}
\text{AMORT}_{\text{POP}} &= \text{ACTUAL}_{\text{POP}} + |\text{stack}|^2_{\text{after}} - |\text{stack}|^2_{\text{before}} \\
&= 1 + (n-1)^2 - n^2 = -2(n-1) \leq 0 = O(n). \\
\text{AMORT}_{\text{PUSH}} &= \text{ACTUAL}_{\text{PUSH}} + |\text{stack}|^2_{\text{after}} - |\text{stack}|^2_{\text{before}} \\
&= 1 + (n+1)^2 - n^2 = 2(n+1) = O(n). \\
\text{AMORT}_{\text{MULTIPOP}(k)} &= \text{ACTUAL}_{\text{MULTIPOP}(k)} + |\text{stack}|^2_{\text{after}} - |\text{stack}|^2_{\text{before}} \\
&= k(k-1)/2 + (n-k)^2 - n^2
\end{aligned}
$$

$$\begin{aligned} &= k^2/2 - k/2 - 2kn \\ &= k(k/2 - 2n - 1/2) < 0 = O(n), \end{aligned}$$

because $k \leq n$.

4. The potential function is $\Phi(H) = t(H) + 2m(H)$, but now no nodes are marked so we always have $m(H) = 0$, making the potential function $\Phi(H) = t(H)$. The amortized costs of insertion, union, consolidation, and extracting the minimum still hold because the number of marks did not change in these operations and the $m(H)$ values before and after simply canceled.

   However the amortized cost of decreasing a key (and hence also of deleting a node) no longer has that $-c$ term in the change of potential to offset the $\Theta(c)$ cost of the cascading cut (pages 521–522 of CLRS); together with exercise 19.4-1 (part of Homework 6), this means that a cascading cut (and hence decreasing a key or deleting a node) can have amortized time $\Theta(n)$ in an n-node heap.