COT5405 Homework 2 Solutions

Maksim Levental

September 23, 2014

4.2-3 Pad with zeroes until you get matrices of size which is a power of 2 then truncate all non-zero entries in the resultant matrix. Let $k = 2^{\lceil \lg n \rceil}$ and pad the $n \times n$ matrices out to $k \times k$. Complexity is $O(k^{\lg 7}) = O\left(\left(2^{\lceil \lg n \rceil}\right)^{\lg 7}\right) = O\left(\left(2^{\lg n}\right)^{\lg 7}\right) = O\left(\left(n^{\lg 2}\right)^{\lg 7}\right) = O\left(n^{\lg 7}\right)$.

4.2-4 If the reduction is from one multiplication to k multiplications of matrices size $n/3$ then the recurrence relation is $T(n) = kT(n/3)+\Theta(n^2)$ because each smaller multiplcation, presumably, still takes $\Theta(n^2)$ time. Where $\log_3 k$ falls in relation to 2 determines whether $T(n) = O(n^{\lg 7})$. If $\log_3 k > 2$ then case 1 of the master theorem applies and $T(n) = \Theta(n^{\log_3 k})$ and $\log_3 k < \lg 7$ when $k < 3^{\lg 7} \approx 21.8$. If $\log_3 k = 2$ and $k = 9$ then case 2 of the master theorem applies and $T(n) = \Theta(n^2 \lg n) = O(n^{\lg 7})$. If $\log_3 k < 2$ and $k < 9$ then case 3 of the master theorem applies and $T(n) = \Theta(n^2) = O(n^{\lg 7})$. So the maximum number is 21 by way of case 1.

4.2-6 Let A be the $kn \times n$ matrix and B be the $n \times kn$ matrix. We can look at A as a k-row column vector of $n \times n$ matrices and B as a k-column row vector of $n \times n$ matrices. I.e

$$A = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_k \end{pmatrix} \quad B = (B_1, B_2, \ldots, B_k)$$

Then $A \cdot B$ is the outerproduct of the column and row vector and therefore

$$A \cdot B = \begin{pmatrix} A_1 \cdot B_1 & A_1 \cdot B_2 & \cdots & A_1 \cdot B_k \\ A_2 \cdot B_1 & A_2 \cdot B_2 & \cdots & A_2 B_k \\ \vdots & \vdots & \ddots & \vdots \\ A_k \cdot B_1 & A_k \cdot B_2 & \cdots & A_k \cdot B_k \end{pmatrix}$$

where every $A_i \cdot B_j$ is an $n \times n$ matrix product that can be computed using Strassen's algorithm. Therefore the total running time is $k^2 O(n^{\lg 7}) = O(k^2 n^{\lg 7})$. If the input matrices are reversed then $B \cdot A$ is the "dot product" and therefore

$$A \cdot B = A_1 \cdot B_1 + A_2 \cdot B_2 + \cdots + A_k \cdot B_k$$

where every $A_i \cdot B_j$ is an $n \times n$ matrix product that can be computed using Strassen's algorithm. Therefore the total running time is $kO(n^{\lg 7}) = O(kn^{\lg 7})$.

6-3

(a)

	2	8	12	16
3	9	14	∞	
4	∞	∞	∞	
5	∞	∞	∞	

(b) If $Y[1,1] = \infty$ then the only entries that can appear in $Y[1,j]$ for $j > 1$ are ∞ by the requirement that $Y[1,j] \leq Y[1,j+1]$ and similarly the only entries that can appear in $Y[i,1]$ for $i > 1$ are ∞. Hence by induction/recursion all of the entries in the tableau are ∞. If $Y[m,n] < \infty$ then no entries equal to ∞ can appear in $Y[m,j]$ for $j < n$ by the requirement that $Y[1,j] \leq Y[1,j+1]$ and similarly no entries equal to ∞ can appear in $Y[i,n]$ for $i < n$. Hence by induction/recursion Y is full.

(c) The minimum element is at $Y[1,1]$. Remove/extract it and replace it with ∞. Consider the sub-tableaux Y_{ij} of Y with top-left corner at row i and column j. After extracting the minimum $Y[1,1]$ and replacing it with ∞ compute the minima of $Y_{i+1,j}$ and $Y_{i,j+1}$ for $i = j = 1$ and swap the ∞ in $Y[1,1]$ with the $\min\{Y_{1+1,1}, Y_{1,1+1}\}$. This is correct because it does not violate the Young-tableau property Y since you're picking the minimum of $\{Y_{1+1,1}, Y_{1,1+1}\}$. The algorithm terminates when

$$\infty = \min\{Y_{i+1,j}, Y_{i,j+1}\}$$

or $i > m$ and $j > n$. The recurrence relation is

$$T(m+n) \leq \max\{T(m-1+n), T(m+n-1)\} + \Theta(1)$$

Let $p = m + n$ then

$$\begin{aligned} T(p) &\leq \max\{T(p-1), T(p-1)\} + \Theta(1) \\ &= T(p-1) + \Theta(1) \end{aligned}$$

Whose solution is clearly $O(p) = O(m+n)$ since the sub-problem is 1 smaller.

(d) By symmetry the maximum element in a Young-tableau is in the bottom-right entry. To insert replace the maximum element of Y (by part (b) this entry is ∞ if Y isn't full). Then compute the maximum of $Y[i-1,j]$ and $Y[i,j-1]$ for $i = m$ and $j = n$ and swap the entry in $Y[m,n]$ with the $\max\{Y[m-1,n], Y[m,n-1]\}$. If $Y[m-1,n] = Y[m,n-1]$ swap $Y[m-1,n]$ just to be definite. Then repeat. This is correct because you're picking the maximum of $\{Y[m-1,n], Y[m,n-1]\}$ and so at the end having $Y[i-1,j]$ and $Y[i,j-1]$ in the same column or row does not violate the Young-Tableaux property since $\max\{Y[i-1,j], Y[i,j-1]\} \geq Y[i-1,j]$ and $\max\{Y[i-1,j], Y[i,j-1]\} \geq Y[i,j-1]$. The algorithm terminates when the entry to be inserted is $\geq \max\{Y[m-1,n], Y[m,n-1]\}$ or $j < 1$ and $j < 1$. Since at most the algorithm has to make comparisons with every entry in the last column and every entry in the first row minus 1 the running time is $O(m+n)$.

(e) Run Extract-Min for every one of the n^2 elements. The running time is $n^2 O(n+n) = n^2 O(2n) = O(n^3)$.

(f) Start at the top right corner and inspect $Y[1,1]$ and $Y[m,n]$. These are the minimum and the maximum entries in the tableau Y. If the entry you're searching for is outside of that range return false. Otherwise Consider the 2 sub-tableax with with corners (counterclockwise starting from upper right) $Y_1 = \{Y[2,n], Y[m,n], Y[m,1], Y[2,1]\}$ and $Y_2 = \{Y[1,n-1], Y[m,n-1], Y[m,1], Y[1,1]\}$. Y_1 has range (maximum and minimum) $Y[m,n], Y[2,1]$ and Y_2 has range $Y[m,n-1], Y[1,1]$. If the neither ranges contain the element you're searching for return false. Else recurse to the sub-tableau with the smallest range. Per recursion you perform $O(1)$ operations and the most work would be done were you to "walk" along the edges of Y. Hence the running time is $(m+n)O(1) = O(m+n)$.

9.3-5 Just implement the worst case linear time algorithm Select but replace steps 1-3 with your linear time median algorithm. Another way to look at it is to implement Randomized-Select but forget the randomization and change partition to accept a pivot. Then partition on whatever value Linear-Med returns

```
1 Pivot-Partition(A,p,r,i)
2         //A[i] is the entry you want to partition around
```

```
3           swap(A[i],A[r])
4           return Partition(A,p,r)
5
6   Linear-Select(A,p,r,i)
7           if p == r
8                   return A[p]
9           m = Linear-Med(A,p,r)
10          q = Pivot-Partition(A,p,r,m)
11          k = q-p+1
12          if i == k
13                  return A[q]
14          else if i < k
15                  return Linear-Select(A,p,q-1,i)
16          else
17                  return Linear-Select(A,q+1,r,i-k)
```

9.3-7 Modify Select to Select-Index, which return the index of the mth statistic instead of the value. Duplicate the array in linear time. Call the first array A and the duplicated array B. Use Select to find the median m of A in $O(n)$ time then in $O(n)$ time subtract m from all elements in A and compute their absolute value and store them in A. The k smallest elements in this adjusted array then correspond to the closest to the median from the original array (B). Compute the index of the kth order statistic of this new array. Then use a modified version of Partition, Double-Pivot-Partition, that partitions A but also performs the same swaps in B, i.e. partitions B the same way as A. Then the first k elements in B are elements in the original array that were the k closest to the median.

```
1   Double-Partition(A,B,p,r)
2           x = A[r]
3           i = p-1
4           for j = p to r-1
5                   if A[j] <= x
6                           i = i+1
7                           swap(A[i],A[j])
8                           swap(B[i],B[j])
9           swap(A[i+1],A[r])
10          swap(B[i+1],B[r])
11          return i+1
12
13  Double-Pivot-Partition(A,B,p,r,i)
14          //A[i] is the entry you want to partition around
15          swap(A[i],A[r])
16          swap(B[i],B[r])
17          return Double-Partition(A,B,p,r)
18
19  Median-Closest(A,k)
20          //the actual median, not just the index
21          m = Select-Index(A,0,A.size-1,floor((A.size-1)/2))
22          B = A
23          for j=1 to A.size-1
24                  A[j] = A[j]-A[m]
25          //
26          l = Select-Index(A,0,A.size-1,k)
27          //return will be k because l is index of kth OS
```

```
28          //and so it will be placed in the kth position.
29          Double-Pivot-Partition(A,B,0,l)
30          return B[0..k]
```

9.3-8 Let $a_m = A[\lceil A.\text{size}/2 \rceil]$ and $b_m = B[\lceil B.\text{size}/2 \rceil]$ the medians of each array. Let $c_i < c_{i+1}$ be the sorted elements in $A \cup B$ and m be the median of the $\{c_i\}$, i.e. $A \cup B$.

Lemma: In $A \cup B$ it's the case that $a_b \leq m \leq b_m$, that is in the final arrangement of elements the median of both arrays is between (or equal to one or the other other both of) the medians of A and B.

Proof: If $a_m = b_m$ then clearly the median of $A \cup B = a_m = b_m$. Hence without loss of generality assume $a_m \neq b_m$ and $a_m < b_m$. The quantity of c_i such that $c_i \leq a_m$ is at most $\lceil A.\text{size}/2 \rceil - 1 + \lceil B.\text{size}/2 \rceil - 1$, (saturating if $a_m = b_m$). These are all of the elements in $A \leq a_m$ unioned with all of the elements in $B \leq a_m$. Similarly there are at most $\lceil A.\text{size}/2 \rceil - 1 + \lceil B.\text{size}/2 \rceil - 1$ such that $c_j \geq b_m$. Furthermore since $a_m < b_m$ there are at most $\lceil A.\text{size}/2 \rceil - 1 + \lceil B.\text{size}/2 \rceil$ elements in $A \cup B$ such that $a_m \leq c_i$ ($a_m = c_i$ and $b_m = c_{i+1}$) and similarly at most $\lceil A.\text{size}/2 \rceil - 1 + \lceil B.\text{size}/2 \rceil$ elements such that $c_j \leq b_m$. But the median m is the element such that there are exactly $\lceil A \cup B.\text{size}/2 \rceil - 1 = \lceil A.\text{size}/2 \rceil + \lceil B.\text{size}/2 \rceil + 1$ elements less than it and $\lceil A \cup B.\text{size}/2 \rceil - 1 = \lceil A.\text{size}/2 \rceil + \lceil B.\text{size}/2 \rceil + 1$ greater than it, which is a contradiction. Therefore $a_m \leq m \leq b_m$. ☻

To compute the median of $A \cup B$ compute $a_m = A[\lceil A.\text{size}/2 \rceil]$ and $b_m = B[\lceil B.\text{size}/2 \rceil]$, the medians of each array, in constant time. If $a_m = b_m$ return a_m. Otherwise without loss of generality we assume $a_m < b_m$. Then let S_a be all of the elements between a_m and $A[n]$ in A and likewise for S_b all of the elements between b_m and $B[n]$ in B. Finally by the Lemma above we know that the median $m \in S_a \cup S_b$ and since each of $S_a$and S_b are already sorted we can recurse into them.

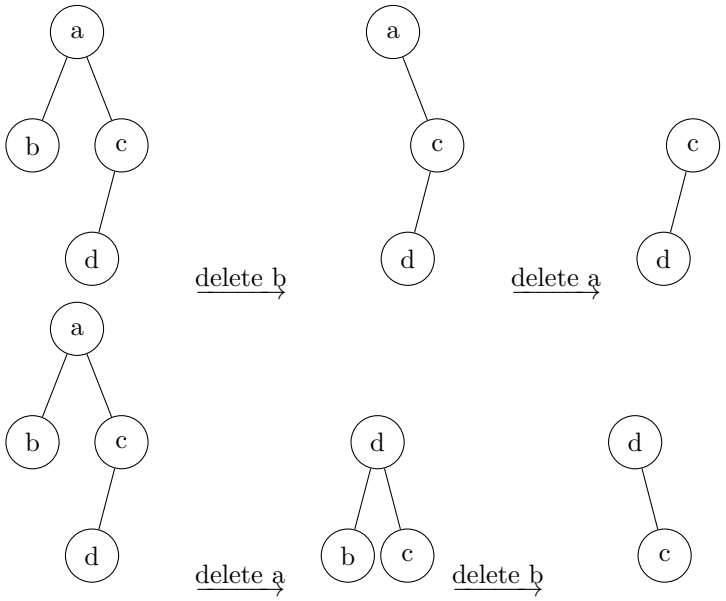
At each recursion medians of each sub-array are computed in constant time, and since we're truncating around the medians of the subarray half of the entries are truncated and so we have the recursion relation $T(n) \approx T(n/2) + O(1)$ whose solution is $O(\lg n)$.

10-1 The complexities:

Operation	unsorted, singly linked	sorted, singly linked	unsorted, doubly linked	sorted, doubly linked
Search(L, k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Insert(L, x)	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Delete(L, x)	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Successor(L, x)	$O(n)$	$O(1)$	$O(n)$	$O(1)$
Predecessor(L, x)	$O(n)$	$O(n)$	$O(n)$	$O(1)$
Minimum(L)	$O(n)$	$O(1)$	$O(n)$	$O(1)$
Maximum(L)	$O(n)$	$O(n)$	$O(n)$	$O(1)$

12.2-5 Proof by contradiction. Let X be the node in question and S be the successor of node X. Assume S has a left child $S.left$. But the successor of X must be in the right subtree of X and hence the $S.left$ is simultaneously less than S, by the binary search tree property (because it's the left child of S), and greater than X (because it's in the right subtree of X). But then S is not the successor. For predecessor let P be the predecessor of X. Assume P has a right child $P.right$. But the predecessor of X must be in the left subtree of X and hence the $P.right$ is simultaneously greater than P, by the binary search tree property (because it's the right child of P), and less than X (because it's in the left subtree of X). But then P is not the predecessor.

12.3-4 Deletion is not commutative. Counterexample:



12.3-5 Search is not affected because it doesn't use the parent pointer in any of the nodes.

Insertion isn't affected either except that now instead $z.p = y$ we need to set $z.succ$ to something appropriate. If z is inserted as a left child then its successor is its parent p, since p is certainly greater and if there were a node $p' < p$ and $z < p'$ then z would have been inserted as the left child of p'. If z is inserted as a right child then its successor is the successor of its parent p and p's successor is now z, since being inserted as a right child implies $p < z$ therefore z is "closer" to p's successor than p and there are no other elements between p and z.

Hence

```
1  Tree-Insert(T,z)
2          y = NIL
3          x = T.root
4          while x != NIL
5                  y = x
6                  if z.key < x.key
7                          x = x.left
8                  else
9                          x = x.right
10         if y == NIL
11                 T.root = z
12                 z.succ = NIL
13         else if z.key < y.key
14                 y.left = z
15                 z.succ = y
16         else
17                 y.right = z
18                 z.succ = y.succ
19                 y.succ = z
```

Running time of Tree-Insert is still $O(\lg n)$ since we've only added constant time operations to it.

In order to implement Delete, which calls Transpant (which makes heavy use of parent pointers), we implement another auxilliary routine Parent(T,u). The easiest way to find u's parent is to search for u while keeping a trailing pointer

```
1  Parent(T,u)
2          y = NIL
3          x = T.root
4          while x != u
5                  y = x
6                  if u.key < x.key
7                          x = x.left
8                  else
9                          x = x.right
10         return y
```

Parent has the same structure as Iterative-Tree-Search except that it matches on node rather than key (in case there are several nodes with the same value) and hence is correct and runs in $O(\lg n)$ time. Then we implment transplant with calls to Parent wherever there's a parent pointer being referenced

```
1  Transplant(T,u,v)
2          uParent = Parent(T,u)
3          if uParent == NIL
4                  T.root = V
5          else if u == uParent.left
6                  uParent.left = v
7          else
8                  uParent.right = v
9          if v != NIL
10                 vParent = Parent(T,v)
11                 vParent = uParent
```

Because the only thing changed is how parent pointers are found Transplant is correct for the same reason Transplant from CLRS is correct. Furthermore Transplant(T,u,v) runs in at most $O(\lg n)$ time since it makes at most 2 calls to Parent. And similarly for Tree-Delete

```
1  Tree-Delete(T,z)
2          if z.left == NIL
3                  Transplant(T,z,z.right)
4          else if z.right == NIL
5                  Transplant(T,z,z.left)
6          else
7                  y = z.succ
8                  yParent = Parent(T,y)
9                  if yParent != z
10                         Transplant(T,y,y.right)
11                         y.right = z.right
12                         yRightParent = Parent(T,y.right)
13                         yRightParent = y
14                 Transplant(T,z,y)
15                 y.left = z.left
16                 yLeftParent = Parent(T,y.left)
17                 yLeftParent = y
```

Tree-Delete has the same structure as Tree-Delete from CLRS and hence is correct and runs in at most $O(\lg n)$ time since it makes at most 3 calls to Transplant and 3 calls to Parent, each of which are run in at most $O(\lg n)$ time themselves.