COT5405 Homework 1 Solutions

Maksim Levental

September 13, 2014

2.3-4 For this "recursive" implementation of insertion sort $T(n) = T(n-1) + O(n)$. The inductive hypothesis is $T(m) \leq c_1 m^2$ for all $m < n$. Then

$$\begin{aligned} T(n) &= T(n-1) + c_2 n \\ &\leq c_1(n-1)^2 + c_2 n \\ &= c_1 n^2 - 2c_1 n + 2c_1 + c_2 n \qquad \text{choose } c_1 \text{ s.t. } 2c_1 n \geq c_2 n + 2c_1 \\ &\leq c_1 n^2 \\ T(n) &= O(n^2) \end{aligned}$$

Furthermore we choose the base case such that $2c_1 n \geq c_2 n + 2c_1$ is satisfied.

2.3-5 The pseudo-code is

```
1  Binary-Search(A,p,r,x)
2    if p==r and A[p] != x
3      return -1
4    pivot = floor((p-r+1)/2)
5    else if A[pivot] == x
6      return pivot
7    else if A[pivot] < x
8      Binary-Search(A, pivot+1,r,x)
9    else
10     Binary-Search(p, pivot-1,x)
11     //only if A[floor(A.length/2)] > x
12     //since first branch checks ==
13     //so no need to include element
14     //A[0:floor(A.length/2)]
```

Binary-Search(A,x) returns the index of a match or -1 if the search fails. The recursion relation for the running time is $T(n) = T(n/2)+O(1)$. Since $f(n) = 1 = \Theta\left(n^{log_2 1}\right)$ the master theorem immediately yields $T(n) = \Theta(n^{log_2 1} \lg n) = \Theta(n^0 \lg n) = \Theta(\lg n)$.

2.3-6 This in fact doesn't work but I didn't realize it until I finished the analysis. Read the comment at the end. Reimagine the array in which all the elements are stored as linked-list and rewrite Binary-Search to return a pointer to the last node recursed to. Then the value to be inserted by Insertion-Sort-Binary-Search(A) inserts a node in the linked list whose child is the last node returned by Binary-Search and whose parent is the parent of the node return by Binary-Search-LL. That is

```
1  Binary-Search-LL(A,p,r,x)
2    //linked-list overloads [] operator
3    pivot = floor((p-r+1)/2)
4    if p==r
5      return pivot
6    else if A[pivot] == x
7      return pivot
8    else if A[pivot] < x
9      Binary-Search(A,pivot+1,r,x)
10   else
11     Binary-Search(p,pivot-1,x)
```

Then Insertion-Sort-Binary-Search(A) is

```
1  Insertion-Sort-Binary-Search(A)
2    for j=2..A.length
3      key=A[j]
4      pivot = Binary-Search-LL(A,0,j-1,key)
5      A.insert(key,pivot-1) //insert key just "behind"
6                            //node returned by binary search
```

Then $T(n) = c_1 n + c_2(n-1) + (n-1)\Theta(\lg n) + (n-1)g(n)$. So the question of whether Insert-Sort-Binary-Search(A) is $\Theta(n \lg n)$ a matter of how long it takes to insert into A implemented as a linked-list. From http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html we see that ListIterator.add(E element) inserts in $O(1)$ so

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + (n-1)\Theta(\lg n) + (n-1)g(n) \\ &= c_1 n + c_2(n-1) + (n-1)\Theta(\lg n) + (n-1)O(1) \\ &= (n-1)\Theta(\lg n) \\ &= \Theta(n \lg n) \end{aligned}$$

The problem with this analysis is that linked-list **doesn't actually overload** [] and so Binary-Search-LL(A,p,r,x) isn't possible. Therefore while the number **comparisons** done by Insertion-Sort with a binary search finding the insertion point would be $O(n \lg n)$ the algorithm as a whole would still run in $O(n^2)$ in the worst case because there would still be $O(n^2)$ **swaps** that would need to be done in order to actually insert.

2.4 (a) (2,1), (3,1), (8,1), (6,1), (8,6)

(b) Reverse sorted, i.e. $\{n, n-1, \cdots, 1\}$. The number of inversions is $n-1$ for 1 because there are $n-1$ elements in the array which are larger than 1 but preceed it in the array, $n-2$ for 2 because there $n-2$ elements which are larger than 2 (exceptions are 2 and 1) but preceed it, and so on. So for $i = 1, 2, \cdots, n$ the number of inversions induced is $n-i$. In sum $\sum_{i=1}^{n} n - i = n\sum_{i=1}^{n} 1 - \sum_{i=1}^{n} i = n^2 - \frac{n(n+1)}{2} = n^2 - \frac{(n^2+n)}{2} = \frac{n(n-1)}{2}$

(c) Call an inversion of type (j,i) induced by an element $A[j]$ if j is such that $A[j] < A[i]$ and $i < j$ and let $|(j,i)|$ be the number of such inversions. Then $|(j,i)|$ is the number of swaps that will have to be performed on element $A[j]$ before it is in its proper position. To see that this is the case note that all (j,i) inversions persist through the sorting process, up until $A[j]$ is sorted, since for all $i < j$ element $A[i]$ will be inserted into the sorted portion of the array prior to $A[j]$. Therefore upon inserting $A[j]$ there will still be $|(j,i)|$ inversions and therefore $|(j,i)|$ swaps. Consequently $\sum_j |(j,i)|$ the total number of inversions in the array is the total number of swaps performed by insertion sort, i.e. directly proportional insertion sort's running time.

(d) Modify merge sort such that when the function returns from the two recursive calls, when the "merging" is done, it counts the number of elements in the "left" array each time an element is chosen from the front of the "right" array (after comparison between the leading elements of both arrays). The quantity of elements in the "left" array each time an element from the front of the right array is chosen is by definition the number of elements in the original array that were greater than that chosen element and yet preceeded it. Furthermore there is no double counting because once a merge happens 2 elements in the merged array are never compared again. Take for example the array $[8,7,6,5,4,3,2,1]$ and suppose the recursion bottoms out at 4 elements. Then the first recursion returns $[5,6,7,8]$ and $[1,2,3,4]$. The merging then selects each of the four elements from the "right" array $[1,2,3,4]$ since each of the elements in the "left" array $[5,6,7,8]$ is greater than each in the "right". Manifestly there are 4 inversions per element in the "right" array, 1 for each element in the "left" array, and so the total is 16.

3-1 Lemma: if $k, d \in \mathbb{N}$ and $k \leq d$ then $n^k \leq n^d$.
Proof: $n^k/n^d = n^{k-d} = 1/n^{d-k} \leq 0$ since $k - d \leq 0$.
Corollary: if $k, d \in \mathbb{N}$ and $k > d$ then $n^k > n^d$.

(a) Let $p(n) = \sum_{i=0}^{d} a_i n^i$ and $c = \max\{a_0, a_1, \ldots, a_d\}$. Note that $c \geq 0$ since otherwise all $a_i < 0$ and therefore $p(n) < 0$. Then for all $i \in \{0, 1, \ldots, d\}$ and for $k \geq d$ the lemma above implies $cn^k \geq a_i n^i$. Therefore $\sum_{i=0}^{d} cn^k = c \cdot (d+1) \cdot n^k \geq \sum_{i=0}^{d} a_i n^i = p(n)$.

(b) Let $p(n) = a_d n^d + \sum_{i=0}^{d-1} a_i n^i$. Then by part (a) $\sum_{i=0}^{d-1} a_i n^i \in O(n^{d-1})$ so there exists c_1 such that $\sum_{i=0}^{d-1} a_i n^i \leq c_1 n^{d-1}$. Therefore $p(n) \geq a_d n^d - c_1 n^{d-1}$. Fix any $\epsilon < a_d$ and for $n > c_1/\epsilon$

$$\begin{aligned} p(n) &\geq (a_d - \epsilon)n^d + \epsilon n^d - c_1 n^{d-1} \\ &= (a_d - \epsilon)n^d + n^{d-1}(\epsilon n - c_1) \\ &\geq (a_d - \epsilon)n^d \\ &\geq (a_d - \epsilon)n^k \qquad \text{by the lemma} \end{aligned}$$

Hence if $c_2 = (a_d - \epsilon)$ and $n > c_1/\epsilon$ then $p(n) \geq c_2 n^k$ and therefore $p(n) = \Omega(n^k)$.

(c) Let $p(n) = \sum_{i=0}^{d} a_i n^i$. By part (a) we have that for $k = d$ there exist c_1 and n_0 such that $p(n) \leq c_1 n^k$. By part (b) we have that for $k = d$ there exist c_2 and n_1 such that $p(n) \geq c_2 n^k$. Let $n_2 = \max\{n_0, n_1\}$ then for all $n \geq n_2$ we have that $c_2 n^k \leq p(n) \leq c_1 n^k$. Therefore $p(n) = \Theta(n^k)$.

(d) Let $p(n) = \sum_{i=0}^{d} a_i n^i$. Part (a) guarantees there exist constants c_1 and n_0 such that for $n_0 \geq n$ it's the case that $p(n) \leq c_1 n^d$. Then for $k > d$ and any constant c_2 let n_1 be be such that $n_1^{k-d} > c_1/c_2$. Such an n_1 can be chosen since $k - d > 0 \implies \exists n_j$ s.t. $n_j^{k-d} > c$ for any constant c. Then $c_2 n_1^k - c_1 n_1^d = n_1^d(c_2 n_1^{k-d} - c_1) > 0$. Finally let $n_3 = \max\{n_0, n_1\}$ and for all $n \geq n_3$ we have $c_2 n^k > c_1 n^d \geq p(n)$.

(e) The argument that worked for part (a) works here. For any c let $\epsilon = |a_d - c|$ and $n_0 > c_1/\epsilon$. If $a_d > c$ then there's no problem and the proof from part (b) works. Otherwise let n_1 be such that for $n \geq n_1 \implies n > \sqrt[d-k]{c/(a_d - \epsilon)}$ then

$$n > \sqrt[d-k]{\frac{c}{a_d - \epsilon}} \implies n^{d-k} > \frac{c}{a_d - \epsilon} \implies n^d > \frac{c}{a_d - \epsilon} n^k \implies (a_d - \epsilon)n^d > cn^k$$

Therefore

$$\begin{aligned} p(n) &\geq (a_d - \epsilon)n^d + \epsilon n^d - c_1 n^{d-1} \\ &= (a_d - \epsilon)n^d + n^{d-1}(\epsilon n - c_1) \\ &\geq (a_d - \epsilon)n^d \\ &> cn^k \end{aligned}$$

Hence $p(n) = \omega(n^k)$.

4.5 (a) Let n_b be the number of bad chips and n_g be the number of good chips. Without loss of generality assume there is only one good chip, $n_g = 1$. Then picking there are $1 + n_b$ choices chip is in fact the good one, i.e. any algorithm must distinguish between $1 + n_b$ different cases. Suppose that all bad chips always report any other chip to be "bad". Then there is not information to distinguish between the $1 + n_b$ different cases since the good chip as well always report "bad" (so there is only one bit of information).

(b) Divide the set of chips into pairs. If there is an odd number of chips set the odd one aside. Perform the comparisons within each pair. If both chips in a pair report the other chip "good" then keep either one, otherwise discard both. Repeat the process. Let n_{b_j} and n_{g_j} be the number of bad chips and good chips left after the jth iteration of the algorithm. At any step in the recursive process a majority of the remaining chips will be good since the only way to discard a good chip is if it is paired with a bad chip and therefore discarding both preserves $n_{g_j}/n_{b_j} > 1$ for all j. The recursion bottoms out and the invariant is preserved so the last chip left must be good. The halves the problem size since half the chips are discarded at every step.

(c) By part one we can find at least one good chip in $T(n) \approx T(n/2) + O(n)$. Let c_1 be such that $T(n) \leq T(n/2) + c_1 n$ and c_2 such that $c_2 > 2c_1$. Then the inductive hypothesis is $T(n) \leq c_2 n$. Then we have

$$T(n) \leq \frac{c_2 n}{2} + c_1 n = c_2 n - n\left(\frac{c_2}{2} - a\right) \leq c_2 n$$

Hence it takes $T(n) = O(n)$ to find one good chip, which you can then use to sort the rest of $n-1$ chips using $n-1$ comparisons, stopping early if you've found all of the good chips. Hence total time is $\Theta(n)$.

7.6 This problem was solved with the help of http://alumni.media.mit.edu/~dlanman/courses/cs157/HW3.pdf

(a) Let points[i][j] be the $n \times 2$ array with points[i][0] = a_i and points[i][1] = b_i. Let Find-Intersection be a function that computes intersections of subsets of intervals

```
1  Find-Intersection(points,p,s)
2          j = random(p,s)
3          swap(points[p],points[j])
4          a = points[p][0]
5          b = points[p][1]
6          for i=p to s
7                  //              a|---------|b
8                  //   x|--------|y
9                  //              u|------|v
10                 // the only requirement for intersection is
11                 // that both the left end point of interval
12                 // being checked is <= b
13                 // and the right end point is >= a so
14                 if points[i][0] <= b and points[i][1] >= a
15                         // check if narrower on the left
16                         if points[i][0] > a
17                                 a = points[i][0]
18                         // check if narrower on the right
19                         if points[i][1] < b
20                                 b = points[i][1]
21         // the intersection interval
22         return (a,b)
```

The call to random on line 2 is inorder to improve expected running time. If all intervals overlap then Find-Intersection returns the largest region of overlap. Otherwise on average it returns the largest possible overlap of a subset of intervals. Then let Partition-Intervals be essentially the same as Partition from Quicksort except it takes a key to compare against, a flag to pick whether to compare a_i or b_i, and a comparator $\{\leq, \geq, <, >\}$.

```
1  Partition-Intervals(points,p,r,flag,key,Comparator)
2          i = p-1
3          for j=p to s
4                  // compare a_i or b_i
5                  if points[j][flag] Comparator key
6                          i++
7                          // swap intervals in the array
8                          swap(points[i],points[j])
9          return i
```

Finally Fuzzy-Quicksort

```
1  Fuzzy-Quicksort(points,x,y)
2          if x<y
3                  (a,b) = Find-Intersection(points,x,y)
4                  // right
5                  r = Partition-Intervals(points,x,y,0,a,<=)
6                  // left-middle
7                  q = Partition-Intervals(points,x,r,1,b,<)
8                  Fuzzy-Quicksort(points,x,q-1)
9                  Fuzzy-Quicksort(points,r+1,y)
```

The first call to Partition-Intervals sorts the intervals by their a_i (hence flag = 0) on $\leq$ around the a returned by Find-Intersection. After this first call all intervals to the right of r are outside of the intersection. Why? If there were an interval $[a_i, b_i]$ such that some portion of it overlapped the intersection and $a_i \leq a$ then the it would have been considered in Find-Intersection on line 16. Hence all intervals in points[i] with indices $i > r$ are outside of the intersection. Therefore all intervals in points[i] with indices $i < r$ are either inside the intersection or outside the intersection but on the left. The second call to Partition-Intervals sorts the intervals by their b_i (hence flag = 1) on $<$ around the b returned by Find-Intersection. After this second call all intervals to the left of q are outside the intersection. Why? If there were an interval $[a_i, b_i]$ such that some portion of it overlapped the intersection and $b_i < b$ then it would have been considered by Find-Intersection on line 19. Therefore all intervals in points[i] with indices $i < q$ are outside the intersection but on the left. Therefore all intervals in points[i] with $q \leq i \leq r$ are in the intersection and their permutation is irelevant because we know they intersect and therefore either a or b returned by Find-Intersection is a correct c^* such that $[a_j, b_j] \forall j \in \{[a_q, b_q], \ldots, [a_r, b_r]\}$.

(b) The macroscopic structure of Fuzzy-Quicksort is almost the same as Randomized-Quicksort . Find-Intersection is clearly $\Theta(n)$ since it considers every interval. Note also that when all intervals are disjoint Find-Intersection returns just the first interval, the one chosen by random on line 2. Therefore in such an instance Find-Intersection has exactly the same effect as wrapper Randomized-Partition for Partition on page 179 of CLRS. Partition-Intervals is almost exactly the same as Quicksort 's Partition and so runs in $\Theta(n)$. Therefore worst case is similarly $\Theta(n \lg n)$. Best case is when all intervals intersect, in which case Find-Intersection returns an intersection that's contained every interval and the first call to Partition-Intervals returns index y and the second call returns index x. Then the two recursive calls return immediately since $x \not\leq x - 1$ and $y + 1 \not\leq y$. Therefore in this case, when all the intervals overlap the algorithm runs in $\Theta(n)$.

8.4 (a) Compare every red jug to every blue jug. Exactly n^2 units of time. Hence $\Theta(n^2)$.

(b) Fix a permutation of the red jugs, call it $(r_1, r_2, \ldots, r_n)$. Note that it is not necessarily the case that $r_i \leq r_j$ if $i < j$ or anything like this. Then the problem is tantamount to finding a permutation of blue jugs, $(b_{\pi(1)}, b_{\pi(2)}, \ldots, b_{\pi(n)})$ such that $b_{\pi(i)} = r_i$. Note there are $n!$ such permutations. Any comparison based algorithm must distinguish between 3 different cases at each step: either $b_{\pi(i)} < r_j$ or $b_{\pi(i)} > r_j$ or $b_{\pi(i)} = r_j$ and overall must distinguish between $n!$ permutations of the b_i. Thus the decision tree that a comparison based algorithm must traverse is a 3-ary tree with minimum height $h \geq \log_3 n!$. Then by **eqn. 3.19** in CLRS we have that $h \geq \Theta(n \lg n)$.

(c) Pick a random blue jug, the "blue pivot", and use Randomized-partition to partition all the red jugs into bigger, smaller, and equal. Then use the red jug that's equal in size to the blue pivot, call it the "red pivot" to similarly partition all blue jugs. Then recurse choosing a random new "blue pivot" in each of the "smaller-than" and "bigger-than" sets. The recurrence relations is the same as for Randomized-quicksort except the divide step uses Randomized-partition twice instead of once. So absorbing the constant factor the recurrence relation is $T(n) = \max_{0 \leq q \leq n-1}(T(q) + T(n-q-1)) + 2\Theta(n) = \max_{0 \leq q \leq n-1}(T(q) + T(n-q-1)) + \Theta(n)$. Hence by the analysis of Quicksort we have have expected running time of $O(n \lg n)$, with worst case running $\Theta(n^2)$.