Evan Louie
Austin Yen
4/29/16

Homework 3

ejlouie, auyen

1. $T(n) \leq 7T(n/3) + n^2$ Master Theorem

$a = 7 \quad b = 3 \quad f(n) = n^2$

$n^{\log_3 7} \approx n^{1.77}$

for $f(n) = n^2 = n^{1.77}$ must add constant $\epsilon = 0.23$

$f(n) = \Omega(n^{\log_3 7 + \epsilon})$ so $T(n) = \Theta(f(n))$

$T(n) = O(n^2)$

$T(n) \leq 7T(n/3) + n$ Master Theorem

$a = 7 \quad b = 3 \quad f(n) = n$

$n^{\log_3 7} \approx n^{1.77}$

for $f(n) = n^1 = n^{1.77}$ must subtract constant $\epsilon = 0.77$

$f(n) = O(n^{\log_3 7 - \epsilon})$ so $T(n) = \Theta(n^{\log_b a})$

$T(n) = O(n^{\log_3 7})$

$T(n) \leq 7T(n/3) + 1$ Master Theorem

$a = 7 \quad b = 3 \quad f(n) = 1$

$n^{\log_3 7} \approx n^{1.77}$

for $f(n) = n^0 = n^{1.77}$ must subtract constant $\epsilon = 1.77$

$f(n) = O(n^{\log_3 7 - \epsilon})$ so $T(n) = \Theta(n^{\log_b a})$

$T(n) = O(n^{\log_3 7})$

2. $T(n) < 2T(n/2) + \sqrt{n}$, $T(1) = 1$

Induction Hypothesis: $T(n) \leq n$

Base Case: $n_0 = 2$, $T(2) < 2T(\frac{2}{2}) + \sqrt{2}$

$T(2) < 2 + \sqrt{2}$

$T(2) < 2$ from IH

Inductive Step: $T(n) < 2T(\frac{n}{2}) + \sqrt{n}$

$< 2\frac{n}{2} + \sqrt{n}$

$< n + \sqrt{n}$

$= O(n)$

3. Initialize count = 0 as global counter

```
Inversions (A)
  n = A.length
  if n = 1 return 0
  L = [1 ... n/2]
  R = [n/2 + 1 ... n]
  Linvert = Inversions (L)
  Rinvert = Inversions (R)
  return merge (Linvert, Rinvert)

Merge (B, C)
  m = B.length
  p = C.length
  B[m+1] = ∞
  C[p+1] = ∞
  initialize D as empty array
  while B[i] < ∞ and C[j] < ∞
    if B[i] < C[j]
      D[k] = B[i]
      m = m - 1          // If it less, then we place B[i]
      i = i + 1          // into D[k], then we subtract 1 from
    else                 // m to update how many elements are in B
      D[k] = C[j]
      count = count + m  // If it more, then we increment
      j = j + 1          // the count by how many elements
    k = k + 1            // are currently in B
  return D
```

The algorithm Inversions is based off of the algorithm mergesort. The only differences are that we have a global counter for the number of inversions, and we are keeping track of how many elements are in array B after an element of B is placed inside of the sorted array D.
When merge is called within Inversions, the function compares the 2 arrays. If B[i] is less than C[j], then what is in B[i] is placed into D[k], the sorted array. In this case we also subtract 1 from m, which is the length or how many are in array B. If B[i] is less than C[j], then we place C[j] into the sorted array D, but also increament the global count by the current length of B.
When Merge is finished, we return the array D.
The time complexity of this algorithm is O(nlogn).
In class we have already shown that the time complexity of mergesort is O(nlogn). Since our algorithm is identical, with the addition of adding in a count, which is a O(1) function, then Inversions is also O(nlogn)

This algorithm works based on the placement of elements from the given array. In the merge function, array B is the left side while array C is the right side. If you choose an element from C before an element from B, that element from C is smaller. and therefore if there are any elements in B that have not been placed in array D, that is however many inversions you have between the two arrays B and C for that one element from C. After these two arrays have been combined to form D, we consider the next pair of arrays. This method is useful because pairs that have already been deemed to be inversions are not counted again. For example, the D we described above is now the new C in a new instance of Merge. We only compare this to the new B, and the elements in C do not interact with each other, and this doesn't matter because we have already counted the number of inversions within C (and also the new B as well).

Q4 Array A, n elements
want to find k^{th} element
Pivot is A[1]
After calling Partition(A)

$[1, \ldots, i-1]$ i $[i+1, \ldots, n]$

Array a Pivot Array b

if $i > k$, then discard array b and run partition on array a
if $i = k$, then A[i] is the kth element
if $i < k$, then discard array a and run partition on array b

Part 1

```
Quicksort (A, k)
  j = Partition (A)
  if j > k
      L = A[1 ... j-1]
      Quicksort (L)
  else if j < k
      R = A[j+1 ... n]
      Quicksort (R)
  else if j = k
      Return A[j]

Partition (A)   // same as given in class; not changed
  n = A.length
  pivot = A[1]
  i = 2
  j = n
  while (true)
      do i++ while A[i] < pivot
      do j-- while A[j] > pivot
      if i > j
          break
      swap A[i] and A[j]
  swap A[1] and A[j]
  return j
```

Part 2 The worst case would be when k is the last element in an ordered (either sorted or reverse sorted) array; in other words, you are searching for the largest element in an ordered array. Like standard quicksort, the time complexity in this case would be $O(n^2)$.

1 2 3 4 [5] 5 4 3 2 [1]

Part 3 When the pivot is randomly chosen, there is a smaller chance of encountering the worst case. In an average case, this algorithm acts like binary search, except that the array is not sorted. It partitions, compares k to the pivot, and returns the pivot if it equals k. If not, it only returns the appropriate half-array to be partitioned again. This process has a time complexity of $O(\log n)$. Partition has a time complexity of $O(n)$. Therefore, this algorithm has a time complexity of $O(n \log n)$

Q5. Quicksort is not a stable sorting algorithm. In this modified version, it only compares the values of the pivots to k. If it assigns the pivot to something equal to the value of the kth smallest value, it returns that value. When you ask for the kth smallest number, you generally want the first instance of the kth smallest number you come across. Not only does quicksort not stably sort, but it might randomly select the wrong instance of the kth smallest element and return that one.

Algorithm on Back

```
Partition(A)
  n = A.length
  pivot = A[1]
  i = 2
  j = n
  x = 2
  while (True)
    while A[i] ≤ pivot
      if A[i] = A[1]
        swap A[x] and A[i]   // move anything equal to pivot to front
        x++
      i++
    while A[j] > pivot
      j--
    if i > j
      break
    swap A[i] and A[j]
    for k = 1 to x
      swap A[k] and A[j-k]   // move everything equal to pivot next to pivot
    return (j-x-1) x100 + j   // encode, only works for A.length < 100


Quicksort(A, k)
  j = Partition(A)
  left = ⌊j/100⌋
  L = [1 ... left]
  Middle = j - left
  M = [Left+1 ... Middle]   // Array holding everything equal to j
  R = [Middle+1 ... n]
  if j > k
    Quicksort(L)
  if j < k
    Quicksort(R)
  //else it is in M if j = k
```