Henry Daniels-Koch
September 30 2016
Worked with Bo Bleckel

Algorithms: Lab 3

Homework Problems

1). If we chose the element at index n/2 as our pivot instead of the last element, a sequence that is already sorted would run in $\theta(nlg(n))$. This is because each partition would allow half of the numbers in the array to be less than the pivot and half greater than the pivot. In each partition, the array would continue to be sorted creating evenly split arrays at each call to quickSort. With evenly split arrays, we observe $\theta(nlg(n))$ as in the case of where the entry at the last index in a normal version of quickSort is the middle number in the array.

However, there exist sequences in which this version of quickSort would run in $\theta(n^2)$. These sequences could be sorted in ascending order within each half of the array with the largest element at index n/2. In this case, all elements would be swapped with themselves and the partition would split the array into 0 and n-1 element sized pieces.

e.g. $5, 6, 7, 8|1, 2, 3, 4$

These sequences could also be sorted in descending order within each half of the array with the smallest element at index n/2. In this case, no elements would be swapped and the partition would split the array into n-1 and 0 element sized pieces.

e.g. $4, 3, 2, 1|8, 7, 6, 5$

2a). We aim to create an algorithm that sorts an array of 0's and 1's in $O(n)$ time, while leaving equal elements in the same order in which they were intially (stable). We utilize a linked list and iterate the array. If we encounter a 0, we put is on the leftmost node of the list. If we encounter a 1, we put it on the right most node

pseudocode
```
Create a linked list B
for i from i=0 to n-1
    if A[i] == 0 : A[i]->B
    else : B->A[i]
```

b). We aim to create an algorithm that sorts an array of 0's and 1's in $O(n)$ time that also sorts in place. We iterate through the array and swap the first 0 with the element at the first index of the array, the second 0 with the element at the second index of the array and so on. We iterate until the end of the array, which automatically leaves the ones on the right side and 0's on left.

pseudocode
```
zeroSwaps = 0
for i from i=0 to n-1
    if A[i] == 0 :
        swap A[i],A[zeroSwaps]
        zeroSwaps ++
```

c). We aim to create an algorithm that sorts an array of 0's and 1's in place, while leaving equal elements in the same order in which they were intially (stable). We can utilize a bubbleSort algorith that swaps adjacent elements if the element on the left is a 1 and the element on the right is a 0.

pseudocode
```
for i from i=0 to n-2
```

```
#Do a bubble pass
for j=0 to n-2
    if A[i]>A[i+1] : swap A[i],A[i+1]
```

d). We aim to create an algorithm that sorts an array of 0's,1's and 2's in $O(n)$ time that also sorts in place. We can use a similar algorithm to the one described in (b), however we must keep track of the number of swaps with 0's and 2's in order to swap elements from the outside in. We iterate through the array and swap the first 0 with the element at the first index and the first 2 with the element at the last index. We keep track of this swap and next time swap the second 2 with the element at the second index. Similarly, we swap the second 2 with the element at the second to last index. We iterate until the end of the array, which automatically leaves the ones in the middle (and 0's on left and 2's on right).

```
pseudocode
leftSwaps, rightSwaps = n-1
for i from i=0 to n-1
    if A[i] == 0 :
        swap A[i],A[leftSwaps]
        leftSwaps ++
    elseif A[i] == 2 :
        swap A[i],A[rightSwaps]
        rightSwaps –
```

3a). Stooge-Sort correctly sorts the input array A[1...n] where n=length[A] by splitting the array recursively into 2/3 length sections. The algorithm first compares the first and last elements of the array and swaps them if necessary. Next the algorithm recursively calls itself on the first 2/3 of the array. The algorithm will break the array down until a section only has two elements while continuously comparing the first index to the last index of the array which effectively finds the minimum of the array. It is helpful to conceptualize stooge-sort by acting on a 3 element array, which occurs right before the base case. At this point, it compares the first two elements, swaps them and moves on to the next call of Stooge-Sort(A,i+k,j). This acts on the last 2/3 of the array or in this 3 element case, it acts on the 2nd and 3rd element. To finish ordering this 3 element array, one must then compare the first and second element again to account for the 2nd comparison. One can see an example of this attached.

To understand how this algorithm can be generalized from the micro-scale case described above to a larger scale, we now describe a large array A of 100 elements. In general, this algorithm sorts the first 2/3 of the array, then sorts the last 2/3, then sorts the first 2/3 again. With 3 recursive calls, we could sort any fraction of the array with $\geq 2/3$. However, we need to sort chunks of at least 2/3 in order to provide every element the chance to traverse across the array. Futhermore, the first sort gives elements in the first third of the array a chance to make it to middle third of the array. The second sort allows these elements in the middle third to make it to the last third in sorted order while also giving elements in the last third the chance to move to the middle third. Finally, the last sort is required to allow these new elements in the middle third the chance to move to the first third of the array. We provide an example attached.

b). To obtain T(n), we get 3 recurrence terms from the 3 recursions on 2/3 of the array as follows: $T(n) = T(2/3n) + T(2/3n) + T(2/3n) + \theta(1)$. As shown on the attached paper, this yields a tight asymptotic bound of $\theta(n^2.7)$.

c). This worst case running is much worse than insertion sort $\theta(n^2)$, merge sort $\theta(nlgn)$, heapsort $\theta(nlgn)$, and quicksort $\theta(n^2)$. Thus, the professors should be laughed at and made into adjunct professors.

4). We aim to merge k sorted lists into one sorted list with n total elements in all of the input lists. We

first build a min-heap of the minimums (first elements of the sorted lists) in $\theta(k)$ time since there are k minimum elements. We then delete the absolute minimum of the minimums from the k lists which is automatically located at the root of the min-heap as in the first step of the heap-delete-min function. We insert it as the first element of a soon to be sorted array B. Just as in the heap delete-min function, we move the element from the rightmost leaf on the lowest level to the root and delete the leaf. We then repeatedly swap elements with the smaller of the children elements until the heap property is satisfied (heapify). We continue deleting the root and inserting it as the next element in the array until the sorted array is full, leaving the heap empty. This algorithm uses the fact that each list is sorted because we can find the list's minimum in constant time. This algorithm runs in $\theta(nlgk)$ time because for n elements that we delete from the heap of minimums, we need to heapify a heap of height k since there are k minimums. This height k corresponds to $\theta(lgk)$ run time.

Pseudocode:

```
mergeSortedLists(lists,n)
#Create array of minimums from all lists
for i=0 to k-1
    A[i] = k_i[0]
#Build a heap of minimums from k sorted lists
for i=n/2 down to 1:
    heapify(i)
for i=0 to n-1
    B[i] = heap.root
    deleteMin
#deleteMin takes the element from the rightmost leaf, puts it in the root and heapifies the heap. Implicitly, the new minimum for each list is just the next element in the array since the array is in order.
```