Algorithms and Data Structures Final exam

1.

a)

Algorithm F reverses the n elements of an array. It works in a way such that the first element becomes the last and the second element becomes the second to the last, the third element becomes the third to the last, etc. So in simple, what is happening is the first and last elements are switched, next the second element and the second last one, and so on and so forth. It's akin to two invisible lines, one starting from the left and the other from the right, pointing toward each other and narrowing down to reach the middle of the array, reversing elements. Recursive calls are carried out on the elements remaining to be reversed. Further notes to point out are the initial check to see if the array consists of at least two elements in the subarray (the base case): if this condition fails, the block below won't run and the program terminates. Variables p and r can be seen as low and high, while x serves as a termporary variable to be used in swapping the two. The recursive call (which increments p and decrements r – the narrowing in on array effect mentioned earlier) at the end basically recurs on the rest remember. And if an even array is the input, the base case is reached as p == r+1; otherwise, the base case it reached since p==r for an odd array..

b)

1 5 7 9 3 4 1

1 5 7 9 3 4 1

1 4 7 9 3 5 1

1 4 3 9 7 5 1

1 4 3 9 7 5 1

c)

[Assume an arrow pointing downward to each of the subsequent recursive calls shown below; for example, F(A,0,6) has an arrow pointing down toward it, and it points down to F(A,1,5), and so on. Going bottom up, each recursive call returns to the one above it; so we're just returning from the recursive calls ultimately.]

F(A,0,6)

F(A,1,5)

F(A,2,4)

F(A,3,3)

d)

Since we’re simply returning from the recursive calls, this indicates the algorithm is tail recursive. as there are no operations in what can be imagined as invisible return arrows pointing to the calls above them, ascending upwards.

e)

As only O(1) extra space is needed for the swapping of the elements (noting we’re only really investigating two indices each time we’re going to reverse), the algorithm is in-place.

f)

The time complexity of the iterative version is linear, O(n). (Most operations are just O(1) for the swaps; iteration of the while loop is O(n) with n as the number of elements.)

```
F(A, p, r)
  while p < r
    A[p], A[r] = A[r], A[p]
    p+1
    r-1
```

2.

For both answers, we base on the following definition from the Cormen book:

“Let f(n) and g(n) be functions mapping positive integers to positive real numbers.We say that f(n) is O(g(n)) if there is a real constant c > 0 and an integer constant   n0 ≥ 1 such that   f(n) ≤ c · g(n), for n ≥ n0.”

a)

True: 29 is a constant O(1) so it’s upper bounded by logn.

b)

False: max is n^3 which has order 3 and its Big-O is not n^2 which is of a lower order (2). If we used min of the two, we could say it’s the case.

3.

a)

Counting sort involves three arrays: A (from to 0), B (same size as a but it stores the sorted solution) . It takes three steps: it counts the recurrences of the values in the input array, recording it in an array (say C); secondly, the counts are modified to include a running sum and the modified count array shows the position of each object in output sequence. Third and last, the running sum is used to copy elements to B. So effectively it sorts integers in range 0 to k but without comparisons in O(n+k).

The sorted array is 1,2,3,4,4,5,7

b)

Radix sort works by sorting each digit from least significant digit to the most significant. In base 10, for example, it sorts by digits in 1's place then in 10's place, etc. It makes use of counting sort as its subroutine to sort the digits. Linear time is O(n).

Visually speaking, it looks like this:

Imagine Index on the and three digits on the right. We start from the right column and stable sort on it. Next we do it on the middle column and finally the leftmost one. I provide the base point of it. Just imagine sorting using this procedure until we get the final sorted array.

0 802

1 256

2 958

3 938

4 693

5 405

6 684

7 858

The sorted array will be: 256, 405, 684, 693, 802, 854, 938, 958.

4.

a)

[Note that S.top starts of at -1; also, assume the five elements given below go from index 0 to 5.

- Originally, no elements are containined.

-1 | 0 1 2 3 4   [These are just the indexes, only shown this one time; the array has nothing in it currently.]

- First, push 5 and increment S.top to point at 5, storing 5 in the array

5 - - - -   [There is only the 5 now, the rest is - meaning empty.]

- Second, push 2 and increment S.top to point at 2

5 2 - - -

- Third, push 4 and increment S.top to point at 4

5 2 4 - -

- Fourth, pop out the most recently added element (4), returning its value and decrementing S.top to point at 2 [the 4 is still there but can be overridde]

5 2 - - -

- Fifth, push 7 and increment S.top to point at 7

5 2 7 - -

- Sixth, pop out the most recently added element (7), returning its value and decrementing S.top to point at 2 (the previous 4 got overridden by the 7; the 7 is there there but like the 4 previously it's just not shown in the third – I drew for step 4 and this one)

5 2 - - -

b)

[Imagine S.top on the left pointing to a nil originally]

- First, push 5 so that S.top points to it

S.top -> 5

- Second, push 2: S.top is pointing to 5 but now also make the pushed 2 point to the 5, then point S.top to the 2, removing S.top's pointer to 5

S.top -> 2 -> 5

Third, push 4: S. top is pointing to 2 but now also make the pushed 4 point to the 2, then point S.top to the 4, removing S.top's pointer to to 2

S.top -> 4 -> 2 -> 5

Fourth, remove S.top's pointer to 4 and point it to 2 instead and remove 4's pointer to 2 so it's popped off (remembering that popping will return 4)

Fifth, push 7 (the process is similar to the previous pushes so I'll simply be writing the output as the idea is the same)

S.top -> 7 -> 2 -> 5

Sixth, pop 7 (similar idea to the popping in the fourth step so I'll just write the output of it; remember that popping returns the 7)

S.top -> 2 -> 5

5.

a)

A hash table is a data structure for the storage of key/value pairs. It has two main components: a hash table which is an array of fixed size and a hash function which maps elements from they keys' universe to a subset. Keys are mapped to the indexes of the array (noting that the set of keys is smaller than the hash table size). Hash operations include insertion, deletion and searching (all of which are constant time on average). (An extra note about hash tables is they perform worst case O(n) .)

b)

Because the universal key Is a lot bigger, a key might end up being mapped to the same position in the array as another– this is known as a hash collision. (Good hash functions may be able to reduce the chances of such collisions.) Hash table storage is used in file systems and cryptography.

c)

In the chaining method, a doubly linked list (which serves as a 'chain', supporting fast deletion) is used to store pairs that collide in the same bucket. Below it's visualised (arbitrarily): left part is the index and right has Key|Value. The <-> shows next pair in the linked list. Insertion is O(1) which is an advantage, thanks to the linked list.

d)

Imagine the left is the index part and the right shows the insertions, <-> shows the links.

0

1 -> 10 | Cat <-> 20 |Dog

2 -> 45 |Zebra

d)

0

1 -> 28 <-> 19 <-> 10

2 -> 20

3 -> 12

4

5 -> 5

6 -> 15 <-> 33

7

8 -> 17

e)

In open addressing, if there’s a collision, other cells are tried out (probed) until an empty cell is discovered. It does not use linked lists to resolve any hash collisions and pointers are avoided to provide more space for the slots. In determining which slot to probe, the hash function carries a probe index (beginning from 0) as a second input. Every position is tried until the table fills and the function tries to find a slot to insert in where there are no collisions.

f)

Imagine index is on the left and table’s slots on the right.

0	700
1	50
2	85
3	92
4	73
5	101
6	76