CS302/307 EXAM 1 Name: ANSWER KEY

Thursday September 26, 2019

The exam is pencil and paper only. Notes, books, and electronic devices of any form are not allowed.

Problem 1 [8pts]: Sorting: Big picture

[Circle the correct answer.]

True **False** Selection sort can detect sorted data in O(N) time but takes O(N^2) time to actually sort.

True False Bubble sort can detect sorted data in O(N) time but takes O(N^2) time to actually sort.

True **False** Insertion sort takes O(N^2) time to sort regardless of the input data.

True False Shell sort wraps insertion sort inside a gap-based outer loop that allows data to move greater distances faster. Some gap sequences lead to worst case run times of O($N^{1.5}$).

True **False** Quicksort is a divide-and-conquer algorithm that runs in O(N log N) time for any data.

True False Mergesort is a divide-and-conquer algorithm that runs in O(N log N) time for any data.

True False Divide-and-conquer algorithms recursively reduce a problem to non-overlapped smaller problems. Thus avoiding duplication of work improves the computational efficiency.

True **False** Comparison based sorting algorithms have a lower bound of O(N) comparisons.

Problem 2 [8pts]: Insertion sort: Implementation

(a [5]) Underline code problems. Provide corrected code to the right of the problem line.

```
template <typename T>
void insertion_sort(vector<T> &A) {
  int h, k, N=A.size();
  for ( h=1; h<=N; h++ ) {                         h < N
    T tmp = A[ h ];
    for ( k=N-1; h<k && tmp < A[ k-1 ]; k-- )      k = h,  0 < k
      A[ k ] = A[ k+1 ];                           k-1
    A[ 0 ] = tmp;                                  A[k]
  }
}
```

(b [3]) Why is vector A passed by reference and not by value?

To prevent creating a copy of A on the call stack

Problem 3 [12pts]: Quicksort: Algorithm

(a [4]) Describe the major steps of quicksort, then summarize the mode of operation in one statement.

Quicksort uses a pivot to partition array into subarrays, then recursively calls itself on said subarrays

Each call produces $\{\leq\}$, pivot, $\{\geq\}$ which places pivot in its proper location in the sorted data.

(b [4]) Under what circumstances does quicksort perform at its best? Why is it impractical to pursue such an implementation? Name two different methods for approximating the desired behavior?

When pivot = median value. Requires sorting to obtain

Alt 1: median-of-three selection (left, middle, right)

Alt 2: random selection

(c [2]) Under what circumstances does quicksort perform at its worst?

When pivot = min/max value (leads to empty sublists)

(d [2]) In its standard implementation, quicksort does not produce a stable sort. What does that mean?

The relative order of equal sort data is not preserved

Problem 4 [5pts]: Quicksort: Big-O cost

State the recurrence relations and their solutions for quicksort's best and worst case runtimes. Include base case cost(s). Do not show how to obtain the solution.

Best: $T(1) = 1$, $T(N) = 2T(N/2) + cN = O(N \log N)$

Worst: $T(1) = 1$ $T(N) = T(N-1) + (N-1) = O(N^2)$

Problem 5 [10pts]: Quicksort: Numerical example

Show details of the first iteration of quicksort applied to the sequence A = {D L R Q V E G}. Use median-of-three based pivot selection. Underline and sort the sublist of "three elements", put a square around the element which becomes a pivot. Use arrows to indicate the subsequent left-to-right search for elements to swap during partitioning.

0	1	2	3	4	5	6	
<u>D</u>	L	R	<u>Q</u>	V	E	<u>G</u>	left, middle, right
D	L	R	$\boxed{G}$	V	E	Q	sort
D	L	R	E	V	$\boxed{G}$	Q	move pivot to right-1
							partition / swap
D	E	R	L	V	$\boxed{G}$	Q	
D	E	$\boxed{G}$	L	V	R	Q	move pivot to middle of {≤}, {≥}
{≤}			{≥}				

Problem 6 [8pts]: Mergesort: Algorithm

(a [4]) Describe the two major steps of mergesort.

1. Recursively split array in half until subarray produced contains just a single element

2. Merge subarrays together while sorting

(b [2]) Comment on storage requirements.

Aux. array needed when merging (for typical implementation)

(c[2]) State whether the standard implementation of mergesort produces a stable sort or not.

Stable sort

Problem 7 [12pts]: Mergesort: Big-O cost

(a [8]) State the recurrence relation that describes mergesort's best case computational cost, then solve it using "telescoping." Show details of your work and include comments that explain critical steps.

$$T(1) = 1$$

$$T(N) = 2T(N/2) + cN$$

$$\Downarrow$$

$$\left.\begin{aligned} \frac{T(N)}{N} &= \frac{T(N/2)}{N/2} + c \\ \frac{T(N/2)}{N/2} &= \frac{T(N/4)}{N/4} + c \\ &\vdots \\ \frac{T(2)}{2} &= \frac{T(1)}{1} + c \end{aligned}\right\}$$

telescoping: log N expressions

(common terms cancel out)

$$\Downarrow$$

$$\frac{T(N)}{N} = T(1) + c \log N$$

$$\Downarrow$$

$$T(N) = N + cN \log N = O(N \log N)$$

(b [4]) Explain what storing the data in a linked list does to the computational cost. Which part of the computation readily supports such data. Which part of the computation becomes more cumbersome, and what is the impact on the overall big-O cost of the mergesort algorithm?

Merging readily supported by linked list (no aux array)

Linear search for where to split required

$\Downarrow$

Change in constant c above, but not big-O cost itself

Problem 8 [4pts]: Insertion sort: Big-O cost

State the big-O cost for insertion sort when each element need to be moved at most *k* positions.

(a [2]) Assume *k* is constant.

$$T(N) = k + k + \dots + k = kN = O(N)$$

(b [2]) Assume $k=pN$ where $0 < p \leq 1$.

$$T(N) = pN + pN + \dots + pN = pN^2 = O(N^2)$$

```
template <typename RandomAccessIterator>
void nth_element(RandomAccessIterator i0, RandomAccessIterator nth, RandomAccessIterator i1);
```

nth_element(i0, nth, i1) rearranges the elements in the range [i0, i1) such that the element at the nth position is where it would be in a sorted sequence. The other elements are ordered such that those in the range [i0,nth) are not greater than the nth element and those in the range [nth,i1) are not smaller than it. Average run time is O(N).

Problem 9 [4pts]: Nth_element : Algorithm

Give a simple description of how nth_element( ) described above could be derived from quicksort.

Partition function places pivot at its proper location in the sort order, say as kth element. If $k == n$, then done. Otherwise if $n < k$, recursively (or iteratively) descend into left subarray. Alt. if $n > k$, continue in right subarray.

Problem 10 [12pts]: Binary tree based sorting: Big-O cost

(a [4]) A balanced binary search tree supports O(N log N) sorting. State the big-O cost for building the binary search tree as well as the big-O cost for traversing it to produce the sorted data.

Building: $O(N \log N)$ Traversal: $O(N)$

(b [4]) A binary min-heap supports sorting data in O(N log N) time. State the big-O cost for heapifying array data into a binary min-heap as well as the big-O cost for producing the sorted data.

Heapify: $O(N)$ Extraction $O(N \log N)$

(c [4]) Explain how to use a binary min-heap to carry out a partial sort of the just the first *k* elements. State the associated big-O cost.

Build min-heap in $O(N)$ using heapify, then extract each of the k elements in $O(\log N)$

$\Downarrow$

Total cost is $O(N + k \log N)$

Problem 11 [4pts]: Indirect sorting: Implementation

Briefly describe how the concept of a smart pointer can be used to sort linked list data using any of the sorting algorithm that applies to array data.

The smart pointer is a wrapper class that translates p1 < p2 into *p1 < *p2. Sorting of smart pointers is thus equiv. to sorting data itself, except indirectly.

The sorted smart pointers are ultimately used to correctly order the data pointed to.

Problem 12 [10pts]: Search trees: Fill in the blanks

Balanced by design, AVL and red-black binary search trees require O(log N) operations for each lookup/insertion/deletion. Tree imbalance is detected using height and color properties, respectively. Balance is in both cases restored thru one or more rotation operations. For slow-access data such as file systems and databases, B-trees are more efficient. Based on multi-way branching, a search tree of this type can hold $m^{H} - 1$ keys where H denotes the tree height and m denotes branching factor. Let m = 2d + 1. Then d denotes the min. number of keys stored by an internal node while 2d denotes the max. number of keys stored by any node including the root which can hold as little as a single key. All leaf nodes must be at the same depth. The result is slow growing tree of limited height.

Problem 13 [3pts]: File I/O

Briefly explain the difference between formatted and binary file I/O.

Formatted file I/O : data ≡ ASCII (interpretation)

Binary file I/O : data ≡ bytes (no interpretation)