Test 2: CPS 100

Owen Astrachan

April 10, 2000

Name: ________________________________

Login: __________

Honor code acknowledgment (signature) ________________________________

<table>
<thead>
<tr>
<th>Problem</th>
<th>value</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>15 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>15 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>16 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>12 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 5</td>
<td>8 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 6</td>
<td>6 pts.</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>72 pts.</td>
</tr>
</tbody>
</table>

This test has 7 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 50 minutes.

In writing code you do not need to worry about specifying the proper \#include header files. Assume that all the header files we’ve discussed are included in any code you write.

Some common recurrences and their solutions:

- \( T(n) = T(n/2) + O(1) \) \( O(\log n) \)
- \( T(n) = T(n/2) + O(n) \) \( O(n) \)
- \( T(n) = 2T(n/2) + O(1) \) \( O(n) \)
- \( T(n) = 2T(n/2) + O(n) \) \( O(n \log n) \)
- \( T(n) = T(n-1) + O(1) \) \( O(n) \)
- \( T(n) = T(n-1) + O(n) \) \( O(n^2) \)
PROBLEM 1: (Sort stuff)

Part A, 9 points
We discussed three \( O(n^2) \) sorts. For each sort, briefly describe data /a situation for which the sort is preferable than using the other two.

1. Selection sort
2. Insertion sort
3. Bubble sort

Part B, 6 points
An implementation of Merge Sort for vectors is shown below.
(It would be called as mergesort(a, 0, a.size()-1).

```cpp
void mergesort(tvector<string> & a, int left, int right)
// postcondition: a[left] <= ... <= a[right],
{
    if (left < right)
    {
        int mid = (left+right)/2;
        mergesort(a, left, mid);
        mergesort(a, mid+1, right);
        merge(a, left, mid, right);
    }
}
```

Write a recurrence relation for mergesort as shown above. Assume that merge is \( O(n) \) for merging \( n \) elements.

Finding the middle of a linked list takes \( O(n) \) time for an \( n \)-element list, whereas finding the middle of a vector is \( O(1) \). What’s the complexity of mergesort for linked lists and why?
PROBLEM 2:  *(Heap stuff)*

The tree below on the left shows elements of a heap. The heap is actually stored in an array as shown on the right.

Part A, 3 points

What values can be added to the heap above without causing any existing heap elements to move? (describe every such value)

Part B, 3 points

Draw an array diagram of the heap that results from inserting 11 into the heap above (the array diagram you draw will be graded, not a tree picture).

Part C, 3 points

If the minimal element in the original heap above (that has 9 elements) is removed, draw an array diagram of the heap that results.

Part D, 3 points

The code below sorts a vector using a `tpqueue` (which is implemented with a heap). What is the complexity of this sort, briefly justify your answer.

```c++
void sort(vector<string>& a)
// post: a[0] <= a[1] <= ... <= a[a.size()-1]
{
    tpqueue<string> pq;
    int k;
    for(k=0; k < a.size(); k++)
    {   pq.insert(a[k]);
    }
    for(k=0; k < a.size(); k++)
    {   pq.deletemin(a[k]);
    }
}
```

Part E, 3 points

If the implementation of `tpqueue` is changed to use a balanced search tree rather than a heap does the complexity change? why?
PROBLEM 3:  (Huff Stuff)

Part A, 4 points
The table below shows the frequency of occurrence of every character in a text file. Draw the huffman tree for these characters and occurrences.

<table>
<thead>
<tr>
<th>character</th>
<th># occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>5</td>
</tr>
<tr>
<td>'t'</td>
<td>3</td>
</tr>
<tr>
<td>'s'</td>
<td>1</td>
</tr>
<tr>
<td>'o'</td>
<td>1</td>
</tr>
<tr>
<td>'e'</td>
<td>8</td>
</tr>
<tr>
<td>'r'</td>
<td>2</td>
</tr>
</tbody>
</table>

Part B, 4 points
In writing code to compress using Huffinan coding, each 8-bit chunk was mapped to a string of zeros and ones that was the encoding of the chunk. Rather than using a string, suppose an int is used, where the 0's and 1's in the int represent the encoding. This will work for many files, but not for all files. Why?

Part C, 4 points
Changing the implementation of the class tpqueue from a heap to an unsorted vector doesn’t have a noticeable impact on how long it takes the huff program to run. Why?

Part D, 4 points
In two or three sentences, why was the PSEUDO_EOF character needed/used in the implementation of huffman coding?
PROBLEM 4:  (*Mapquest*)

**Part A, 4 points**
You must determine if two vectors each containing $n$ strings (call the vectors $a$ and $b$) contain exactly the same strings. One easy way to do this is to sort both vectors, then traverse each from left to right examining the vectors to see if the corresponding strings are equal. This will work. What is the complexity in terms of $n$, where $n$ is the number of elements in each vector? Justify your answer briefly.

**Part B, 4 points**
Instead of sorting, suppose you use two maps of strings to ints (one map for each vector) where the map associates each string with the number of times it occurs in the vector. If the map is implemented using a binary search tree to store key/value pairs, what is the big-Oh complexity of this method in the average case? Justify your answer briefly (account for all map operations in analyzing the complexity.)

**Part C, 4 points**
If maps are implemented using hash tables instead of search trees what is the big-Oh complexity of the method in the average case? Justify your answer briefly.
PROBLEM 5: \((\text{Findking})\)

A common task is finding the \(k^{th}\)-largest in a list of items. The \(k^{th}\)-largest is larger than \(k\) elements. This means the 0-th element is the first in a sorted list, it’s larger than 0 elements. In a list of \(n\) elements, the \((n - 1)^{th}\)-largest is the maximal element.

In a sorted vector, it’s easy to determine the \(k^{th}\)-largest, it’s just \(a[k]\). However this requires sorting the vector. In a binary search tree of strings, the code below returns the \(k^{th}\)-largest string. You’ll be asked to determine the complexity of \(\text{findk}\).

```cpp
int numNodes(Tree * t) {
    if (t == 0) return 0;
    return 1 + numNodes(t->left) + numNodes(t->right);
}

string findk(Tree * t, int k) {
    if (t != 0) {
        int leftCount = numNodes(t->left);
        if (leftCount == k) return t->info; // found it
        if (k < leftCount) return findk(t->left, k); // in left subtree

        // we know there are leftCount + 1 nodes smaller, look right
        return findk(t->right, k - leftCount - 1);
    }
    // should never reach this point if precondition satisfied
}
```

Part A, 4 points
What is the complexity of the function \(\text{findk}\)? Justify your answer briefly (assume trees are roughly balanced).

Part B, 4 points
If the elements are copied into a vector and the vector is used for \(\text{findk}\), the complexity is \(O(1)\) for \(\text{findk}\), not including the time to copy into a vector or the extra storage of the vector. Is it worth it? Why?
PROBLEM 6:  (Playing Games (6 points))
This problem involves a small change to the tic-tac-toe playing program we discussed in class. The question that you must answer after the code is “Why is the modification so much faster?”

The main logic of the function bestComputerMove to determine the Computer’s best move using backtracking minimax is shown below.

```c++
int bestScore = Game::HUMAN_WIN; // find max, initialize small
for(k=0; k < myBoard.size(); k++)
{
    if (myBoard.isClear(k))
    {
        myBoard.place(k, Board::X);  // put down an X
        score = bestHumanMove(dontCareMove); // see what happens
        myBoard.unplace(k);          // take the X off
        if (score < bestScore)
        {
            bestScore = score;
            move = k;
        }
    }
}
return bestScore;
```

In the program to play tic-tac-toe, this code results in more than 500,000 boards being examined by recursive calls to this function and the similar bestHumanMove that it calls for calculating the first move.

If the line

```c++
score = bestHumanMove(dontCareMove); // see what happens
```

is replaced with

```c++
string s = myBoard.toString();
if (myMap.contains(s))
{
    score = myMap.get(s);
}
else
{
    score = bestHumanMove(dontCareMove); // see what happens
    myMap.insert(s, score);
}
```

where myMap is defined as

```c++
HMap<string, int> myMap;  // hash-table based map
```

The computer plays the same game but makes only 6,000 recursive calls to calculate the first move. Why is there such a large reduction in recursive calls and why is the program so much faster as a result?