NAME:

University of Washington Tacoma
TCSS 543: Design and Analysis of Algorithms
Martine De Cock
Midterm #1, Thu Oct 29, 2009

- This test is closed book and closed notes. You do not need a calculator.
- Double-check that your exam copy consists of 5 pages (this page included).
- Write your name on this page.
- Read each question carefully before answering. Answer what is asked for in the question, not more and not less. Show your work for partial credit.
- *Write legibly* and organize your answers in a way that is easy to read. Use scratch paper to prepare your answer.
- Use the blank space under the questions for your answers. If you need more space for an answer, use the back of the *previous* page and *indicate that you did so.* If needed, you can also use the additional blank page at the end.
- Don't spend too much time on any one question. Some questions may be harder for you than other questions. The questions' weights give you an indication of how much time you are expected to spend on each of them.
- You can work on this exam until 9:20pm. When you are done and have checked your work carefully, you may turn in your exam copy and leave the classroom. However, don't start conversations with classmates in the hallway outside of the classroom, as this can be very distracting for students who are still working on the exam.

Question 1	/8
Question 2	/7
Question 3	/5
Total	/20

Question 1: the majority element problem (8 points)

An array $A[1 \ldots n]$ is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The tricky part is that the elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $A[i] > A[j]$?". That also means that *you can not sort* the elements in any specific order. (Think of the array elements as GIF files, say.) However you can answer questions of the form: "is $A[i] = A[j]$?" in constant time.

(a) (2 points) Suppose you would split the array A into two arrays A_1 and A_2 of half the size. Does knowing the majority elements of A_1 and A_2 help you figure out the majority element of A? Justify your answer.

(b) (4 points) Design an algorithm that solves the majority element problem in $O(n \log n)$ time. The input to your algorithm is a list $A[1 \ldots n]$ and the output is the majority element of A, if it exists. If there is no majority element, the algorithm should return *nil*. Present your algorithm in pseudocode.

(c) (2 points) Present an analysis of the runtime of your algorithm, showing that it indeed runs in $O(n \log n)$.

Your answers for question 1:

(a) Yes. The majority element of A will show up as either the majority element of A_1 or as the majority element of A_2 (or both). To show this, let $f_A(x)$ be the number of occurrences of x in A, $f_{A_1}(x)$ the number of occurrences of x in A_1, and $f_{A_2}(x)$ the number of occurrences of x in A_2. It holds that

$$f_A(x) = f_{A_1}(x) + f_{A_2}(x)$$

Let x be the majority element, i.e. $f_A(x) > n/2$. We need to show that $f_{A_1}(x) > n/4$ or $f_{A_2}(x) > n/4$. We do this by showing that $f_{A_1}(x) \leq n/4$ implies $f_{A_2}(x) > n/4$. So let us assume that $f_{A_1}(x) \leq n/4$, then

$$f_A(x) - f_{A_1}(x) > n/4$$

in other words $f_{A_2}(x) > n/4$. This shows that if x is not the majority element of A_1, it must be the majority element of A_2.

(b)
```
Algorithm Majority(A)
//input: an array A
//output: the majority element of A, or nil if such an element does not exist
n <- A.length
if n = 1
    return A[1]

let A1 contain the first n/2 elements of A
let A2 contain the remaining elements of A

x <- Majority(A1)
if x <> nil
    // test if it is a majority element for A too
    if Frequency(A,x) > n/2
        return x

x <- Majority(A2)
if x <> nil
    // test if it is a majority element for A too
    if Frequency(A,x) > n/2
        return x

return nil
```

(c) This is a divide-and-conquer algorithm that looks for the majority element in a list A of length n by recursively looking for the majority elements in 2 lists A_1 and A_2 of length $n/2$. In the conquer part, `Frequency(A,x)` is called at most twice. Such a function can be implemented in linear time (traverse list A and keep track of how many times you encounter x). Hence the overall runtime of the algorithm is characterized by $T(n) = 2T(n/2) + O(n)$, i.e. $T(n) \in O(n \log n)$.

Question 2: the hotel reservation problem (7 points)

You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a_1 < a_2 < \ldots < a_n$, where each a_i is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance a_n), which is your destination. To allow you enough rest, you do not want to drive too many miles a day. On the other hand, you also want to make sufficient progress towards your destination every day. You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the penalty for that day is $(200 - x)^2$. You want to plan your trip so as to minimize the total penalty — that is, the sum, over all travel days, of the daily penalties.

(a) (4 points) Give an efficient algorithm that determines an optimal sequence of hotels at which to stop. The input to your algorithm are the mile posts of the available hotels $a_1, a_2, \ldots, a_n$ (sorted in increasing order), and the output is the list of hotels at which you should stop to minimize the total penalty of your trip.

(b) (2 points) Include a brief explanation of why your algorithm is correct.

(c) (1 point) What is the runtime efficiency class of your algorithm? Justify your answer.

Your answers for question 2:

The greedy strategy of picking the hotel that is as close as possible to 200 miles away from your currect location does not seem to work. E.g. let $a_1 = 100$, $a_2 = 199$ and $a_3 = 202$. a_3 is the final destination. The greedy strategy mentioned above would suggest to drive from mile post 0 to a_2 on the first day, and then from a_2 to a_3 on the second day. The optimal solution would however be to drive from mile post 0 straight to a_3.

Some students have added the restriction to the problem that you can not drive more than 200 miles per day. In this case the greedy strategy does not work either, for it would again make you drive from mile post 0 to a_2 on the first day, and then from a_2 to a_3 on the second day (note that the penalty for that last little trip is very, very high). The optimal solution in this case would be to drive from mile post 0 to a_1 on the first day, and then from a_1 to a_3 on the second day.

(a) Let $OPT(j)$ be the smallest total penalty for getting from mile post 0 to hotel a_j. In other words, we are considering the subproblem of getting from mile post 0 to hotel a_j with possible stops at the hotels in between, i.e. we are looking at $a_1 < a_2 < \ldots < a_j$. Then

$$OPT(j) = \min_{0 \le i < j} (OPT(i) + penalty(a_i, a_j)) \qquad (1)$$

with $penalty(a_i, a_j) = (200 - (a_i - a_j))^2$. $OPT(0) = 0$, $a_0 = 0$, and $OPT(n)$ is the minimum total penalty for travelling from a_0 to a_n.

```
Algorithm HotelReservation (a)
//input: array a[0..n] of mile posts
// a[0] = 0, and all other mile posts have hotels
//output: array b of hotels where we should stop
// to keep the total penalty as low as possible
OPT[0] <- 0
for j <- 1 to n
    //compute the value of OPT[j] using formula (1)
    OPT[j] <- infinity
    for i <- 0 to j-1
        if OPT[i] + penalty(a[i],a[j]) < OPT[j]
            OPT[j] <- OPT[i] + penalty(a[i],a[j])
            R[j] <- i //keep track of the best previous stop before a[j]

// the smallest possible total penalty will be in OPT[n]
// we use traceback matrix R to find an optimal sequence of hotels
k <- 0
j <- n
while j > 0
    b[k] <- R[j]
    k <- k+1
    j <- R[j]

return reverse(b)
```

(b) They key to our algorithm is recurrence equation (1). If in an optimal solution to the subproblem of getting from mile post 0 to a_j we would have spent the night before arriving at hotel a_j in hotel a_i, then the minimum total penalty would be $OPT(i) + penalty(a_i, a_j)$. This is the minimum total penalty $OPT(i)$ for getting from mile post 0 to hotel a_i, plus the penalty for driving directly from a_i to a_j.

In (1) we are considering every hotel a_i between mile post 0 and hotel a_j as a candidate hotel to spend the night before arriving at hotel a_j, and take the minimum overall.

(c) The algorithm is quadratic. The number of times $C(n)$ that the instructions in the innermost loop are executed is

$$\sum_{j=1}^{n} \sum_{i=0}^{j-1} 1 = \sum_{j=1}^{n} j = \frac{n \cdot (n+1)}{2}$$

hence $C(n) \in O(n^2)$.

Question 3: cell phone base stations problem (5 points)

Consider a long road with houses scattered along it. You can picture the road as a long line segment with an eastern endpoint and a western endpoint; the houses would be dots on the line. You want to place cell phone base stations at certain points along the road (think of them as dots on the line too), so that every house is within four miles of one of the base stations.

(a) (4 points) Give an efficient algorithm that achieves this goal, using as few base stations as possible. Present your algorithm in pseudocode. You do not have to formally show its correctness, but please provide a brief explanation of the idea underlying your algorithm.

(b) (1 point) What is the runtime efficiency class of your algorithm? Justify your answer.

YOUR ANSWERS FOR QUESTION 3:

(a)
```
Algorithm PlaceBaseStations(A)
//input: array A of locations of houses, sorted from west to east
//output: array B of locations for cell phone base stations
// such that every house is within 4 miles of a base station
// and the number of base stations is minimal
n <- A.length
B[1] <- A[1] + 4 // place first base station 4 miles east from first house
j <- 1 // index in B
for i <- 2 to n
    if A[i] - B[j] > 4
       // house A[i] is no longer in range of previously placed B[j]
       j <- j+1
       B[j] <- A[i] + 4
return B
```

The algorithm traverses the houses from west to east. Whenever an encountered house is not yet in range of a cell phone base station, a new base station is placed 4 miles to the east of that house. This is the furthest distance possible to assure coverage for the house under consideration while also covering as many other not yet encountered houses as possible.

(b) There are $n-1$ iterations of the for-loop. Each iteration takes constant time. The overall runtime of the algorithm is $O(n)$. (Note that if we had not assumed that the houses were already sorted, the overall runtime would have been $O(n \log n)$ because of the additional sorting cost.)