Homework 5
Odd numbered problem solutions
cs161
Summer 2009

Problem 1 (10 points):

Algorithm: First we sort the set $\{x_1,x_2,\ldots x_n\}$ to get a set of sorted real numbers $S=\{y_1,y_2,\ldots y_n\}$

Beginning with y_1, cover the set with the interval $K_1 = [y_1,y_1+1]$, remove any points in S that are covered by K_1. Now suppose that y_i is the smallest real number in S that is not covered by K_1, add in the interval $K_2 = [y_i,y_i+1]$ and remove all points covered by K_2. Repeat this process until all the points are covered. We end up with a set T of intervals that cover S.

Proof of Correctness:

Suppose we are given another set of unit intervals T' that covers S, we claim that T' has at least as many intervals as T.

For each interval K_i in T, we pick out the left endpoint of K_i, which we call z_i, note that by the way we defined our intervals, $Z = \{z_i |\ i=1 \text{ to } i=|T|\}$ is a subset of S.

Now again by the way we defined the intervals in T, each z_i is distance > 1 from any other element in Z. Because Z is a subset of S, in order to cover S, T' must cover Z.

Now since there are |T| elements in Z We must have at least |T| unit intervals in T' to cover Z, one for each element in Z. If this is not the case, then there must be a unit interval in T' that covers more than 1 point in Z, this is impossible because any two points in Z are more than distance 1 from each other. QED

Running Time:

Use your favorite $O(n\log(n))$ sorting algorithm to sort the set. The algorithm itself runs in $O(n)$ time. Hence the overall running time is $O(n\log(n))$

Problem 3 (10 points):

Given the sequence s, we try to find the longest subsequence LP(s) of s such that LP(s) is a palindrome.

We find LP(s) recursively:

Given s, if s(1) = s(n) where n is the length of s, then LP(s) =s(1) + LP(s(2,n-1))+s(n), where s(2,n-1) is the subsequence of s with the first and last characters removed and '+' denotes concatenation

If s(1) !=s(n), then LP(s) = LP(s(1,n-1)) if $|LP(s(1,n-1))| \geq |LP(s(2,n))|$,

else: LP(s) = LP(s(2,n))

To include the base cases, we define LP(s(i)) = s(i) for any i from 1 to n

To implement this using dynamic programming, we need to have an nxn lookup matrix where for the (i, j) entry we store s(i) if i=j (i.e. the diagonals) and fill in the i<j entries using our recursive definition, the (1,n) entry would be the solution. We keep track of the path through the matrix that the algorithm dictates down to the diagonal and fill in the needed entries from the diagonal up to the (1,n) entry to get the longest palindrome.

The computational cost is $O(n^2)$

Problem 5 (10 points):

We solve this using dynamic programming:

Suppose on the first day, there are k hotels within range (i.e. within distance d from the starting point), for each of these hotels, we compute the penalty if we end up staying at that hotel, which we designate h_i. We define Cost(h_i) as the minimum total penalty for a trip that begins at hotel i. Hence, our sub-problems consist of trips that begin at hotels (Without loss of generality, we assume that each pair of consecutive hotels is within distance d of each other)

So the recursion is:

$$Cost(h_i) = \min_{\{k:k < i \text{ and } x\_i - x\_k <= d\}} ((d-x_k)^2 + Cost(h_k))$$

Pseudocode:

Initialize 1x(n+1) Cost array

Cost[0] = 0

```
Penalty(x)
        If x is within range,
                Return (d-x)^2
        Else
                Return ∞

for i = 1 to n
        min = Penalty[i]
        for j = 1 to i-1
                newpenalty = Cost[j] + Penalty(x_i-x_j)
                        if newpenalty < min
                                min = newpenalty
                Cost[i] = min
return Cost[n]
```

Computational Cost: There are essentially two for loops that we have to go over, and we have to store n sub problem solutions in a vector, the total computational cost is $O(n^2)$

CS 161 Summer 2009
Homework #5 Sample Solutions

Regrade Policy: If you believe an error has been made in the grading of your homework, you may resubmit it for a regrade. If the error consists of more than an error in addition of points, please include with your homework a detailed explanation of which problems you think you deserve more points on and why. We reserve the right to regrade your entire homework, so your final grade may either increase or decrease.

Problem 2 [15 points]

(a) (10 points) We order the subproblem by size. Let V be a one dimension boolean array. Define $V(i) = true$ if and only if the substring $s_1, s_2, \ldots, s_i$ is valid. That is, it can be decomposed into a sequence of words from the dictionary. In this problem we are interested in computing $V(n)$.
Now, we fill up V bottom up.

- Base case: $V(0) = false$;
- Recurrence: $V(i+1) = \bigvee_{1 \le k \le i+1} [V(k-1) \wedge \text{DICT}(S, k, i+1)]$, for $0 \le i < n$.

Here $\bigvee$ is the OR boolean operation. String $s_1, s_2, \ldots, s_{i+1}$ is valid as long as there exists some k, s.t. substring $s_1, s_2, \ldots, s_{k-1}$ is valid AND $s_k, \ldots, s_{i+1}$ is a word from the dictionary (actually the last word for this string).
Running time: $O(n^2)$. That is because the running time for computing each $V(i)$ is $O(i)$, for all $1 \le i \le n$.

(b) (5 points) In addition to the table V, we also store a table B, where if $V(i) = true$, then $B(i)$ is the index of the beginning index of the word ending at index i. We can modify the algorithm as follows: Initialize B with all entries as 0; Whenever we set entry $V(i+1) = true$, we set $B(i+1) = k$, where k is an index such that $[V(k-1) \wedge \text{DICT}(S, k, i+1)] = true$. (If there are more than one k that satisfy this condition, we can just pick one arbitrarily.)
After the recursive algorithm ends, if $V(n) = true$, we traverse B backwards and then print the string as described in the PRINT algorithm below.
The total running time of the modified algorithm is still $O(n^2)$.

Algorithm 1 PRINT(S, B)

```
1: i = n
2: while i > 0 do
3:    temp = B(i)
4:    B(i) = −1
5:    i = temp
6: end while
7: for i = 1 to n do
8:    PRINT s_i
9:    if i < n and B[i + 1] == −1 then
10:      PRINT ' '
11:   end if
12: end for
```

Problem 4 [10 points]

The optimal strategy is the obvious greedy one. Beginning from the start point, we should go to the farthest hotel that we can get to within d miles. Stay there for one night. Then go to the farthest hotel we can get to within d miles of where stayed, and rest there, and so on. If there are n hotels on the map, we need to inspect each one just once. The running time is therefore $O(n)$.

To prove the optimality of our algorithm, we use a "stays ahead" argument (as called in class).

For the first day, suppose there are k hotels beyond the start that are within d miles of the start. Our greedy algorithm chooses the k-th hotel as its first stop. No hotel beyond the k-th works as a first stop, since we will run out of the d miles upper limit. If a solution chooses a hotel $j < k$ as its first stop, then we could choose the k-th hotel instead, having at least as much miles to the destination when we leave the k-th hotel as if we'd chosen the j-th hotel. Therefore, we would get at least as far without stopping again if we had chosen the k-th hotel.

Now, suppose there are m possible hotels. Consider an optimal solution with s hotels and whose first stop is at the k-th hotel. Then the rest of the optimal solution must be an optimal solution to the subproblem of the remaining $m-k$ hotels. Otherwise, if there were a better solution to the subproblem, i.e., one with fewer than $s-1$ stops, we could use it to come up with a solution with fewer than s stops for the full problem, contradicting our supposition of optimality.

Thus our algorithm is optimal.

Alternately this argument can be viewed as follows: The first choice made by our algorithm is no worse than any other solution as in any solution we are better off postponing the first stop as much as possible. Next, observe that given the first stop, the remaining choices

can be viewed as a subproblem. As before, the first stop in this subproblem is chosen to be the farthest possible choice without go beyond d miles.

Extra Credit [5 points]

We can consider dynamic programming approach to solve the problem.
Assume W can be divided by 3. Otherwise we can return NotSolvable. We can construct a three-dimension boolean matrix M of size $(\frac{W}{3}+1) \times (\frac{W}{3}+1) \times (n+1)$, indexed from 0. Define $M[x, y, k] = true$ if and only if there are two disjoint subsets $I, J \subseteq \{1, \ldots, k\}$ such that $\sum_{i \in I} a_i = x$ and $\sum_{j \in J} a_j = y$. We construct the matrix as follows,

- Base case $k = 0$:
  $M[0, 0, 0] = true$;
  $M[x, y, 0] = false$, for $x + y > 0$.
- The recursive step:
  $M[x, y, k] = M[x - a_k, y, k-1] \vee M[x, y - a_k, k-1] \vee M[x, y, k-1]$, for $k = 1, \ldots, n$

Here $\vee$ means OR operation. That is, there are three possible ways to place number a_k.
(1) If we put a_k into the first partition (that is to insert k into I), then $M[x, y, k] = M[x - a_k, y, k-1]$;
(2) If we put a_k into the second partition (that is to insert k into J), then $M[x, y, k] = M[x, y - a_k, k-1]$;
(3) If we put a_k into the third partition, then $M[x, y, k] = M[x, y, k-1]$.
If we construct M accordingly, the answer of this problem is in the entry $M[\frac{W}{3}, \frac{W}{3}, n]$.

The running time is $O(W^2 n)$, since there are $(\frac{W}{3}+1)^2(n+1)$ entries in M and each can be filled in $O(1)$ time.