CS 170 Algorithms
Spring 2014 Elchanan Mossel
HW 7

1. (15 pts.) Cutting cloth

Subproblems: Define XY subproblems. For $1 \le i \le X$ and $1 \le j \le Y$, let $C(i,j)$ be the best return that can be obtained from a cloth of shape $i \times j$. Define also a function `rect` as follows:

$$\texttt{rect}(i,j) = \begin{cases} \max_k c_k \text{ for all products } k \text{ with } a_k = i \text{ and } b_k = j \\ 0 \text{ if no such product exists} \end{cases}$$

Algorithm and Recursion: Then the recursion is:

$$C(i,j) = \max\{\max_{1\le k<i}\{C(k,j)+C(i-k,j)\}, \max_{1\le h<j}\{C(i,h)+C(i,j-h)\}, \texttt{rect}(i,j)\}$$

It remains to initialize the smallest subproblems correctly:

$$C(1,j) = \max\{0, \texttt{rect}(1,j)\}$$
$$C(i,1) = \max\{0, \texttt{rect}(i,1)\}$$

The final solution is then the value of $C(X,Y)$.

Correctness and Running Time: For proving correctness, notice that $C(i,j)$ trivially has the intended meaning for the base cases with $i = 1$ or $j = 1$. Inductively, $C(i,j)$ is solved correctly, as a rectangle $i \times j$ can only be cut in the $(i-1)+(j-1)$ ways considered by the recursion or be occupied completely by a product, which is accounted for by the $\texttt{rect}(i,j)$ term. The running time is $O(XY(X+Y+n))$ as there are XY subproblems and each takes $O(X+Y+n)$ to evaluate.

2. (15 pts.) (Optimal binary search tree) Let $S(i,j)$ be the cost of cheapest tree formed by words i to j, for $1 \le i, j \le n$. Also, initialize $S(i,j)$ to 0 if $i > j$. Then $S(i,j)$ will be the minimum cost of the tree over all choices of word k, $i \le k \le j$, to place at the root. If word k is at the root, the cost of the left subtree will be $S(i,k-1)$ and the cost of right subtree will be $S(k+1,j)$. Moreover, all words will need to pay one comparison at the root node, so the total cost of the tree will be $\sum_{t=i}^{j} p_t + S(i,k-1) + S(k+1,j)$. Hence:

$$S(i,j) = \min_{i\le k\le j}\{\sum_{t=i}^{j} p_t + S(i,k-1) + S(k+1,j)\}$$

Finally, the cost of the optimal tree will be $S(1,n)$. To reconstruct the tree, it suffices to keep track of which root k minimized the expression in the recursion for each subproblem and backtrack from $S(1,n)$.

Running time: The running time for this algorithm is $O(n^3)$. There are $O(n^2)$ subproblems (one for each possible i,j pair), and each subproblem takes $O(n)$ to compute, since it requires taking a min over $O(n)$ values (the values of $\sum_{t=i}^{j} p_t$ can be memoized along with the values of $S(i,j)$).

3. (20 pts.) Timesheets Part 2

Preprocessing: Sort all jobs in the order of decreasing $\frac{P_i}{R_i}$ so we have the permutation $\Pi(1),...,\Pi(N)$ of $1,...,N$.

Subproblem: Let $K(t,j)$ be the maximum reward achievable by time t completing a subset of jobs from jobs labeled $\Pi(1),...,\Pi(j)$.

Initialization: $K(0,0)=0$, $K(0,j)=0$ for all $1\leq j\leq N$, $K(t,0)=0$ for all $1\leq t\leq T$

Recursion: $K(t,j)=\max\{K(t-R_{\Pi(j)},j-1)+V_{\Pi(j)}-t*P_{\Pi(j)},K(t,j-1),K(t-1,j)\}$

Solution: The answer is given by $K(T,N)$.

Correctness: We sort in this order for the same reason as last week's problem without the time limit. Given the subset of jobs we want to process, we still want to process them in the most optimal way. The rest of the algorithm is a variation of the knapsack problem. The recursion chooses job $\Pi(j)$ to either be in the subset of jobs done or toss the job out and not complete it. The last term in the recursion is just there to make sure that the maximum sum propagates to the end even if the jobs chosen to be completed do not fill the maximum time T.

Running time: The running time of the algorithm is $O(N\ log\ N)$ for sorting in the beginning and $O(NT)$ for the recursion. Therefore, the total running time of the algorithm is $O(N(T+log\ N))$.

4. (15 pts.) Sorting with errors

Let $\{e_1,e_2,\ldots,e_n\}$ be our set of elements and let $Score(S)$ be the optimal score for some $S\subseteq\{e_1,e_2,\ldots,e_n\}$. We can express $Score(S)$ in terms of smaller sub-problems as follows. If e_j is the last element in the optimal ordering of S. The elements in $S-\{e_j\}$ must appear in the same order in the optimal orderings of $S-\{e_j\}$ and S since adding e_j at the end of any ordering of $S-\{e_j\}$ will add the same value to the score which is $\#\{e_i\in S-\{e_j\}:e_i>e_j\}$.

So, $Score(\{e_i\})=0$ for $i=1,\ldots,n$.

and the recurrence is

$$Score(S)=\min_{e_j\in S}(Score(S-\{e_j\})+\#\{e_i\in S-\{e_j\}:e_i>e_j\})$$

The running time is $O(2^n n^2)$ as we have $O(2^n)$ subproblems and each of them takes $O(n^2)$ time to compute.

5. (15 pts.) Exon chaining

For $i\in\{1,\cdots,n\}$, let $W(i)$ be the the weight of the best subset of consistent partial matches in $x[1,\cdots,i]$. To compute $W(i)$, we consider the two following cases:

- the best subset of partial matches contains a match j with $r_j=i$,
- the best subset of partial matches does not contain such j

In the first case, $W(i)$ will be the sum of w_j and the weight of the best match on the remaining of the string, i.e. $W(l_j-1)$. In the second case, we will just have $W(i)=W(i-1)$. This shows that the following recursion is correct:

$$W(i)=\max\{W(i-1),\max_{j:r_j=i}\{W(l_j-1)+w_j\}\}$$

The algorithm will then proceed computing $W(i)$ in ascending order of i and will output $W(n)$ as best total weight achievable. To reconstruct the actual sequence of partial matches, it suffices to keep track, for all

$W(i)$ of which j maximizes the expression in the recursion, when the second maximum is the larger. We can then follow these pointers from $W(n)$ backwards to identify the optimal alignment. The running time is $O(n+m)$, where m is the number of partial matches, as we have n subproblems and each partial match is considered once in the maximizations.

6. (20 pts.) Time and space complexity of dynamic programming

Assume $m = O(n)$.

a) Consider the usual matrix of subproblems $E(i, j)$. If we update the values column by column, at every point we only need the current column and the previous column to perform all calculations. Hence, if we are just interested in the final value $E(m,n)$ we may keep only two columns at every time, using space $O(n)$. Note that we are not able now to have a pointer structure to recover the optimal alignment, as we would need pointers for all subproblems, which would take space mn.

b) Together with the subproblem solution $L(i, j)$, for each $j \geq m/2$, in each of the active two columns, maintain a pointer to the index k at which the optimal path leading to (i, j) crossed the $m/2$ column. Such pointer can be easily updated at every recursion by copying the pointer of the subproblem from which the optimal solution is derived.

c) Consider the space requirement first. At any time during the running of this scheme, a single dynamic programming data structure is active, taking up space $O(n)$. All that is left to store is the values k for all the subproblems on which we recurse. These are at most m as every values corresponds to an index of x matched to one of y in the minimum edit distance alignment.. Hence, the total space required is $O(n)$. For the running time analyisis, notice that level $i+1$ of the recursion takes half the time of level i. Hence, the total running time will be bounded above by $O(mn)(1 + \frac{1}{2} + \frac{1}{4} + \cdots) = 2O(mn)$.