Design and Analysis of Algorithms
Massachusetts Institute of Technology — 6.046J/18.410J
Debayan Gupta, Aleksander Madry, and Bruce Tidor — Handout 10

Recitation 10: Dynamic Programming

As you already know from 6.006, dynamic programming is an algorithmic technique which allows problems which naively require exponential time to be solved in polynomial time, at the expense of using additional space.

Dynamic programming is used for any problems that exhibit *optimal substructure*, the property that the optimal solution to a problem can be constructed from the optimal solutions to subproblems. The extra space is used to *memoize* the solutions to these subproblems.

1 Difference Array

Given a non-negative array U of length n as input, your goal is to create another array A of length n, for which the sum of differences between consecutive elements is maximized and $1 \le a_i \le u_i \forall i$. Formally, you want to find A that satisfies

$$\max \sum_{i=2}^{n} |a_i - a_{i-1}|$$
$$\text{s.t. } 1 \le a_i \le u_i, 1 \le i \le n$$

The above optimization problem can't be solved with a linear program due to the absolute values in the objective function, so we have to use other methods.

At first it may seem intuitive that we need only consider cases where the values of A alternate between the maximum allowed and 1. However this intuition turns out to be incorrect, as shown by the case $U = [5, 2, 2, 5]$. Alternating between the maximum and minimum yields a score of 6, but the optimal answer is $A = [5, 1, 1, 5]$ with a score of 8. Although this intuition turned out to be wrong, it suggests some new intuition which is in fact correct: we need only consider cases where each a_i is either 1 or u_i.

Lemma 1 *Consider an element a_i which is not an extreme value. We show that a_i can be changed to an extreme value without decreasing the array's score. WLOG, let $a_{i-1} \le a_{i+1}$.*

- *Case 1: $a_{i-1} \le 1, u_i \le a_{i+1}$. In this case, all choices of a_i result in the same score, so a_i can be set to an extreme value.*
- *Case 2: $a_{i-1} \le 1 \le a_{i+1} \le u_i$. Choosing a_i such that $a_{i-1} \le a_i \le a_{i+1}$ results in a score of $a_{i+1} - a_{i-1}$, but choosing $a_i = u_i$ results in a score of $2(u_i - a_{i+1}) + (a_{i+1} - a_{i-1})$, so a_i can be set to u_i.*

- *Case 3:* $1 \leq a_{i-1} \leq u_i \leq a_{i+1}$. *This case is analogous to case 2.*
- *Case 4:* $1 \leq a_{i-1} \leq a_{i+1} \leq u_i$. *Choosing* $a_i = u_i$ *results in score* $2(u_i - a_{i+1}) + (a_{i+1} - a_{i-1})$, *choosing* $a_i = 1$ *results in score* $2(a_{i-1} - 1) + (a_{i+1} - a_{i-1})$. *Moving* a_i *away from one of the extreme values always decreases the score, so we set* a_i *to whichever extreme value has the highest score.*

Naively, we could check all 2^n choices for the values a_i, but we can do much better since this problem exhibits optimal substructure. Let $f(i,1)$ denote the optimal score for a subarray ending at position i and whose last value is u_i, and $f(i,0)$ be the analogous score where the last value is 1. Then,

$$\begin{aligned} f(1,0) &= f(1,1) = 0 \\ f(i,0) &= \max(|1 - u_{i-1}| + f(i-1,1), |1-1| + f(i-1,0)) \\ f(i,1) &= \max(|u_i - u_{i+1}| + f(i-1,1), |u_i - 1| + f(i-1,0)) \\ f(i) &= \max(f(i,0), f(i,1)) \end{aligned}$$

If we simply implement the recurrence above, the resulting program will still have an exponential runtime. However, if we memoize the calls to $f(i,b)$, the runtime becomes linear:

```
DP1($U$)
1: MEMO[$n$][2] $=$ 0 {set MEMO to zero everywhere}
2: for $i$ in $2 \ldots n$ do
3:     MEMO[$i$][0] $= \max(|1 - u_{i-1}| + MEMO(i-1,1), |1-1| + MEMO(i-1,0))$
4:     MEMO[$i$][1] $= \max(|u_i - u_{i-1}| + MEMO(i-1,1), |u_i - 1| + MEMO(i-1,0))$
5: end for
6: return $\max(DP[n][0], DP[n][1])$
```

2 String Segmentation

In this problem, you are given a string s and a dictionary D of words. Your goal is to determine whether or not s can be segmented into words contained in the dictionary. As before, this problem exhibits optimal substructure. In particular, the substring starting at position i can be segmented if and only if there exists a substring starting at i and ending at $j > i$ which is contained in the dictionary, and the substring starting at $j+1$ can itself be segmented.

More formally, let $f(i)$ be true if the substring of s starting at position i can be segmented into words contained in D. Then,

$$f(i) = \bigvee_{j=i+1}^{n} (s[i:j] \in D \land f(j))$$

Like before, a naive implementation of the above recurrence leads to an exponential runtime. This is easiest to see be considering the case where $s = aaa...ab$ and $D = \{a, aa, aaa, ...\}$, up to $n-1$ as. The final b will prevent the algorithm from terminating, and every possible segmentation of the first $n-1$ as will be tried.

Below, is an implementation of the memoized version of the algorithm.

```
DP2($s, D$)
 1: $MEMO[n+1] = 0$ {set MEMO to zero everywhere}
 2: $MEMO[n] = 1$
 3: for $i$ in $n \ldots 0$ do
 4:    for $j$ in $i+1 \ldots n$ do
 5:       $prefix\_cond = s[i:j] \in D$
 6:       $suffix\_cond = MEMO[j]$
 7:       $MEMO[i] = MEMO[i] \lor (prefix\_cond \land suffix\_cond)$
 8:    end for
 9: end for
10: return $MEMO[0]$
```

If we assume substring lookups are constant time, it's easy to see that the implementation above runs in $O(n^2)$.

3 Bank Heist

You and your $n-1$ friends are attempting to break into a bank. By analyzing employee schedules, you have found a moment when the guards will be gone for T minutes. In order to successfully break into the bank, each student s_i will have to pick the vault for t_i minutes. However, the alarm will go off if more than two of you are inside the bank at once. Can you find a plan that will allow you to break in?

The problem can be rephrased as follows: you're given an array S of times for each student and asked to find a partitioning A, B of the elements such that $\sum(A) \leq T$ and $\sum(B) \leq T$. If $\sum(S) > 2T$, the problem is clearly impossible. If not, you want to find the subset A of S which has the largest possible sum that does not exceed T. Once you find this subset, you can successfully rob the bank if $\sum(S) - \sum(A) \leq T$.

So we've reduced the problem to finding the largest subset sum which does not exceed T. This can actually be solved using a similar algorithm to the one for the knapsack problem. Note that there exists a subset of S that sums to a total t if there exists an element s_i with value v and there exists a subset of $S \backslash \{s_i\}$ that sums to $t - v$, or if S itself sums to t. This means that if we let $sack(t, i) = 1$ if the first i elements of S can form a subset that sums to t and 0 otherwise, the following equation holds.

$$sack(t, i) = sack(t - s_i, i - 1) \lor sack(t, i - 1)$$

This means that we can simply instantiate a table of size MN and methodically fill it using the above equation.

```
DP3($S, T$)
if $sum(S) > 2T$ then
    return $null$
end if
$N = len(S)$
$M = T$
$MEMO[M + 1][N + 1] = 0$ {set MEMO to zero everywhere}
$MEMO[0] = 1$ {set first row to 1}
$max\_val = 0$
for $i$ in $1 \ldots N$ do
    for $j$ in $1 \ldots M$ do
        $MEMO[j][i] = MEMO[j - a_i][i - 1] \lor MEMO[j][i - 1]$
        if $MEMO[j][i] == 1$ then
            $max\_val = \max(max\_val, j)$
        end if
    end for
end for
return $sum(S) - max\_val <= T$
```

In reality, the above pseudocode should be modified to prevent out-of-bounds errors, but this has been left out for clarity. It's easy to see that the resulting algorithm runs in $O(NM)$ time. However, the input to the problem was actually of size N! Since this problem runs in time which is polynomial in the numerical value of the input rather than the number of elements in the input, we say it runs in pseudopolynomial time.