Dynamic Programming & Greedy

Examples

Ugur Dogrusoz, CS Dept, Bilkent U.

Longest Palindromic Subsequence

In a palindromic subsequence, elements read the same backward and forward.

A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements. Longest Palindromic Subsequence (or LPS) of a given sequence $X=<x_1,x_2,..,x_n>$ is the longest of all palindromic subsequences of X.

In other words, given a sequence $X=<x_1,x_2,..,x_n>$, we would like to find the length of its Longest Palindromic Subsequence (or LPS) $Y=<y_1,y_2,...,y_m>$, where indices of Y $<i_1,i_2,...,i_m>$ match indices of X $<j_1,j_2,..,j_m>$ with $1\leq j_1< j_2...< j_m\leq n$ and $y_{j1}=y_{jm}$,$y_{j2}=y_{jm-1}$,...

Longest Palindromic Subsequence

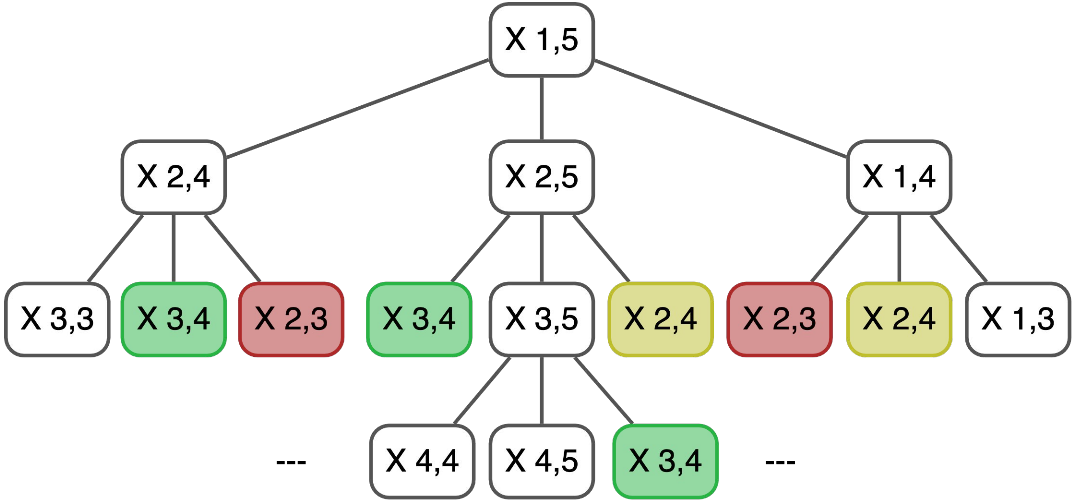
Example: for sequence $X=\langle\underline{a},a,\underline{b},c,\underline{d},e,\underline{b},\underline{a},f\rangle$, a LPS of X is $Y_1=\langle a,b,d,b,a\rangle$ with length 5. Another LPS of X is $Y_2=\langle a,b,e,b,a\rangle$.

Longest Palindromic Subsequence: **D-n-C solution**

```
int findLPSLength(X, p, r)
    if p > r return 0
    // a sequence with single element is a palindrome of length 1
    if p == r return 1
    // beginning and end elements are same
    if X[p] == X[r]
        return 2 + findLPSLength(X, p+1, r-1)
    // ignore one non-matching element either from beginning or from end
    l1 = findLPSLength(X, p+1, r)
    l2 = findLPSLength(X, p, r-1)
    return max(l1, l2)
```

Longest Palindromic Subsequence: **D-n-C solution**

Rather inefficient due to non-independent / duplicate subproblems. For example, for a sequence X of length 5:



Longest Palindromic Subsequence: **DP solution**

```
int findLPSLength(X, n)
    L[0..n-1,0..n-1] = {0}  // L[i,j]: optimal solution for X_i,j
    for i = n-1  to 0
        L[i,i] = 1
        for j = i+1  to n-1
            if X[i] == X[j]
                L[i,j] = L[i+1,j-1] + 2
            else
                L[i,j] = max(L[i+1,j],L[i,j-1]
    return L[0][n-1]
```

Longest Palindromic Subsequence: **DP solution** (memoized)

```
int findLPSLength(L, X, p, r)
    if p > r return 0
    if p == r return 1
    if L[p][r] == null // if not already solved
        if X[p] == X[r]
            L[p][r] = 2 + findLPSLength(L, X, p+1, r-1)
        else
            c1 = findLPSLength(L, X, p+1, r)
            c2 = findLPSLength(L, X, p, r-1)
            L[p][r] = max(c1, c2)
    return L[p][r]
```

Subset Sum Problem

Given a set of integers $X = \{x_1, x_2, \dots, x_n\}$, and an integer B, find a subset of X that has maximum sum not exceeding B.

$S_{n,B} = \langle\{x_1, x_2, \dots, x_n\} : B\rangle$ is the subset-sum problem, where we choose from integers x_i to obtain the desired sum B.

Example: $S_{12,59}$: $\langle X=\{10, 20, 4, 60, 30, 40, 5, 15, 70, 50, 0, 85\} : B=59\rangle$

An optimal solution: {10, 4, 30, 15} with 10+4+30+15=59

Subset Sum Problem: **DP solution**

c[i, b]: the value of an optimal solution for $S_{i,b} = \{x_1, \ldots, x_i: b\}$

Similar to? 0-1 Knapsack problem

$$c[i,b] = \begin{cases} 0 & \textit{if } \mathrm{i} = 0 \text{ or } \mathrm{b} = 0 \\ c[i-1,b] & \textit{if } \mathrm{x}_i > b \\ Max\{x_i + c[i-1, b-x_i], c[i-1,b]\} & \textit{if } \mathrm{i} > 0 \text{ and } \mathrm{b} \geq \mathrm{x}_i \end{cases}$$

Subset Sum Problem: DP solution

```
int findSubsetSum(X, n, B)
    for b=0 to B do // no numbers available for making sum b
        c[0,b]=0
    for i=1 to n do // trying to make a sum of zero with available numbers
        c[i,0]=0
    for i=1 to n do
        for b=1 to B do
            if X[i] ≤ b then
                c[i,b]=max{X[i]+c[i-1,b-X[i]], c[i-1,b]}
            else // choosing x_i would make sum exceed b
                c[i,b]=c[i-1,b]
    return c[n,B]
```

Coin Change Problem

Given (unlimited number of) coins with different denominations like 1¢, 5¢ and 10¢, we want to make an amount by using these coins such that a ***minimum number of coins*** is used.

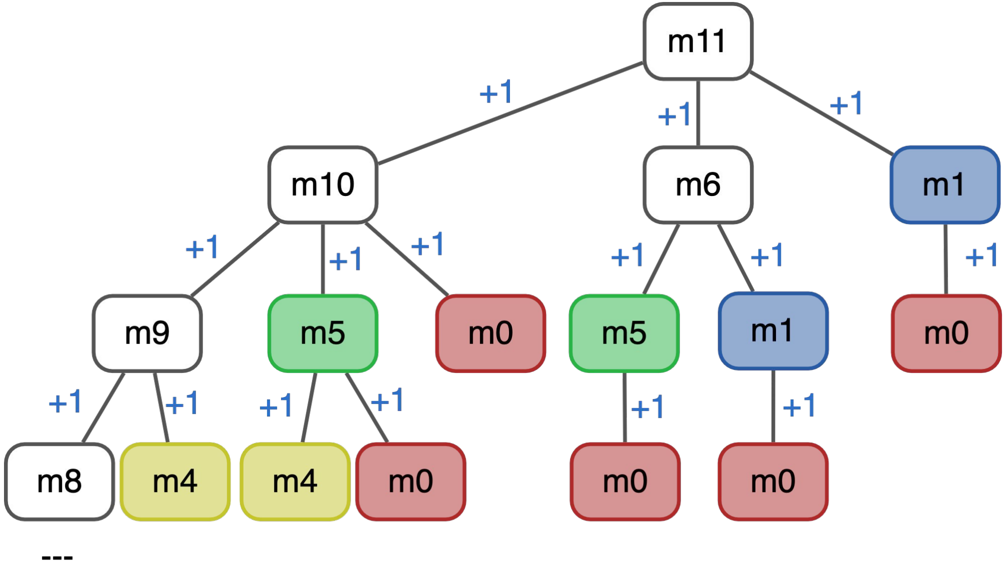
Example: d_1=1¢ < d_2=5¢ < d_3=10¢, 11¢ can be made in following ways:

- all 1¢'s (11 coins)
- 5¢+1¢+1¢+1¢+1¢+1¢+1¢ (7 coins)
- 5¢+5¢+1¢ (3 coins)
- 10¢+1¢ (2 coins) <= optimal

Coin Change Problem

Given denominations $d_1 < d_2 < \dots < d_k$ and an amount n, let m[i], $1 \le i \le n$, denote the minimum number of denominations to make n, and m[0]=0.

Then, $m[i] = \min_{1 \le j \le k}(m[i-d_j]+1)$



Coin Change Problem

Simple D-n-C will not be efficient due to redundancies

Will greedy work? Not always

Coin Change Problem

In real-life, choice of denominations allow a greedy approach to work nicely. Arbitrary choice of denominations, however, will not work.

Example: 1¢, 3¢, and 4¢, greedy approach will find 4¢+1¢+1¢ for 6¢, whereas 3¢ +3¢ works as well.

Need the DP approach!

Coin Change Problem: **Greedy solution** example

Use <1¢,5¢,10¢> to make change for 37¢

37¢=3 x 10¢ + 1 x 5¢ + 2 x 1¢

Greedy algorithm run time complexity?

Coin Change Problem: **DP solution** example

Use <1¢,3¢,4¢> to make change for 12¢

DP algorithm run time complexity?

0¢	*1¢*	*2¢*	*3¢*	*4¢*	*5¢*	*6¢*	*7¢*	*8¢*	*9¢*	*10¢*	*11¢*	*12¢*
0	1	2	1	1	2	2	2	2	3	3	3	3

Optimal for 6¢[2]+4¢[1] or 7¢[2]+3¢[1], not 9¢[3]+1¢[1]

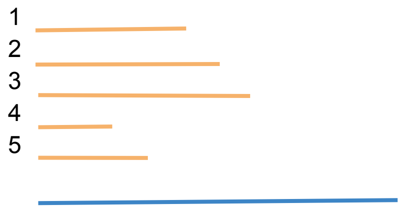
Coin Change Problem: **DP algorithm**

```
in makeChange(d, k, n)  // make n from denominations d=<d1,d2,...,dk>
    m[0] = 0
    for i in 1 to n
        min = INF
        for j in 1 to k
            if i >= d[j]
                min = min(min, 1+m[i-d[j]])  // min_{1≤j≤k}(m[i-d_j]+1)
        m[i] = min
    return m[n]
```

Maximizing Tasks

Given n different tasks with different time requirements, your goal is to perform a maximum number of tasks in a given period of time.

A **greedy** algorithm for this is to always perform a task requiring the least amount of time. This algorithm allows you to maximize the number of tasks performed.

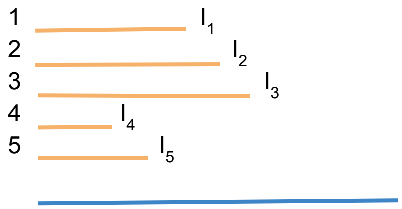


Maximizing Profits

Suppose now we also have a varying income for each task described in the previous example and we would like to maximize our profit within specified time interval.

Tasks can be performed in fractions?

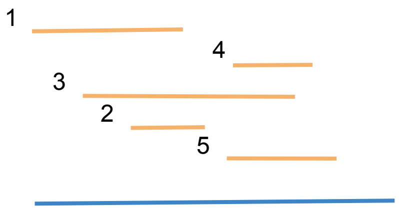
- *Yes*: a greedy algorithm (similar to fractional knapsack problem) for this is to always perform a task rewarding the most profit with per unit of time.
- *No*: a DP algorithm (similar to 0-1 knapsack problem)



Maximizing Fixed-Time Task Profits

Suppose for the previous problem, now we also have fixed, specified time periods for each task (cannot do them when we like!) and the income is the same.

A **greedy** algorithm (similar to activity selection problem) for this is to choose a task with earliest finish time.



Maximizing Profit w/ Varying-Time Tasks w/ Varying Profits

Now we have different scheduled events to choose from, but each event yields a different profit (activity selection with varying rewards).

The greedy approach will *not* work for this problem. **DP** can be used as follows:

- OPT(j) = value of optimal solution to problem with events {1, 2, . . . , j } ordered w.r.t. finish times
- q_j=largest index i < j such that event i is compatible with j
- OPT(0)=0 and OPT(j)=max(I_j+OPT(q_j), OPT(j-1))