# Dynamic Programming \& Greedy 

Examples

## 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 $X=\left\langle x_{1}, x_{2}, . ., x_{n}>\right.$ is the longest of all palindromic subsequences of $X$.

In other words, given a sequence $X=\left\langle x_{1}, x_{2}, \ldots, x_{n}\right\rangle$, we would like to find the length of its Longest Palindromic Subsequence (or LPS) $Y=\left\langle y_{1}, y_{2}, \ldots, y_{m}\right\rangle$, where indices of $Y$ $<\mathrm{i}_{1}, \mathrm{i}_{2}, \ldots, \mathrm{i}_{\mathrm{m}}>$ match indices of $\mathrm{X}<\mathrm{j}_{1}, \mathrm{j}_{2}, \ldots, \mathrm{j}_{\mathrm{m}}>$ with $1 \leq \mathrm{j}_{1}<\mathrm{j}_{2} \ldots<\mathrm{j}_{\mathrm{m}} \leq \mathrm{n}$ and $\mathrm{y}_{\mathrm{j} 1}=\mathrm{y}_{\mathrm{jm}}, \mathrm{y}_{\mathrm{j} 2}=\mathrm{y}_{\mathrm{jm}-1}, \ldots$

## Longest Palindromic Subsequence

Example: for sequence $X=<\underline{a}, a, \underline{b}, c, \underline{d}, e, \underline{b}, \underline{a}, \mathbf{f}>$, a LPS of $X$ is $Y_{1}=<a, b, d, b, a>$ with length 5. Another LPS of $X$ is $Y_{2}=<a, b, e, b, a>$.

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

```
int findLPSLength(X, P, r)
```



```
    // a sequence with single element is a palindrome of length 1
    if }\textrm{P}==r\mathrm{ 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
11 = findLPSLength(X, p+1, r)
12 = 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 :


Ugur Dogrusoz, CS Dept, Bilkent U.

## Longest Palindromic Subsequence: DP solution (memoized)

```
int findLPSLength(L, X, p, r)
    if }\textrm{P}>>r\mathrm{ 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 $X=\left\{x_{1}, x_{2}, \ldots, x_{n}\right\}$, and an integer $B$, find a subset of $X$ that has maximum sum not exceeding $B$.
$S_{n, B}=\left\langle\left\{x_{1}, x_{2}, \ldots, x_{n}\right\}: B>\right.$ is the subset-sum problem, where we choose from integers $x_{i}$ to obtain the desired sum $B$.

Example: $\mathrm{S}_{12,59}:<X=\{10,20,4,60,30,40,5,15,70,50,0,85\}: B=59>$
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 $S_{i, b}=\left\{x_{1}, \ldots, x_{i}\right.$ : $\left.b\right\}$
Similar to? 0-1 Knapsack problem

$$
c[i, b]= \begin{cases}0 & \text { if } \mathrm{i}=0 \text { or } \mathrm{b}=0 \\ c[i-1, b] & \text { if } \mathrm{x}_{i}>b \\ \operatorname{Max}\left\{x_{i}+c\left[i-1, b-x_{i}\right], c[i-1, b]\right\} & \text { if } \mathrm{i}>0 \text { and } \mathrm{b} \geq \mathrm{x}_{i}\end{cases}
$$

## 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] \leq b then
            c[i,b]=max{X[i]+c[i-1,b-X[i]], c[i-1,b]}
            else // choosing x }\mp@subsup{\textrm{i}}{\textrm{i}}{}\mathrm{ 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 $1 \phi, 5 \phi$ and $10 \phi$, we want to make an amount by using these coins such that a minimum number of coins is used.

Example: $d_{1}=1 \phi<d_{2}=5 \phi<d_{3}=10 \phi, 11 \phi$ can be made in following ways:

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


## Coin Change Problem

Given denominations $\mathrm{d}_{1}<\mathrm{d}_{2}<\ldots<\mathrm{d}_{\mathrm{k}}$ and an amount n , let $\mathrm{m}[\mathrm{i}], 1 \leq \mathrm{i} \leq \mathrm{n}$, denote the minimum number of denominations to make $n$, and $m[0]=0$.

Then, $m[i]=\min _{1 \leq j \leq k}\left(m\left[i-d_{j}\right]+1\right)$


Ugur Dogrusoz, CS Dept, Bilkent U.

## 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: $1 \phi$, $3 \phi$, and $4 \phi$, greedy approach will find $4 \phi+1 \phi+1 \phi$ for $6 \phi$, whereas $3 \phi$ $+3 \phi$ works as well.

Need the DP approach!

## Coin Change Problem: Greedy solution example

Use $<1 \phi, 5 \phi, 10 \phi>$ to make change for $37 \phi$
$37 \phi=3 \times 10 \phi+1 \times 5 \phi+2 \times 1 \phi$
Greedy algorithm run time complexity?

## Coin Change Problem: DP solution example

Use $<1 \phi, 3 \phi, 4 \phi>$ to make change for $12 \phi$
DP algorithm run time complexity?


Optimal for $6 \phi[2]+4 \phi[1]$ or $7 \phi[2]+3 \phi[1]$, not $9 \phi[3]+1 \phi[1]$

## Coin Change Problem: DP algorithm

in makeChange $(d, k, n) / /$ make $n$ from denominations $d=\left\langle d_{1}, d_{2}, \ldots, d_{k}\right\rangle$

```
m[0] = 0
for i in 1 to n
    min = INF
    for j in 1 to k
        m[i] = min
    return m[n]
```

            if i \(>=d[j]\)
                        \(\min =\min (m i n, 1+m[i-d[j]]) \quad / / \min _{1 \leq j \leq k}\left(m\left[i-d_{j}\right]+1\right)\)
    
## 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/ Fixed-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:

- $\operatorname{OPT}(\mathrm{j})=$ value of optimal solution to problem with events $\{1,2, \ldots, j\}$ ordered w.r.t. finish times
- $\mathrm{q}_{\mathrm{j}}=$ largest index $\mathrm{i}<\mathrm{j}$ such that event i is compatible with j
- OPT(0)=0 and OPT(j)=max ( $\left.\mathrm{l}_{\mathrm{j}}+\mathrm{OPT}\left(\mathrm{q}_{\mathrm{j}}\right), \mathrm{OPT}(\mathrm{j}-1)\right)$


