CS253- Dynamic Programming vs Greedy Algorithms

It is a divide-and–conquer strategy.

Chapter 15,16 Topics

Chapter 15,16 Topics

Dynamic -- $16.2, $15.4

Greedy -- $15.1, $16.1

- Principle of Optimality (PoO)
  - Know how a problem can be formulated so that it satisfies the PoO
  - Be able to develop a recurrence for a dynamic program
  - Show how a recurrence can be “unrolled” into a loop program for some problems.
- Example Problems
  - $15.1 Car Scheduling-- **dynamic ‘programming’**
  - $15.4String matching: acd, adc -- **dynamic ‘programming’**
  - $16.1 Real-Time deadline schedules – **greedy algorithm**
  - $16.2 0-1 Knapsack: 1,2,3,4 W=5 -- **dynamic ‘programming’**
  - $16.2 fractional Knapsack: 1,2,3,4 W=5 -- **greedy algorithm**

Things to Know

- When can dynamic programming be used?
- Dynamic programming provides an exact solution, but if the problem is HARD, it will take a long time
- Simplifications include avoiding re-computation of subproblems and storage of subproblem solutions by integer indexing.
- Invariants for a dynamic program are exactly those that are expressed by the PoO.

We want to optimize the solution to a problem: Maximize or minimize
Knapsack problem maximize the sack value.
Shortest paths problem minimize the distance.

Example

W=7,w= 6,5,4,3, 2 -- greedy algorithm (take the largest first) will give 6; but 4,3 or, 5, 2 is optimal which can be obtained from dynamic programming. The first weight ‘6’ does not lead to solution, not useful.

W=7,w= 2,3,4,5,6 -- greedy algorithm(take the smallest first) will give 5; but 3,4; 2,5; is optimal which can be obtained from dynamic programming. The first weight ‘1’ does lead to an optimal solution, hence it is useful.

Note. optimal value is unique, but optimal solution is not.

Question? Should we solve both the subproblems or one of them?
If we solve **both these subproblems**, this can lead to a binary tree of computations. It reduces the problem to 2^n sub problems. It amounts to **Exponential runtime O(2^n) complexity** from doubling the # of problems. This is **dynamic programming**. It will lead to optimal solution but it may be very time consuming.
Remember, there are 2^n subsets of n items, find the optimal solution from 2^n solutions. This means finding a solution from all the solutions.

KnapSack problem is an example of both dynamic programming and Greedy algorithm.

Example
Formally state
0-1 KNAPSACK PROBLEM
If $w_1,w_2,\ldots,w_n$ are sizes of objects, W is the size of knapsack, then the problem of optimizing the load with a subset of the n objects $(w_1,w_2,\ldots,w_n)$ can be written as $P(w_1,w_2,\ldots,w_n;W)$. The optimal load is obtained from optimal solution of two subproblems:

$P(w_2,\ldots,w_n;W)$ – exclude w_1
and
$P(w_2,\ldots,w_n;W-w_1)$, w_1□W -- include w_1

or
If $w_1,w_2,\ldots,w_n$ are sizes of objects, $v_1,v_2,\ldots,v_n$ are values of objects, W is the size of knapsack, then the problem of optimizing the load with a subset of the n objects $(w_1,w_2,\ldots,w_n)$ can be written as $P(w_1,w_2,\ldots,w_n; v_1,v_2,\ldots,v_n;W)$.

The optimal load is obtained from optimal solution of two sub problems:

$P(w_2,\ldots,w_n; v_2,\ldots,v_n;W)$ – exclude w_1
and
$P(w_2,\ldots,w_n; v_2,\ldots,v_n;W-w_1)$, w_1□W -- include w_1

If we choose one sub problem and try to solve the whole from this choice. It may yield optimal solution or it may not. It will also depend on the how 'useful' w_1 is? Making a choice, then solving the rest of the problem is called as **Greedy algorithm.**
Optimal strategy: use optimal solutions to sub problems and get optimal solution of the larger problem.
It depends on making choices:
Dynamic programming: solve sub problems, then make a choice, proceed to next upper level problem. **LCS, Assembly line**.
Dynamic programming uses optimal structure is in **bottom-up** manner.
Greedy Algorithms: make a choice, then solve the chosen sub problem, proceed to next lower level problem. **0-1 Knapsack problem.**

Think Recursive, Implement sequential.

We first discuss Greedy algorithm then dynamic programming. Greedy algorithm sometimes gives optimal solution, sometimes not.
Dynamic programming always yields optimal solution, but complexity is too large.

Greedy Algorithms Problem
Given as sack with weight **capacity** W, n items with **values** v_i, and **weights** w_i, for i=1,2,…,n
Goal:

- take most valuable load up to capacity of sack.

or

- take whatever comes first to load up to capacity of sack.

There are two types of Knapsack problem: 0-1 and fractional
0-1 Knapsack problem.
For each item **take it or leave it** (binary, 0 1 game).

Fractional Knapsack problem. one can take fractional weight also as far as possible.

Example
Suppose the thief goes to a store with an empty bag to steal some things.
He can start filling it with whatever size comes first, or he can start with largest item first, then next smaller, then next smaller so on.
Or
If he is a professional he knows the values of things, he can maximize the value instead of the load or weight.
If he is in a hurry, for each item he can take it or leave it.
Otherwise he can take a nap, can pick fractional items also to maximize the value.

Notation:
Capacity of bag W
Items weights : w_j
Items values : v_j
Value per unit of weight v_j/w_j

Questions can you maximize stolen value?
Does the load have to include w_j -- i.e. maximize the value for W- w_j from the rest
or can we do without including it i.e. excluding it – maximize value for W from the rest.

0-1 Knapsack problem.
Optimal structure strategy: select an item say w_j , now we are left with the problem of selecting from the remaining n-1 weights to maximize a load to a capacity of W- w_j
Fractional Knapsack problem.
Optimal structure strategy: select w from an item say w_j , now we are left with the problem of selecting from the remaining n-1 weights and w_j- w to a capacity of W- w

The fractional problem can be solved by greedy strategy, whereas 0-1 cannot be.
For fractional problem, first compute the value per unit of weight v_j/w_j

First take the item with greatest value per pound. This algorithm involves sorting values: □(nlgn)

If at any time, weight w > W'(remaining W') will not fit, take W'/w of weight w, and the corresponding value take W'/w of v.

What is the methodology? Sorting? Tabel O(nW)
Sorting works on fractional, not necessarily on 0-1
Table works
Example: W =50,
$w_1 = 10$, $w_2 = 20$, $w_3 = 30$,
$v_1 = 60$, $v_2 = 100$, $v_3 = 120$,
$v_1/w_1 = 6$, $v_2/w_2 = 5$, $v_3/w_3 = 4$

Under 0-1 strategy, you can have

$w_1 + w_2 = 30$,	value 160,	or
$w_1 + w_3 = 40$,	value 180,	or
$w_2 + w_3 = 50$,	value 220,	

Under fractional strategy, you can have
$w_1 + w_2 + 2/3\, w_3 = 50$, value 60+100+80= 240

With ordering, Optimal load with optimal value $P(w_1,w_2,\ldots,w_k; W)$ is $v_1+v_2+\ldots+v_k$ if the sum of weights is less than or equal to W.

Example 0-1 Knapsack
W = 5, and 4 weights and their values
Greedy
$W_2 + W_4 = 3$ **$v_2 + v_4 = 25$**
Dynamic
$W_3 + W_4 = 5$ **$v_3 + v_4 = 35$**

Sorting the value/weight and taking them in that order may not optimize:

i	w_i	v_i	v_i / w_i
1	3	12	4
2	1	10	10
3	3	20	6.66
4	2	15	7.5

Example 0-1 Knapsack different w1
W = 5, and 4 weights and their values
Greedy
$W_2 + W_4 + W_1 = 5$ **$v_2 + v_4 + v_1 = 37$**
Dynamic
$W_2 + W_4 + W_1 = 5$ **$v_2 + v_4 + v_1 = 37$**

i	w_i	v_i	v_i / w_i

1	2	12	6
2	1	10	10
3	3	20	6.66
4	2	15	7.5

Greedy strategy is to sort $w_1,w_2,\ldots,w_n$ and choose one of the two problems.
Two ways to view the knapsack problem
0-1 or fractional Knapsack.
Dynamic Programming
How can we speed up the process?
Create **(n+1)x(W+1)** table.
Let c[i,j] be the optimal value of a sack of size j by using a subset of i items
c[i,j] = if i=0 or j=0
Recursive calls
c[i,j] values are calculated as follows:
$c[i,j] = c[i-1,j]$ if $w_i>j$, -- w_i cannot be used
$c[i,j] = \max(v_i+ c[i-1,j- w_i], c[i-1,j])$ otherwise -- whether w_i will lead to optimal value

Note. For implmentation uses, if $w_i=j$, -- either w_i may not have impact or w_i may have larger value.

This is equivalent to solving the sub problems in the recursive fashion.
Fortunately, it is possible to improve the running time of these kinds of problems.

Example Create table for optimal sack with weights and values.
Example: W =5,
$w_1 =1$, $w_2 = 2$, $w_3 = 3$,
$v_1 = 6$, $v_2 = 10$, $v_3 = 12$,

Table for 0-1 sack optimal value: Weights ordered by value per unit weight {1, 2, 3}.

i		j→	**0**	**1**	**2**	**3**	**4**	**5**
0	v_i	w_i	0	0	0	0	0	0
1	6	1	0	6	6	6	6	6
2	10	2	0	6	**10**	16	16	16
3	12	3	0	6	10	16	18	**22**

Table for 0-1 sack optimal value: weights given in the order {2, 1, 3}.

		j→	**0**	**1**	**2**	**3**	**4**	**5**
i	v_i	w_i	0	0	0	0	0	0
1	**10**	**2**	0	0	**10**	10	10	10
2	**6**	**1**	0	6	10	16	16	16
3	**12**	**3**	0	6	10	16	18	**22**

Question? Do we need to fill the whole last row? Not necessarily? just c[n,m] needs to be computed.

How to trace the path is same way as for weights.
How to create the table. Think recursive

```
int c[i,j]
if i=1,
    return ($w_i$ >j?0:$v_i$)
else
    if $w_i$ >j
        then return (c[i-1,j])  -- ignore $w_i$
        else  return ( max(c[i-1,j], $v_i$ + c[i-1,j-$w_i$] ) – test whether to ignore or include $w_i$
```

How to create the array? Sequential implementation
AlgorithmDynamic 0-1 KnapSack

c[i,j]= optimal value of a sack of size j from a subset of i weights

Invariant: a sack of size j using a subset of empty set of weights has optimal value 0

```
Initialize c[0..n,0]
for i=0 to n
    c[i,0]=0
    Invariant: a sack of size zero using a subset of n weights  has optimal value 0

Initialize c[0,0..W]
for j=0 to W
    c[0,j]=0
    Invariant: a sack of size j using a subset of empty set of weights has optimal value 0

for i=1 to n
Invariant: c[i-1,0..W] are optimal values of sacks of sizes 0..W for (i-1)-th row (from subset
    of first i-1 weights)
    for j=1 to W
    Invariant: c[i,0..j-1] are optimal values of sacks of sizes 0..j-1 upto column j-1 of i-th
        row (from subset of first i weights)
        if $w_i$ >j
            then c[i,j] =  c[i-1,j] -- ignore $w_i$
        else  if  $v_i$ + c[i-1,j-$w_i$] $\leq$ c[i-1,j]
            then    c[i,j] =  c[i-1,j] -- ignore $w_i$
            else    c[i,j] = $v_i$ + c[i-1,j-$w_i$] -- include $w_i$
    // c[i,j] = $w_i$ >j ? c[i-1,j]: ( c[i-1,j] $\geq$ ($v_i$ + c[i-1,j-$w_i$])? c[i-1,j]: $v_i$ + c[i-1,j-$w_i$] )
    //to optimize calculations if  weight at c[i,W] =W, then c[i+1..n,W]=c[i,W]=W, return
```

Invariant: c[i,0..j] are optimal values of sacks of sizes 0..j upto column j of i-th row (from subset of first i weights)
Invariant: c[i,0..W] are optimal values of sacks of sizes 0..W for (i)-th row (from subset of first i weights)

Post condition: c[1..n,1..W], are optimal values

Trace path: we are interested in determining which weights are included.
Trace path complexity: in (n) time
Optimal value is obtained when c[i,j] becomes overall optimal first time.

```
start with c[n, m]
    S= Solution=null
    i=n, j=m
    while( i >0 and j>0 )
    S contains weights for optimal solution
        while(c[i,j] = c[i-1,j]), i--;
        i--, j= j- $w_i$;    -- look for next weight in first optimal.
        S= {$w_i$,} U S, --Weight is added to the solution
    S contains weights for optimal solution
S is optimal solution set
```

Dynamic Programming Problem
Find the Longest Common Subsequence in
The notion of longest common subsequence of two strings is used to compare file. The diff command of the UNIX system is based on this notion where the lines of the files are considered as the symbols of the alphabet.
Subsequence is not consecutive.
Common Subsequence in

$X_m = (x_1, x_2, \ldots, x_m)$
$Y_n = (y_1, y_2, \ldots, y_n)$
is
$Z_k = (z_1, z_2, \ldots, z_k)$

so that $k \le \min(m,n)$, and $z_p = x_{i_p} = y_{j_p}$ where i_p, j_p are in ascending order.
$i_p < i_{p+1}$ $\quad$ $j_p < j_{p+1}$

Let c[i,j] be the length of LCS in X_i and Y_j

Complexity of Longest Common Subsequence
Recursive topdown algorithm

c(m,n): length of lcs in X_m and Y_n

$c(m,n) = c(m-1,n-1)+1$ if $x_m=y_n$
$c(m,n) = \max(c(m,n-1), c(m-1,n))$ if $x_m \neq y_n$

This leads to a ternary tree, **each node** in the tree has **3 branches.** The worst case **height is m+n,**
Complexity of search is the total number of nodes which becomes $3^{(m+n)}$

Alternate way to search: Longest Common Subsequence length takes mn computations instead of $3^{(m+n)}$ The search in the worst case is m+n cells.

Worst case arises when $x_1=y_1$ and $x_m=y_n$. The LCS length is min(m,n) in worst case.
Best case arises when m=n and $X_m=Y_n$ The LCS length is min(m,n) in best case.

Example Create a (m+1)x(n+1) table c[i,j]
Iterative/sequential implementation bottom up

```
Set     c[i,0]= 0, c[0,j]= 0,
        if( $x_i$==$y_j$), set c[i,j]= c[i-1,j-1]+1;
        else set c[i,j]= max(c[i,j-1], c[i-1,j])
```

Examples: GRAMPRO and PROGRAM: many common subsequences here, LCS is GRAM
2613564 and 5642613: many common subsequences here, LCS is 2613
the subsequence is suggested by **symbols in the sequance** above(or left) of **red marked 1,2,3,4**

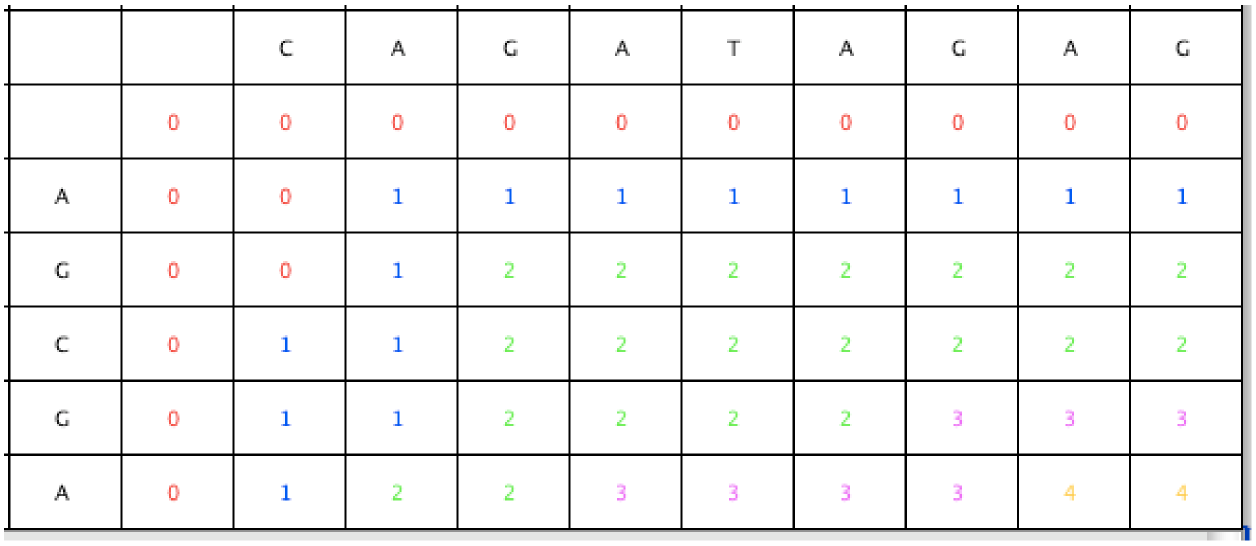
j		0	1	2	3	4	5	6	7
i			G	R	A	M	P	R	O
0		0	0	0	0	0	0	0	0
1	P	0	0	0	0	0	1	1	1
2	R	0	0	1	1	1	1	2	2
3	O	0	0	1	1	1	1	2	3
4	G	0	1	1	1	1	1	2	3
5	R	0	1	2	2	2	2	2	3
6	A	0	1	2	3	3	3	3	3
7	M	0	1	2	3	4	4	4	4

Trick to fill the table:
first mark the match entries, then fill the rest of the row looking at the top and left entries (whichever is bigger) and (at check mark, diagonal entry plus 1).

Exercise find LCS of 1232412 and 243121: 2412, 2312, 2321

j		0	1	2	3	4	5	6	7
i	□	□	1	2	3	2	4	1	2
0	□	0	0	0	0	0	0	0	0
1	2	0	0	1	1	1	1	1	1
2	4	0	0	1	1	1	2	2	2
3	3	0	0	1	2	2	2	2	2
4	1	0	1	1	2	2	2	3	3
5	2	0	1	2	2	3	3	3	4
6	1	0	1	2	2	2	2	4	4

Example LCS: AGGA
AGCGA
CAGATAGAG



		C	A	G	A	T	A	G	A	G
	0	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1	1
G	0	0	1	2	2	2	2	2	2	2
C	0	1	1	2	2	2	2	2	2	2
G	0	1	1	2	2	2	2	3	3	3
A	0	1	2	2	3	3	3	3	4	4

Ref. Applet http://www-igm.univ-mlv.fr/~lecroq/seqcomp/node4.html

Dynamic programming—Sequential Implementation

Let c[m,n] = length of **Longest Common Subsequence of X and Y. The** matrix c[m,n] can be built in **mn** computations.

Sequential approach

```
LCS-Length(X,Y)
m = length(X);
n = length(Y);
Initialize c[i,0]
for i=1 to m
    c[i,0]=0;
    Invariant: The length of LCS of $X_i$ and $Y_0$ is zero
Initialize c[0,j]
for j=1 to n
    c[0,j]=0;
    Invariant: The length of LCS of $X_0$ and $Y_j$ is zero

for i=1 to m
Invariant: c[i-1, 1..n]  are the lengths of LCSs of $X_{i-1}$ and $Y_{1..n}$
    for j=1 to n
    Invariant: c[i, j-1]  is the length of LCS of $X_i$ and $Y_{j-1}$
        if(xi==yj),
            c[i,j]= c[i-1,j-1]+1;
    //  else c[i,j]= max(c[i,j-1], c[i-1,j])
        else
            c[i,j]= c[i,j-1] > c[i-1,j] ? c[i,j-1] : c[i-1,j]
    Invariant: c[i, j]  is the length of LCS of $X_i$ and $Y_j$
Invariant: c[i, 1..n]  are the lengths of LCSs of $X_i$ and $Y_{1..n}$

Post Condition: c[0..m, 0..n]  are the lengths of LCSs of $X_{1..m}$ and $Y_{1...n}$
```

What if you want the subsequence itself, and not just its length? This is important for some but not all of the applications we mentioned. Once we have filled in the array L described above, we can find the sequence by working forwards through the array.

Path can be searched in m+n steps(worst case) by following the diagonal, rows and columns of the matrix.

How to determine a path from the optimal length?
C[m,n] is the length of the LCS **in** X_m and Y_n
One possible solution Path

```
    //vertically followed by horizontally
    i=m, j=n   -- can you have a for loop
    s=null
    while (i>0,j>0) -- can you have a for loop, for k= 1 to length c[m,n]
        while(c(i,j) = c(i-1,j-1),i, j>0),i--, j--;
        while(c(i,j) = c(i-1,j), i>0), i--;
        while(c(i,j) = c(i,j-1), j>0), j--;
        if i>0 and j>0,  s={x_i, s}, i--,j--
```

Note. You can't use i--,j or i, j-- —because you run into the problem of using same row(column) symbol twice.

Alternate possible solution path

```
    horizontally followed by vertically
    i=m, j=n
    s=null
    while (i>0,j>0)
        while(c(i,j) = c(i-1,j-1),i, j>0),i--, j--;
        while(c(i,j) = c(i,j-1), j>0), j--;
        while(c(i,j) = c(i-1,j), i>0), i--;
        if i>0 and j>0, s={x_i, s}, i--,j--
```

Implement Hybrid algorithm to find all possible paths. Create a data structure Tree with nodes of type ={node left, node up, value list}. The root of the tree has a list value nil, there are two possible branches for growth. If a node has no left and up children its list value is the path.
Tree node has {node, node, list}

```
Trace (Tree,i,j)
Find value by traversing left followed by up
findLeftUp (i,j)
        while(c(i,j) = c(i-1,j-1),i, j>0),i--, j--;
        while(c(i,j) = c(i,j-1), j>0), j--;
        while(c(i,j) = c(i-1,j), i>0), i--;
        retrun (x_i,i--,j--)
Find value by traversing up followed by left
findUpLeft (i,j)
        while(c(i,j) = c(i-1,j-1),i, j>0),i--, j--;
        while(c(i,j) = c(i-1,j), i>0), i--;
        while(c(i,j) = c(i,j-1), j>0), j--;
    retrun (x_i,i--,j--)
```

Trace(Tree, m,n)
where Tree node has the form (left,up,s), initially Tree=(nil,nil,nil)

```
 If c[i,j]= c[i-1,j]= c[i,j-1]
// it is possible both branches lead to the same pace or different places
        Then
        (x,p,q) =FindUpLeft(i,j)
                s= {x}Us, Tree.up = new node(nil,nil, s)
                Trace (Tree.up,p,q)
        (x,p,q) =FindLeftUp (i,j)
                s= {x}Us, Tree.left = new node(nil,nil, s)
                Trace (Tree.left,p,q)
ElseIf c[i,j]= c[i-1,j]
        Then
        (x,p,q) =FindUpLeft(i,j)
                s= {x}Us, Tree.up = new node(nil,nil, s)
                Trace (Tree.up,p,q)
ElseIf c[i,j]= c[i,j-1]
        (x,p,q) =FindLeftUp (i,j)
                s= {x}Us, Tree.left = new node(nil,nil, s)
                Trace (Tree.left,p,q)
Else arbitrarily choose one branch
                s= {x}Us, Tree.up = new node(nil,nil, s)
                Trace (Tree.u,p,q)
```

Solutions are at the leaf nodes of the tree, that is, if node.left=node.up=null, the path is node.s