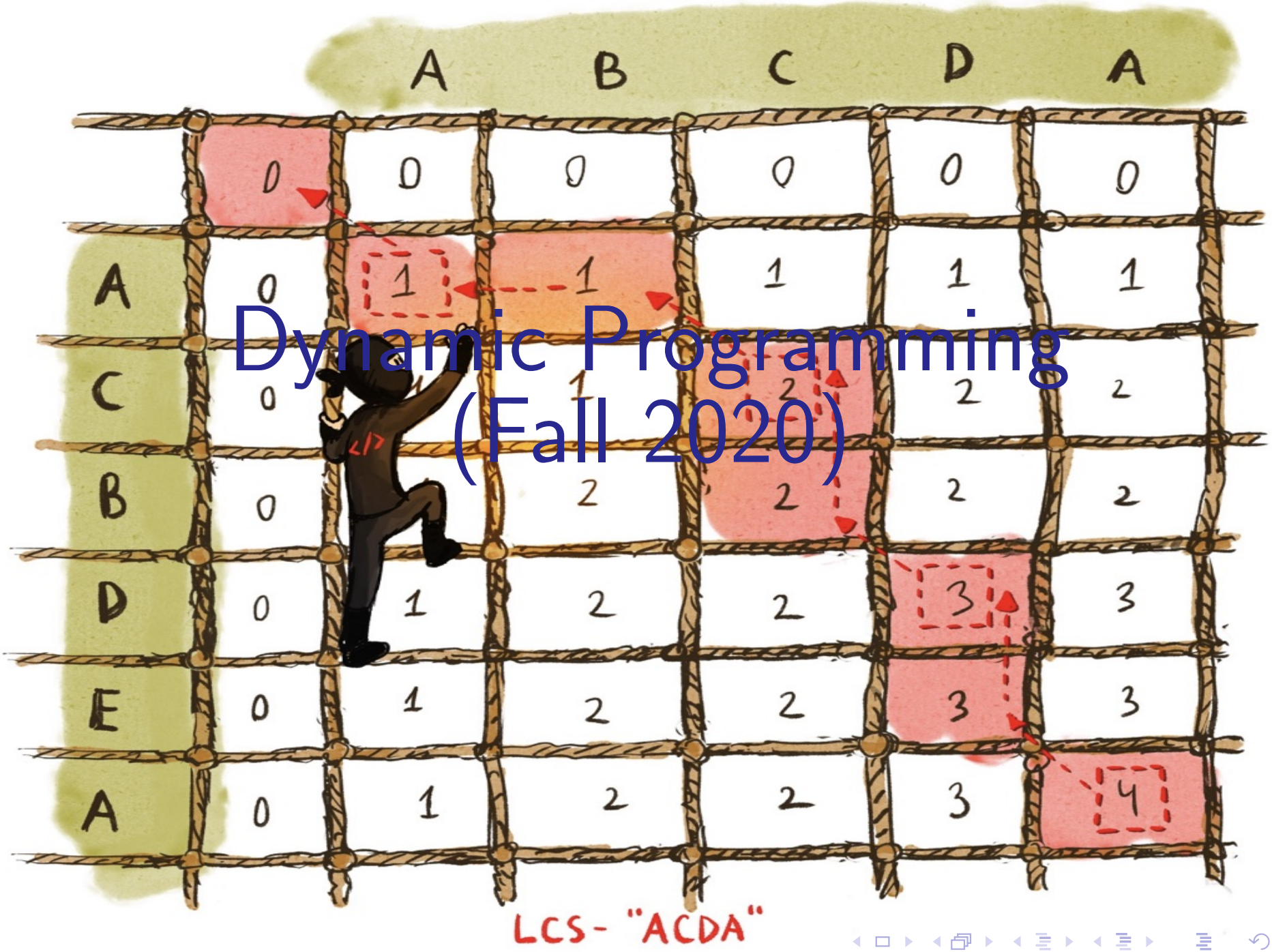


Dynamic Programming (Fall 2020)

Multiplying a Sequence of Matrices

(This example is from Section 15.2 in CormenLRS' book.)
Multiplication of n matrices Given as input a sequence of n matrices $(A_1 \times A_2 \times \cdots \times A_n)$. Minimize the number of operation in the computation $A_1 \times A_2 \times \cdots \times A_n$
Recall that Given matrices A_1, A_2 with $\dim(A_1) = p_0 \times p_1$ and $\dim(A_2) = p_1 \times p_2$, the basic algorithm to $A_1 \times A_2$ takes time $p_0 p_1 p_2$

Example:

$$\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix} = \begin{bmatrix} 13 & 18 & 23 \\ 18 & 25 & 32 \\ 23 & 32 & 41 \end{bmatrix}$$

Multiplying a Sequence of Matrices

- Matrix multiplication is NOT commutative, so we can not permute the order of the matrices without changing the result,
- but it is associative, so we can put parenthesis as we wish.
- In fact, the solution the problem of how to multiply is equivalent to the problem of how to parenthesize
- We want to find the way to put parenthesis so that the product requires the minimum numbe rof operations.

Example Consider $A_1 \times A_2 \times A_3$, where dim $(A_1) = 10 \times 100$ dim $(A_2) = 100 \times 5$ and dim $(A_3) = 5 \times 50$.

- $((A_1 A_2)A_3)$ takes $(10 \times 100 \times 5) + (10 \times 5 \times 50) =$ *7500* operations,
- $(A_1(A_2 A_3))$ takes $(100 \times 5 \times 50) + (10 \times 100 \times 50) =$ 75000 operations.

The order in which we make the computation of products of two matrices makes a big difference in the total computation's time.

Multiplying a Sequence of Matrices

How to parenthesize $(A_1 \times \ldots \times A_n)$?

- If $n = 1$ we do not need parenthesis.
- Otherwise, decide where to break the sequence $((A_1 \times \cdots \times A_k)(A_{k+1} \times \cdots \times A_n))$ for some k, $1 \leq k < n$.
- Combine any way to parenthesize $(A_1 \times \cdots \times A_k)$ with any way to parenthesize $(A_{k+1} \times \cdots \times A_n)$.

Using this, we can count the number of ways to parenthesize $(A_1 \times \cdots \times A_n)$ as well as to define a backtracking algorithm that goes over all thos ways to parenthesize and eventually to a brute force recursive algorithm to solve the problem of computing efficiently the product.

How many ways to parenthesize $(A_1 \times \cdots \times A_n)$?

Let $P(n)$ be the number of ways to paranthesize $(A_1 \times \cdots \times A_n)$. Then,

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{si } n \geq 2 \end{cases}$$

How many ways to parenthesize $(A_1 \times \cdots \times A_n)$?

Let $P(n)$ be the number of ways to paranthesize $(A_1 \times \cdots \times A_n)$. Then,

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{si } n \geq 2 \end{cases}$$

with solution $P(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n/n^{3/2})$

The Catalan numbers.

How many ways to parenthesize $(A_1 \times \cdots \times A_n)$?

Let $P(n)$ be the number of ways to paranthesize $(A_1 \times \cdots \times A_n)$. Then,

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{si } n \geq 2 \end{cases}$$

with solution $P(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n/n^{3/2})$

The Catalan numbers.

Brute force will take too long!

1.- Structure of an optimal solution

- We want to compute $(A_1 \times \cdots \times A_n)$ efficiently.
- In an optimal solution the last matrix product must correspond to a break at some position k, $((A_1 \times \cdots \times A_k)(A_{k+1} \times \cdots \times A_n))$ Let $A_{i-j} = (A_i A_{i+1} \cdots A_j)$.
- The parenthesization of the subchains $(A_1 \times \cdots \times A_k)$ and $(A_{k+1} \times \cdots \times A_n)$ within the optimal parenthesization must be an optimal paranthesization of $(A_1 \times \cdots \times A_k)$, $(A_{k+1} \times \cdots \times A_n)$.
  as
  $\text{cost}(A_1 \ldots A_n) = \text{cost}(A_1 \ldots A_k) + \text{cost}(A_{k+1} \ldots A_n) + p_0 p_k p_n$.
- An optimal solution is formed by optimal solution of subproblems.
- However, we do not known the good value for k.

2.- Cost Recurrence

- Let $m[i,j]$ the minimum cost of computing $(A_i \times \ldots \times A_j)$, for $1 \leq i < j \leq n$.
- $m[i,j]$ will be defined by the $k, i \leq k < j$ that minimizes $m[i,k] + m[k+1,j] +$ cost $((A_i \times \ldots \times A_k) \times (A_{k+1} \times \ldots \times A_j))$.
- That is,

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{otherwise} \end{cases}$$

3.- Computing the optimal costs

Straightforward recursive implementation of the previous recurrence:
The input is given by $P =< p_0, p_1, \ldots, p_n >$,

MCR(i, j)
if $i = j$ **then**
  return 0
$m[i, j] = \infty$
for $k = i$ **to** $j - 1$ **do**
  $q = \text{MCR}(i, k) + \text{MCR}(k + 1, j) + P[i - 1] * P[k] * P[j]$
  if $q < m[i, j]$ **then**
    $m[i, j] = q$
return $(m[i, j])$

Complexity: $T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n \sim \Omega(2^n)$.

Can we apply dynamic programming?

- We have an optimal recursive algorithm which takes exponential time.

Can we apply dynamic programming?

- We have an optimal recursive algorithm which takes exponential time.
- Subproblems?

Can we apply dynamic programming?

- We have an optimal recursive algorithm which takes exponential time.
- Subproblems?
  The subproblems are identified by the two inputs in the recursive call, the pair (i, j).

Can we apply dynamic programming?

- We have an optimal recursive algorithm which takes exponential time.
- Subproblems?
  The subproblems are identified by the two inputs in the recursive call, the pair (i, j).
- How many subproblems?

Can we apply dynamic programming?

- We have an optimal recursive algorithm which takes exponential time.
- Subproblems?
  The subproblems are identified by the two inputs in the recursive call, the pair (i, j).
- How many subproblems?
  As $1 \leq i < j \leq n$, we have only $O(n^2)$ subproblems.

Can we apply dynamic programming?

- We have an optimal recursive algorithm which takes exponential time.
- Subproblems?
  The subproblems are identified by the two inputs in the recursive call, the pair (i, j).
- How many subproblems?
  As $1 \leq i < j \leq n$, we have only $O(n^2)$ subproblems.
- We can use DP!

Dynamic programming: Memoization

MCP(P)
for all $1 \leq i < j \leq n$ **do**
  $m[i,j] = -1$
for $i = 1$ **to** n **do**
  $m[i,i] = 0$
MCR$(1,n)$
return $(m[1,n])$

MCR(i,j)
if $m[i,j]! = -1$ **then**
  return $(m[i,j])$
$m[i,j] = \infty$
for $k = i$ **to** $j-1$ **do**
  $q = \text{MCR}(i,k) + \text{MCR}(k+1,j) +$
  $P[i-1] * P[k] * P[j]$
  if $q < m[i,j]$ **then**
    $m[i,j] = q$
return $(m[i,j])$

$T(n) = \Theta(n^3)$ additional space $\Theta(n^2)$.

Dynamic programming: Tabulating

To compute the element $m[i, j]$ the base case is when $i = j$, we need to access $m[i, k]$ and $m[k + 1, j]$. We can achieve that by filling the (half) table by diagonals.

Dynamic programming: Tabulating

To compute the element $m[i,j]$ the base case is when $i = j$, we need to access $m[i,k]$ and $m[k+1,j]$. We can achieve that by filling the (half) table by diagonals.

MCP(P)
for $i = 1$ **to** n **do**
  $m[i,i] = 0$
for $d = 2$ **to** n **do**
  for $i = 1$ **to** $n - d + 1$ **do**
    $j = i + d - 1$
    $m[i,j] = \infty$
    for $k = i$ **to** $j - 1$ **do**
      $q =$
      $m[i,k] + m[k+1,j] + P[i-1] * P[k] * P[j]$
      if $q < m[i,j]$ **then**
        $m[i,j] = q$
return $(m[1,n])$

$T(n) = \Theta(n^3)$,
space $= \Theta(n^2)$.

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1				
2				
3				
4				

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1	0			
2		0		
3			0	
4				0

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1	0	45		
2		0	30	
3			0	24
4				0

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1	0	45	60	
2		0	30	70
3			0	24
4				0

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1	0	45	60	84
2		0	30	70
3			0	24
4				0

4.- Recording more information about the optimal solution

We have been working with the recurrence

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{otherwise} \end{cases}$$

To keep information about the optimal solution the algorithm will keep additional information about the value of k that provides the optimal cost.

$$s[i,j] = \begin{cases} i & \text{if } i = j \\ \arg\ \min_{i \leq k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{otherwise} \end{cases}$$

Dynamic programming: Memoization

MCP(P)
for all $1 \leq i < j \leq n$ **do**
  $m[i,j] = -1$
for $i = 1$ **to** n **do**
  $m[i,i] = 0$; $s[i,i] = i$;
MCR$(1,n)$
return m, s

MCR(i,j)
if $m[i,j] != -1$ **then**
  return $(m[i,j])$
$m[i,j] = \infty$
for $k = i$ **to** $j-1$ **do**
  $q = \text{MCR}(i,k) + \text{MCR}(k+1,j) +$
  $P[i-1] * P[k] * P[j]$
  if $q < m[i,j]$ **then**
    $m[i,j] = q$; $s[i,j] = k$;
return $(m[i,j])$

Dynamic programming: Tabulating

MCP(P)
for $i = 1$ **to** n **do**
  $m[i,i] = 0$; $s[i,i] = 0$;
for $d = 2$ **to** n **do**
  for $i = 1$ **to** $n - d + 1$ **do**
    $j = i + d - 1$
    $m[i,j] = \infty$
    for $k = i$ **to** $j - 1$ **do**
      $q =$
      $m[i,k] + m[k+1,j] + P[i-1] * P[k] * P[j]$
      if $q < m[i,j]$ **then**
        $m[i,j] = q$; $s[i,j] = k$;
return m, s.

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1				
2				
3				
4				

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1	0 1			
2		0 2		
3			0 3	
4				0 4

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1	0 1	45 1		
2		0 2	30 2	
3			0 3	24 3
4				0 4

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1	0 1	45 1	60 1	
2		0 2	30 2	70 3
3			0 3	24 3
4				0 4

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1	0 1	45 1	60 1	84 3
2		0 2	30 2	70 3
3			0 3	24 3
4				0 4

5.- Constructing an optimal solution

We need to construct an optimal solution from the information in s. $s[i,j]$ contains the value of k that decomposes optimally the product:

$$A_i \times \cdots \times A_j = (A_i \times \cdots \times A_{s[i,j]})(A_{s[i,j]+1} \times \cdots \times A_j).$$

Moreover, $s[i, s[i,j]]$ determines the position to compute optimally the first term and $s[s[i,j]+1, j]$ determines the one for the second term.

Therefore,

$$A_1 \times \cdots \times A_n = (A_1 \times \cdots \times A_{s[1,n]})(A_{s[1,n]+1} \times \cdots \times A_n).$$

We can use a recursive algorithm to perform the product in an optimal way.

The product algorithm

The input is the sequence of matrices $A = A_1, \ldots, A_n$ and the table s computed before.

Product(A, s, i, j)
if $j > 1$ **then**
  $X =$**Product**$(A, s, i, s[i, j])$
  $Y =$**Product**$(A, s, s[i, j] + 1, j)$
  return $(X \times Y)$
else
  return (A_i)

The total number operations required to compute the product is $m[1, n]$ and the cost of the complete algorithm is
$T(n) = O(n^3 + m[1, n])$

Example.

We wish to compute $A_1 \times A_2 \times A_3 \times A_4$ with $P =< 3, 5, 3, 2, 4 >$

$i \setminus j$	1	2	3	4
1	0 1	45 1	60 1	84 3
2		0 2	30 2	70 3
3			0 3	24 3
4				0 4

The optimal way to minimize the number of operations is

$$(((A_1) \times (A_2 \times A_3)) \times (A_4))$$

0-1 Knapsack

(This example is from Section 6.4 in Dasgupta,Papadimritriou,Vazirani's book.)

0-1 Knapsack: Given as input a set of n items that can NOT be fractioned, item i has weight w_i and value v_i, and a maximum permissible weight W.

QUESTION: select the items $S \subseteq I$ to maximize the profit.

Recall that we can NOT take fractions of items.

Characterize structure of optimal solution and define recurrence

Input: $(w_1, \ldots, w_n)$, $(v_1, \ldots, v_n)$, W.

- Let $S \subseteq \{1, \ldots, n\}$ be an optimal solution to the problem The optimal benefit is $\sum_{i \in S} v_i$
- With respect to the last item we have two cases:
  - $n \notin S$, then S is an optimal solution to the problem $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, W
  - $n \in S$, then $S - \{n\}$ is an optimal solution to the problem $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, $W - w_n$
- in both cases we get an optimal solution of a problem in which the last item is removed and in which the maximum weight can be W or a value smaller than W.
- This identifies subproblems of the form $[i, x]$ that are knapsack instances in which the set of items is $\{1, \ldots, i\}$ and the maximum weight that can hold the knapsack is x.

Characterize structure of optimal solution and define recurrence

Let $v[i, x]$ be the maximum value (optimum) we can get from objects $\{1, 2, \ldots, i\}$ within total weight $\leq x$.

To compute $v[i, x]$, the two possibilities we have considered give raise to the recurrence:

$$v[i, x] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max v[i-1, x-w_i] + v_i, v[i-1, x] & \text{otherwise} \end{cases}$$

DP algorithm

Define a table $P[n+1, W+1]$

Knapsack(i, x)
for $i = 0$ **to** n **do**
  $P[i, 0] = 0$
for $x = 1$ **to** W **do**
  $P[0, x] = 0$
for $i = 1$ **to** n **do**
  for $x = 0$ **to** W **do**
    $P[i, x] = \max\{P[i-1, x], P[i-1, x-w[i]] + v[i]\}$
return $P[n, W]$

The number of steps is $O(nW)$.

Example.

i	1	2	3	4	5
w_i	1	2	5	6	7
v_i	1	6	18	22	28

$W = 11$.

		w										
		0	1	2	3	4	5	6	7	8	9	10	11
	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	1	1	1	1	1	1	1	1	1	1	1
	2	0	1	6	7	7	7	7	7	7	7	7	7
I	3	0	1	6	7	7	18	19	24	25	25	25	25
	4	0	1	6	7	7	18	22	23	28	29	29	40
	5	0	1	6	7	7	18	22	28	29	34	35	40

For instance,

$v[4,10] = \max\{v[3,10], v[3,10-6]+22\} = \max\{25, 7+22\} = 29.$

$v[5,11] = \max\{v[4,11], v[4,11-7]+28\} = \max\{40, 4+28\} = 40.$

Recovering the solution

To compute the actual subset $S \subseteq I$ that is the solution, we modify the algorithm to compute also a Boolean table $K[n+1, W+1]$, so that $K[i, x]$ is 1 when the max is attained in the second alternative ($i \in S$), 0 otherwise.

$x = W$, $S = \emptyset$
for $i = n$ **downto** 1 **do**
  if $K[i, x] = 1$ **then**
    $S = S \cup \{i\}$
    $x = x - w_i$
Output S

Complexity: $O(nW)$

	0	1	2	3	4	5	6	7	8	9	10	11
0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
1	0 0	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1
2	0 0	1 0	6 1	7 1	7 1	7 1	7 1	7 1	7 1	7 1	7 1	7 1
3	0 0	1 0	6 0	7 0	7 0	18 1	19 1	24 1	25 1	25 1	25 1	25 1
4	0 0	1 0	6 0	7 0	7 0	18 1	22 1	23 1	28 1	29 1	29 1	40 1
5	0 0	1 0	6 0	7 0	7 0	18 0	22 0	28 1	29 1	34 1	35 1	40 0

$K[5, 11] \to K[4, 11] \to K[3, 5] \to K[2, 0]$. So $S = \{4, 3\}$

Complexity

The 0-1 KNAPSACK is NP-complete. Does it mean P=NP?

- An algorithm runs in pseudo-polynomial time if its running time is polynomial in the numerical value of the input, but it is exponential in the length of the input,
- Recall that given $n \in \mathbb{Z}$ the value is n but the length of the representation is $\lceil \lg n \rceil$ bits.
- 0-1 KNAPSACK, has complexity $O(nW)$, and its length is $O(n \lg M)$ taking $M = \max\{W, \max_i w_i, \max_i v_i\}$.
- If W requires k bits, the cost and space of the algorithm is $n2^k$, exponential in the length W. However the DP algorithm works fine when $W = \Theta(n)$, here $k = O(\log n)$.
- Consider the unary knapsack problem, where all integers are coded in unary (7=1111111). In this case, the complexity of the DP algorithm is polynomial on the size, i.e., UNARY KNAPSACK $\in$P.

Dynamic Programming in Trees

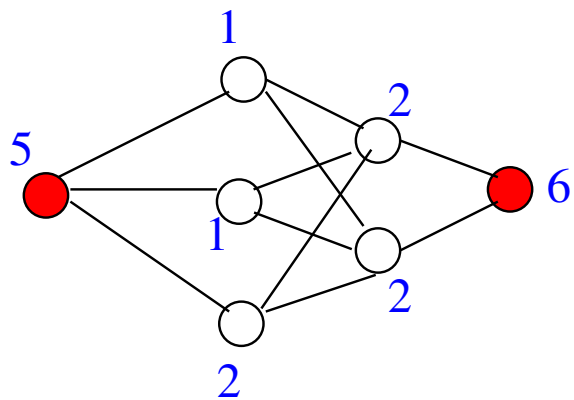
Trees are nice graphs easily adapted to recursion.
Once you root the tree the computation can go recursively from the root to the leaves.
The tree is a good representation of the recursive call. We can identify subproblems with the nodes in the tree .

We can use Dynamic Programming to give polynomial solutions to "difficult" graph problems when the input is restricted to be a tree, or to have a treee-like structure (small treewidth).

In this case instead of having a global table we can think that each node in the tree keeps additional information about the associated subproblem.

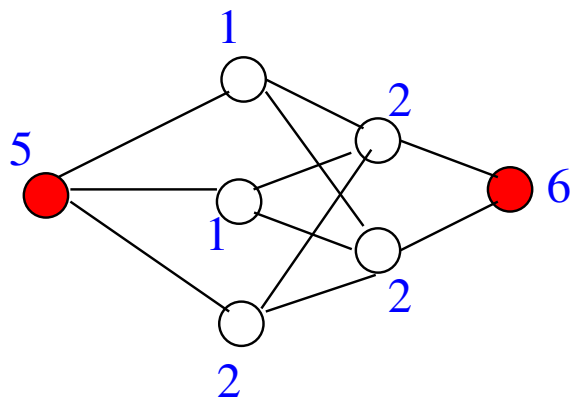
The Maximum Weight Independent Set (MWIS)

Given as input $G = (V, E)$, together with a weight $w : V \to \mathbb{R}$. Find the heaviest $S \subseteq V$ such that no two vertices in S are connected in G.



The Maximum Weight Independent Set (MWIS)

Given as input $G = (V, E)$, together with a weight $w : V \to \mathbb{R}$. Find the heaviest $S \subseteq V$ such that no two vertices in S are connected in G.

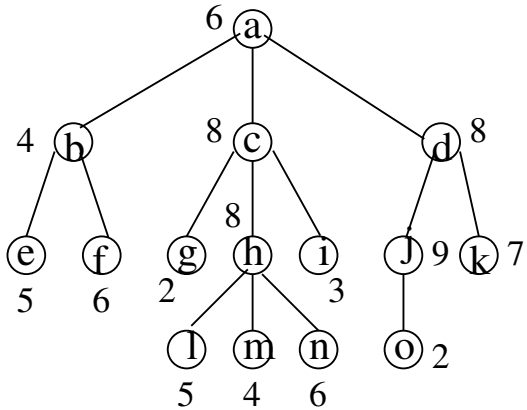


For general graphs, the problem is hard, even for the case in which all vertex have weight 1, i.e. Maximum Independent Set is NP-complete.

Maximum Weight Independent Set on Trees

Given a tree $T = (V, E)$ choose a $r \in V$ and root it from r

i.e. Given a rooted tree $T = (V, E, r)$ and a set of weights $w : V \to \mathbb{R}$, find the independent set of nodes with maximum weight.



Notation:

- For $v \in V$, let T_v be the subtree rooted at v. $T = T_r$.
- Given $v \in V$ let $F(v)$ be the set of children of v, and $N(v)$ be the set of grandchildren of v.

Characterization of the optimal solution

Key observation: An IS can't contain vertices which are father-son.
Let S be an optimal solution.

- If $r \in S$: then $F(r) \not\subseteq S_r$. So $S - \{r\}$ contains an optimum solution for each T_v, with $v \in N(r)$.
- If $r \notin S$: S contains an optimum solution for each T_u, with $u \in F(r)$.

Recursive definition of the optimal solution

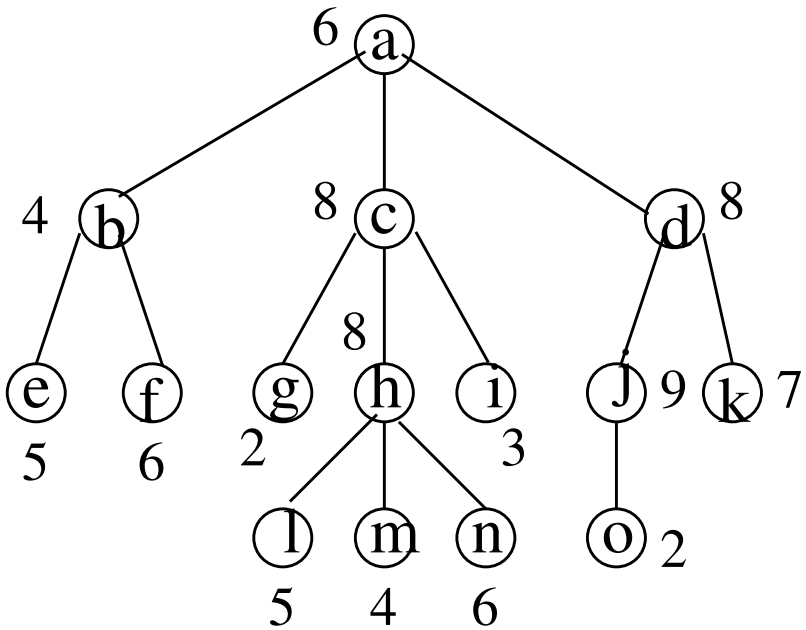
To implement DP, tor every node v, we add one value,
$v.M$: the value of the optimal solution for T_u
Following the recursive structure of the solution we have the following recurrence

$$v.M = \begin{cases} w(v) & \text{if } v \text{ a leaf,} \\ \max\{(\sum_{u \in F(v)} u.M), (w(v) + \sum_{u \in N(v)} u.M)\} & \text{otherwise.} \end{cases}$$

Notice that for any $v \in T$: we have to compute $\sum_{u \in N(v)} u.M$ and for this we must access to the children of its children
To avoid this we add another value to the node
$v.M'$: the sum of the values of the optimal solution of its children, i.e., $\sum_{u \in F(v)} u.M$.

Post-order traversal of a rooted tree

To perform the computation we can follow a DFS, post-order, traversal of the nodes in the tree, computing the additional values at each node.



Post−Order

e f b g l m n h i c o j k d a

DP Algorithm to compute the optimal weight

Let $v_1, \ldots, v_n = r$ be the post-order traversal of T_r

WIS T_r

Let $v_1, \ldots, v_n = r$ the post-order traversal of T_r

for $i = 1$ **to** n **do**

if v_i is a leaf **then**

$v_i.M = w[v_i], v_i.M' = 0$

else

$v_i.M' = \sum_{u \in N(v)} u.M$

$aux = \sum_{u \in N(v)} u.M'$

$v_i.M = \max\{aux, w[v_i] + v_i.M'\}$

return r.M

Complexity: space $= O(n)$, time $= O(n)$

Top-down traversal to obtain an optimal IS

RWIS(v)
if v is a leaf **then**
  return $(\{v\})$
if $M(v_i) = M'(v_i) + w(v_i)$ **then**
  $S = S \cup \{v_i\}$
  for $w \in N(v)$ **do**
    $S = S \cup$ **RWIS**(w)
else
  for $w \in F(v)$ **do**
    $S = S \cup$ **RWIS**(w)
return S

RWIS(r) provides an optimal solution in time $O(n)$

Total cost $O(n)$ and additional space $O(n)$