Divide and Conquer
Dynamic Programming
Greedy Algorithm

Algorithmic Paradigms

- Divide-and-conquer. Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
- Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.
- Greedy. Build up a solution incrementally, myopically optimizing some local criterion.

Dynamic Programming History

- Bellman. Pioneered the systematic study of dynamic programming in the 1950s.
- Etymology.
  - Dynamic programming = planning over time.
  - Secretary of Defense was hostile to mathematical research.
  - Bellman sought an impressive name to avoid confrontation.
    - "it's impossible to use dynamic in a pejorative sense"
    - "something not even a Congressman could object to"

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography.*

Dynamic Programming Applications

- Areas.
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, systems, ….
- Some famous dynamic programming algorithms.
  - Viterbi for hidden Markov models.
  - Unix diff for comparing two files.
  - Smith-Waterman for sequence alignment.
  - Bellman-Ford for shortest path routing in networks.
  - Cocke-Kasami-Younger for parsing context free grammars.

Dynamic Programming

General idea: try to avoid computing the solution of the same sub problems more than once, by storing the computed sub solutions.

This technique is used for optimization problems.

Computing is done from bottom to top.

There are usually 4 steps in any dynamic programming algorithm:

- Characterize the structure of the optimal solution.
- Define the optimal solution recursively
- Compute the optimal solution from bottom to top.
- Combine the solutions for smaller problems stocked in the array to construct an optimal solution of the original problem

Dynamic Programming

Problem: Matrix-chain multiplication

How to compute efficiently the product: $M = M_1 M_2 \dots M_n$ $M_i : d_i, d_{i+1}$

$M = (\dots(M_1 * M_2)* M_3)\dots M_n)$

$M = (M_1 * M_2)*(M_3 * M_4) * \dots M_n)$

....

$M = (M_1 * (M_2 * \dots (M_{n-1} * M_n)\dots)$

To compute the product of 2 matrices A(p,q) and B(q,r) we need p*q*r multiplications.

$A = A_1 \times A_2 \times A_3 \times A_4$

10x20 20x50 50x1 1x100

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

$Cost(A_3 \times A_4) = 50 \times 1 \times 100$

$Cost(A_2 \times (A_3 \times A_4)) = 20 \times 50 \times 100$

$Cost(A_1 \times (A_2 \times (A_3 \times A_4))) = 10 \times 20 \times 100$

Total Cost = 125000

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

Coût$(A_2 \times A_3) = 20 \times 50 \times 1$

Coût$(A_1 \times (A_2 \times A_3)) = 10 \times 20 \times 1$

Coût$((A_1 \times (A_2 \times A_3)) \times A_4) = 10 \times 1 \times 100$

Total Cost = 2200

Dynamic Programming

How to find the most efficient way to insert the paranthesis. A direct approach by checking all possibilities will not work.

$$M = \underbrace{(M_1\, M_2 \ldots M_i)}_{T(i)}\ \underbrace{(M_{i+1}\, M_{i+2} \ldots M_n)}_{T(n-i)}$$

$T(1) = 1 \qquad T(n) = \sum_{i=1}^{n-1} T(i)\, T(n-i)$

$$T(n) = 1/n \binom{2n-2}{n-1}$$

$T(n) \in \Omega\, (4^n/n^{3/2})$

Dynamic Programming

- Dynamic programming approach

1) Structure of an optimal solution

If $M = (M_1\, M_2 \ldots M_i)\, (M_{i+1}\, M_{i+2} \ldots M_n)$ is an optimal solution then :
$(M_1\, M_2 \ldots M_i)$ is an optimal solution for $M_1 \ldots M_i$
AND
$(M_{i+1}\, M_{i+2} \ldots M_n)$ is an optimal solution for $M_{i+1} \ldots M_n$

For $1 \le i \le j \le n$, $m[i, j]$ denotes the optimal solution for the product $M_i\, M_{i+1} \ldots M_j$

$$m[i,j] = \begin{cases} 0 & (i = j) \\ \min\limits_{i \le k < j} \{ m[i,k] + m[k+1, j] + d_{i-1} d_k d_j \} & (i \le j) \end{cases}$$

$$T(n) = \sum_{k=1}^{n-1} (T(k) + T(n-k) + O(1)) \quad = \Omega(2^n)$$

Dynamic Programming

- Dynamic programming approach
- How to compute m[1, 6]?

$$m[i, j] = \min_{i \le k < j} \{ m[i,k] + m[k+1, j] + d_{i-1} d_k d_j \}$$

	1	2	3	4	5	6
1	0	x	x	x	x	x
2		0	x	x	x	x
3			0	x	x	x
4				0	x	x
5					0	x
6						0

Solution m[1,6]

Dynamic Programming

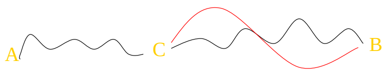
```
Matrix-Chain-Order( p, n )
{
    for i = 1 to n
    m[i,i] = 0
    for len = 2 to n
    for i = 1 to n - len + 1
          j = i + len - 1
          m[i,j] = 8
    for k = i to j-1
          q = m[i,k] + m[k+1,j] + d[i-1]*d[k]*d[j]
          if q < m[i,j] then
              m[i,j] = q
              s[i,j] = k
    return s
}
```


Dynamic Programming

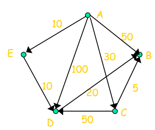
Shortest path: Floyd algorithm

Let G = (V,E) be a connected weighted graph, find all shortest paths between any two vertices in G.

Structure of an optimal solution: If a shortest path between A and B contains a vertex C then the sub paths between A and C and between C and B are optimal.



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & \infty & \infty \\ 2 & 50 & 0 & 15 & 5 \\ 3 & 30 & \infty & 0 & 15 \\ 4 & 15 & \infty & 5 & 0 \end{array}$$



Dynamic Programming

The algorithm: We maintain a 2 dimensional array D (initialized with L). Vertices of the graph are numbered from 1 to n. There are n iterations. After iteration k, D will contain the shortest paths between any two vertices (i, j), using only vertices from the set {1, 2, ..., k}.

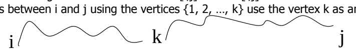
At iteration k and for every pair of vertices (i, j) we check whether or not the introduction of vertex k will give a better shortest path.

Phase k:

$$L = D_0 , D_1 , ...D_n = D$$

Case 1. Adding vertex k does not give a better solution: Dk [i,j] = Dk-1[i,j]

Case 2. All shortest paths between i and j using the vertices {1, 2, ..., k} use the vertex k as an intermediary vertex.



Dk [i,j] = Dk-1[i,k]+ Dk-1[k,j]}

Dk[i,j] = min (Dk-1[i,j] , Dk-1[i,k] + Dk-1[k,j])

We need a second 2 dimensional array P in order to keep track of the actual shortest path between any two vertices:

```
if D[i,k]+D[k,i] < D[i,j]
    then
    D[i,j] = D[i,k]+D[k,i]
    P[i,j] = k
```

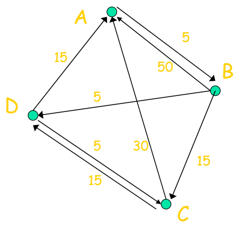
Dynamic Programming

Example:

$$D_0=L \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

$$D_1 \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_2 \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$



Dynamic Programming

Cont ...

$$D_2 \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$P \begin{pmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

$$D_3 \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D=D_4 \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Dynamic Programming

```
Floyd(n, L[ ] [ ], P[ ][ ])
  array D  of size n x n
  array P  of size n x n
  D = L; P=0
  for(k=1; k<=n; k++)                          n
     for(i=1; i<=n; i++)                       n
        for(j=1; j<=n; j++)                    n
           if D[i,k]+D[k,j] < D[i,j]
                D[i,j] = D[i,k]+D[k,j]  }  Constant
                P[i, j] = k             }
  return D
```

$O(n^3)$

Greedy Algorithm

Problem: Making change for n cents using the least number of coins.

Greedy algorithm: At each step, return the largest possible coin

Example 1: Coins: 25c 10c 5c 1c
Making change of 87c: 3 x 25c + 1 x 10c + 2 x 1c

Example 2: Coins: 25c 10c 12c 5c 1c
Making change of 16c: 1 x 12c + 4 x 1c
(Optimal solution: 1x 10c + 1x 5c + 1x 1c (3 coins))

```
function change(C: set {coins}; s: integer): set with repititions
  S := ∅ {solution}
  while s > 0 do
     p := largest coin in C such that its value does not exceed s;
     S := S ∪ {p}
     s := s-p
  return S
```


Greedy Algorithm

A greedy algorithm go through a sequence of steps with a set of choices at each step. It makes the choice that looks best at the moment.

Greedy algorithms are usually simple and easy to implement.

Typically used for optimization problems.

Greedy algorithms do not always lead to an optimal solution

General Scheme of a greedy algorithm:

```
C: set of candidates
S := ∅
while S is not a solution and C ≠ ∅
   x := element in C that maximizes the selection criteria
   C := C - x
   if (S ∪ {x}) is realizable then S := S ∪ {x}

if S is a solution then return S
   else return no-solution
```


Greedy Algorithm

Greedy algorithm for a task scheduling problem: minimizing the total waiting time.

We have a set of n tasks {1, 2, …, n} to be processed sequentially by a single processor. Each task has a processing time ti. The goal is to minimize the total waiting time of all tasks

$$\sum_{i=1}^{n} (\text{time spent by client i in the system: from start to finishing of task i})$$

Example: 3 tasks

$t_1 = 5$ There are 6 possible orders:

$t_2 = 10$ 123: 5 + (5 + 10) + (5 + 10 + 3) = 38
132: 5 + (5 + 3) + (5 + 3 + 10) = 31
$t_3 = 3$ 213: 10 + (10 + 5) + (10 + 5 + 3) = 43
231: 10 + (10 + 3) + (10 + 3 + 5) = 41
312: 3 + (3 + 5) + (3 + 5 + 10) = 29
321: 3 + (3 + 10) + (3 + 10 + 5) = 34

Greedy Algorithm

Greedy algorithm: at each step we choose a task, among the ones left, with a minimum processing time.

Theorem. The greedy algorithm always lead to an optimal scheduling

Proof. Let $I = (i_1, i_2, \ldots, i_n)$ an arbitrary permutation of the tasks $\{1,2, \ldots, n\}$.

If the tasks are scheduled according to the order I then the total waiting time is: $T(I) = t_{i1} + (t_{i1} + t_{i2}) + (t_{i1} + t_{i2} + t_{i3}) + \ldots = \sum_{k=1}^{n} (n - k + 1)\, t_{ik}$

Suppose I: 1 2 ……… a ……….. b ……………. n

where $a < b$ and $t_{ia} > t_{ib}$

If we inverse the positions of a and b in I

I': 1 2 ……… b ……….. a ……………. n

ti1 ti2 ………tib ………..tia …………. tin

$$T(I') = (n - a + 1)\, t_{ib} + (n - b + 1)\, t_{ia} + \sum_{\substack{k=1 \\ k \neq \{a,b\}}}^{n} (n - k + 1)\, t_{ik}$$

Greedy Algorithm

Thus: $T(I) = \sum_{k=1}^{n} (n - k + 1)\, t_{ik}$

$$T(I') = (n - a + 1)\, t_{ib} + (n - b + 1)\, t_{ia} + \sum_{\substack{k=1 \\ k \neq \{a,b\}}}^{n} (n - k + 1)\, t_{ik}$$

$$T(I) - T(I') = (n - a + 1)(t_{ia} - t_{ib}) + (n - b + 1)(t_{ib} - t_{ia}) = (b-a)(t_{ia} - t_{ib}) > 0$$

Therefore the scheduling I' is better than I. Thus the greedy algorithm is always optimal, and any optimal solution is necessarily constructed using the greedy algorithm.

Greedy Algorithm

Another Scheduling problem: n unit-time tasks with deadlines and penalties. A set S={1, 2, …, n} of n unit-tasks.

A set of n integer deadlines d1, d2, …, dn, such that 1≤di≤n for each i, and task i is supposed to finish by time di

A set of n non negative weights g1, g2, …, gn such that a gain gi is incurred if task i is finished by time di.

i	gi	di
1	50	2
2	10	1
3	15	2
4	30	1

Possibles Seq:	gain
1	50
2	10
3	15
4	30
1,3	65
2,1	60
2,3	25
3,1	65
4,1	80
4,3	45

The sequences 1 et 1, 2, 3 corresponds to the same gain. Only task 1 leads to a gain.

Greedy Algorithm

A set of tasks is realizable if there exists at least one ordering of the tasks T that permits to all tasks in T to be executed before the delay.

Theorem: A set of tasks T is realizable if and only if all tasks of T are executed before the delay when the tasks of T are sorted in increasing order with respect to delay.

Proof: Assume that S is realizable with some order:

t1 t2 ... ti ... tj ... tk with $d(tj) < d(ti)$.

By exchanging ti et tj i the order of S we obviously obtain a realizable order:

t1 t2 ... tj ... ti ... tk

Repetitively, we obtain an increasing order which is realizable. The converse is obviously true.

How do we know if a set of tasks is realizable?

It is enough to test the tasks in one specific order: increasing order with respect to delay.

Greedy Algorithm

Greedy Algorithm:

At each step, take the task not considered yet, and with the largest value gi. If the new set of tasks is realizable, add that task to the solution, otherwise disregard the task.

Sort the set of tasks in decreasing order T with respect to gain

S:= ∅ O(n log n)

repeat

Choose the task $e \in T$ with the largest gain

T := T - e

If $S \cup e$ is realizable

S := $S \cup e$

until T is empty

$$\sum_{i=2}^{n} [(i-1) + i] = n^2 - 1$$

Number of steps n-1. Need (i-1) comparisons to add the ith task, i comparisons to verify that if the set is realizable

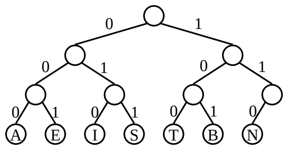
Greedy Algorithm

Huffman Code

- A statistical compression of data, it permits to reduce the length of the coding of an alphabet.
- The Huffman code (1952) is an optimal variable length, that is the average length of the coded text is minimal.
- We notice a size reduction of the order of 20 to 90%.

Greedy Algorithm

- Codes with fixed length: each character is coded with the same number of bits.
- For C characters we need $\lceil \log_2 C \rceil$ bits
- Simple, and easy to manipulate

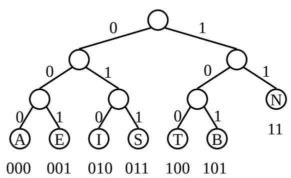


000	A
001	E
010	I
011	S
100	T
101	B
110	N

Greedy Algorithm

- Example 2

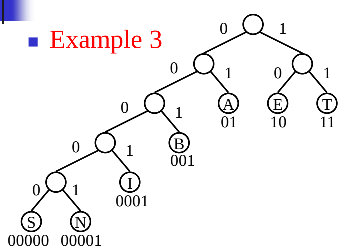
variable length



000	A
001	E
010	I
011	S
100	T
101	B
11	N

Greedy Algorithm

- Example 3



Symbol	Code
A	01
E	10
I	0001
S	00000
T	11
B	001
N	00001

TEST = 11100000011

Greedy Algorithm

Char.	Frequency	Fixed length		Variable length	
		Code	Total Bits	Code	Total Bits
A	41	000	123	01	82
E	35	001	105	10	70
I	12	010	36	0001	48
S	4	011	12	00000	20
T	32	100	96	11	64
B	18	101	54	001	54
N	5	110	15	0000	25
			441	1	363

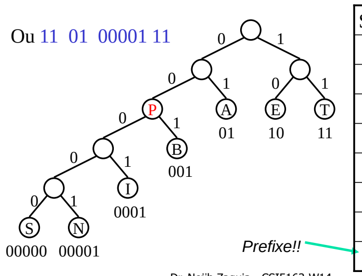
Greedy Algorithm

- A coding should have the prefix property to avoid the ambiguity: a binary sequence cannot be the code of a characters and in the same time the start of the code of another character.
- Prefix codes could be represented by binary trees. Characters are placed as leafs and the code is the reading of the unique path from the root to the leaf.

Greedy Algorithm

11010000111 = 11 01 00 001 11

Ou 11 01 00001 11



Symbole	Code
A	01
E	10
I	0001
S	00000
T	11
B	001
N	00001
P	**00**

Greedy Algorithm

Problem:

- Input
  - A list of characters with frequencies.
- Output
  - The binary tree for the variable length code
    - Total cost = $\sum_s l_s f_s$
    - l_s = number of bits of the code of character s
    - f_s = frequency of the character s

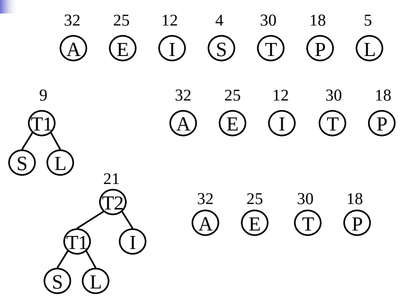
CSI 3505
Algorithmes Voraces

Huffman Algorithm(1952)

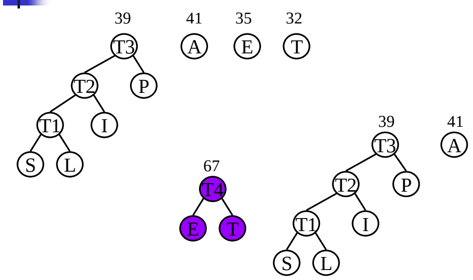
Greedy algorithm for variable length optimal code

- The algorithm is of greedy type: construct a longer path for a character with smaller frequency.
- For each character we construct a tree with a single vertex labelled with the frequency of the character.
- At each stage of the algorithm, we merge the trees with the smallest labels on their root into a new tree with a new root. The new root is labelled with the sum of the labels of the old roots. The roots of the old trees become the left and right child of the new root.
- The algorithm ends when we arrive to a unique tree.

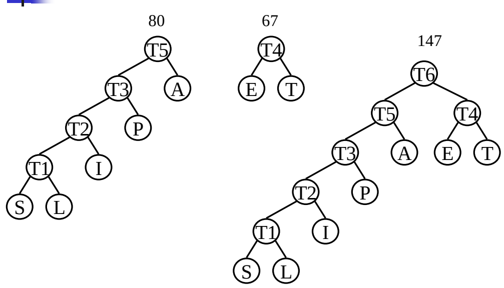
Greedy Algorithm



Greedy Algorithm

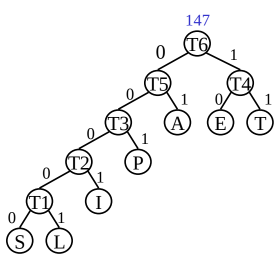


Greedy Algorithm



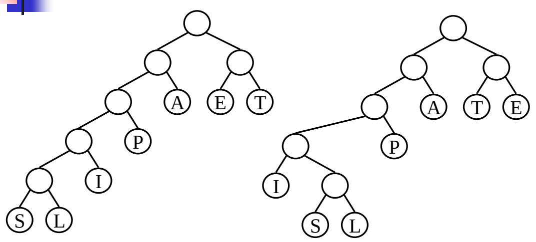
Greedy Algorithm

Code



Symbol	Code
A	01
E	10
I	0001
S	00000
T	11
P	001
L	00001

Greedy Algorithm



Greedy Algorithm

An efficient implementation: use heaps for vertices « priority list »

Huffman (C)

n = number of characters in C

Initialization of the heap Q, (using the frequencies as priority) O(n)

for i in 1..n-1 do

- z = new vertex on tree
- x = Extract-Minimum (Q)
- y = Extract-Minimum (Q)
- left-child z = x
- right=child z = y
- f[z] = f[x] + f[y]
- Insert (Q, z)

log (n)

end for

Return tree

Total Complexity: O(n logn)

Another example: Dynamic Programming

Longest common subsequence problem

Sequences X =<x1, x2, ..., xs> and Y=<y1, y2, ..., yt>

Z =<z1, z2, ..., zk> is a subsequence of X if there exists a strictly increasing sequence <i(1), i(2), ..., i(k)> of indices of X such that for all j, $x_{i(j)} = z_j$

<A, C, Z, K> is a subsequence of <C, A, F, C, C, Z, K, Y> with index sequence <2, 4, 6, 7> or <2, 5, 6, 7>

Z is a common subsequence of X and Y if Z is a subsequence of both X and Y.

X=<A, B,C , B, D, A, B> Y=<B, D, C, A, B, A>

A COMMON SUBSEQUENCE (CS) B,C, A

A longest CS (LCS) B, C, B, A

Another example: Dynamic Programming

Brute force: try all common subsequence– takes exponential time

X =<x1, x2, ..., xi> "Xi is the i-th prefix of X"

Theorem: Let $Z=<z_1, z_2, \ldots, z_k>$ be any LCS of X and Y.

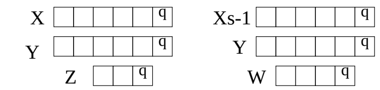
(1) If $x_s = y_t$ then $x_k=x_s=y_t$ and Z_{k-1} is an LCS of X_{s-1} and Y_{t-1}

(2) If $x_s \neq y_t$ then $z_k \neq x_s$ implies that Z is an LCS of X_{s-1} and Y

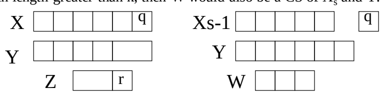
(3) If $x_s \neq y_t$ then $z_k \neq y_t$ implies that Z is an LCS of X and Y_{t-1}

Another example: Dynamic Programming

Proof. (1) if $z_k \neq x_s$, then we could append $x_s=y_t$, to Z and gets a CS of length k+1. If there were a CS W of X_{s-1} and Y_{t-1} with length k, then we could append $x_s=y_t$ to W and get a CS of X and Y of length k+1. Thus, a CS of X_{s-1} and Y_{t-1} with length k is an LCS i.e. Zk-1 is an LCS of X_{s-1} and Y_{t-1}



(2) if $z_k \neq x_s$, then that Z is a CS of X_{s-1} and Y. If there were a CS W of X_{s-1} and Y with length greater than k, then W would also be a CS of X_s and Y.



(3) Same argument as (2)

Another example: Dynamic Programming

Recursive solution for finding LCS of X and Y

If $x_s=y_t$ then find an LCS of X_{s-1} and Y_{t-1}, and then append $x_s=y_t$ to this LCS

If $x_s \neq y_t$ then solve two problems:
(1) Find a LCS of X_{s-1} and Y; (2) find an LCS of X and Y_{t-1};
whichever of these two LCS's is longer, that's an LCS of X and Y.

c[i, j]=length of an LCS of the sequences X_i and Y_j
when i=0 or j=0 the c[i, j] =0
when i>0, j>0, and $x_i = y_j$ then c[i, j]= c[i-1, j-1]+1
when i>0, j>0 and $x_i \neq y_j$ the c[i, j] = max {c[i, j-1], c[i-1, j]}

Complexity: O(st)

Another example: Dynamic Programming

LCS-Length(X, Y)

set c[i, 0]'s and c[0, j]'s to 0

for i=1 to s

  for j=1 to t

    if $x_i=x_j$ the c[i, j] = c[i-1, j-1]+1

    else c[i, j]=max{ c[i, j-1], c[i-1, j]}

Greedy vs. Dynamic Programming

- 0-1 knapsack problem

A thief robbing a store finds n items

i-th item has weight w[i] and is worth v[i] dollars *(w[i] and v[i] are integers).*

If the thief can carry at most L weight in his knapsak, what items should he take to make the most profitable heist?

- Fractional knapsack problem: same as above, except that the thief can take fractions of items (e.g. an item might be gold dust)

Greedy vs. Dynamic Programming

Optimal substructures property:

Consider the most valuable load that weights ≤ *L* pounds in 0-1 problem, if we remove item j from this load, the remaining load must be the most valuable load weighting at most *L=w[j]* that can be taken from the n-1 original items excluding j.

In fractional problem, if we remove *w* pounds of item j, the remaining load must be the most valuable load weighting at *L=w* that can be taken from the n-1 original items plus *w[j] – w* pounds of item j.

Greedy vs. Dynamic Programming

A greedy solution for the fractional problem

Compute the *v[i]/w[i]* for each item i.

Sort in decreasing order the items according to the values of *v[i]/w[i]*

For items i=1 to n

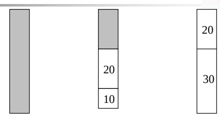
take as much of item I as there while not exceeding weight limit L

Running time: O(n log n)

The same greedy algorithm does not work for the 0-1 knapsack problem

Greedy vs. Dynamic Programming

Items	Weight	Gain
Item 1	10	60
Item 1	20	100
Item 1	30	120



Take the most expensive does not work: fails for L=30

Dynamic programming solution for 0-1 problem:
Sort items by increasing weight. Let i be the highest-numbered item in an optimal solution S for an L kilo knapsack and items 1..n.
Then S'=S-{i} must be an optimal solution for items 1..i-1 and an L-w[i] kilos knapsacks.
The value of solution S is v[i] plus the value of solution S'.
Let c[i, wj]=the value of the solution for items 1..i and maximum weight w
- when i=0 or w=0 the c[i, w] =0
- when w[i] >w, then c[i, w]= c[i-1, w]
- when i>0 and w w[i], then c[i, w] = max {v[i]+c[i-1, w-w[i]], c[i-1, w]}

Greedy vs. Dynamic Programming

Dynamic 0-1-Knapsack:

```
for w=0 to L
   c[0, w]=0
for i=1 to n
    c[i, 0]=0
    for w=1 to L
        if w[i] ≤ w then
          if v[i] + c[i-1, w-w[i]] > c[i-1, w]
                then c[i, w] = v[i] + c[i-1, w-w[i]]
                else c[i, w]=c[i-1, w]
        else c[i, w] = c[i-1, w]
```

Complexity: O(nL)