CH08-320201

Algorithms and Data Structures

ADS

Lecture 19

Dr. Kinga Lipskoch

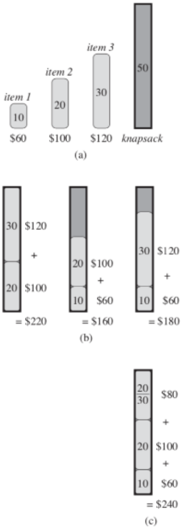
Spring 2019

Conclusions: Greedy Approach for the Knapsack Problem

- As already mentioned, the locally optimal choice of a greedy approach does not necessary lead to a globally optimal one.
- For the knapsack problem, the greedy approach actually fails to produce a globally optimal solution.
- However, it produces an approximation, which sometimes is good enough.

0-1 vs. Fractional Knapsack Problem

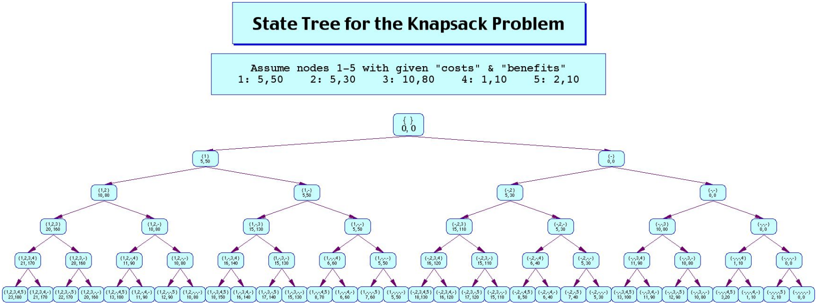
- 0-1 knapsack problem
  - Either take (1) or leave an object (0)
  - Greedy fails to produce global optimum
- fractional knapsack problem
  - You can take fractions of an object
  - Greedy strategy: value per weight v/w $\rightarrow$ begin taking as much as possible of item with greatest v/w , then with next greater v/w, ...
  - Leads to global optimum (proof by contradiction)
- What is the difference?



Alternatives for 0-1 Knapsack (1)

Brute-Force:

- Benefit: it finds the optimum
- Drawback: it takes very long - $O(2^n)$
- Because recomputing the results of the same subproblems over and over again



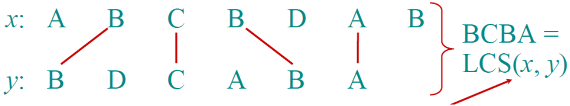
Alternatives for 0-1 Knapsack (2)

Dynamic programming:

- Optimal substructure:
  - optimal solution to problem consists of optimal solutions to subproblems
- Overlapping subproblems:
  - few subproblems in total, many recurring instances of each
- Main idea:
  - use a table to store solved subproblems

Dynamic Programming: Problem

- Given two sequences $x[1..m]$ and $y[1..n]$, find a longest subsequence common to both of them.
- Example:



Brute-Force Solution

Check every subsequence of $x[1..m]$ to see if it is also a subsequence of $y[1..n]$.

Analysis:

- Checking per subsequence is done in $O(n)$.
- As each bit-vector of m determines a distinct subsequence of x, x has 2^m subsequences.
- Hence, the worst-case running time is $O(n \cdot 2^m)$, i.e., it is exponential.

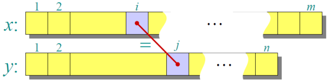
Strategy

- Look at length of longest-common subsequence.
- Let $|s|$ denote the length of a sequence s.
- To find $LCS(x, y)$, consider prefixes of x and y (i.e., we go from right to left)
- Definition: $c[i, j] = |LCS(x[1..i], y[1..j])|$.
  In particular, $c[m, n] = |LCS(x, y)|$.
- Theorem (recursive formulation):

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

Proof (1)

Case $x[i] = y[j]$:



Let $z[1..k] = LCS(x[1..i], y[1..j])$ with $c[i,j] = k$.
Then, $z[k] = x[i] = y[j]$ (else z could be extended).
Thus, $z[1..k-1]$ is CS of $x[1..i-1]$ and $y[1..j-1]$.
Claim: $z[1..k-1] = LCS(x[1..i-1], y[1..j-1])$.

- Assume w is a longer CS of $x[1..i-1]$ and $y[1..j-1]$, i.e., $|w| > k-1$.
- Then the concatenation $w++z[k]$ is a CS of $x[1..i]$ and $y[1..j]$ with length $> k$.
- This contradicts $|LCS(x[1..i], y[1..j])| = k$.
- Hence, the assumption was wrong and the claim is proven.

Hence, $c[i-1, j-1] = k-1$, i.e., $c[i,j] = c[i-1, j-1] + 1$.

Proof (2)

Case $x[i] \neq y[j]$:

Then, $z[k] \neq x[i]$ or $z[k] \neq y[j]$.

- $z[k] \neq x[i]$:
  Then, $z[1..k] = LCS(x[1..i-1], y[1..j])$.
  Thus, $c[i-1, j] = k = c[i, j]$.
- $z[k] \neq y[j]$:
  Then, $z[1..k] = LCS(x[1..i], y[1..j-1])$.
  Thus, $c[i, j-1] = k = c[i, j]$.

In summary, $c[i, j] = \max\{c[i-1, j], c[i, j-1]\}$.

Dynamic Programming Concept (1)

Step 1: Optimal substructure.
An optimal solution to a problem contains optimal solutions to subproblems.

Example:
If $z = LCS(x, y)$, then any prefix of z is an LCS of a prefix of x and a prefix of y.

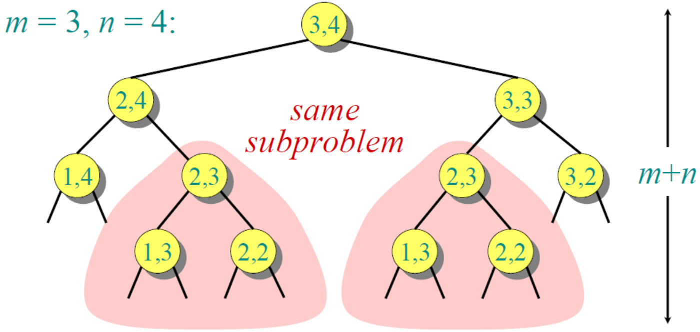
Recursive Algorithm

- Computation of the length of LCS:

```
LCSlength(x,y,i,j):
  if i=0 or j=0
    return 0
  else if x[i] = y[j]
    return LCSlength(x,y,i-1,j-1)+1
  else return max {LCSlength(x,y,i-1,j),
                   LCSlength(x,y,i,j-1)}
```

- Remark: if $x[i] \neq y[j]$, the algorithm evaluates two subproblems that are very similar.

Recursive Tree



Height $= m + n \Rightarrow$ work potentially exponential, but we're solving subproblems already solved!

Dynamic Programming Concept (2)

Step 2: Overlapping subproblems.
A recursive solution contains a "small" number of distinct subproblems repeated many times.

Example:
The number of distinct LCS subproblems for two prefixes of lengths m and n is only $m \cdot n$.

Memoization Algorithm

Memoization:

- After computing a solution to a subproblem, store it in a table.
- Subsequent calls check the table to avoid repeating the same computation.

Recursive Algorithm with Memoization

Computation of the length of LCS:

```
LCSlength (x,y,i,j):
  if c[i,j] = NIL
    then if i=0 or j=0
      c[i,j] = 0
    else if x[i] = y[j]
      c[i,j] = LCSlength (x,y,i-1,j-1)+1
    else c[i,j] = max {LCSlength (x,y,i-1,j),
                       LCSlength (x,y,i,j-1)}
  return c[i,j]
```

Dynamic Programming (1)

Compute the table bottom-up:

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Dynamic Programming (2)

Compute the table bottom-up:

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

Complexity

- Time complexity: $T(m, n) = \Theta(m \cdot n)$
- Space complexity: $S(m, n) = \Theta(m \cdot n)$

Reconstructing LCS

- Trace backwards:

i		j	0	1	2	3	4	5	6
		y_j		B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑0	↑0	↑0	↖1	←1	↖1
2	B		0	↖1	←1	←1	↑1	↖2	←2
3	C		0	↑1	↑1	↖2	←2	↑2	↑2
4	B		0	↖1	↑1	↑2	↑2	↖3	←3
5	D		0	↑1	↖2	↑2	↑2	↑3	↑3
6	A		0	↑1	↑2	↑2	↖3	↑3	↖4
7	B		0	↖1	↑2	↑2	↑3	↖4	↑4

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

- Time complexity: $O(m + n)$

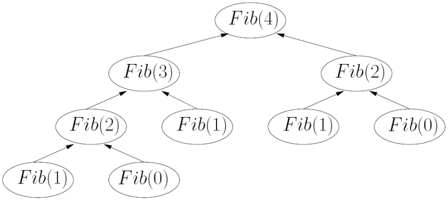
Fibonacci Numbers Revisited (1)

Recall:

- Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

- Recursion tree of brute-force implementation:

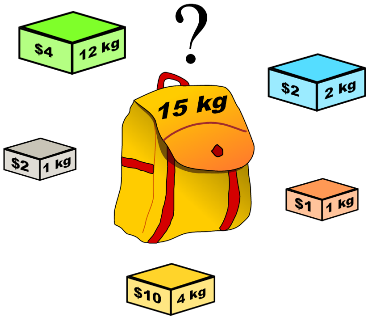


Fibonacci Numbers Revisited (2)

Dynamic programming solution:

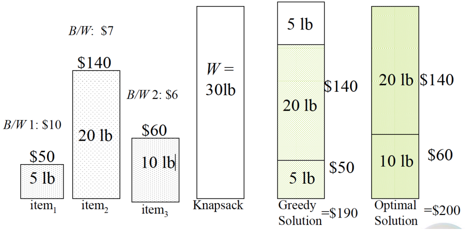
- Avoid re-computations of same terms.
- Store results of subproblems in a table.
- Thus, $Fib(k)$ is computed exactly once for each k.
- This basically leads to the previously discussed bottom-up approach.
- Computation time is $T(n) = \Theta(n)$.

Knapsack Problem (Revisited)



Knapsack Problem: Greedy Algorithm

- Greedy approaches make a locally optimal choice.
- There is no guarantee that this will lead to a globally optimal solution.
- In the 0-1 Knapsack Problem it did not.



Knapsack Problem: Dynamic Programming Approach (1)

- Let us try a dynamic programming approach.
- We need to carefully identify the subproblems.
- If items are labeled $1..n$, then a subproblem would be to find an optimal solution for $S_k = \{\text{items labeled } 1, 2, ..., k\}$.

Knapsack Problem: Dynamic Programming Approach (2)

Max weight: W = 20

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_4 = 3$ $b_4 = 4$	

For S_4:

Total weight: 14

Maximum benefit: 20

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_5 = 9$ $b_5 = 10$

For S_5:

Total weight: 20

Maximum benefit: 26

Item #	Weight w_i	Benefit b_i
1	2	3
2	4	5
3	5	8
4	3	4
5	9	10

S_4

S_5

Solution for S_4 is not part of the solution for S_5

Knapsack Problem: Dynamic Programming Approach (3)

- Re-define the subproblem by also considering the weight that is given to the subproblem.
- The subproblem then will be to compute $V[k, w]$, i.e., to find an optimal solution for $S_k = \{\text{items labeled } 1, 2, ..k\}$ in a knapsack of size w, with $w \leq W$.
- $V[k, w]$ denotes the overall benefit of the solution.
- Question: Assuming we know $V[i, j]$ for $i = 0, 1, 2, ..., k-1$ and $j = 0, 1, 2, ..., w$, how can we derive $V[k, w]$?
- Answer:

$$V[k,w] = \begin{cases} V[k-1,w] & \text{if } w_k > w \\ \max\{V[k-1,w], V[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$