COT 6405 Introduction to Theory of Algorithms

Topic 13. Dynamic programming

Dynamic Programming (DP)

- Like divide-and-conquer, solve problem by combining the solutions to sub-problems.
- Divide-and-conquer vs. DP:
  - divide-and-conquer: Independent sub-problems
    - solve sub-problems independently and recursively, (→ so same sub-problems solved repeatedly)
  - DP: Sub-problems are dependent
    - sub-problems share sub-sub-problems
    - every sub-problem is solved just once
    - solutions to sub-problems are stored in a table and used for solving higher level sub-problems.

Overview of DP

- Not a specific algorithm, but a technique (like divide-and-conquer).
- Doesn’t really refer to computer programming
- Application domain of DP
  - Optimization problem: find a solution with the optimal (maximum or minimum) value

Matrix-chain multiplication problem

- Given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices
  - where for $i = 1, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$
  - fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.
- What is the minimum number of multiplications required to compute $A_1 \cdot A_2 \cdot \ldots \cdot A_n$?
- What order of matrix multiplications achieves this minimum? This is our goal !

Matrix-chain multiplication problem

- Consider the problem of a chain $\{A_1, A_2, A_3\}$.
  - The dimensions of the matrices are 10×100, 100×5, and 5×50, respectively
- $((A_1 A_2) A_3)$:
  - 10*100*5 +10*5*50 = 7500 scalar multiplications
- $(A_1 (A_2 A_3))$:
  - 100*5*50 +10*100*50 = 75,000 scalar multiplications

A Possible Solution

- Exhaustively checking all possible parenthesizations
  - Not an efficient algorithm at all!
- P(*n*): the number of alternative parenthesizations of a sequence of n matrices
  - The split may occur between the *k*th and (*k*+1)st matrices for any *k* = 1, 2, ..., n-1
  - Ω (2^n)

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

Four-step method

1. Characterize the structure of an optimal solution
   - Optimal solutions incorporate solutions to subproblems
   - The problem must have an optimal structure
2. Recursively define the value of an optimal solution
   - Combine solutions to subproblems
3. Compute the value of an optimal solution
   - typically in a bottom-up fashion
   - Get rid of recurrences
4. Construct an optimal solution from computed information
   - Trace back the solution steps

Step 1: Find the structure of an optimal parenthesization

- Finding the optimal substructure and using it to construct an optimal solution to the problem based on optimal solutions to subproblems.

Both must be **Optimal** for sub-chain

$$((A_1 A_2 \cdots A_k)(A_{k+1} A_{k+2} \cdots A_n))$$

Then combine them for the original problem

- The key is to find k ; then, we can build the global optimal solution

Step 2: A recursive solution to define the cost of an optimal solution

- Define $m[i, j]$ = the minimum number of multiplications needed to compute the matrix $A_{i..j} = A_i A_{i+1} \cdots A_j$
- Goal: to compute $m[1, n]$
- Basis: m(i, i) = 0
  - Single matrix, no computation
- Recursion: How to define m[i, j] recursively?
  - $((A_i A_2 \cdots A_k)(A_{k+1} A_{k+2} \cdots A_j))$

Step2: Defining $m[i,j]$ Recursively

- Consider all possible ways to split A_i through A_j into two pieces: $(A_i \cdot \ldots \cdot A_k) \cdot (A_{k+1} \cdot \ldots \cdot A_j)$
- Compare the costs of all these splits:
  - best case cost for computing the product of the two pieces
  - plus the cost of multiplying the two products
  - Take the best one
  - $m[i,j] = \min_k \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \}$

Step2: Defining $m[i,j]$ Recursively (Cont'd)

- $m[i,j] = \min_k\{ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \}$

$$\underbrace{((A_i A_2 \cdots A_k)}_{B_1}\underbrace{(A_{k+1} A_{k+2} \cdots A_j))}_{B_2}$$

- minimum cost to compute B_1 is $m(i, k)$
- minimum cost to compute B_2 is $m(k+1, j)$
- for $i = 1,\ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$
- The dimension of B_1 is $p_{i-1}p_k$, The dimension of B_2 is p_kp_j
- Therefore, cost to compute $B_1 \cdot B_2$ is $p_{i-1}p_kp_j$

Step 3: Computing the Optimal Cost by Finding Dependencies Among Subproblems

- $m[i,j] = \min_k\{ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \}$
- k ranges between i and j-1
- Computing $m[i,j]$ uses k = i, i+1, i+2,…,j-1
- **$m[i,k]$:** $m[i,i]$, $m[i,i+1]$, …, $m[i,j-1]$
- **$m[k+1,j]$:** $m[i+1,j]$, $m[i+2,j]$,…,$m[j,j]$

Step 3: Computing the Optimal Cost by Finding Dependencies Among Subproblems (cont'd)

m[]

	1	2	3	4	5
1	0				
2	n/a	0			
3	n/a	n/a	0		
4	n/a	n/a	n/a	0	
5	n/a	n/a	n/a	n/a	0

GOAL: m(1,5)

- *m*[*i,i*], *m*[*i,i+1*], …, *m*[*i,j-1*]: everything in same row to the left
- *m*[*i,j*], *m*[*i+1,j*],…,*m*[*j,j*]: everything in same column below:

Identify Order for Solving Subproblems

- Solve the subproblems (i.e., fill in the table entries) along the diagonal

	1	2	3	4	5
1	0				
2	n/a	0			
3	n/a	n/a	0		
4	n/a	n/a	n/a	0	
5	n/a	n/a	n/a	n/a	0

14

An example

	1	2	3	4
1	0	1200		
2	n/a	0	400	
3	n/a	n/a	0	10000
4	n/a	n/a	n/a	0

A1 is 30x1
A2 is 1x40
A3 is 40x10
A4 is 10x25
p0 = 30, p1 = 1
p2 = 40, p3 = 10
p4 = 25

m[1,2] = A1A2 : 30X1X40 = 1200,
m[2,3] = A2A3 : 1X40X10 = 400,
m[3,4] = A3A4: 40X10X25 = 10000

An example (cont'd)

	1	2	3	4
1	0	1200	700	
2	n/a	0	400	
3	n/a	n/a	0	10000
4	n/a	n/a	n/a	0

A1 is 30x1
A2 is 1x40
A3 is 40x10
A4 is 10x25
p0 = 30, p1 = 1
p2 = 40, p3 = 10
p4 = 25

$$m[i,j] = \min_k\{\ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\ \}$$

m[1,3]: i = 1, j = 3, k = 1, 2
= min{ m[1,1]+m[2,3]+p0*p1*p3, m[1, 2]+m[3,3]+p0*p2*p3}
= min{0 + 400 + 30*1*10, 1200+0+30*40*10} = 700

An example (cont'd)

	1	2	3	4
1	0	1200	700	
2	n/a	0	400	650
3	n/a	n/a	0	10000
4	n/a	n/a	n/a	0

A1 is 30x1
A2 is 1x40
A3 is 40x10
A4 is 10x25
p0 = 30, p1 = 1
p2 = 40, p3 = 10
p4 = 25

$$m[i,j] = \min_k\{\ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\ \}$$

m[2,4]: i = 2, j = 4, k = 2, 3
= min{ m[2,2]+m[3,4]+p1*p2*p4, m[2, 3]+m[4,4]+p1*p3*p4}
= min{0 + 10000 + 1*40*25, 400+0+1*10*25} = 650

An example (cont'd)

	1	2	3	4
1	0	1200	700	1400
2	n/a	0	400	650
3	n/a	n/a	0	10000
4	n/a	n/a	n/a	0

A1 is 30x1
A2 is 1x40
A3 is 40x10
A4 is 10x25
p0 = 30, p1 = 1
p2 = 40, p3 = 10
p4 = 25

$$m[i,j] = \min_k\{\ m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\ \}$$

m[1,4]: i = 1, j = 4, k = 1, 2, 3
= min{ m[1,1]+m[2,4]+p0*p1*p4, m[1,2]+m[3,4]+p0*p2*p4,
m[1,3]+m[4,4]+p0*p3*p4}
= min{0+650+30*1*25, 1200+10000+30*40*25, 700+0+30*10*25}
= 1400

Step 3: Keeping Track of the Order

- We know the cost of the cheapest order, but what is that cheapest order?
  - Use another array s[]
  - update it when computing the minimum cost in the inner loop
- After m[] and s[] are done, we call a recursive algorithm on s[] to print out the actual order

Example

m[]

s[]

	1	2	3	4
1	0	1200 1	700 1	1400 1
2	n/a	0	400 2	650 3
3	n/a	n/a	0	10,000 3
4	n/a	n/a	n/a	0

A1 is 30x1

A2 is 1x40

A3 is 40x10

A4 is 10x25

keep track of cheapest split point found so far: between A_k and A_{k+1}

An example

	1	2	3	4
1	0	1		
2	n/a	0	2	
3	n/a	n/a	0	3
4	n/a	n/a	n/a	0

A1 is 30x1
A2 is 1x40
A3 is 40x10
A4 is 10x25
p0 = 30, p1 = 1
p2 = 40, p3 = 10
p4 = 25

m[1,2] = A1A2 : 30X1X40 = 1200, s[1,2] = 1
m[2,3] = A2A3 : 1X40X10 = 400, s[2,3] = 2
m[3,4] = A3A4: 40X10X25 = 10000, s[3,4] = 3

An example (cont'd)

	1	2	3	4
1	0	1	1	
2	n/a	0	2	
3	n/a	n/a	0	3
4	n/a	n/a	n/a	0

A1 is 30x1
A2 is 1x40
A3 is 40x10
A4 is 10x25
p0 = 30, p1 = 1
p2 = 40, p3 = 10
p4 = 25

m[1,3]: *i* = 1, *j* = 3, *k* = 1, 2
= min{ m[1,1]+m[2,3]+p0*p1*p3, m[1, 2]+m[3,3]+p0*p2*p3}
= min{0 + 400 + 30*1*10, 1200+0+30*40*10} = 700
m[1,3] is the minimum value when k = 1, so s[1,3] = 1

An example (cont'd)

	1	2	3	4
1	0	1	1	
2	n/a	0	2	3
3	n/a	n/a	0	3
4	n/a	n/a	n/a	0

A1 is 30x1
A2 is 1x40
A3 is 40x10
A4 is 10x25
p0 = 30, p1 = 1
p2 = 40, p3 = 10
p4 = 25

m[2,4]: *i* = 2, *j* = 4, *k* = 2, 3
= min{ m[2,2]+m[3,4]+p1*p2*p4, m[2, 3]+m[4,4]+p1*p3*p4}
= min{0 + 10000 + 1*40*25, 400+0+1*10*25} = 650
m[2,4] is the minimum value when k = 3, so s[2,4] = 3

An example (cont'd)

	1	2	3	4
1	0	1	1	1
2	n/a	0	2	3
3	n/a	n/a	0	3
4	n/a	n/a	n/a	0

A1 is 30x1
A2 is 1x40
A3 is 40x10
A4 is 10x25
p0 = 30, p1 = 1
p2 = 40, p3 = 10
p4 = 25

m[1,4]: *i* = 1, *j* = 4, *k* = 1, 2, 3
= min{ m[1,1]+m[2,4]+p0*p1*p4, m[1,2]+m[3,4]+p0*p2*p4,
m[1,3]+m[4,4]+p0*p3*p4}
= min{0+650+30*1*25, 1200+10000+30*40*25, 700+0+30*10*25}
= 1400
m[1,4] is the minimum value when k = 1, so s[1,4] = 1

Step 4: Using S to Print Best Ordering (cont'd)

	1	2	3	4
1	0	1	1	1
2	n/a	0	2	3
3	n/a	n/a	0	3
4	n/a	n/a	n/a	0

A1 A2 A3 A4
s[1,4] = 1 - > A1 (A2 A3 A4)
s[2,4] = 3 - > (A2 A3) A4
A1 (A2 A3 A4) -> A1 ((A2 A3) A4)

Step 3: Computing the optimal costs

```
MATRIX-CHAIN-ORDER(p)
  n = length[p] − 1
  Let m [1..n, 1..n] and s[1.. n-1, 2..n] be new tables
  for i = 1 to n
        m[i, i] = 0
  for l = 2 to n
        for i = 1 to (n − l + 1)
              j = i + l − 1
                 m[i, j] = ∞
                 for k = i to (j − 1)
                      q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                      if q < m[i, j]
                           m[i, j] = q
                           s[i, j] = k
  return m and s
```

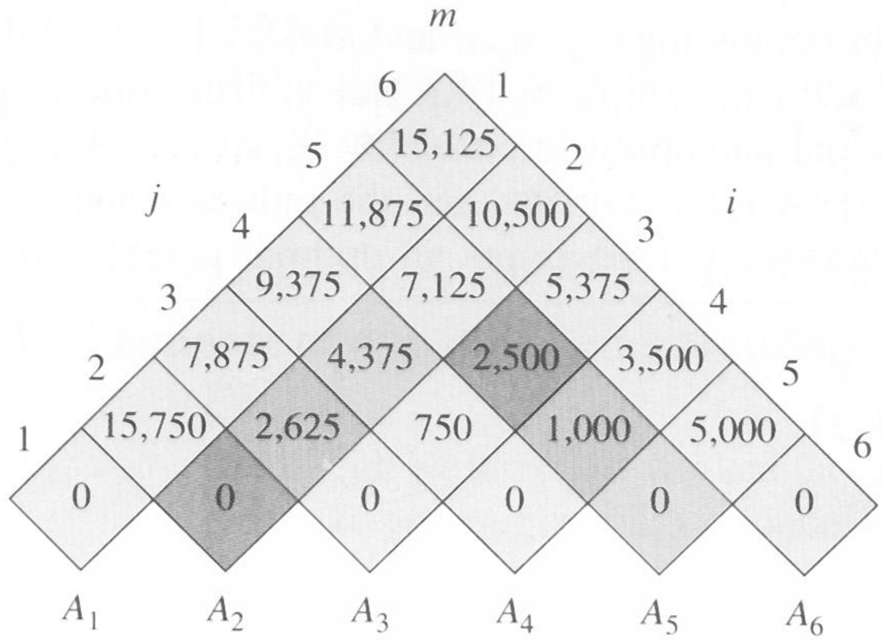
Complexity: $O(n^3)$ Space: $\Theta(n^2)$

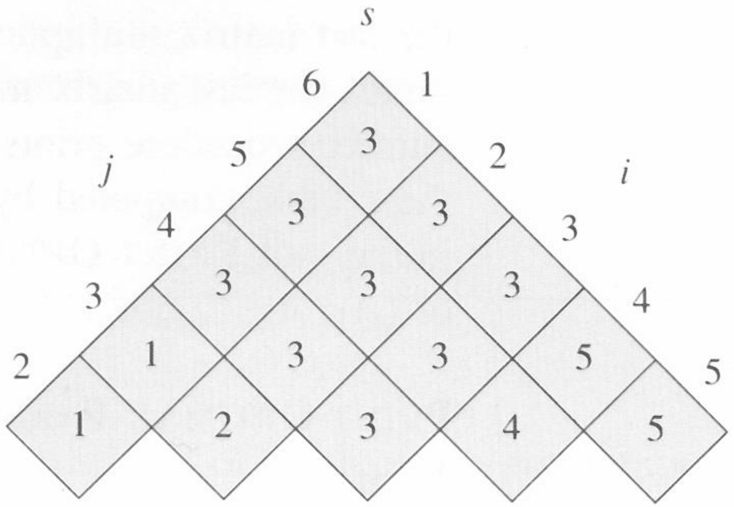
Step 4: Using S to Print Best Ordering

- $s[i,j]$ is the split position for $A_iA_{i+1}\ldots A_j \rightarrow A_i\ldots A_{s[i,j]}$ and $A_{s[i,j]+1}\ldots A_j$
- Call Print-Optimal-PARENS(s, 1, n)

```
Print-Optimal-PARENS (s, i, j)
  if (i == j) then
        print "A" + i              //+ is string concatenation
  else
        print "("
        Print-Optimal-PARENS (s, i,  s[i, j] )
        Print-Optimal-PARENS (s,  s[i, j]+1, j)
        Print ")"
```

An example





A1 A2 A3 A4 A5 A6

s[1,6] = 3 - > (A1 A2 A3) (A4 A5 A6)

s[1,3] = 1 - > A1 (A2 A3)

s[4,6] = 5 -> (A4 A5) A6

(A1 A2 A3) (A4 A5 A6) -> ((A1 (A2 A3))((A4 A5) A6))

16.3 Elements of dynamic programming

- Optimal substructure
  - a problem exhibits optimal substructure if an optimal solution to the problem contains within its optimal solutions to subproblems.
  - Example: Matrix-multiplication problem

- Overlapping subproblems
  - The space of subproblems is “small” in that a recursive algorithm for the problem solves the same subproblems over and over.
  - Total number of distinct subproblems is typically polynomial in input size

- Reconstructing an optimal solution

Follow a common pattern in discovering optimal substructure

1. We show that a solution to the problem consists of making a choice. Making this choice leaves one or more subproblems to be solved.
2. We suppose that for a given problem, we are given the choice that leads to an optimal solution.
3. Given this choice, we determine which subproblems ensue and how to best characterize the resulting space of subproblems.
4. We show that the solutions to the subproblems used for the optimal solution to the problem must be optimal by using a "*cut-and-paste*" technique.

Characterize Subproblem Space

- Try to keep the space as simple as possible
- In matrix-chain multiplication, subproblem space $A_1A_2 \ldots A_j$ will not work
  - $A_1A_2 \ldots A_k$ *and* $A_{k+1}A_2 \ldots A_j$ → not a single form $A_1A_2 \ldots A_k$
  - *The second subproblem does not start from 1*
- Instead, $A_iA_{i+1} \ldots A_j$ (vary at both ends) works.

Optimal substructure varies across problem domains in two ways

1. how many subproblems are used in an optimal solution to the original problem, and
2. how many choices we have in determining which subproblem(s) to use in an optimal solution
3. Example: Matrix multiplication problem: (j-i) choices, 2 subproblems

- DP solve the problem in bottom-up manner

Running Time for DP Programs

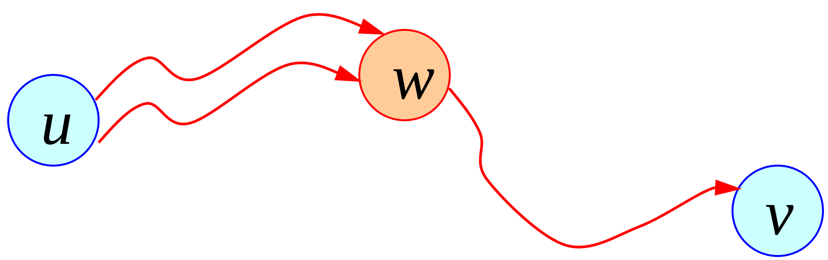
- # of overall subproblems × # of choices
  - In matrix-chain multiplication, $O(n^2) \times O(n) = O(n^3)$
- The cost = costs of solving subproblems + cost of making the choice
  - In matrix-chain multiplication, the cost of a choice k is $p_{i-1}p_kp_j$.

Optimal structure may not exist

- We cannot assume it when it is not there
- Consider the following two problems. in which we are given a directed graph $G = (V, E)$ and vertices $u, v \in V$
  - P1: Unweighted shortest path (USP)
    - Find a path from u to v consisting of the fewest edges. Good for Dynamic programming.
  - P2: Unweighted longest simple path (ULSP)
    - A path is simple if all vertices in the path are distinct
    - Find a simple path from u to v consisting of the most edges. Not good for Dynamic programming.

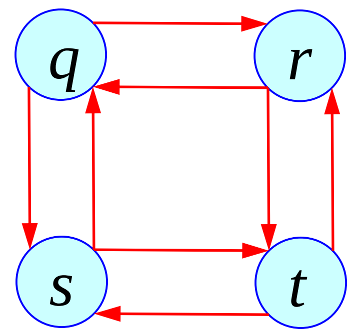
DP is good to find shortest path

- Given a shortest path from *u* to *v*, there must exist an intermediate vertex *w*, so that we can decompose the path *u*⇢*v* to *u*⇢*w* and *w*⇢*v*
  - where both u ⇢ *w* and *w* ⇢ *v* are both (optimal) shortest paths
  - Another path u ⇢ *w* cannot be an optimal solution → otherwise, *cut-and-paste*



DP is not good to find Unweighted longest simple path

- Path $q \rightarrow r \rightarrow t$ is a longest simple path from q to t, but the subpath $q \rightarrow r$ is not a longest simple path from q to r (should be $q \rightarrow s \rightarrow t \rightarrow r$)
  - nor is the subpath $r \rightarrow t$ a longest simple path from r to t (should be $r \rightarrow q \rightarrow s \rightarrow t$).



- However, when we combine the longest simple path $q \rightarrow s \rightarrow t \rightarrow r$ and $r \rightarrow q \rightarrow s \rightarrow t$, we get $q \rightarrow s \rightarrow t \rightarrow r \rightarrow q \rightarrow s \rightarrow t$ which is not simple.

Overlapping Subproblems

- Second ingredient: an optimization problem must have for DP is that the space of subproblems must be "small", in a sense that
  - A recursive algorithm solves the same subproblems over and over, rather than generating new subproblems.
  - The total number of distinct subproblems is polynomial in the input size
  - DP algorithms use a table to store the solutions to subproblems and look up the table in a constant time

Overlapping Subproblems (Cont'd)

- In contrast, a problem for which a divide-and-conquer approach is suitable when the recursive steps always generate new problems at each step of the recursion.
- Examples: Mergesort and Quicksort.
  - Sorting on smaller and smaller arrays (each recursion step work on a different subarray)

A Recursive Algorithm for Matrix-Chain Multiplication

RECURSIVE-MATRIX-CHAIN(p,i,j), called with($p,1,n$)

1. **if** ($i == j$) **then return** 0

2. $m[i,j] = \infty$

3. **for** $k = i$ to (j-1)

4. $q =$ RECURSIVE-MATRIX-CHAIN(p,i,k)

+ RECURSIVE-MATRIX-CHAIN($p,k+1,j$) + $p_{i-1}p_k p_j$

5. **if** ($q < m[i,j]$) **then** $m[i,j] = q$

6. **return** $m[i,j]$

The running time of the algorithm is $O(2^n)$.

The recursion tree

for k= i to (j-1)

q = RECURSIVE-MATRIX-CHAIN(p,i,k)

+ RECURSIVE-MATRIX-CHAIN(p,k+1,j) + $p_{i-1}p_kp_j$

RECURSIVE-MATRIX-CHAIN(p,1,4)

i =1, j = 4, k = 1, 2, 3 (i to j-1)

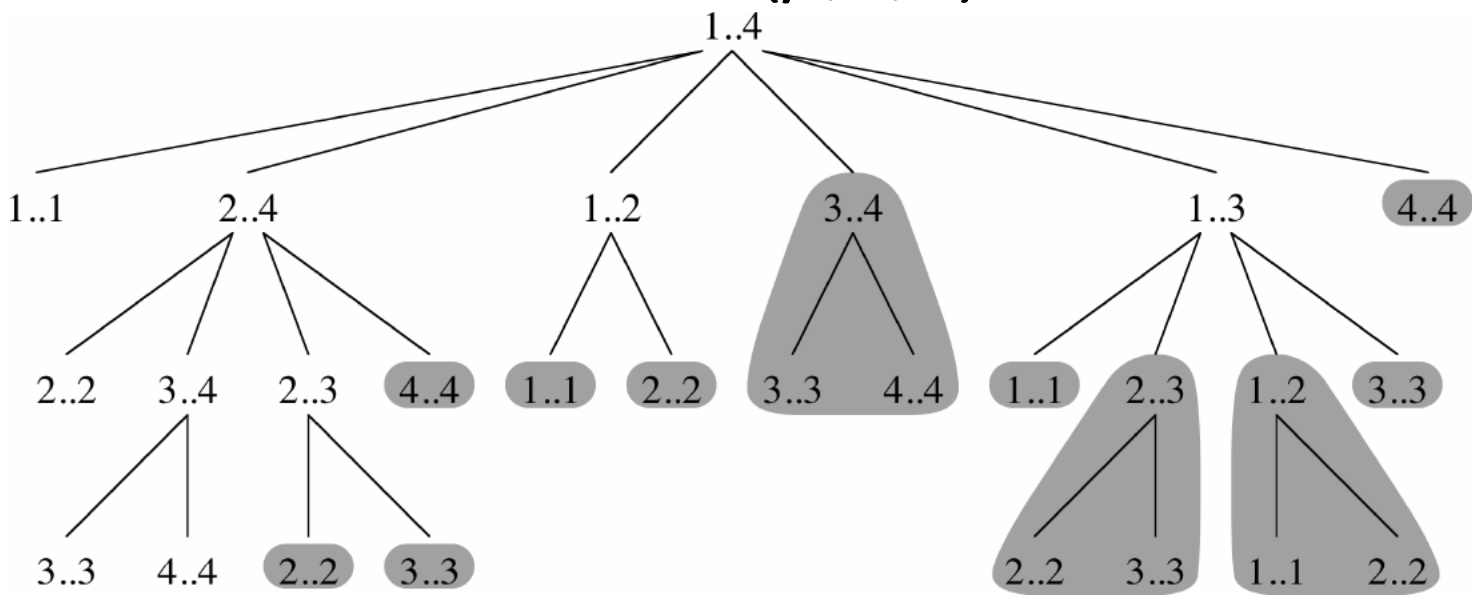
needs to solve (1, k) (k+1, 4)

k = 1 - > (1, 1) (2, 4)

k = 2 - > (1, 2) (3, 4)

K = 3 -> (1, 3) (4, 4)

Recursion tree of RECURSIVE-MATRIX-CHAIN(p,1,4)



- This divide-and-conquer recursive algorithm solves the overlapping problems over and over.
  - DP solves the same subproblems only once
  - The computations in darker color are replaced by table look up in MEMOIZED-MATRIX-CHAIN(p,1,4).
- The divide-and-conquer is better for the problem which generates brand-new problems at each step of recursion.

General idea of Memoization

- A variation of DP
- Keep the same efficiency as DP
- But in a top-down manner.
- Idea:
  - When a subproblem is first encountered, its solution needs to be solved, and then is stored in the corresponding entry of the table.
  - If the subproblem is encountered again in the future, just look up the table to take the value.

Memoized Matrix Chain

MEMOIZED-MATRIX-CHAIN(p)

```
n ← length[p] − 1
for i ← 1 to n
      do for j ← i to n
              do m[i, j] ← ∞
return LOOKUP-CHAIN(p, 1, n)
```

LOOKUP-CHAIN(p,i,j)

1. **if** $m[i,j] < \infty$ **then return** $m[i,j]$
2. **if** $(i == j)$ **then** $m[i,j] = 0$
3. **else for** $k = i$ to $j-1$
4. q=LOOKUP-CHAIN(p,i,k)+
5. LOOKUP-CHAIN($p,k+1,j$) + $p_{i-1}p_kp_j$
6. **if** $(q < m[i,j])$ **then** $m[i,j] = q$
7. **return** $m[i,j]$

DP VS. Memoization

- MCM can be solved by DP or Memoized algorithm, both in $O(n^3)$
  - Total $\Theta(n^2)$ subproblems, with $O(n)$ for each.
- If all subproblems must be solved at least once, DP is better by a constant factor due to no recursive involvement as in memorized algorithm
- If some subproblems may not need to be solved, Memoized algorithm may be more efficient
  - since it only solve these subproblems which are definitely required.

Longest Common Subsequence (LCS)

- DNA analysis to compare two DNA strings
- DNA string: a sequence of symbols A,C,G,T
  - S =ACCGGTCGAGCTTCGAAT
- Subsequence of X is X with some symbols left out
  - Z =CGTC is a subsequence of X =ACGCTAC
- Common subsequence Z of X and Y: a subsequence of X and also a subsequence of Y
  - Z =CGA is a common subsequence of X =ACGCTAC and Y =CTGACA
- Longest Common Subsequence (LCS): the longest one of common subsequences
  - Z' =CGCA is the LCS of the above X and Y
- LCS problem: given $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, find their LCS

LCS Intuitive Solution – brute force

- LCS problem: given $X = <x_1, x_2,..., x_m>$ and $Y = <y_1, y_2,..., y_n>$, find their LCS
- List all possible subsequences of X, check whether they are also subsequences of Y, keep the longer one each time.
- What is the run-time complexity?
- Each subsequence corresponds to a subset of the indices $\{1,2,…,m\}$, there are 2^m

LCS DP step 1: Optimal Substructure

- Characterize optimal substructure of LCS
- Theorem 15.1: $X_m = <x_1, x_2, \ldots, x_m>$, $Y_n = <y_1, y_2, \ldots, y_n>$, let $Z_k = <z_1, z_2, \ldots, z_k>$ be any LCS of X_m and Y_n
  1. if $x_m == y_n$, then $z_k = x_m = y_n$, and Z_{k-1} is the LCS of X_{m-1} and Y_{n-1}
  2. if $x_m \neq y_n$, then $z_k \neq x_m$ implies Z_k is the LCS of X_{m-1} and Y_n
  3. if $x_m \neq y_n$, then $z_k \neq y_n$ implies Z_k is the LCS of X_m and Y_{n-1}

Optimal Substructure

- if $x_m \neq y_n$, we have four cases if $x_m \neq y_n$
- $z_k \neq x_m$ and $z_k \neq y_n$
- $z_k \neq x_m$ and $z_k = y_n$
- $z_k \neq y_n$ and $z_k \neq x_m$
- $z_k \neq y_n$ and $z_k = x_m$
- The four cases can be reduced to two cases

$\boxed{z_k \neq y_n}$

$\boxed{z_k \neq x_m}$

LCS DP step 2: Recursive Solution

- What the theorem says:
  - If $x_m == y_{n,}$ find LCS of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$, then append x_m
  - If $x_m \neq y_{n,}$ find (1) the LCS of $X_{m\text{-}1}$ and Y_n and (2) the LCS of X_m and $Y_{n\text{-}1}$; then, take which one is longer
- Overlapping substructure:
  - Both LCS of $X_{m\text{-}1}$ and Y_n and LCS of X_m and $Y_{n\text{-}1}$ will need to solve LCS of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$ first
- $c[i,j]$ is the length of LCS of X_i and Y_j

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0, \text{ or } j = 0 \\ c[i\text{-}1, j\text{-}1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{ c[i\text{-}1, j], c[i, j\text{-}1] \} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

LCS DP step 3: Computing the Length of LCS

$$c[i,j] = \begin{cases} 0 & \text{if } i=0, \text{ or } j=0 \\ c[i\text{-}1, j\text{-}1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{ c[i\text{-}1,j], c[i,j\text{-}1] \} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

- *c*[0..*m*, 0..*n*], where *c*[*i*,*j*] is defined as above.
  - *c*[*m*,*n*] is the answer (length of LCS)
- *b*[1..*m*, 1..*n*], where *b*[*i*,*j*] points to the table entry corresponding to the optimal subproblem solution chosen when computing *c*[*i*,*j*].
  - From *b*[*m*, *n*] backward to find the LCS.

LCS DP Algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if x_i = y_j
            then c[i, j] ← c[i − 1, j − 1] + 1
                b[i, j] ← "↖"
            else if c[i − 1, j] ≥ c[i, j − 1]
                then 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
```

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

O(m*n), since each c[i,j] is calculated in constant time, and there are m*n elements in the array

LCS Example

We'll see how LCS algorithm works on the following example: X = ABCB Y = BDCAB. What is the Longest Common Subsequence of X and Y?

LCS(X, Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

ABCB
BDCAB

LCS Example (0)

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,6]

LCS Example (1)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for i = 1 to m   c[i,0] = 0
for j = 1 to n   c[0,j] = 0

LCS Example (2)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**				
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if (Xi == Yj)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (3)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (Xi == Yj)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (4)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (Xi == Yj)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (5)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1** →	**1**
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (Xi == Yj)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (6)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**				
3	**C**	**0**					
4	**B**	**0**					

if (Xi == Yj)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (7)

ABCB
BDCAB

j		0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	
3	**C**	**0**					
4	**B**	**0**					

if (Xi == Yj)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (8)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**					
4	**B**	**0**					

if (Xi == Yj)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (10)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**			
4	**B**	**0**					

if (Xi == Yj)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (11)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**		
4	**B**	**0**					

if (Xi == Yj)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (12)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**					

if (Xi == Yj)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (13)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**				

if (Xi == Yj)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (14)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	

if (Xi == Yj)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (15)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

if (Xi == Yj)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

How to find actual LCS

- So far, we have just found the *length* of LCS, but not LCS itself.
- We want to modify this algorithm to make it output Longest Common Subsequence of X and Y
- Each *c[i,j]* depends on *c[i-1,j]* and *c[i,j-1]* or *c[i-1, j-1].*
- For each c[i,j], we can say how it was acquired:

2	2
2	3

For example, here
c[i,j] = c[i-1,j-1] +1 = 2+1=3

How to find actual LCS

- Remember that

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

  - So we can start from *c[m,n]* and go backwards
  - Whenever *c[i,j] = c[i-1, j-1]+1*, remember *x[i]* (because *x[i]* is a part of LCS)
  - When i=0 or j=0 (i.e. we reached the beginning), output remembered letters in reverse order

Finding LCS

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

LCS: **B C B**

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

Figure 15.8 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner; the path is shaded. Each "↖" on the path corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

Summary

- Dynamic programming and where it can be applied
  - Optimal substructure
  - Overlapping subproblems
  - Four steps to construct a DP AL

Greedy Algorithms

- We have learned two design techniques
  - Divide-and-conquer
  - Dynamic Programming
- Now, the third → Greedy Algorithms
  - Optimization often goes through some choices
  - Make local best choices → hope to achieve global optimization
  - Many times, this works; Other times, does NOT!
    - Minimum spanning tree algorithms
  - We must carefully examine if we can apply this method

An activity-selection problem

- Activity set $S = \{a_1, a_2, ..., a_n\}$
- n activities wish to use a single resource
- Each activity a_i has a **start time** s_i and a **finish time** f_i, where $0 \leq s_i < f_i < \infty$
- If selected, activity a_i take place during the half-open time interval $[s_i, f_i)$
- Activities a_i and a_j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap
  - a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$

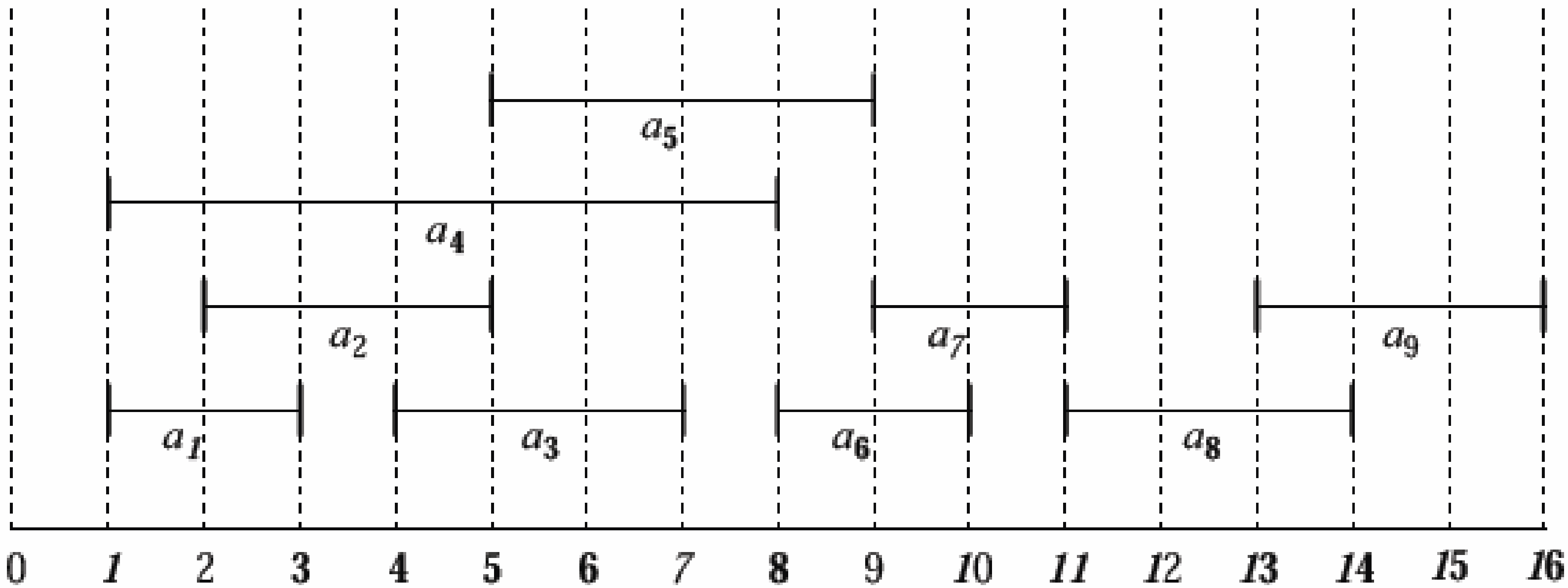
Activity-selection Problem

- To select a maximum-size subset of mutually compatible activities
- Activities sorted in finishing times, e.g.,

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- $\{a_3, a_9, a_{11}\}$ works, but it is not the Max set
- What is the MAX set? How do we get it?
  - $\{a_2, a_4, a_9, a_{11}\}$

Another example

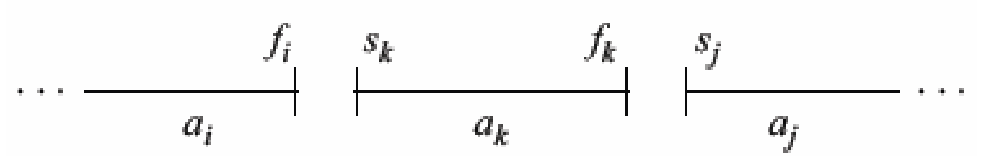


Solving Activity-selection Problem in different methods

- 1st, a dynamic programming solution: we combine optimal solutions to two subproblems to form an optimal solution to the original problem
  - Select one activity, divide the set into two subsets
  - We have n choices; two subproblems: k and (n-k-1)
- 2nd, we then observe that we need only consider one choice – the greedy choice
  - this greedy choice guarantee that one of the subproblems is empty → so that only one nonempty subproblem remains.

Optimal substructure

- $S_{i\,j}$ is the subset of activities that can
  - start after activity a_i finishes
  - and finish before activity a_j starts
  - $S_{i\,j} = \{ a_k \in S: f_i \leq s_k < f_k \leq s_j \}$
  - $f_0 = 0$ and $s_{n+1} = \infty$. Then $S = S_{0,n+1}$, and the ranges for i and j are given by $0 \leq i, j \leq n+1$
- Define A_{ij} as the maximum compatible set in S_{ij}
  - Selecting a_k in the optimal solutions generates two subproblems
  - $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$
  - $|A_{ij}| = |A_{ik}| + 1 + |A_{kj}|$



A recursive solution

- Define A_{ij} as the maximum compatible set in S_{ij}
  - Selecting a_k in the optimal solutions generates two subproblems
  - $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$
- C[i, j] denoted the size of optimal solution for S_{ij}

$$c[i, j] = c[i, k] + c[k, j] + 1$$

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\limits_{i<k<j}\{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

Can we do better?

- Can we solve the problem without solving all the subproblems?
- Intuition: Choose an activity that leaves the resource available for as many other activities as possible
- It must finish as early as possible: greedy

The greedy choice

- Let S_k =$\{a_i \in S : s_i >= f_k\}$ be the set of activities that start after activity a_k finishes
- If we make the greedy choice of activity a_1 (i.e., a_1 is the first activity to finish), then S_1 remains as the only subproblem to solve.
  - Let A_1 be the maximum-size subset of mutually compatible activities in S_1
  - $\mathbf{a_1} + \mathbf{A_1}$ must be the maximum compatible set for S
  - Is this correct?

Converting a dynamic-programming solution to a greedy solution

- **Theorem 16.1** Consider any nonempty subproblem S_k, and let a_m be the activity in S_k with the earliest finish time: $f_m = \min \{ f_x : a_x \in S_k \}$. Then a_m is used in some maximum-size subset of mutually compatible activities of S_k
- Let A_k be the maximum-size subset of mutually compatible activities in S_k
- Let a_j be the activity in A_k with the earliest finish time
- If $a_j == a_m$, we are done.
- Otherwise, $A'_k = A_k - \{a_j\} \cup \{a_m\}$
- We have new A_k with a_m

A recursive top-down greedy algorithm

RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)

// s[] are start times, f[] are finish times

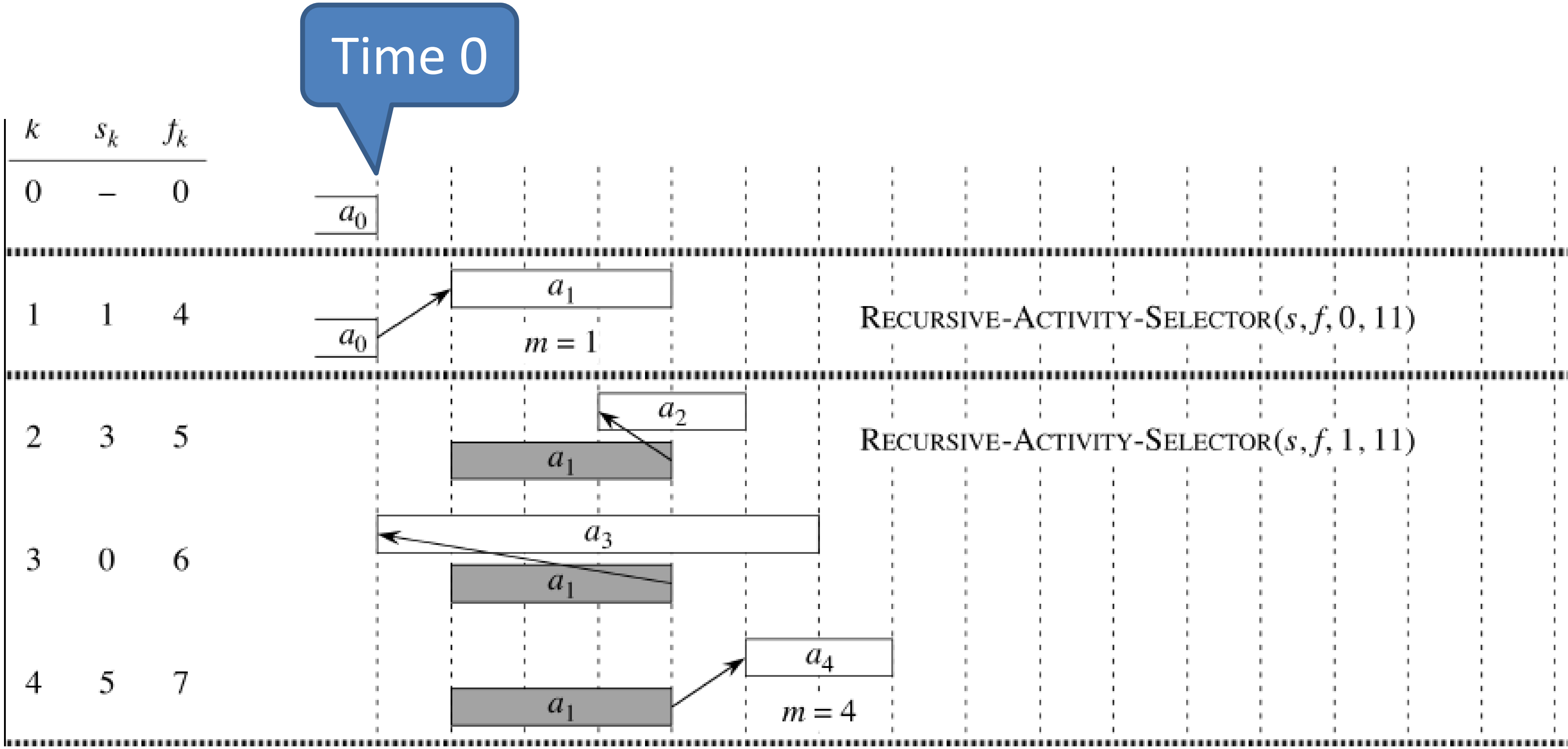
// k is the current subproblem index

// n is the original problem size

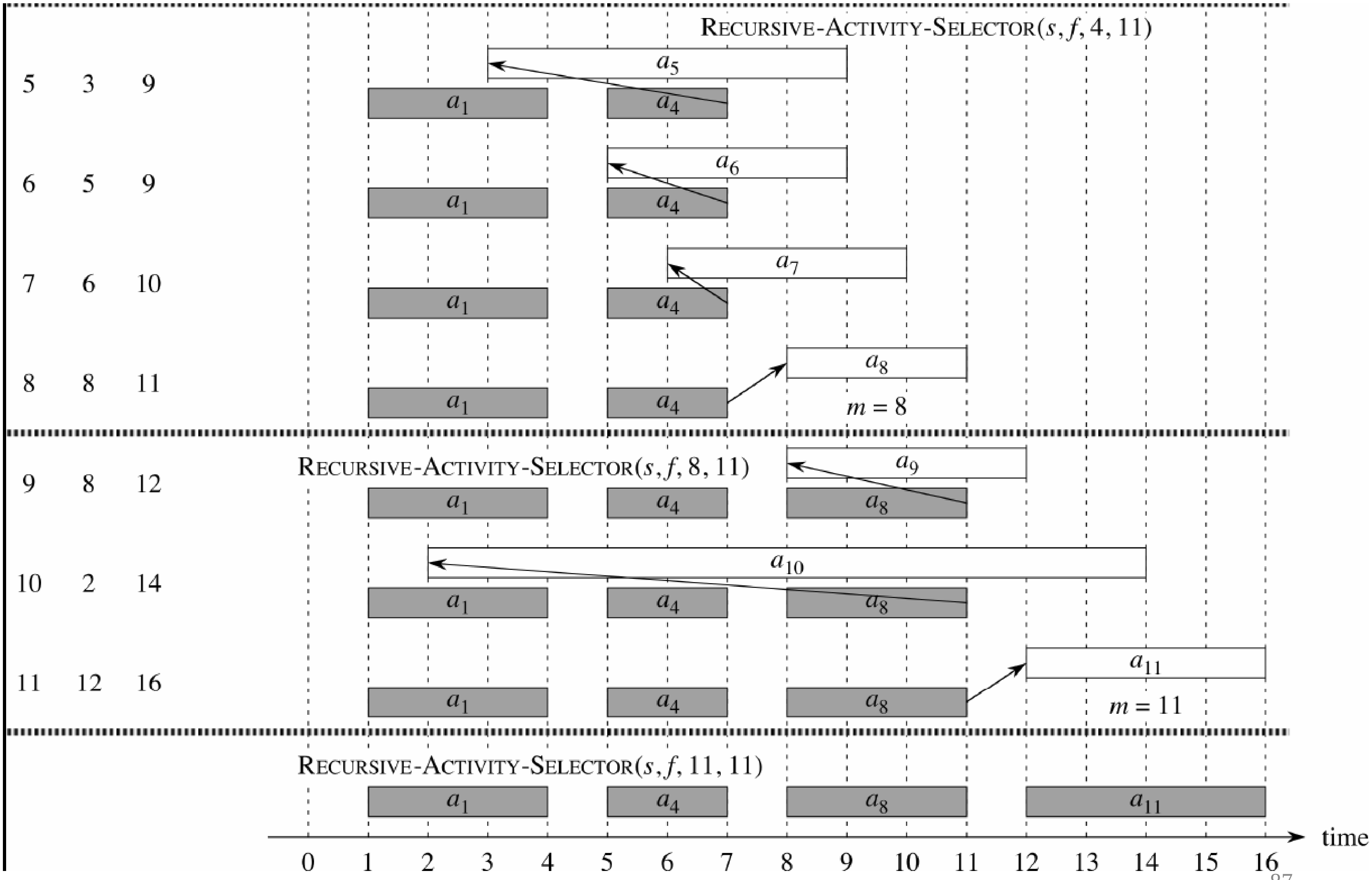
```
m = k + 1
while m ≤ n and s_m < f_k    //Find the first activity in S_k
      m = m + 1              // right after a_m
if m ≤ n
      return {a_m} ∪ RECURSIVE-ACTIVITY-
                      SELECTOR (s, f, m, n)
else return ø         // m==n, no better results
```

Initial call

REC-ACTIVITY-SELECTOR$(s, f, 0, n)$.



i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14


RECURSIVE-ACTIVITY-SELECTOR(s, f, 4, 11)
5 3 9
6 5 9
7 6 10
8 8 11
m = 8
RECURSIVE-ACTIVITY-SELECTOR(s, f, 8, 11)
9 8 12
10 2 14
11 12 16
m = 11
RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 11)
time
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

An example

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- {a1, a4, a8, a11}

An iterative greedy algorithm

GREEDY-ACTIVITY-SELECTOR(s, f)

n = s.*length*

$A = \{a_1\}$

k = 1

for $m = 2$ to n

    if $s_m \geq f_k$

        then $A = A \cup \{a_m\}$

            $k = m$

return A

16.2 Elements of the greedy strategy

- We need make a choice at each step: local best→ greedy choice
- Common Steps for greedy Als
  1. Determine the optimal substructure of the problem.
  2. Develop a recursive solution
  3. Show that if we make the greedy choice, then only one subproblem remains; others are empty
  4. Prove that one of the optimal choices is the greedy choice at any stage of the recursion. Thus, it is always safe to make the greedy choice.
  5. Develop a recursive algorithm to implement it
  6. Convert the recursive algorithm to an iterative one

Designing a greedy algorithm

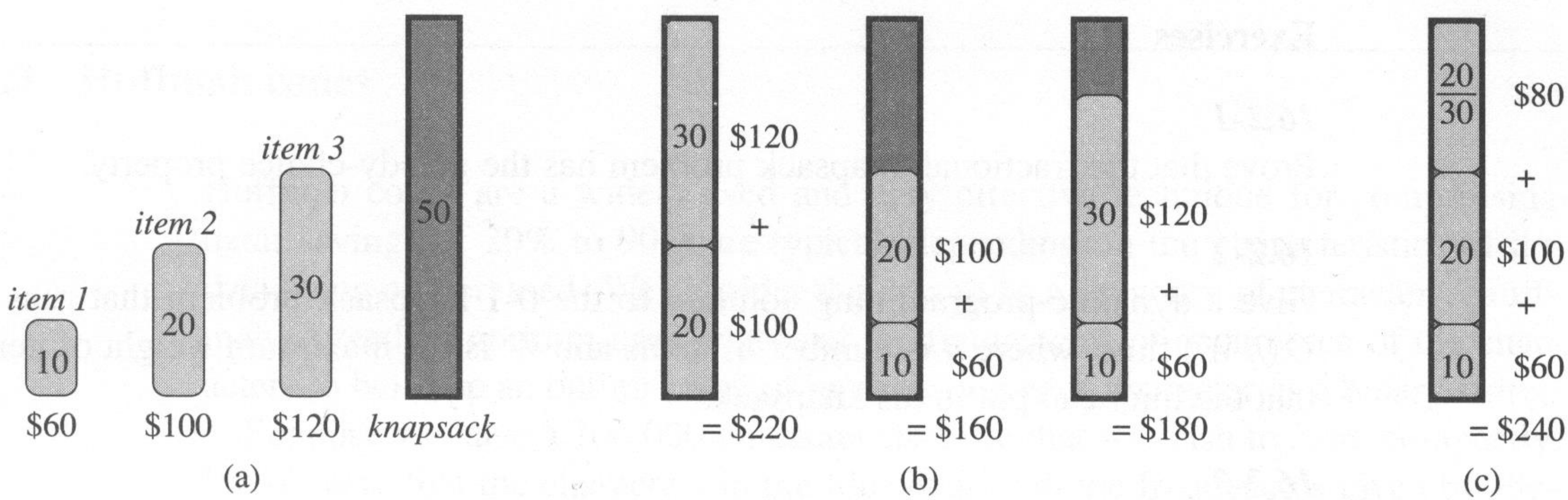
1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem, we arrive at an optimal solution to the original problem.

knapsack problem

- knapsack problem
  - There are n items
  - The i-th item has value v_i and weight w_i
  - A thief only can carry W pounds
  - Which items should he take?
- 0-1 knapsack problem: take one item or not
- fractional knapsack problem: take fractions
  - Greedy choice: max value v_i / w_i

The greedy strategy does not work for the 0-1 knapsack

- Per unit value: item 1, $6, item 2, $5, item 3, $4
- Greedy choice will be Item 1



Fractional knapsack problem

FRACTIONAL-KNAPSACK(v, w, W)

load $=0$
$i=1$
while **load** $< W$ **and** $i \le n$
    if $w_i \le W -$ **load**
        take all of item i
    else **take** $(W -$ **load**$)/w_i$ **of item** i
    add what was **taken to load**
    $i = i + 1$

Summary: ingredients of greedy ALs

- **Greedy-choice property:** A global optimal solution can be achieved by making a local optimal choice.
  - Without considering results of subproblems
- **Optimal substructure:** An optimal solution to the problem within its optimal solution to subproblem