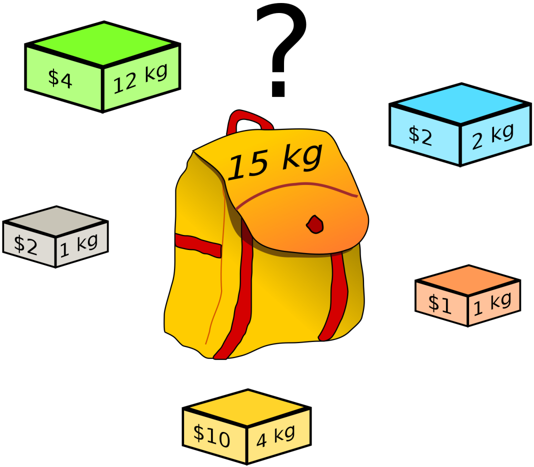
Technical Slide

Module 6: Dynamic Programming ||

Knapsack Problem



Goal

Maximize value ($) while limiting total weight (kg)

Applications

- Classical problem in combinatorial optimization with applications in resource allocation, cryptography, planning

Applications

- Classical problem in combinatorial optimization with applications in resource allocation, cryptography, planning
- Weights and values may mean various resources (to be maximized or limited):

Applications

- Classical problem in combinatorial optimization with applications in resource allocation, cryptography, planning
- Weights and values may mean various resources (to be maximized or limited):
    - Select a set of TV commercials (each commercial has duration and cost) so that the total revenue is maximal while the total length does not exceed the length of the available time slot

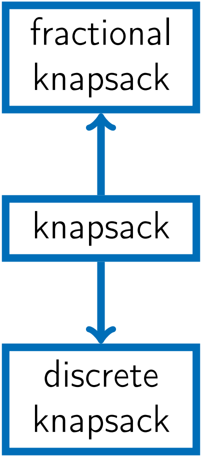
Applications

- Classical problem in combinatorial optimization with applications in resource allocation, cryptography, planning
- Weights and values may mean various resources (to be maximized or limited):
    - Select a set of TV commercials (each commercial has duration and cost) so that the total revenue is maximal while the total length does not exceed the length of the available time slot
    - Purchase computers for a data center to achieve the maximal performance under limited budget

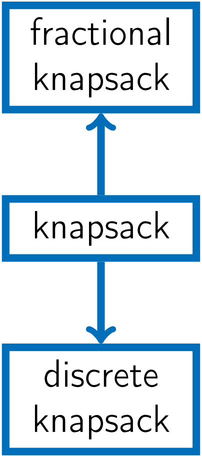
Problem Variations

knapsack

Problem Variations



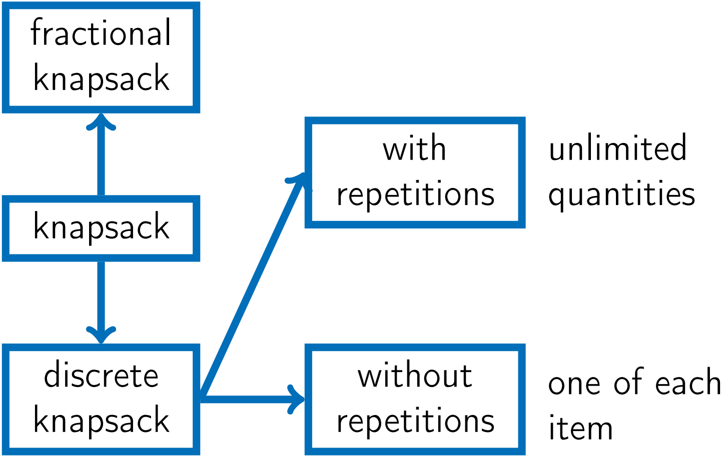
Problem Variations



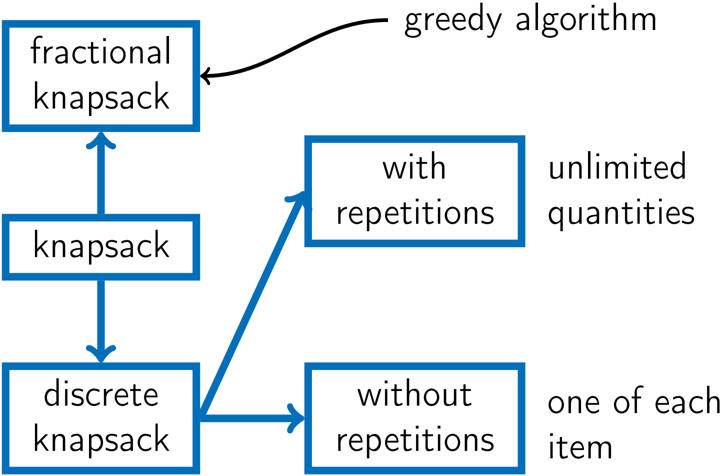
can take fractions of items

each item is either taken or not

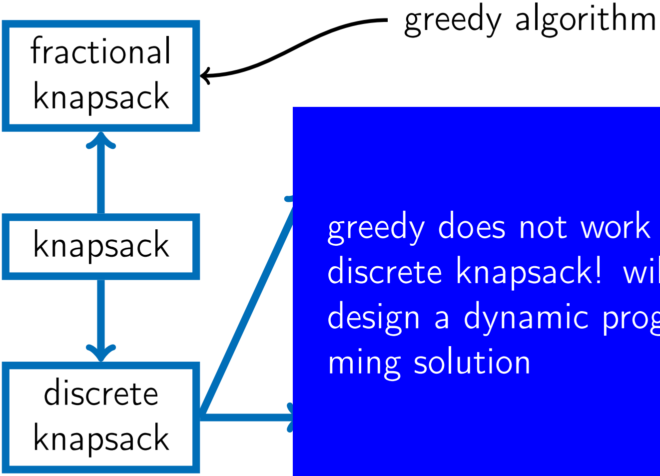
Problem Variations



Problem Variations

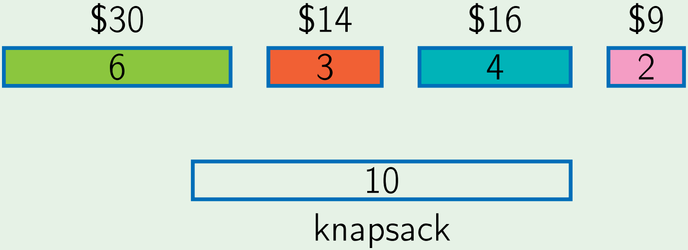


Problem Variations

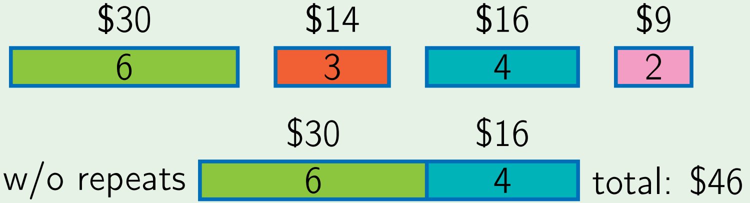


greedy does not work for discrete knapsack! will design a dynamic programming solution

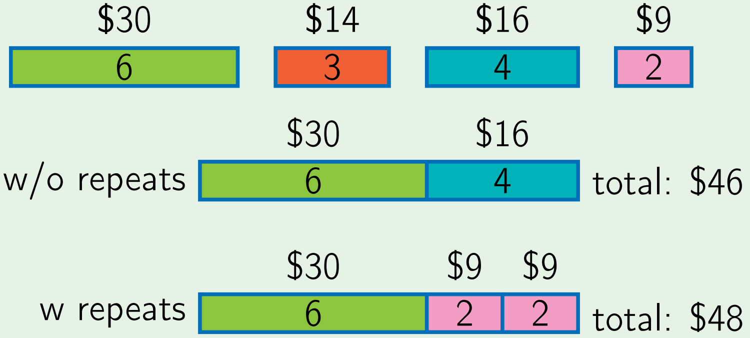
Example



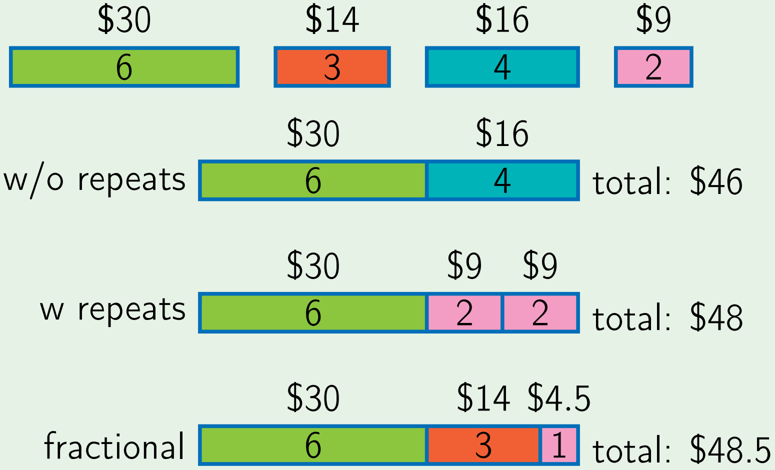
Example



Example

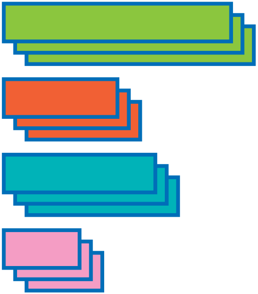

$30
6
$14
3
$16
4
$9
2
w/o repeats
$30
6
$16
4
total: $46
w repeats
$30
6
$9
2
$9
2
total: $48

Example

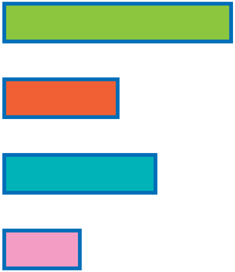


$30
$14
$16
$9
6
3
4
2
w/o repeats
$30
$16
6
4
total: $46
w repeats
$30
$9
$9
6
2
2
total: $48
fractional
$30
$14
$4.5
6
3
1
total: $48.5

With repetitions:
unlimited quantities



Without repetitions:
one of each item



Knapsack with repetitions problem

Input: Weights $w_0, \ldots, w_{n-1}$ and values $v_0, \ldots, v_{n-1}$ of n items; total weight W (v_i's, w_i's, and W are non-negative integers).

Output: The maximum value of items whose weight does not exceed W. Each item can be used any number of times.

Analyzing an Optimal Solution

- Consider an optimal solution and an item in it:

Analyzing an Optimal Solution

- Consider an optimal solution and an item in it:

- If we take this item out then we get an optimal solution for a knapsack of total weight $W - w_i$.

Subproblems

- Let *value*(u) be the maximum value of knapsack of weight u

Subproblems

- Let *value*(u) be the maximum value of knapsack of weight u
-

$$value(u) = \max_{i \colon w_i \leq w} \{value(u - w_i) + v_i\}$$

Subproblems

- Let $value(u)$ be the maximum value of knapsack of weight u
-

$$value(u) = \max_{i\colon w_i \leq w} \{value(u - w_i) + v_i\}$$

- Base case: $value(0) = 0$

Subproblems

- Let *value*(u) be the maximum value of knapsack of weight u
- $$value(u) = \max_{i\colon w_i \le w} \{value(u - w_i) + v_i\}$$
- Base case: $value(0) = 0$
- This recurrence relation is transformed into a recursive algorithm in a straightforward way

Recursive Algorithm

```
T = dict()

def knapsack(w, v, u):
  if u not in T:
    T[u] = 0

    for i in range(len(w)):
      if w[i] <= u:
        T[u] = max(T[u],
          knapsack(w, v, u - w[i]) + v[i])

  return T[u]


print(knapsack(w=[6, 3, 4, 2],
               v=[30, 14, 16, 9], u=10))
```

Recursive into Iterative

- As usual, one can transform a recursive algorithm into an iterative one

Recursive into Iterative

- As usual, one can transform a recursive algorithm into an iterative one
- For this, we gradually fill in an array T:
  $T[u] = value(u)$

Recursive Algorithm

```
def knapsack(W, w, v):
    T = [0] * (W + 1)

    for u in range(1, W + 1):
        for i in range(len(w)):
            if w[i] <= u:
                T[u] = max(T[u], T[u - w[i]] + v[i])

    return T[W]


print(knapsack(W=10, w=[6, 3, 4, 2],
               v=[30, 14, 16, 9]))
```

Example: $W = 10$

\$30	\$14	\$16	\$9
6	3	4	2

0	1	2	3	4	5	6	7	8	9	10
0	0	9	14	18	23	30	32	39	44	

Example: $W = 10$

\$30	\$14	\$16	\$9
6	3	4	2

0	1	2	3	4	5	6	7	8	9	10
0	0	9	14	18	23	30	32	39	44	
				+30		+16	+14	+9		

Example: $W = 10$

\$30	\$14	\$16	\$9
6	3	4	2

0	1	2	3	4	5	6	7	8	9	10
0	0	9	14	18	23	30	32	39	44	48
				+30		+16	+14	+9		

Subproblems Revisited

- Another way of arriving at subproblems: optimizing brute force solution

Subproblems Revisited

- Another way of arriving at subproblems: optimizing brute force solution
- Populate a list of used items one by one

Brute Force: Knapsack with Repetitions

```python
def knapsack(W, w, v, items):
    weight = sum(w[i] for i in items)
    value = sum(v[i] for i in items)

    for i in range(len(w)):
        if weight + w[i] <= W:
            value = max(value,
                        knapsack(W, w, v, items + [i]))

    return value

print(knapsack(W=10, w=[6, 3, 4, 2],
               v=[30, 14, 16, 9], items=[]))
```

Subproblems

- It remains to notice that the only important thing for extending the current set of items is the weight of this set

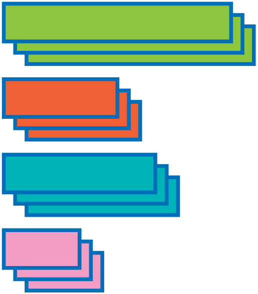
Subproblems

- It remains to notice that the only important thing for extending the current set of items is the weight of this set
- One then replaces `items` by their weight in the list of parameters

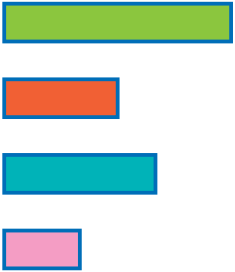
Technical Slide

Module 6: Dynamic Programming II

With repetitions:
unlimited quantities



Without repetitions:
one of each item

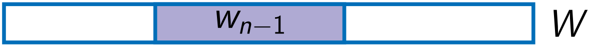


Knapsack without repetitions problem

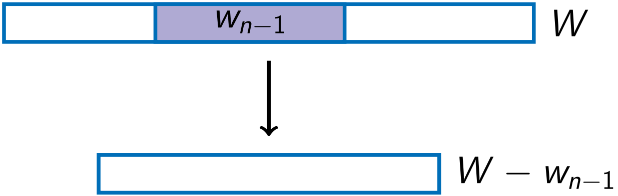
Input: Weights $w_0, \ldots, w_{n-1}$ and values $v_0, \ldots, v_{n-1}$ of n items; total weight W (v_i's, w_i's, and W are non-negative integers).

Output: The maximum value of items whose weight does not exceed W. Each item can be used at most once.

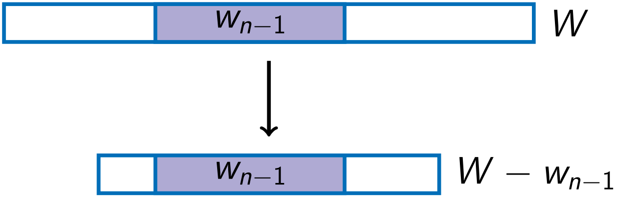
Same Subproblems?



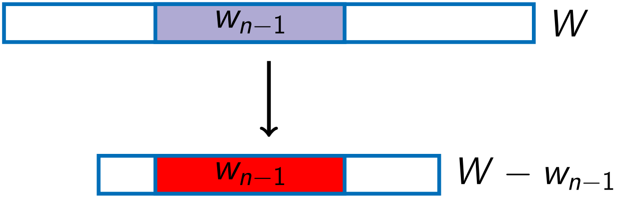
Same Subproblems?



Same Subproblems?



Same Subproblems?



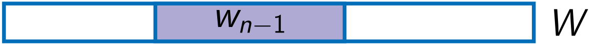
Subproblems

- If the last item is taken into an optimal solution:

then what is left is an optimal solution for a knapsack of total weight $W - w_{n-1}$ using items $0, 1, \ldots, n-2$.

Subproblems

- If the last item is taken into an optimal solution:



  then what is left is an optimal solution for a knapsack of total weight $W - w_{n-1}$ using items $0, 1, \ldots, n-2$.

- If the last item is not used, then the whole knapsack must be filled in optimally with items $0, 1, \ldots, n-2$.

Subproblems

- For $0 \leq u \leq W$ and $0 \leq i \leq n$, $value(u, i)$ is the maximum value achievable using a knapsack of weight u and the first i items.

Subproblems

- For $0 \leq u \leq W$ and $0 \leq i \leq n$, $value(u, i)$ is the maximum value achievable using a knapsack of weight u and the first i items.
- Base case: $value(u, 0) = 0$, $value(0, i) = 0$

Subproblems

- For $0 \leq u \leq W$ and $0 \leq i \leq n$, $value(u, i)$ is the maximum value achievable using a knapsack of weight u and the first i items.
- Base case: $value(u, 0) = 0$, $value(0, i) = 0$
- For $i > 0$, the item $i - 1$ is either used or not: $value(u, i)$ is equal to

  $$\max\{value(u - w_{i-1}, i - 1) + v_{i-1}, value(u, i - 1)\}$$

Recursive Algorithm

```
T = dict()

def knapsack(w, v, u, i):
  if (u, i) not in T:
    if i == 0:
      T[u, i] = 0
    else:
      T[u, i] = knapsack(w, v, u, i - 1)
      if u >= w[i - 1]:
        T[u, i] = max(T[u, i],
          knapsack(w, v, u - w[i - 1], i - 1) + v[i - 1])

  return T[u, i]


print(knapsack(w=[6, 3, 4, 2],
               v=[30, 14, 16, 9], u=10, i=4))
```

Iterative Algorithm

```python
def knapsack(W, w, v):
  T = [[None] * (len(w) + 1) for _ in range(W + 1)]

  for u in range(W + 1):
    T[u][0] = 0

  for i in range(1, len(w) + 1):
    for u in range(W + 1):
      T[u][i] = T[u][i - 1]
      if u >= w[i - 1]:
        T[u][i] = max(T[u][i],
          T[u - w[i - 1]][i - 1] + v[i - 1])

  return T[W][len(w)]


print(knapsack(W=10, w=[6, 3, 4, 2],
               v=[30, 14, 16, 9]))
```

Analysis

- Running time: $O(nW)$

Analysis

- Running time: $O(nW)$
- Space: $O(nW)$

Analysis

- Running time: $O(nW)$
- Space: $O(nW)$
- Space can be improved to $O(W)$ in the iterative version: instead of storing the whole table, store the current column and the previous one

Reconstructing a Solution

- As it usually happens, an optimal solution can be unwound by analyzing the computed solutions to subproblems

Reconstructing a Solution

- As it usually happens, an optimal solution can be unwound by analyzing the computed solutions to subproblems
- Start with $u = W$, $i = n$

Reconstructing a Solution

- As it usually happens, an optimal solution can be unwound by analyzing the computed solutions to subproblems
- Start with $u = W$, $i = n$
- If $value(u, i) = value(u, i - 1)$, then item $i - 1$ is not taken. Update i to $i - 1$

Reconstructing a Solution

- As it usually happens, an optimal solution can be unwound by analyzing the computed solutions to subproblems
- Start with $u = W$, $i = n$
- If $value(u, i) = value(u, i-1)$, then item $i-1$ is not taken. Update i to $i-1$
- Otherwise $value(u, i) = value(u - w_{i-1}, i-1) + v_{i-1}$ and the item $i - i$ is taken. Update i to $i-1$ and u to $u - w_{i-1}$

Subproblems Revisited

- How to implement a brute force solution for the knapsack without repetitions problem?

Subproblems Revisited

- How to implement a brute force solution for the knapsack without repetitions problem?
- Process items one by one. For each item, either take into a bag or not

```python
def knapsack(W, w, v, items, last):
  weight = sum(w[i] for i in items)

  if last == len(w) - 1:
    return sum(v[i] for i in items)

  value = knapsack(W, w, v, items, last + 1)
  if weight + w[last + 1] <= W:
    items.append(last + 1)
    value = max(value,
                knapsack(W, w, v, items, last + 1))
    items.pop()

  return value

print(knapsack(W=10, w=[6, 3, 4, 2],
               v=[30, 14, 16, 9],
               items=[], last=-1))
```

Technical Slide

Module 6: Dynamic Programming II

Recursive vs Iterative

- If all subproblems must be solved then an iterative algorithm is usually faster since it has no recursion overhead

Recursive vs Iterative

- If all subproblems must be solved then an iterative algorithm is usually faster since it has no recursion overhead
- There are cases however when one does not need to solve all subproblems and the knapsack problem is a good example: assume that W and all w_i's are multiples of 100; then $value(w)$ is not needed if w is not divisible by 100

Polynomial Time?

- The running time $O(nW)$ is not polynomial since the input size is proportional to $\log W$, but not W

Polynomial Time?

- The running time $O(nW)$ is not polynomial since the input size is proportional to $\log W$, but not W
- In other words, the running time is $O(n2^{\log W})$.

Polynomial Time?

- The running time $O(nW)$ is not polynomial since the input size is proportional to $\log W$, but not W
- In other words, the running time is $O(n2^{\log W})$.
- E.g., for

$$W = 10\,345\,970\,345\,617\,824\,751$$

(twenty digits only!) the algorithm needs roughly 10^{20} basic operations

Polynomial Time?

- The running time $O(nW)$ is not polynomial since the input size is proportional to $\log W$, but not W
- In other words, the running time is $O(n2^{\log W})$.
- E.g., for

$$W = 10\,345\,970\,345\,617\,824\,751$$

(twenty digits only!) the algorithm needs roughly 10^{20} basic operations
- Solving the knapsack problem in truly polynomial time is the essence of the P vs NP problem, the most important open problem in Computer Science (with a bounty of \$1M)

Technical Slide

Module 6: Dynamic Programming ||

Chain matrix multiplication

Input: Chain of n matrices $A_0, \ldots, A_{n-1}$ to be multiplied.

Output: An order of multiplication minimizing the total cost of multiplication.

Clarifications

- Denote the sizes of matrices $A_0, \ldots, A_{n-1}$ by

  $$m_0 \times m_1, m_1 \times m_2, \ldots, m_{n-1} \times m_n$$

  respectively. I.e., the size of A_i is $m_i \times m_{i+1}$

Clarifications

- Denote the sizes of matrices $A_0, \ldots, A_{n-1}$ by

$$m_0 \times m_1, m_1 \times m_2, \ldots, m_{n-1} \times m_n$$

  respectively. I.e., the size of A_i is $m_i \times m_{i+1}$
- Matrix multiplication is not commutative (in general, $A \times B \neq B \times A$), but it is associative: $A \times (B \times C) = (A \times B) \times C$

Clarifications

- Denote the sizes of matrices $A_0, \ldots, A_{n-1}$ by

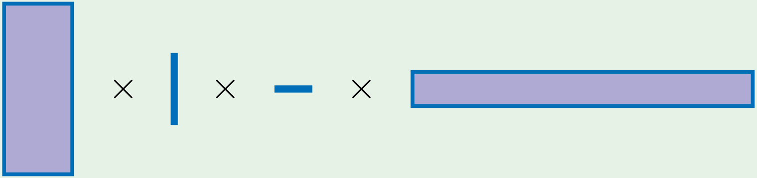
$$m_0 \times m_1, m_1 \times m_2, \ldots, m_{n-1} \times m_n$$

  respectively. I.e., the size of A_i is $m_i \times m_{i+1}$
- Matrix multiplication is not commutative (in general, $A \times B \neq B \times A$), but it is associative: $A \times (B \times C) = (A \times B) \times C$
- Thus $A \times B \times C \times D$ can be computed, e.g., as $(A \times B) \times (C \times D)$ or $(A \times (B \times C)) \times D$

Clarifications

- Denote the sizes of matrices $A_0, \dots, A_{n-1}$ by
$$m_0 \times m_1, m_1 \times m_2, \dots, m_{n-1} \times m_n$$
respectively. I.e., the size of A_i is $m_i \times m_{i+1}$
- Matrix multiplication is not commutative (in general, $A \times B \neq B \times A$), but it is associative: $A \times (B \times C) = (A \times B) \times C$
- Thus $A \times B \times C \times D$ can be computed, e.g., as $(A \times B) \times (C \times D)$ or $(A \times (B \times C)) \times D$
- The cost of multiplying two matrices of size $p \times q$ and $q \times r$ is pqr

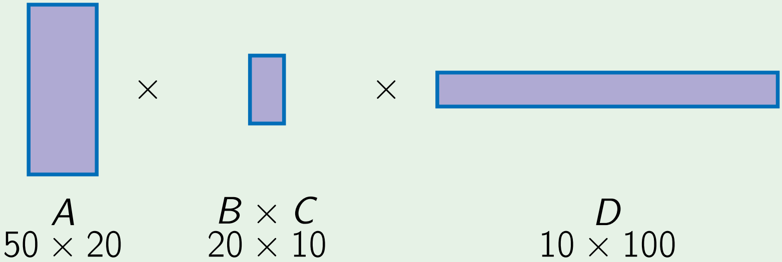
Example: $A \times ((B \times C) \times D)$



A 50×20 B 20×1 C 1×10 D 10×100

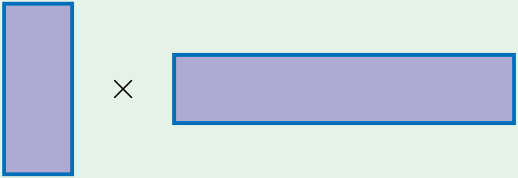
cost:

Example: $A \times ((B \times C) \times D)$



cost: $20 \cdot 1 \cdot 10$

Example: $A \times ((B \times C) \times D)$



A $\quad$ $B \times C \times D$

50×20 $\quad$ 20×100

cost: $20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100$

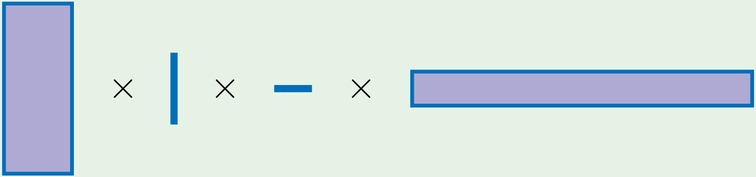
Example: $A \times ((B \times C) \times D)$

$A \times B \times C \times D$

50×100

cost: $20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100 = 120\,200$

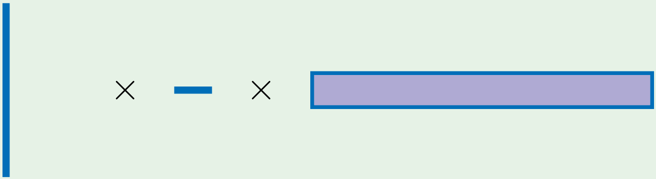
Example: $(A \times B) \times (C \times D)$



A	B	C	D
50×20	20×1	1×10	10×100

cost:

Example: $(A \times B) \times (C \times D)$



$A \times B$ $\quad$ C $\quad$ D

50×1 $\quad$ 1×10 $\quad$ 10×100

cost: $50 \cdot 20 \cdot 1$

Example: $(A \times B) \times (C \times D)$

$A \times B$ $C \times D$

50×1 1×100

cost: $50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100$

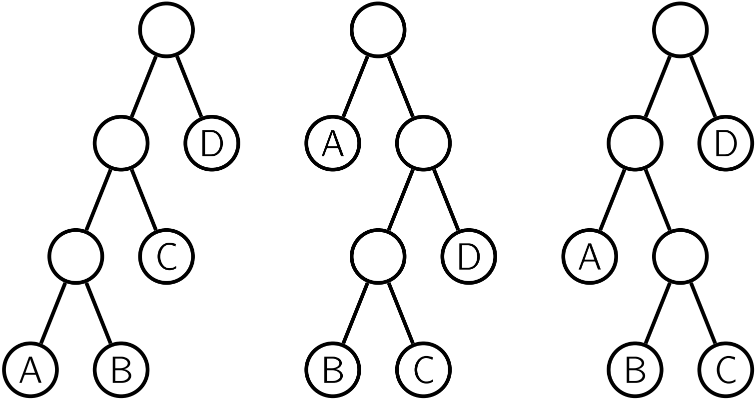
Example: $(A \times B) \times (C \times D)$

$A \times B \times C \times D$

50×100

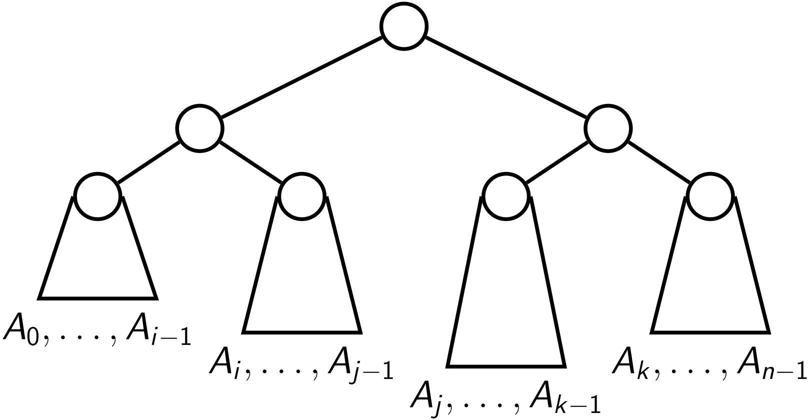
cost: $50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100 = 7\,000$

Order as a Full Binary Tree



$((A \times B) \times C) \times D$ $\quad$ $A \times ((B \times C) \times D)$ $\quad$ $(A \times (B \times C)) \times D$

Analyzing an Optimal Tree



each subtree computes
the product of $A_p, \ldots, A_q$ for some $p \leq q$

Subproblems

- Let $M(i, j)$ be the minimum cost of computing $A_i \times \cdots \times A_{j-1}$

Subproblems

- Let $M(i, j)$ be the minimum cost of computing $A_i \times \cdots \times A_{j-1}$
- Then

$$M(i, j) = \min_{i<k<j} \{M(i, k) + M(k, j) + m_i \cdot m_k \cdot m_j\}$$

Subproblems

- Let $M(i,j)$ be the minimum cost of computing $A_i \times \cdots \times A_{j-1}$
- Then

$$M(i,j) = \min_{i<k<j} \{M(i,k) + M(k,j) + m_i \cdot m_k \cdot m_j\}$$

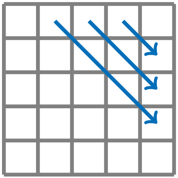
- Base case: $M(i,i+1) = 0$

Recursive Algorithm

```
T = dict()

def matrix_mult(m, i, j):
  if (i, j) not in T:
    if j == i + 1:
      T[i, j] = 0
    else:
      T[i, j] = float("inf")
      for k in range(i + 1, j):
        T[i, j] = min(T[i, j],
          matrix_mult(m, i, k) +
          matrix_mult(m, k, j) +
         m[i] * m[j] * m[k])

  return T[i, j]

print(matrix_mult(m=[50, 20, 1, 10, 100], i=0, j=4))
```

Converting to an Iterative Algorithm

- We want to solve subproblems going from smaller size subproblems to larger size ones
- The size is the number of matrices needed to be multiplied: $j - i$
- A possible order:



Iterative Algorithm

```python
def matrix_mult(m):
  n = len(m) - 1
  T = [[float("inf")] * (n + 1) for _ in range(n + 1)]

  for i in range(n):
    T[i][i + 1] = 0

  for s in range(2, n + 1):
    for i in range(n - s + 1):
      j = i + s
      for k in range(i + 1, j):
        T[i][j] = min(T[i][j],
          T[i][k] + T[k][j] +
          m[i] * m[j] * m[k])

  return T[0][n]

print(matrix_mult(m=[50, 20, 1, 10, 100]))
```

Final Remarks

- Running time: $O(n^3)$

Final Remarks

- Running time: $O(n^3)$
- To unwind a solution, go from the cell $(0, n)$ to a cell $(i, i+1)$

Final Remarks

- Running time: $O(n^3)$
- To unwind a solution, go from the cell $(0, n)$ to a cell $(i, i+1)$
- Brute force search: recursively enumerate all possible trees

Technical Slide

Module 6: Dynamic Programming ||

Step 1 (the most important step)

Define subproblems and write down a recurrence relation (with a base case)

- either by analyzing the structure of an optimal solution, or
- by optimizing a brute force solution

Subproblems: Review

1. Longest increasing subsequence: $LIS(i)$ is the length of longest common subsequence ending at element $A[i]$
2. Edit distance: $ED(i,j)$ is the edit distance between prefixes of length i and j
3. Knapsack: $K(w)$ is the optimal value of a knapsack of total weight w
4. Chain matrix multiplication $M(i,j)$ is the optimal cost of multiplying matrices through i to $j-1$

Step 2

Convert a recurrence relation into a recursive algorithm:

- store a solution to each subproblem in a table
- before solving a subproblem check whether its solution is already stored in the table

Step 3

Convert a recursive algorithm into an iterative algorithm:

- initialize the table
- go from smaller subproblems to larger ones
- specify an order of subproblems

Step 4

Prove an upper bound on the running time. Usually the product of the number of subproblems and the time needed to solve a subproblem is a reasonable estimate.

Step 5

Uncover a solution

Step 6

Exploit the regular structure of the table to check whether space can be saved

Recursive vs Iterative

- Advantages of iterative approach:
  - No recursion overhead
  - May allow saving space by exploiting a regular structure of the table
- Advantages of recursive approach:
  - May be faster if not all the subproblems need to be solved
  - An order on subproblems is implicit