Analysis and Design of Algorithms

Dynamic Programming

Instructor: **Morteza Zakeri**

Slide by: Alexander S. Kulikov

Modified by: Morteza Zakeri

Outline

Dynamic Programming

- Extremely powerful algorithmic technique with applications in optimization, scheduling, planning, economics, bioinformatics, etc.

Dynamic Programming

- Extremely powerful algorithmic technique with applications in optimization, scheduling, planning, economics, bioinformatics, etc.
- At contests, probably the most popular type of problems.

Dynamic Programming

- Extremely powerful algorithmic technique with applications in optimization, scheduling, planning, economics, bioinformatics, etc
- At contests, probably the most popular type of problems
- A solution is usually not so easy to find, but when found, is easily implementable

Dynamic Programming

- Extremely powerful algorithmic technique with applications in optimization, scheduling, planning, economics, bioinformatics, etc
- At contests, probably the most popular type of problems
- A solution is usually not so easy to find, but when found, is easily implementable
- Need a lot of practice!

Fibonacci numbers

Fibonacci numbers

$$F_n = \begin{cases} 0, & n = 0, \\ 1, & n = 1, \\ F_{n-1} + & n > 1. \end{cases}$$

$'\ {}_{n-2},$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .

Computing Fibonacci Numbers

Computing F_n

Input: An integer $n \geq 0$.

Output: The n-th Fibonacci number F_n.

Computing Fibonacci Numbers

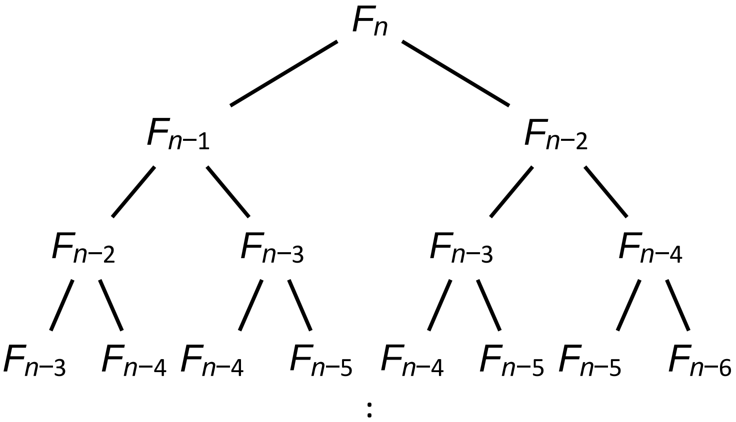
Computing F_n

Input: An integer $n \geq 0$.

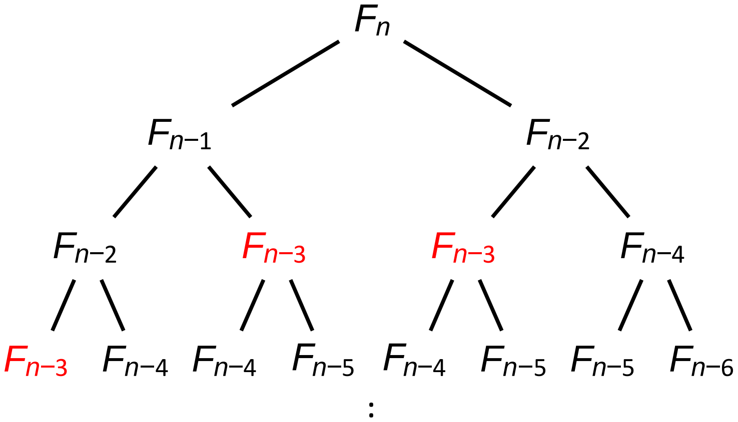
Output: The n-th Fibonacci number F_n.

```
def fib(n):
  if n <= 1:
    return n
  return fib(n − 1) + fib(n − 2)
```

Recursion Tree



Recursion Tree



Running Time

- Essentially, the algorithm computes F_n as the sum of F_n 1's

Running Time

- Essentially, the algorithm computes F_n as the sum of F_n 1's
- Hence its running time is $O(F_n)$

Running Time

- Essentially, the algorithm computes F_n as the sum of F_n 1's
- Hence its running time is $O(F_n)$
- But Fibonacci numbers grow exponentially fast: $F_n \approx \varphi^n$, where $\varphi = 1.618\ldots$ is the golden ratio

Running Time

- Essentially, the algorithm computes F_n as the sum of F_n 1's
- Hence its running time is $O(F_n)$
- But Fibonacci numbers grow exponentially fast: $F_n \approx \varphi^n$, where $\varphi = 1.618\ldots$ is the golden ratio
- E.g., F_{150} is already 31 decimal digits long

Running Time

- Essentially, the algorithm computes F_n as the sum of F_n 1's
- Hence its running time is $O(F_n)$
- But Fibonacci numbers grow exponentially fast: $F_n \approx \varphi^n$, where $\varphi = 1.618\ldots$ is the golden ratio
- E.g., F_{150} is already 31 decimal digits long
- The Sun may die before your computer returns F_{150}

Reason

- Many computations are repeated

Reason

- Many computations are repeated
- *"Those who cannot remember the past are condemned to repeat it."* (George Santayana)

Reason

- Many computations are repeated
- *"Those who cannot remember the past are condemned to repeat it."* (George Santayana)
- A simple, but crucial idea: instead of recomputing the intermediate results, let's store them once they are computed

Memoization

```
def fib(n):
  if n <= 1:
    return n
  return fib(n − 1) + fib(n − 2)
```

Memoization

```
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

```
T = dict()

def fib(n):
    if n not in T:
        if n <= 1:
            T[n] = n
        else:
            T[n] = fib(n- 1)+fib(n- 2)

    return T[n]
```

Hm...

- But do we really need all this fancy stuff (recursion, memoization, dictionaries) to solve this simple problem?

Hm...

- But do we really need all this fancy stuff (recursion, memoization, dictionaries) to solve this simple problem?
- After all, this is how you would compute F_5 by hand:

Hm...

- But do we really need all this fancy stuff (recursion, memoization, dictionaries) to solve this simple problem?
- After all, this is how you would compute F_5 by hand:
    1. $F_0 = 0, F_1 = 1$

Hm...

- But do we really need all this fancy stuff (recursion, memoization, dictionaries) to solve this simple problem?
- After all, this is how you would compute F_5 by hand:
    1. $F_0 = 0, F_1 = 1$
    2. $F_2 = 0 + 1 = 1$

Hm...

- But do we really need all this fancy stuff (recursion, memoization, dictionaries) to solve this simple problem?
- After all, this is how you would compute F_5 by hand:
  1. $F_0 = 0, F_1 = 1$
  2. $F_2 = 0 + 1 = 1$
  3. $F_3 = 1 + 1 = 2$

Hm...

- But do we really need all this fancy stuff (recursion, memoization, dictionaries) to solve this simple problem?
- After all, this is how you would compute F_5 by hand:
  1. $F_0 = 0, F_1 = 1$
  2. $F_2 = 0 + 1 = 1$
  3. $F_3 = 1 + 1 = 2$
  4. $F_4 = 1 + 2 = 3$

Hm...

- But do we really need all this fancy stuff (recursion, memoization, dictionaries) to solve this simple problem?
- After all, this is how you would compute F_5 by hand:
  1. $F_0 = 0, F_1 = 1$
  2. $F_2 = 0 + 1 = 1$
  3. $F_3 = 1 + 1 = 2$
  4. $F_4 = 1 + 2 = 3$
  5. $F_5 = 2 + 3 = 5$

Iterative Algorithm

```
def fib(n):
  T = [None] * (n + 1)
  T[0], T[1] = 0, 1

  for i in range(2, n + 1):
    T[i] = T[i - 1] + T[i - 2]

  return T[n]
```

Hm Again…

But do we really need to waste so much space?

Hm Again...

But do we really need to waste so much space?

```
def fib(n):
    if n <= 1:
        return n

    previous, current = 0, 1
    for _ in range(n - 1):
        new_current = previous + current
        previous, current = current, new_current
    return current
```

Running Time

- $O(n)$ additions

Running Time

- $O(n)$ additions
- On the other hand, recall that Fibonacci numbers grow exponentially fast: the binary length of F_n is $O(n)$

Running Time

- $O(n)$ additions
- On the other hand, recall that Fibonacci numbers grow exponentially fast: the binary length of F_n is $O(n)$
- In theory: we should not treat such additions as basic operations

Running Time

- $O(n)$ additions
- On the other hand, recall that Fibonacci numbers grow exponentially fast: the binary length of F_n is $O(n)$
- In theory: we should not treat such additions as basic operations
- In practice: just F_{100} does not fit into a 64-bit integer type anymore, hence we need bignum arithmetic

Summary

- The key idea of dynamic programming: **avoid recomputing the same thing again!**

Summary

- The key idea of dynamic programming: avoid recomputing the same thing again!
- At the same time, the case of Fibonacci numbers is a slightly artificial example of dynamic programming since it is clear from the very beginning what intermediate results we need to compute the final result

Outline

Longest Increasing Subsequence

Longest increasing

Input: An array $A = [a_0, a_1, \ldots, a_{n-1}]$.

Output: A longest increasing subsequence (LIS), i.e., $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ such that $i_1 < i_2 < \ldots < i_k$, $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$, and k is maximal.

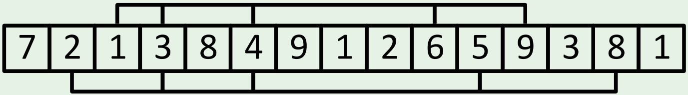
Example

Example

7	2	1	3	8	4	9	1	2	6	5	9	3	8	1

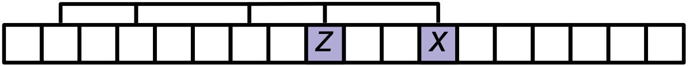
Example

Example



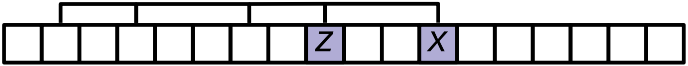
Analyzing an Optimal Solution

- Consider the last element *x* of an optimal increasing subsequence and its previous element *z*:



Analyzing an Optimal Solution

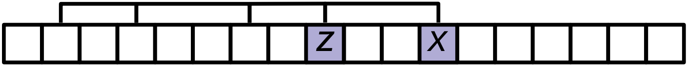
- Consider the last element x of an optimal increasing subsequence and its previous element z:



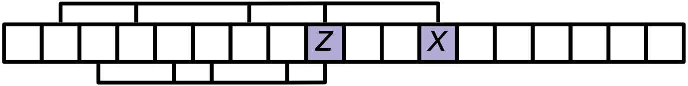
- First of all, $z < x$

Analyzing an Optimal Solution

- Consider the last element x of an optimal increasing subsequence and its previous element z:

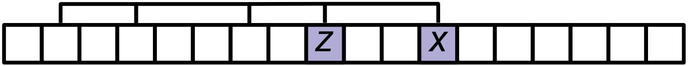


- First of all, $z < x$
- Moreover, the prefix of the IS ending at z must be an optimal IS ending at z as otherwise the initial IS would not be optimal:

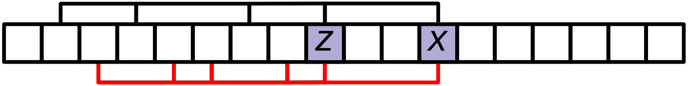


Analyzing an Optimal Solution

- Consider the last element x of an optimal increasing subsequence and its previous element z:

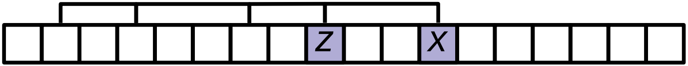


- First of all, $z < x$
- Moreover, the prefix of the IS ending at z must be an optimal IS ending at z as otherwise the initial IS would not be optimal:

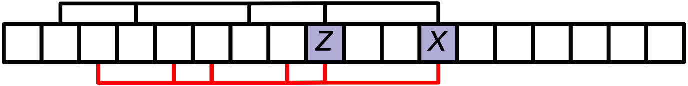


Analyzing an Optimal Solution

- Consider the last element x of an optimal increasing subsequence and its previous element z:



- First of all, $z < x$
- Moreover, the prefix of the IS ending at z must be an optimal IS ending at z as otherwise the initial IS would not be optimal:



- Optimal substructure by “cut-and-paste” trick

Subproblems and Recurrence Relation

- Let $LIS(i)$ be the optimal length of a LIS ending at $A[i]$

Subproblems and Recurrence Relation

- Let $LIS(i)$ be the optimal length of a LIS ending at $A[i]$
- Then

$$LIS(i) = 1 + \max\{LIS(j) : j < i \text{ and } A[j] < A[i]\}$$

Subproblems and Recurrence Relation

- Let $LIS(i)$ be the optimal length of a LIS ending at $A[i]$
- Then

$$LIS(i) = 1+\max\{LIS(j): j < i \text{ and } A[j] < A[i]\}$$

- Convention: maximum of an empty set is equal to zero

Subproblems and Recurrence Relation

- Let $LIS(i)$ be the optimal length of a LIS ending at $A[i]$
- Then

$$LIS(i) = 1 + \max\{LIS(j) : j < i \text{ and } A[j] < A[i]\}$$

- Convention: maximum of an empty set is equal to zero
- Base case: $LIS(0) = 1$

Algorithm

When we have a recurrence relation at hand, converting it to a recursive algorithm with memoization is just a technicality

- We will use a table T to store the results: $T[i] = LIS(i)$

Algorithm

When we have a recurrence relation at hand, converting it to a recursive algorithm with memoization is just a technicality

- We will use a table T to store the results: $T[i] = LIS(i)$
- Initially, T is empty. When $LIS(i)$ is computed, we store its value at $T[i]$ (so that we will never recompute $LIS(i)$ again)

Algorithm

When we have a recurrence relation at hand, converting it to a recursive algorithm with memoization is just a technicality

- We will use a table T to store the results: $T[i] = LIS(i)$
- Initially, T is empty. When $LIS(i)$ is computed, we store its value at $T[i]$ (so that we will never recompute $LIS(i)$ again)
- The exact data structure behind T is not that important at this point: it could be an array or a hash table

Memoization

```
T = dict()

def lis(A, i):
    if i not in T:
        T[i] = 1

        for j in range(i):
            if A[j] < A[i]:
                T[i] = max(T[i], lis(A, j) + 1)

    return T[i]

A = [7, 2, 1, 3, 8, 4, 9, 1, 2, 6, 5, 9, 3]
print(max(lis(A, i) for i in range(len(A))))
```

Running Time

The running time is quadratic ($O(n^2)$): there are n "serious" recursive calls (that are not just table look-ups), each of them needs time $O(n)$ (not counting the inner recursive calls)

Table and Recursion

- We need to store in the table T the value of $LIS(i)$ for all i from 0 to $n-1$

Table and Recursion

- We need to store in the table T the value of $LIS(i)$ for all i from 0 to $n - 1$
- Reasonable choice of a data structure for T: an array of size n

Table and Recursion

- We need to store in the table T the value of $LIS(i)$ for all i from 0 to $n - 1$
- Reasonable choice of a data structure for T: an array of size n
- Moreover, one can fill in this array iteratively instead of recursively

Iterative Algorithm

```
def lis(A):
  T = [None] * len(A)

  for i in range(len(A)):
    T[i] = 1
    for j in range(i):
      if A[j] < A[i] and T[i] < T[j] + 1:
        T[i] = T[j] + 1

  return max(T[i] for i in range(len(A)))
```

Iterative Algorithm

```
def lis(A):
  T = [None] * len(A)

  for i in range(len(A)):
    T[i] = 1
    for j in range(i):
      if A[j] < A[i] and T[i] < T[j] + 1:
        T[i] = T[j] + 1

  return max(T[i] for i in range(len(A)))
```

- Crucial property: when computing $T[i]$, $T[j]$ for all $j < i$ have already been computed

Iterative Algorithm

```python
def lis(A):
  T = [None] * len(A)

  for i in range(len(A)):
    T[i] = 1
    for j in range(i):
      if A[j] < A[i] and T[i] < T[j] + 1:
        T[i] = T[j] + 1

  return max(T[i] for i in range(len(A)))
```

- Crucial property: when computing $T[i]$, $T[j]$ for all $j < i$ have already been computed
- Running time: $O(n^2)$

Outline

Reconstructing a Solution

- How to reconstruct an optimal IS?

Reconstructing a Solution

- How to reconstruct an optimal IS?
- In order to reconstruct it, for each subproblem we will keep its optimal value and a choice leading to this value

Adjusting the Algorithm

```
def lis(A):
   T = [None] * len(A)
   prev = [None] * len(A)

   for i in range(len(A)):
      T[i] = 1
      prev[i] = -1
      for j in range(i):
         if A[j] < A[i] and T[i] < T[j] + 1:
            T[i] = T[j] + 1
            prev[i] = j
```

Unwinding Solution

```
last = 0
for i in range(1, len(A)):
  if T[i] > T[last]:
    last = i

lis= []
current = last
while current >= 0:
  lis.append(current)
  current = prev[current]
lis.reverse()
return [A[i] for i in lis]
```

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
prev	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
prev	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
prev	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
prev	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
prev	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
prev	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
prev	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
prev	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
prev	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
prev	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Example

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
prev	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Unwinding Solution

```
last = 0
for i in range(1, len(A)):
    if T[i] > T[last]:
        last = i

lis= []
current = last
while current >= 0:
    lis.append(current)
    current = prev[current]
lis.reverse()
return [A[i] for i in lis]
```

Reconstructing Again

Reconstructing without `prev`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1

Reconstructing Again

Reconstructing without `prev`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1

Reconstructing Again

Reconstructing without `prev`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1

Reconstructing Again

Reconstructing without `prev`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1

Reconstructing Again

Reconstructing without `prev`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1

Summary

- Optimal substructure property: any prefix of an optimal increasing subsequence must be a longest increasing subsequence ending at this particular element

Summary

- Optimal substructure property: any prefix of an optimal increasing subsequence must be a longest increasing subsequence ending at this particular element
- Subproblem: the length of an optimal increasing subsequence ending at i-th element

Summary

- Optimal substructure property: any prefix of an optimal increasing subsequence must be a longest increasing subsequence ending at this particular element
- Subproblem: the length of an optimal increasing subsequence ending at i-th element
- A recurrence relation for subproblems can be immediately converted into a recursive algorithm with memoization

Summary

- Optimal substructure property: any prefix of an optimal increasing subsequence must be a longest increasing subsequence ending at this particular element
- Subproblem: the length of an optimal increasing subsequence ending at i-th element
- A recurrence relation for subproblems can be immediately converted into a recursive algorithm with memoization
- A recursive algorithm, in turn, can be converted into an iterative one

Summary

- Optimal substructure property: any prefix of an optimal increasing subsequence must be a longest increasing subsequence ending at this particular element
- Subproblem: the length of an optimal increasing subsequence ending at i-th element
- A recurrence relation for subproblems can be immediately converted into a recursive algorithm with memoization
- A recursive algorithm, in turn, can be converted into an iterative one
- An optimal solution can be recovered either by using an additional bookkeeping info or by using the computed solutions to all subproblems

Outline

The Most Creative Part

- In most DP algorithms, the most creative part is coming up with the right notion of a subproblem and a recurrence relation

The Most Creative Part

- In most DP algorithms, the most creative part is coming up with the right notion of a subproblem and a recurrence relation
- When a recurrence relation is written down, it can be wrapped with memoization to get a recursive algorithm

The Most Creative Part

- In most DP algorithms, the most creative part is coming up with the right notion of a subproblem and a recurrence relation
- When a recurrence relation is written down, it can be wrapped with memoization to get a recursive algorithm
- In the previous section, we arrived at a reasonable subproblem by analyzing the structure of an optimal solution

The Most Creative Part

- In most DP algorithms, the most creative part is coming up with the right notion of a subproblem and a recurrence relation
- When a recurrence relation is written down, it can be wrapped with memoization to get a recursive algorithm
- In the previous section, we arrived at a reasonable subproblem by analyzing the structure of an optimal solution
- In this section, we'll provide an alternative way of arriving at subproblems: implement a naive brute force solution, then optimize it

Brute Force: Plan

- Need the longest increasing subsequence? No problem! Just iterate over all subsequences and select the longest one:

Brute Force: Plan

- Need the longest increasing subsequence? No problem! Just iterate over all subsequences and select the longest one:
    - Start with an empty sequence

Brute Force: Plan

- Need the longest increasing subsequence? No problem! Just iterate over all subsequences and select the longest one:
    - Start with an empty sequence
    - Extend it element by element recursively

Brute Force: Plan

- Need the longest increasing subsequence? No problem! Just iterate over all subsequences and select the longest one:
    - Start with an empty sequence
    - Extend it element by element recursively
    - Keep track of the length of the sequence

Brute Force: Plan

- Need the longest increasing subsequence? No problem! Just iterate over all subsequences and select the longest one:
  - Start with an empty sequence
  - Extend it element by element recursively
  - Keep track of the length of the sequence
- This is going to be slow, but not to worry: we will optimize it later

Brute Force: Code

```
def lis(A, seq):
    result = len(seq)

    if len(seq) == 0:
        last_index = -1
        last_element = float("-inf")
    else:
        last_index = seq[-1]
        last_element = A[last_index]

    for i in range(last_index + 1, len(A)):
        if A[i] > last_element:
            result = max(result, lis(A, seq + [i]))

    return result

print(lis(A=[7, 2, 1, 3, 8, 4, 9], seq=[]))
```

Optimizing

- At each step, we are trying to extend the current sequence

Optimizing

- At each step, we are trying to extend the current sequence
- For this, we pass the current sequence to each recursive call

Optimizing

- At each step, we are trying to extend the current sequence
- For this, we pass the current sequence to each recursive call
- At the same time, code inspection reveals that we are not using all of the sequence: we are only interested in its last element and its length

Optimizing

- At each step, we are trying to extend the current sequence
- For this, we pass the current sequence to each recursive call
- At the same time, code inspection reveals that we are not using all of the sequence: we are only interested in its last element and its length
- Let’s optimize!

Optimized Code

```python
def lis(A, seq_len, last_index):
    if last_index == -1:
        last_element = float("-inf")
    else:
        last_element = A[last_index]

    result = seq_len

    for i in range(last_index + 1, len(A)):
        if A[i] > last_element:
            result = max(result,
                         lis(A, seq_len + 1, i))

    return result

print(lis([3, 2, 7, 8, 9, 5, 8], 0, -1))
```

Optimizing Further

- Inspecting the code further, we realize that `seq_len` is not used for extending the current sequence (we don't need to know even the length of the initial part of the sequence to optimally extend it)

Optimizing Further

- Inspecting the code further, we realize that `seq_len` is not used for extending the current sequence (we don't need to know even the length of the initial part of the sequence to optimally extend it)
- More formally, for any *x*, `extend(A, seq_len, i)` is equal to `extend(A, seq_len - x, i) + x`

Optimizing Further

- Inspecting the code further, we realize that `seq_len` is not used for extending the current sequence (we don't need to know even the length of the initial part of the sequence to optimally extend it)
- More formally, for any *x*, `extend(A, seq_len, i)` is equal to `extend(A, seq_len - x, i) + x`
- Hence, can optimize the code as follows:

```
max(result, 1 + seq_len + extend(A, 0, i))
```

Optimizing Further

- Inspecting the code further, we realize that `seq_len` is not used for extending the current sequence (we don't need to know even the length of the initial part of the sequence to optimally extend it)
- More formally, for any *x*, `extend(A, seq_len, i)` is equal to `extend(A, seq_len - x, i) + x`
- Hence, can optimize the code as follows:

```
max(result, 1 + seq_len + extend(A, 0, i))
```

- Excludes `seq_len` from the list of parameters!

Resulting Code

```
def lis(A, last_index):
  if last_index == -1:
    last_element = float("-inf")
  else:
    last_element = A[last_index]

  result = 0

  for i in range(last_index + 1, len(A)):
    if A[i] > last_element:
      result = max(result, 1 + lis(A, i))

  return result

print(lis([8, 2, 3, 4, 5, 6, 7], -1))
```

Resulting Code

```
def lis(A, last_index):
    if last_index == -1:
        last_element = float("-inf")
    else:
        last_element = A[last_index]

    result = 0

    for i in range(last_index + 1, len(A)):
        if A[i] > last_element:
            result = max(result, 1 + lis(A, i))

    return result

print(lis([8, 2, 3, 4, 5, 6, 7], -1))
```

It remains to add memoization!

Summary

- Subproblems (and recurrence relation on them) is the most important ingredient of a dynamic programming algorithm

Summary

- Subproblems (and recurrence relation on them) is the most important ingredient of a dynamic programming algorithm
- Two common ways of arriving at the right subproblem:

Summary

- Subproblems (and recurrence relation on them) is the most important ingredient of a dynamic programming algorithm
- Two common ways of arriving at the right subproblem:
  - Analyze the structure of an optimal solution

Summary

- Subproblems (and recurrence relation on them) is the most important ingredient of a dynamic programming algorithm
- Two common ways of arriving at the right subproblem:
    - Analyze the structure of an optimal solution
    - Implement a brute force solution and optimize it

Outline

Statement

Edit distance

Input: Two strings $A[0 \ldots n-1]$ and $B[0 \ldots m-1]$.

Output: The minimal number of insertions, deletions, and substitutions needed to transform A to B. This number is known as edit distance or Levenshtein distance.

Example: EDITING $\longrightarrow$ DISTANCE

EDITING

Example: EDITING $\longrightarrow$ DISTANCE

$$\begin{array}{c} \text{EDITING} \\ \big\downarrow \text{remove E} \\ \text{DITING} \end{array}$$

Example: EDITING $\longrightarrow$ DISTANCE

$$
\begin{array}{c}
\text{EDITING} \\
\Big\downarrow \text{remove E} \\
\text{DITING} \\
\Big\downarrow \text{insert S} \\
\text{DISTING}
\end{array}
$$

Example: EDITING $\longrightarrow$ DISTANCE

EDITING

↓ remove E

DITING

↓ insert S

DISTING

↓ replace I with by A

DISTANG

Example: EDITING $\longrightarrow$ DISTANCE

$$
\begin{array}{l}
\text{EDITING} \\
\quad \downarrow \text{remove E} \\
\text{DITING} \\
\quad \downarrow \text{insert S} \\
\text{DISTING} \\
\quad \downarrow \text{replace I with by A} \\
\text{DISTANG} \\
\quad \downarrow \text{replace G with C} \\
\text{DISTANC}
\end{array}
$$

Example: EDITING $\longrightarrow$ DISTANCE

EDITING

↓ remove E

DITING

↓ insert S

DISTING

↓ replace I with by A

DISTANG

↓ replace G with C

DISTANC

↓ insert E

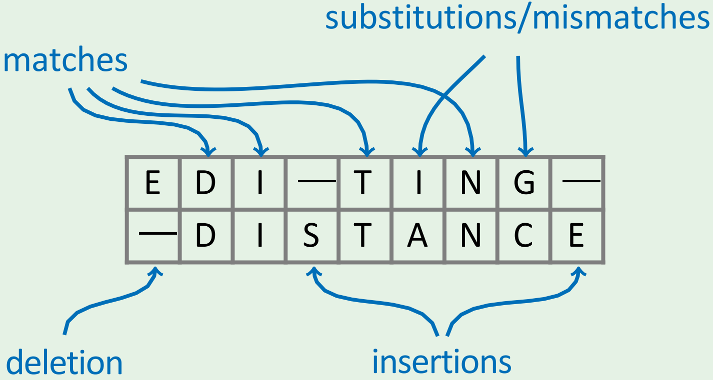
DISTANCE

Example: alignment

E	D	I	—	T	I	N	G	—
—	D	I	S	T	A	N	C	E

cost: 5

Example: alignment



cost: 5

Analyzing an Optimal Alignment

$A[0 \ldots n-1]$
$B[0 \ldots m-1]$

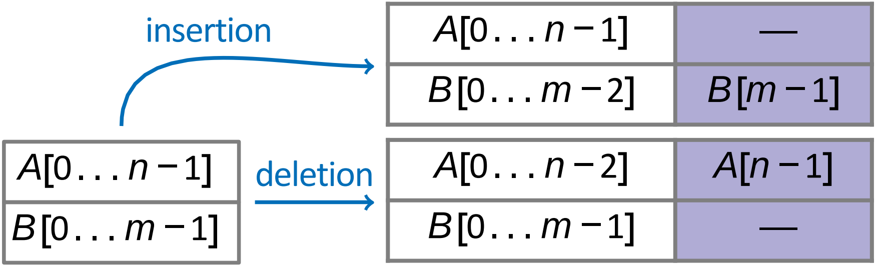
Analyzing an Optimal Alignment

$A[0 \ldots n-1]$
$B[0 \ldots m-1]$

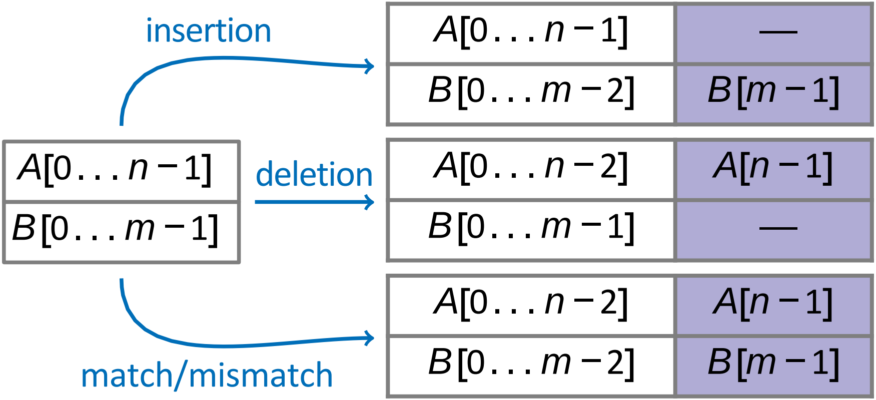
insertion

$A[0 \ldots n-1]$	—
$B[0 \ldots m-2]$	$B[m-1]$

Analyzing an Optimal Alignment



Analyzing an Optimal Alignment



Subproblems

- Let $ED(i, j)$ be the edit distance of $A[0 \ldots i-1]$ and $B[0 \ldots j-1]$.

Subproblems

- Let $ED(i, j)$ be the edit distance of $A[0 \ldots i-1]$ and $B[0 \ldots j-1]$.
- We know for sure that the last column of an optimal alignment is either an insertion, a deletion, or a match/mismatch.

Subproblems

- Let $ED(i, j)$ be the edit distance of $A[0 \ldots i-1]$ and $B[0 \ldots j-1]$.
- We know for sure that the last column of an optimal alignment is either an insertion, a deletion, or a match/mismatch.
- What is left is an optimal alignment of the corresponding two prefixes (by cut-and-paste).

Recurrence Relation

$$ED(i,j) = \min \begin{cases} ED(i, j-1) + 1 \\ ED(i-1, j) + 1 \\ ED(i-1, j-1) + \text{diff}(A[i], B[j]) \end{cases}$$

Recurrence Relation

$$ED(i,j) = \min \begin{cases} ED(i, j-1) + 1 \\ ED(i-1, j) + 1 \\ ED(i-1, j-1) + \mathrm{diff}(A[i], B[j]) \end{cases}$$

Base case: $ED(i,0) = i$, $ED(0,j) = j$

Recursive Algorithm

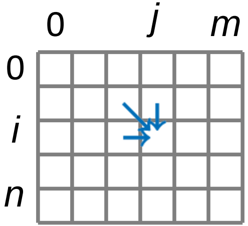
```
T = dict()

def edit_distance(a, b, i, j):
  if not (i, j) in T:
    if i == 0: T[i, j] = j
    elif j == 0: T[i, j] = i
    else:
      diff = 0 if a[i - 1] == b[j - 1] else 1
      T[i, j] = min(
        edit_distance(a, b, i - 1, j) + 1,
        edit_distance(a, b, i, j - 1) + 1,
        edit_distance(a, b, i - 1, j - 1) + diff)

  return T[i, j]


print(edit_distance(a="editing", b="distance",
                    i=7, j=8))
```

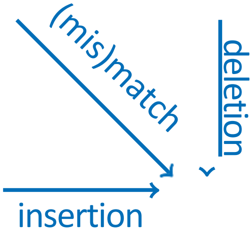
Converting to a Recursive Algorithm

- Use a 2D table to store the intermediate results

Converting to a Recursive Algorithm

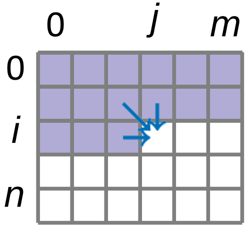
- Use a 2D table to store the intermediate results
- $ED(i, j)$ depends on $ED(i-1, j-1)$, $ED(i-1, j)$, and $ED(i, j-1)$:

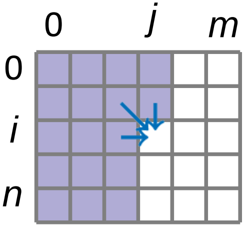




Filling the Table

Fill in the table row by row or column by column:





Iterative Algorithm

```
def edit_distance(a, b):
  T = [[float("inf")] * (len(b) + 1)
       for _ in range(len(a) + 1)]
  for i in range(len(a) + 1):
    T[i][0] = i
  for j in range(len(b) + 1):
    T[0][j] = j

  for i in range(1, len(a) + 1):
    for j in range(1, len(b) + 1):
      diff = 0 if a[i - 1] == b[j - 1] else 1
      T[i][j] = min(T[i - 1][j] + 1,
                    T[i][j - 1] + 1,
                    T[i - 1][j - 1] + diff)

  return T[len(a)][len(b)]


print(edit_distance(a="distance", b="editing"))
```

Example

			E	D	I	T	I	N	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
D	1	1							
I	2	2							
S	3	3							
T	4	4							
A	5	5							
N	6	6							
C	7	7							
E	8	8							

Example

			E	D	I	T	I	N	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
D	1	1							
I	2	2							
S	3	3							
T	4	4							
A	5	5							
N	6	6							
C	7	7							
E	8	8							

Example

			E	D	I	T	I	N	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
D	1	1	1						
I	2	2							
S	3	3							
T	4	4							
A	5	5							
N	6	6							
C	7	7							
E	8	8							

Example

			E	D	I	T	I	N	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
D	1	1	1						
I	2	2							
S	3	3							
T	4	4							
A	5	5							
N	6	6							
C	7	7							
E	8	8							

Example

			E	D	I	T	I	N	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
D	1	1	1	1					
I	2	2							
S	3	3							
T	4	4							
A	5	5							
N	6	6							
C	7	7							
E	8	8							

Example

			E	D	I	T	I	N	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
D	1	1	1	1					
I	2	2							
S	3	3							
T	4	4							
A	5	5							
N	6	6							
C	7	7							
E	8	8							

Example

			E	D	I	T	I	N	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
D	1	1	1	1	2				
I	2	2							
S	3	3							
T	4	4							
A	5	5							
N	6	6							
C	7	7							
E	8	8							

Example

			E	D	I	T	I	N	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
D	1	1	1	1	2	3	4	5	6
I	2	2	2	2	1	2	3	4	5
S	3	3	3	3	2	2	3	4	5
T	4	4	4	4	3	2	3	4	5
A	5	5	5	5	4	3	3	4	5
N	6	6	6	6	5	4	4	3	4
C	7	7	7	7	6	5	5	4	4
E	8	8	7	8	7	6	6	5	5

Brute Force

- Recursively construct an alignment column by column

Brute Force

- Recursively construct an alignment column by column
- Then note, that for extending the partially constructed alignment optimally, one only needs to know the already used length of prefix of A and the length of prefix of B

Outline

Reconstructing a Solution

- To reconstruct a solution, we go back from the cell (n, m) to the cell $(0, 0)$

Reconstructing a Solution

- To reconstruct a solution, we go back from the cell (n, m) to the cell $(0, 0)$
- If $ED(i, j) = ED(i-1, j) + 1$, then there exists an optimal alignment whose last column is a deletion

Reconstructing a Solution

- To reconstruct a solution, we go back from the cell (n, m) to the cell $(0, 0)$
- If $ED(i, j) = ED(i-1, j) + 1$, then there exists an optimal alignment whose last column is a deletion
- If $ED(i, j) = ED(i, j-1) + 1$, then there exists an optimal alignment whose last column is an insertion

Reconstructing a Solution

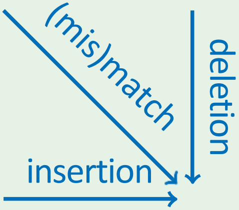
- To reconstruct a solution, we go back from the cell (n, m) to the cell (0, 0)
- If $ED(i, j) = ED(i-1, j) + 1$, then there exists an optimal alignment whose last column is a deletion
- If $ED(i, j) = ED(i, j-1) + 1$, then there exists an optimal alignment whose last column is an insertion
- If $ED(i, j) = ED(i-1, j-1) + \text{diff}(A[i], B[j])$, then match (if $A[i] = B[j]$) or mismatch (if $A[i] \neq B[j]$)

Example

		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5

Example

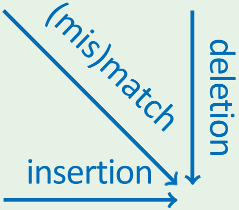
		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5



E
G

Example

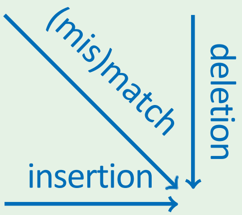
		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5



C	E
-	G

Example

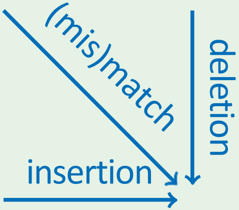
		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5



N	C	E
N	-	G

Example

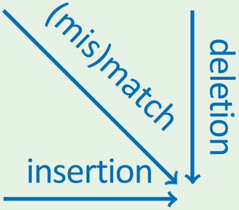
		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5



A	N	C	E
I	N	-	G

Example

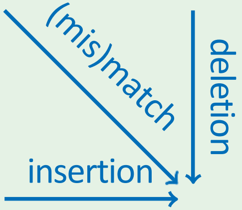
		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5



T	A	N	C	E
T	I	N	-	G

Example

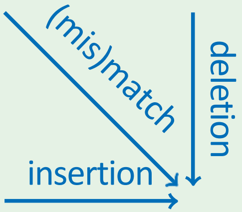
		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5



S	T	A	N	C	E
-	T	I	N	-	G

Example

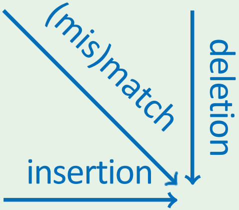
		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5



I	S	T	A	N	C	E
I	-	T	I	N	-	G

Example

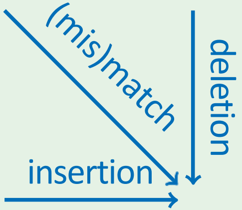
		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5



D	I	S	T	A	N	C	E
D	I	-	T	I	N	-	G

Example

		E	D	I	T	I	N	G
	0	1	2	3	4	5	6	7
D	1	1	1	2	3	4	5	6
I	2	2	2	1	2	3	4	5
S	3	3	3	2	2	3	4	5
T	4	4	4	3	2	3	4	5
A	5	5	5	4	3	3	4	5
N	6	6	6	5	4	4	3	4
C	7	7	7	6	5	5	4	4
E	8	7	8	7	6	6	5	5

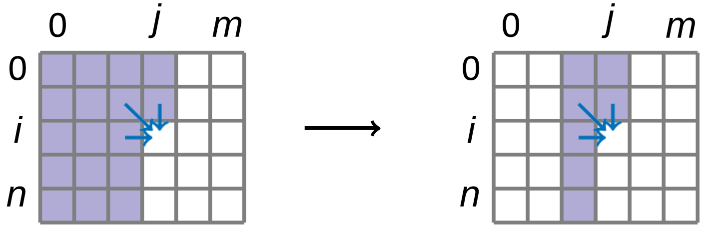


-	D	I	S	T	A	N	C	E
E	D	I	-	T	I	N	-	G

Outline

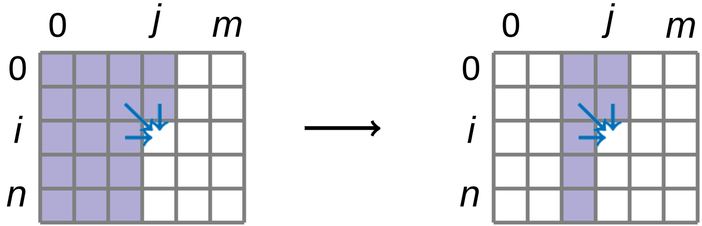
Saving Space

- When filling in the matrix it is enough to keep only the current column and the previous column:



Saving Space

- When filling in the matrix it is enough to keep only the current column and the previous column:



- Thus, one can compute the edit distance of two given strings $A[1 \ldots n]$ and $B[1 \ldots m]$ in time $O(nm)$ and space $O(\min\{n, m\})$.

Reconstructing a Solution

- However we need the whole table to find an actual alignment (we trace an alignment from the bottom right corner to the top left corner)

Reconstructing a Solution

- However we need the whole table to find an actual alignment (we trace an alignment from the bottom right corner to the top left corner)
- There exists an algorithm constructing an optimal alignment in time $O(nm)$ and space $O(n + m)$ (Hirschberg's algorithm)

Weighted Edit Distance

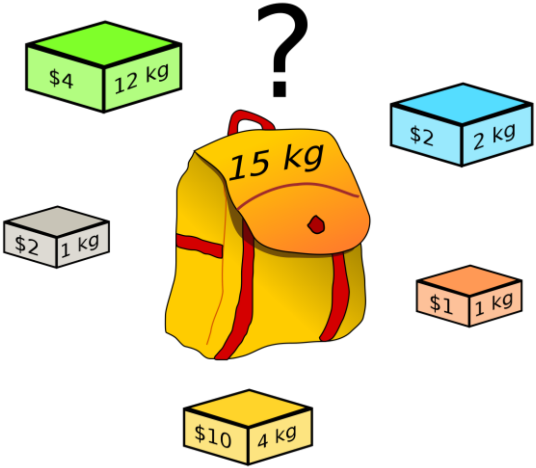
- The cost of insertions, deletions, and substitutions is not necessarily identical
- Spell checking: some substitutions are more likely than others
- Biology: some mutations are more likely than others

Generalized Recurrence Relation

$$\min \begin{cases} ED(i, j-1) + \text{inscost}(B[j]), \\ ED(i-1, j) + \text{delcost}(A[i]), \\ ED(i-1, j-1) + \text{substcost}(A[i], B[j]) \end{cases}$$

Outline

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

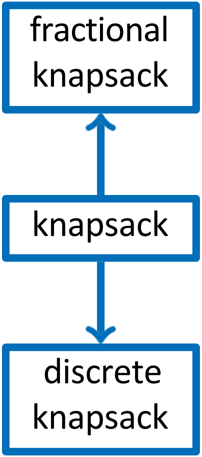
Applications

- 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

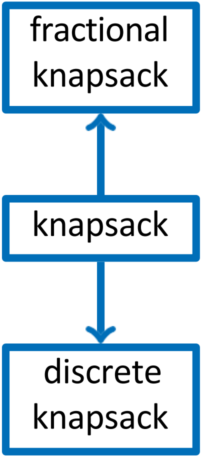
Problem Variations

$\boxed{\text{knapsack}}$

Problem Variations



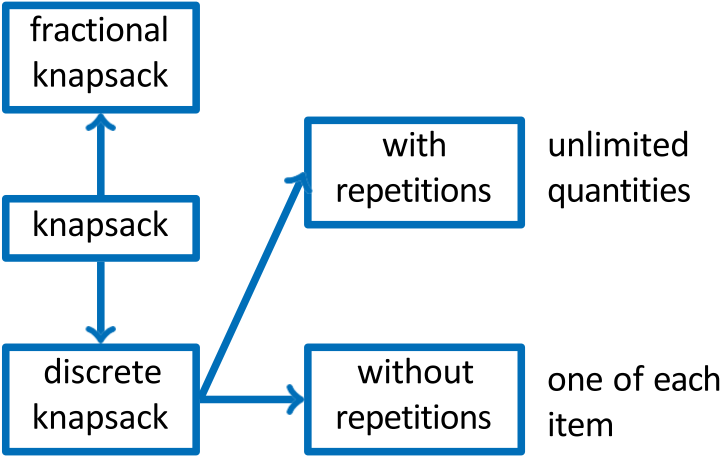
Problem Variations



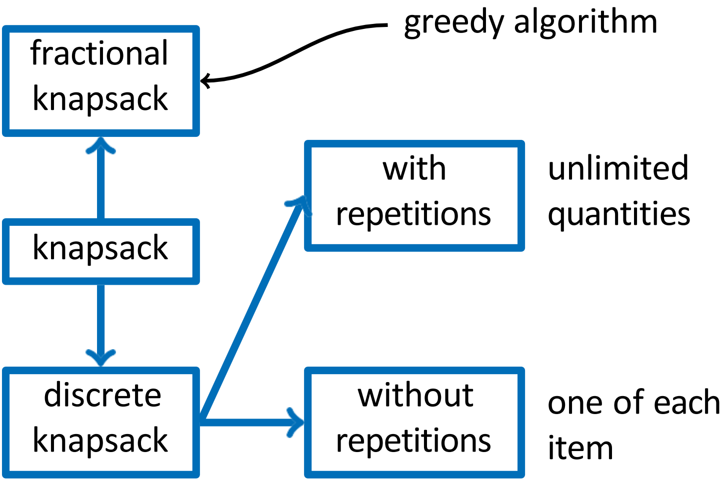
can take fractions of items

each item is either taken or not

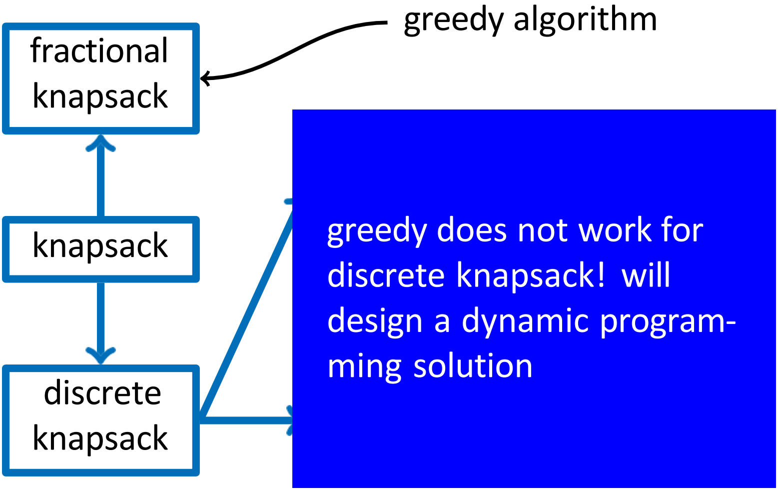
Problem Variations



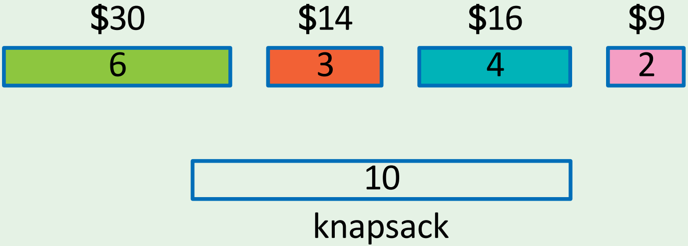
Problem Variations



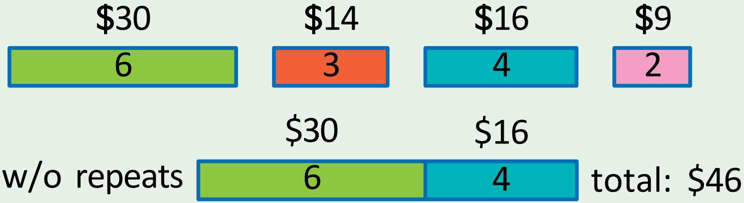
Problem Variations



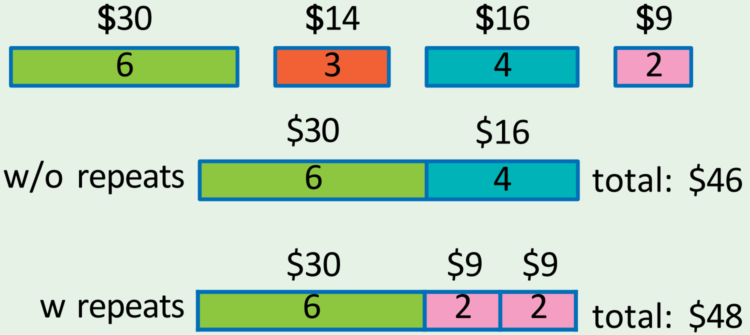
Example



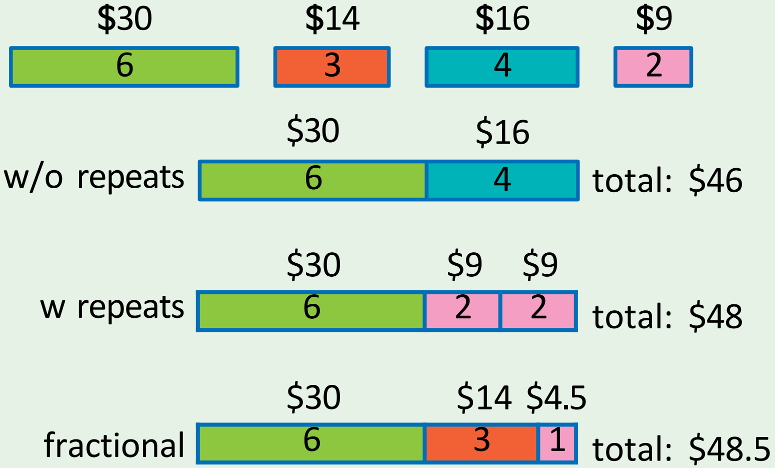
Example



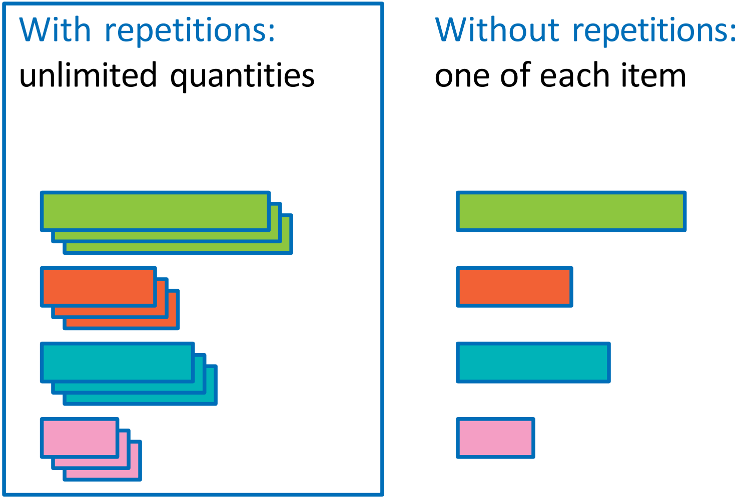
Example


$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

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
fractional
$30
6
$14
3
$4.5
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
-

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

Subproblems

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

$$value(u) = \max_{i\,:\,w_i \le w} \{value(u - w_i) + v_i\}$$

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

Subproblems

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

$$value(u) = \max_{i\,:\,w_i \le w} \{value(u - w_i) + v_i\}$$

- 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

```
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

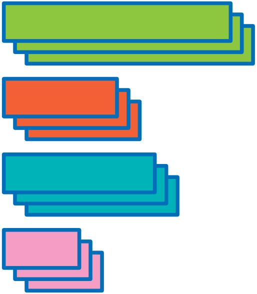
- It remains to notice that the only important thing for extending the current set of items is the weight of this set

Subproblems

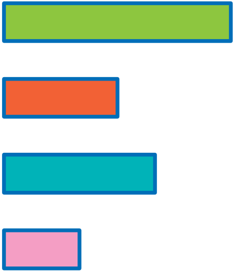
- It remains to notice that the only important thing for extending the current set of items is the weight of this set
- One then replaces `items` by their weight in the list of parameters

Outline

With repetitions:
unlimited quantities



Without repetitions:
one of each item



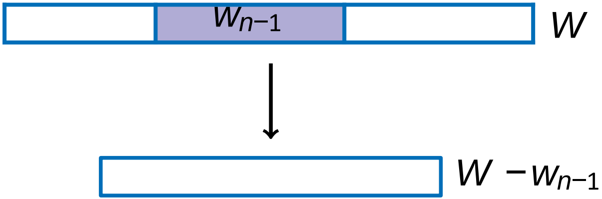
Knapsack without repetitions problem

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).

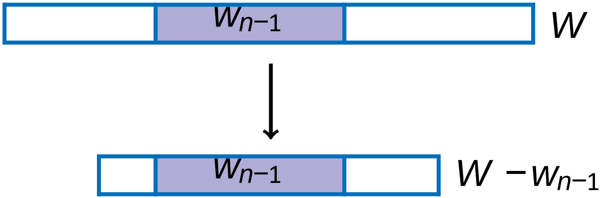
Output: The maximum value of items whose weight does not exceed W. Each item can be used at most once.

Same Subproblems?

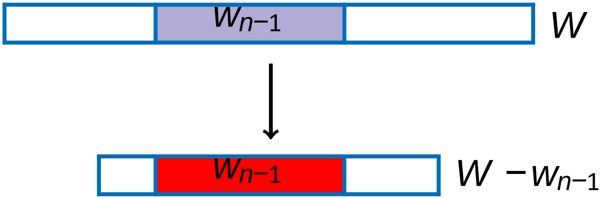
Same Subproblems?



Same Subproblems?

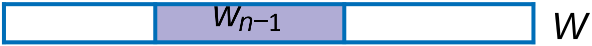


Same Subproblems?



Subproblems

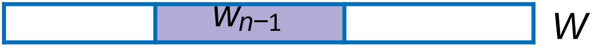
- If the last item is taken into an optimal solution:



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 $0 \le u \le W$ and $0 \le i \le n$, $value(u, i)$ 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 $0 \le u \le W$ and $0 \le i \le n$, $value(u, i)$ 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

```
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))
```

Outline

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)

Fractional Knapsack

The time complexity of the Fractional Knapsack problem is O(N log N).
This complexity arises because the problem is typically solved using a **greedy algorithm**, which involves sorting the items based on their value-to-weight ratio and then adding them to the knapsack in this sorted order until the knapsack's capacity is reached or all items are considered

Outline

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

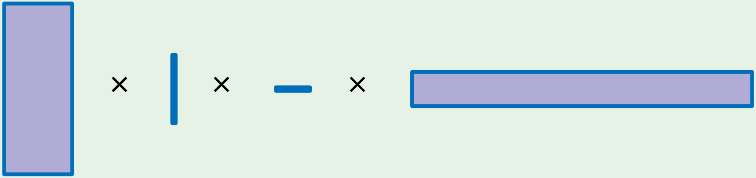
Clarifications

- Denote the sizes of matrices $A_0, \ldots, A_{n-1}$ 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 $(A \times (B \times C)) \times D$
- The cost of multiplying two matrices of size $p \times q$ and $q \times r$ is pqr

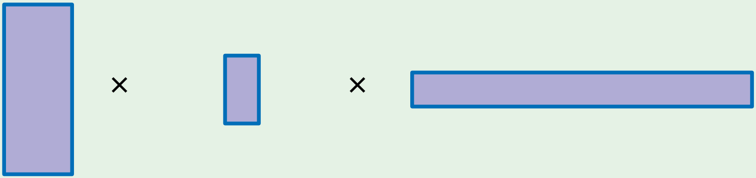
Example: $A \times ((B \times C) \times D)$



A 50×20 B 20×1 C 1×10 D 10×100

cost:

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

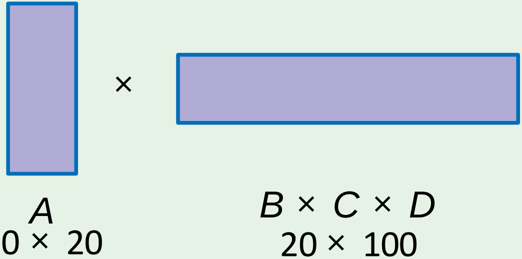


A $B \times C$ D

50×20 20×10 10×100

cost: $20 \cdot 1 \cdot 10$

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



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

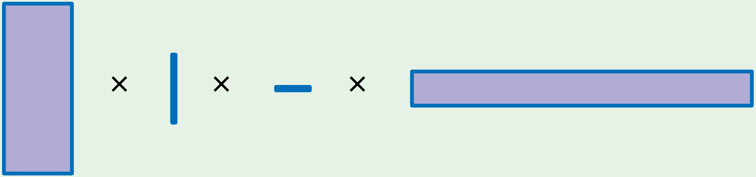
Example: $A \times ((B \times C) \times D)$

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

50×100

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

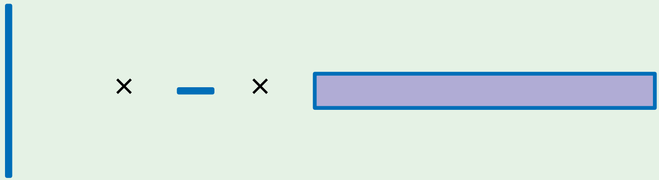
Example: $(A \times B) \times (C \times D)$



A	B	C	D
50×20	20×1	1×10	10×100

cost:

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



$A \times B$	C	D
50×1	1×10	10×100

cost: $50 \cdot 20 \cdot 1$

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

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

50×1 $\qquad\qquad\qquad$ 1×100

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

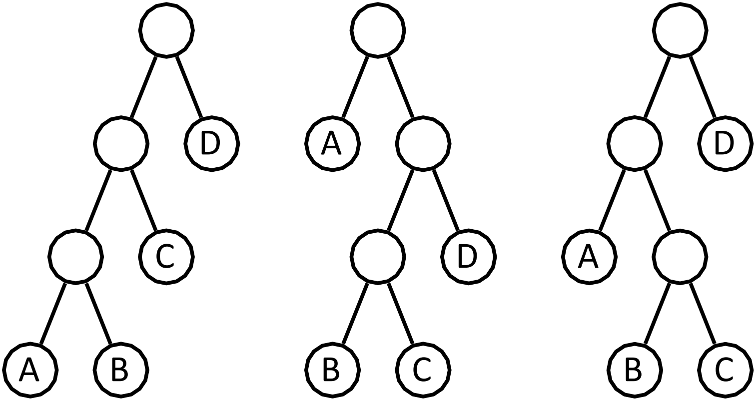
Example: $(A \times B) \times (C \times D)$

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

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

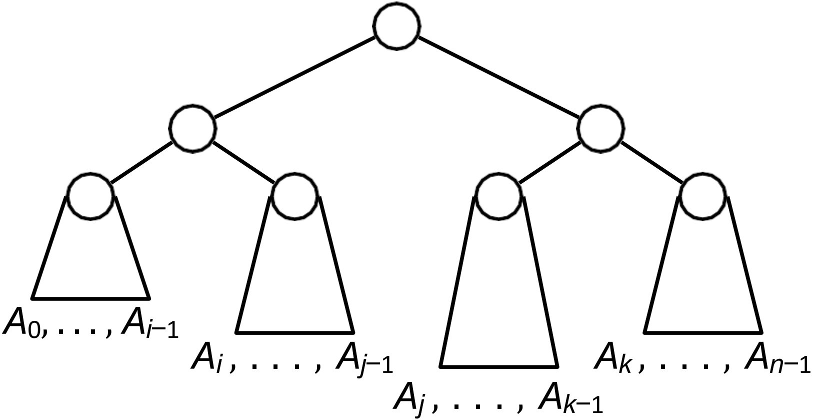


$((A \times B) \times C) \times D$

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

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

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

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

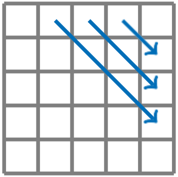
- Base case: $M(i, i+1) = 0$

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))
```

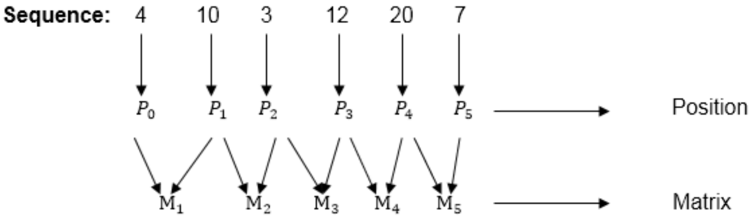
Converting to an Iterative Algorithm

- 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:



Example

The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7



1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

Example

The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7

$$M[2,4] = \min \begin{cases} M[2,3] + M[4,4] + p_1 p_3 p_4 = 360 + 0 + 10.12.20 = 2760 \\ M[2,2] + M[3,4] + p_1 p_2 p_4 = 0 + 720 + 10.3.20 = 1320 \end{cases}$$

M [3, 5] = 1140

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

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

Outline

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