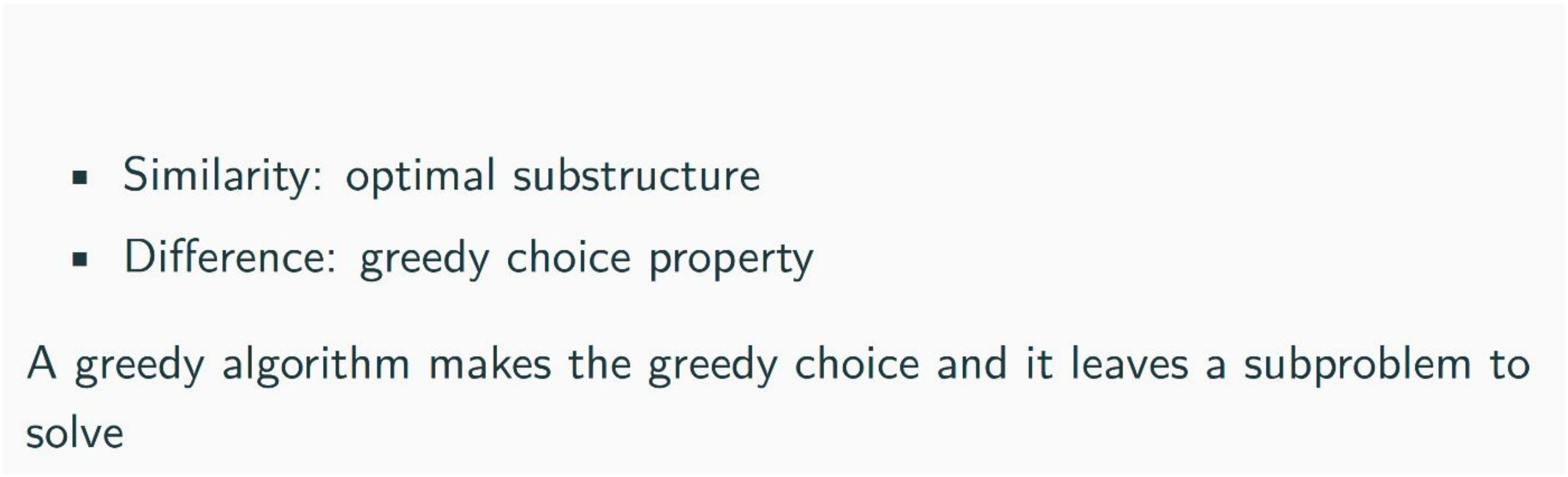
Dynamic programming

CMPSC 465 - Yana Safonova

Intro: the longest increasing subsequence

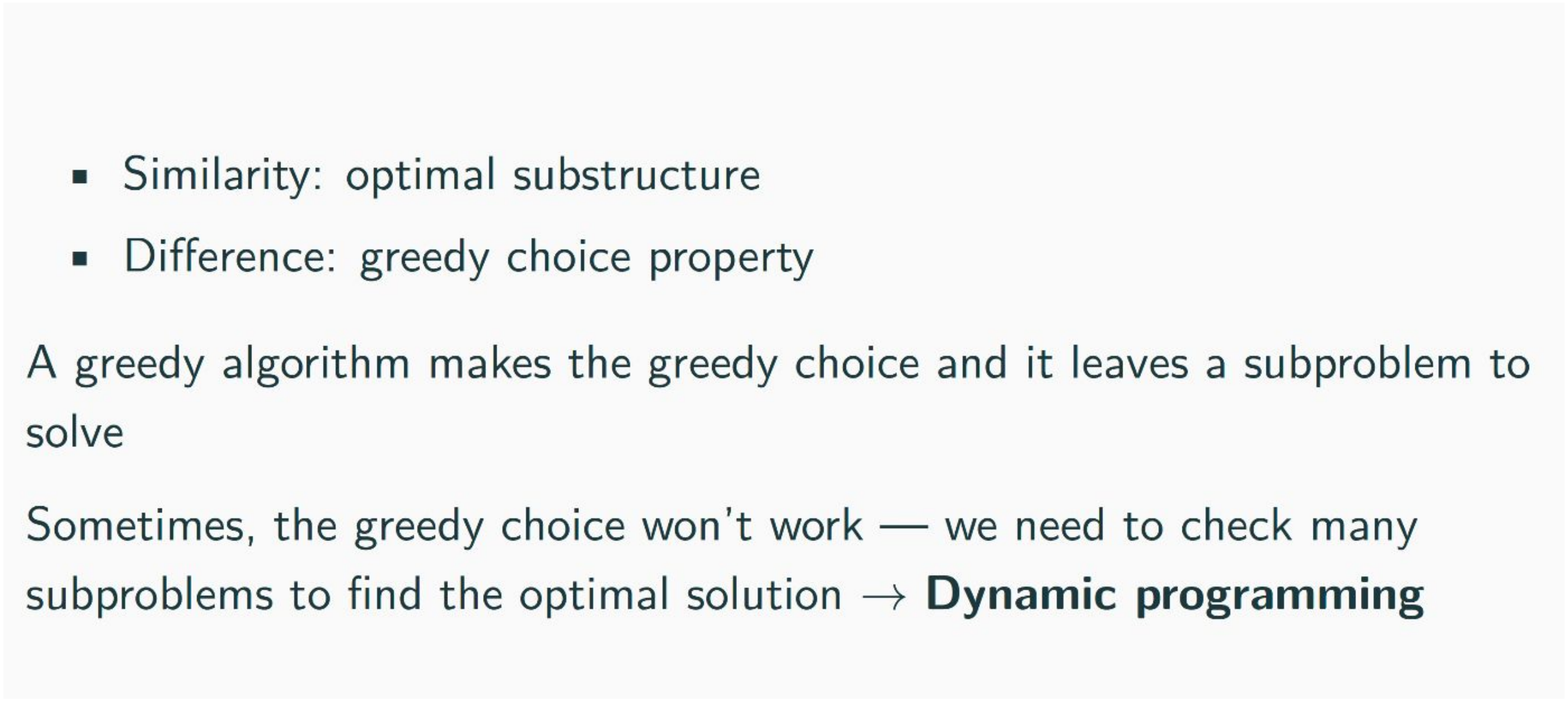
Greedy algorithms vs DP



- Similarity: optimal substructure
- Difference: greedy choice property

A greedy algorithm makes the greedy choice and it leaves a subproblem to solve

Greedy algorithms vs DP

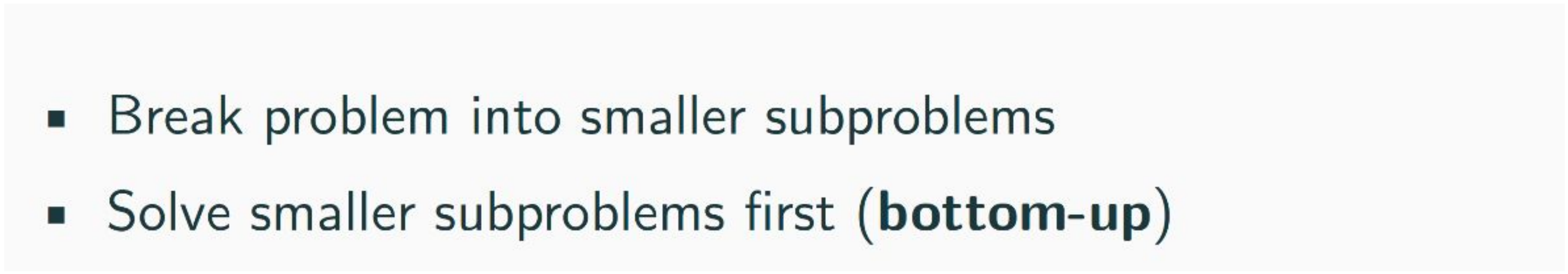


- Similarity: optimal substructure
- Difference: greedy choice property

A greedy algorithm makes the greedy choice and it leaves a subproblem to solve

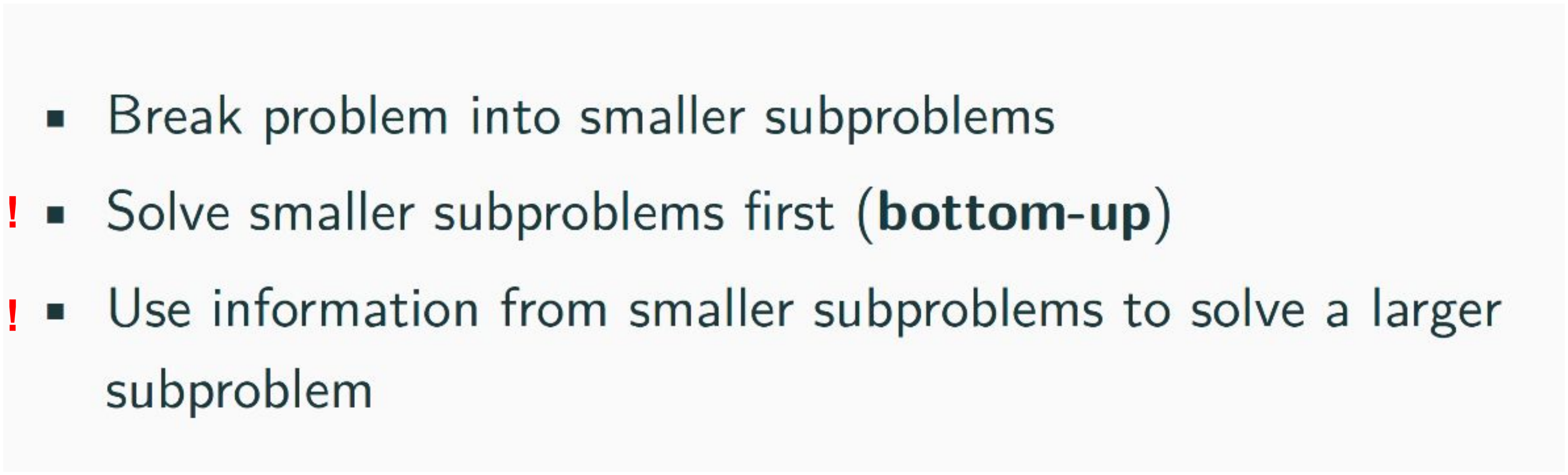
Sometimes, the greedy choice won't work — we need to check many subproblems to find the optimal solution $\rightarrow$ **Dynamic programming**

Main steps of DP



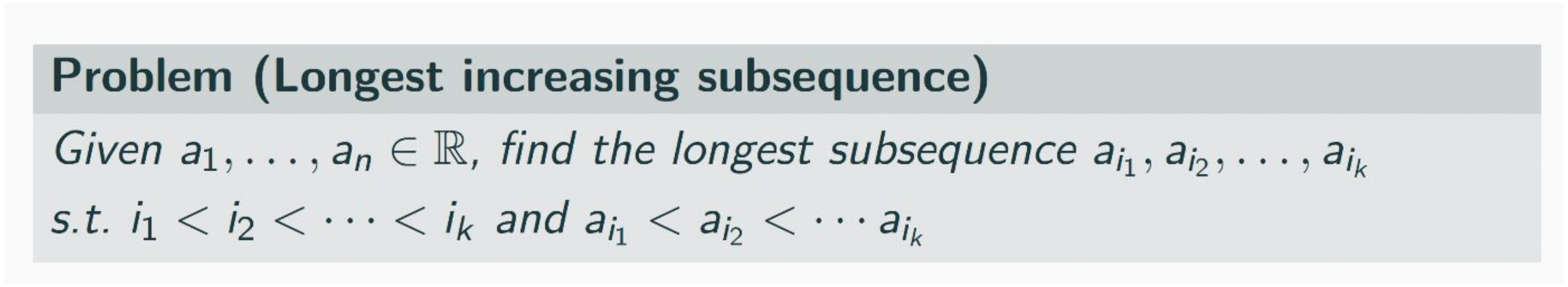
- Break problem into smaller subproblems
- Solve smaller subproblems first (**bottom-up**)

Main steps of DP



- Break problem into smaller subproblems
- ! Solve smaller subproblems first (**bottom-up**)
- ! Use information from smaller subproblems to solve a larger subproblem

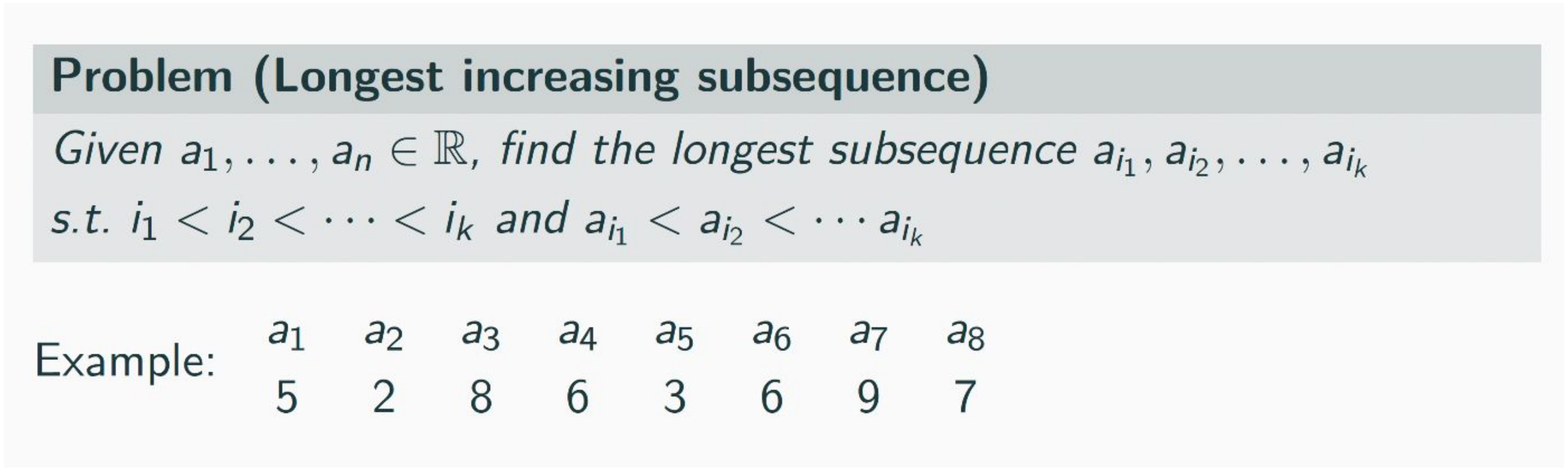
Longest increasing subsequence



Problem (Longest increasing subsequence)

Given $a_1, \ldots, a_n \in \mathbb{R}$, *find the longest subsequence* $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ *s.t.* $i_1 < i_2 < \cdots < i_k$ *and* $a_{i_1} < a_{i_2} < \cdots a_{i_k}$

Longest increasing subsequence



Problem (Longest increasing subsequence)

Given $a_1, \ldots, a_n \in \mathbb{R}$, *find the longest subsequence* $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ *s.t.* $i_1 < i_2 < \cdots < i_k$ *and* $a_{i_1} < a_{i_2} < \cdots a_{i_k}$

Example:

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
5	2	8	6	3	6	9	7

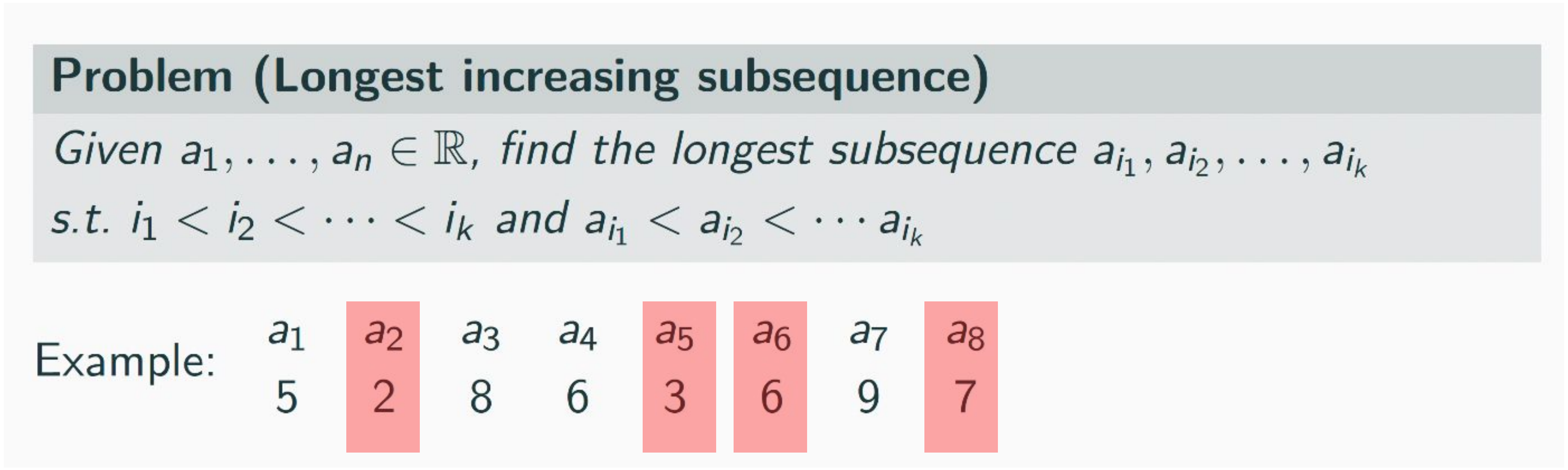
Longest increasing subsequence

Problem (Longest increasing subsequence)

Given $a_1, \ldots, a_n \in \mathbb{R}$, *find the longest subsequence* $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ *s.t.* $i_1 < i_2 < \cdots < i_k$ *and* $a_{i_1} < a_{i_2} < \cdots a_{i_k}$

Example:	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
	5	**2**	8	6	**3**	**6**	**9**	7

Longest increasing subsequence



Problem (Longest increasing subsequence)

Given $a_1, \ldots, a_n \in \mathbb{R}$*, find the longest subsequence* $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ *s.t.* $i_1 < i_2 < \cdots < i_k$ *and* $a_{i_1} < a_{i_2} < \cdots a_{i_k}$

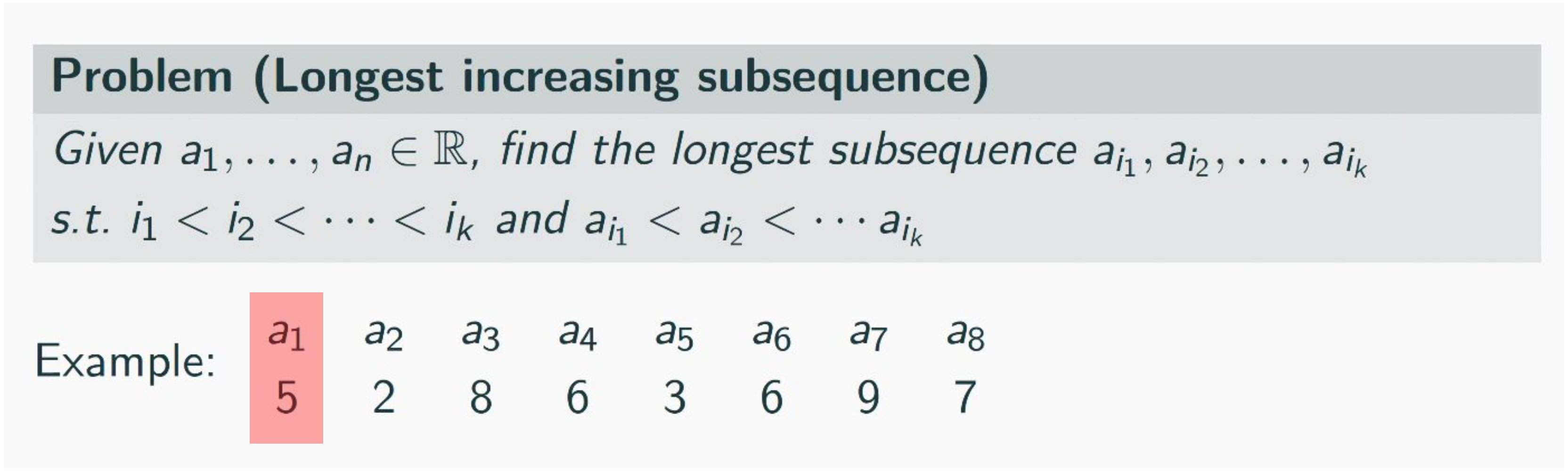
Example:	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
	5	**2**	8	6	**3**	**6**	9	**7**

What would be a greedy algorithm for this problem?

- Start with an empty subsequence LIS = []
- Iterate over all positions of $A = a_1,...,a_n$
- If a_i is greater than the last element of LIS, then LIS += a_i

What would be a greedy algorithm for this problem?

- Start with an empty subsequence LIS = []
- Iterate over all positions of A = $a_1,\ldots,a_n$
- If a_i is greater than the last element of LIS, then LIS += a_i



Problem (Longest increasing subsequence)

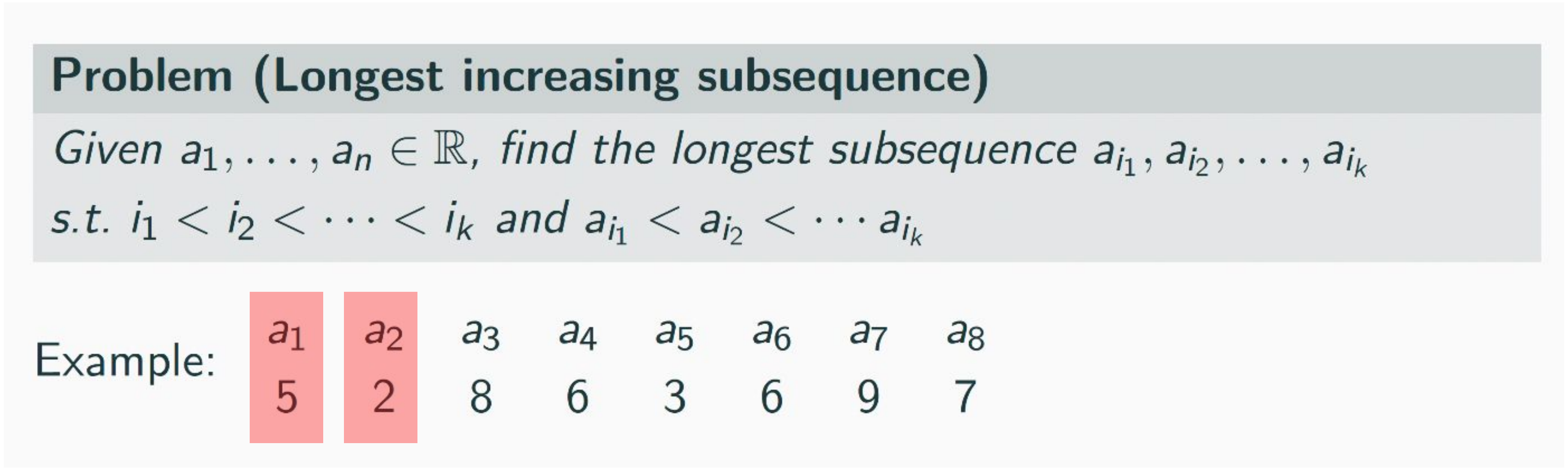
Given $a_1, \ldots, a_n \in \mathbb{R}$, find the longest subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ s.t. $i_1 < i_2 < \cdots < i_k$ and $a_{i_1} < a_{i_2} < \cdots a_{i_k}$

Example:

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
5	2	8	6	3	6	9	7

What would be a greedy algorithm for this problem?

- Start with an empty subsequence LIS = []
- Iterate over all positions of A = $a_1,\ldots,a_n$
- If a_i is greater than the last element of LIS, then LIS += a_i



Problem (Longest increasing subsequence)

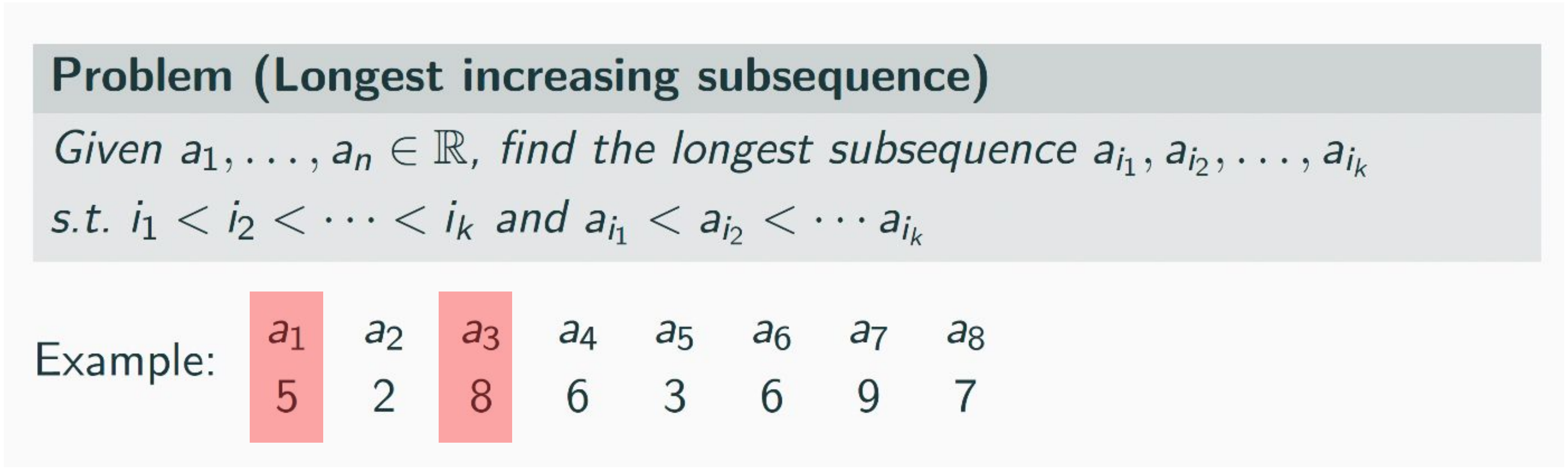
Given $a_1, \ldots, a_n \in \mathbb{R}$, find the longest subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ s.t. $i_1 < i_2 < \cdots < i_k$ and $a_{i_1} < a_{i_2} < \cdots a_{i_k}$

Example:

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
5	2	8	6	3	6	9	7

What would be a greedy algorithm for this problem?

- Start with an empty subsequence LIS = []
- Iterate over all positions of A = $a_1,...,a_n$
- If a_i is greater than the last element of LIS, then LIS += a_i



Problem (Longest increasing subsequence)

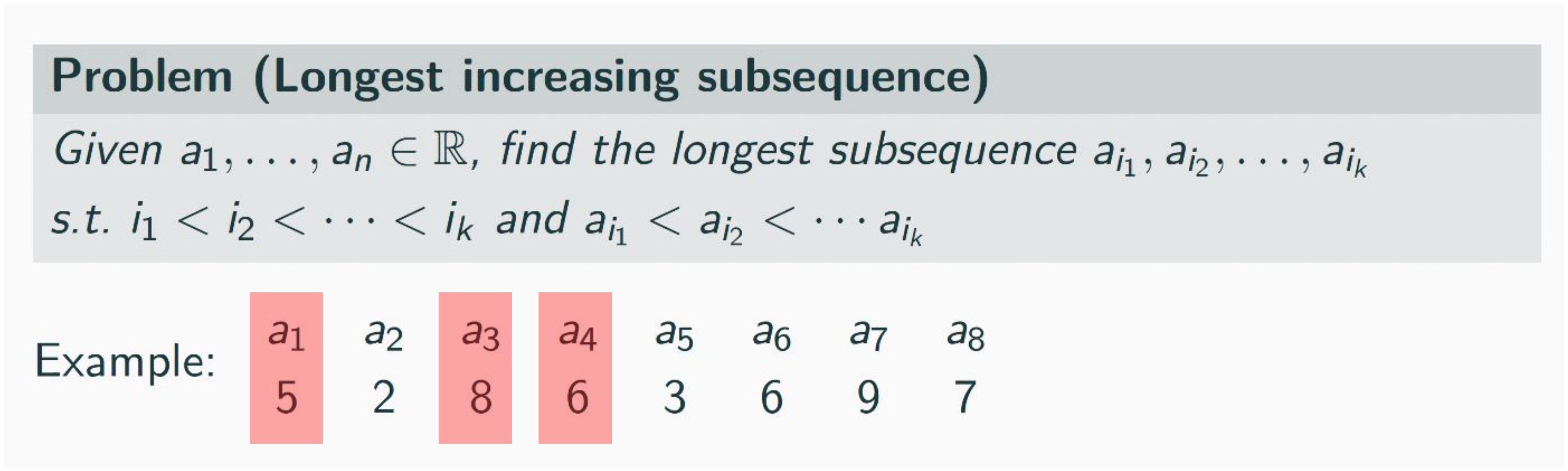
Given $a_1, \dots, a_n \in \mathbb{R}$, find the longest subsequence $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ s.t. $i_1 < i_2 < \cdots < i_k$ and $a_{i_1} < a_{i_2} < \cdots a_{i_k}$

Example:

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
5	2	8	6	3	6	9	7

What would be a greedy algorithm for this problem?

- Start with an empty subsequence LIS = []
- Iterate over all positions of A = $a_1,\dots,a_n$
- If a_i is greater than the last element of LIS, then LIS += a_i



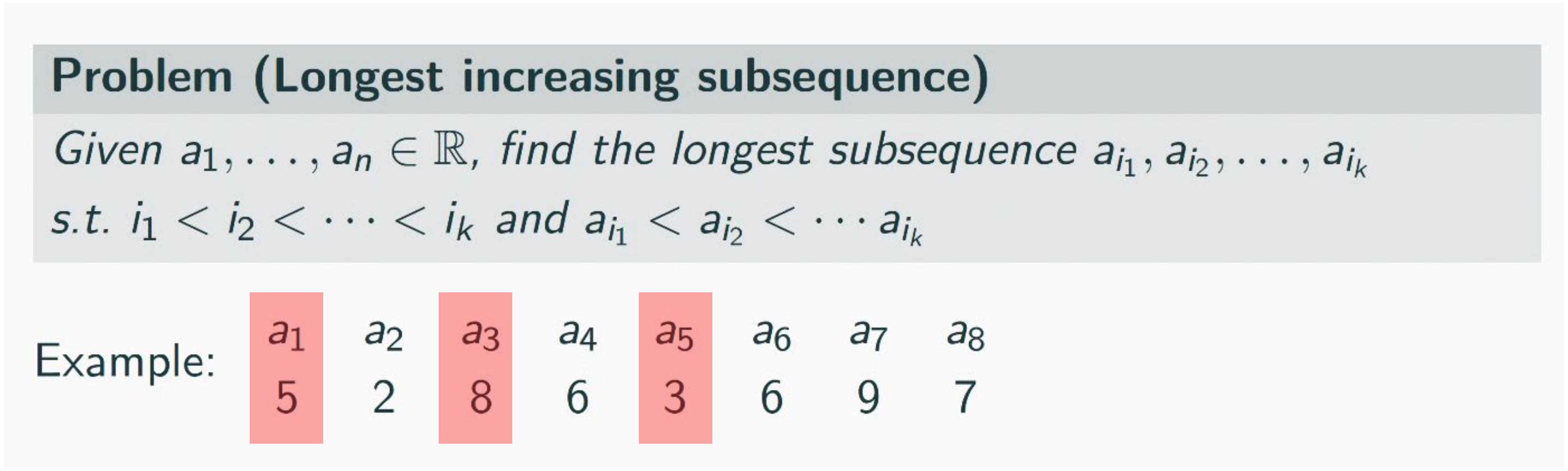
Problem (Longest increasing subsequence)

Given $a_1, \dots, a_n \in \mathbb{R}$, find the longest subsequence $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ s.t. $i_1 < i_2 < \cdots < i_k$ and $a_{i_1} < a_{i_2} < \cdots a_{i_k}$

Example:	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
	5	2	8	6	3	6	9	7

What would be a greedy algorithm for this problem?

- Start with an empty subsequence LIS = []
- Iterate over all positions of A = $a_1,\dots,a_n$
- If a_i is greater than the last element of LIS, then LIS += a_i



Problem (Longest increasing subsequence)

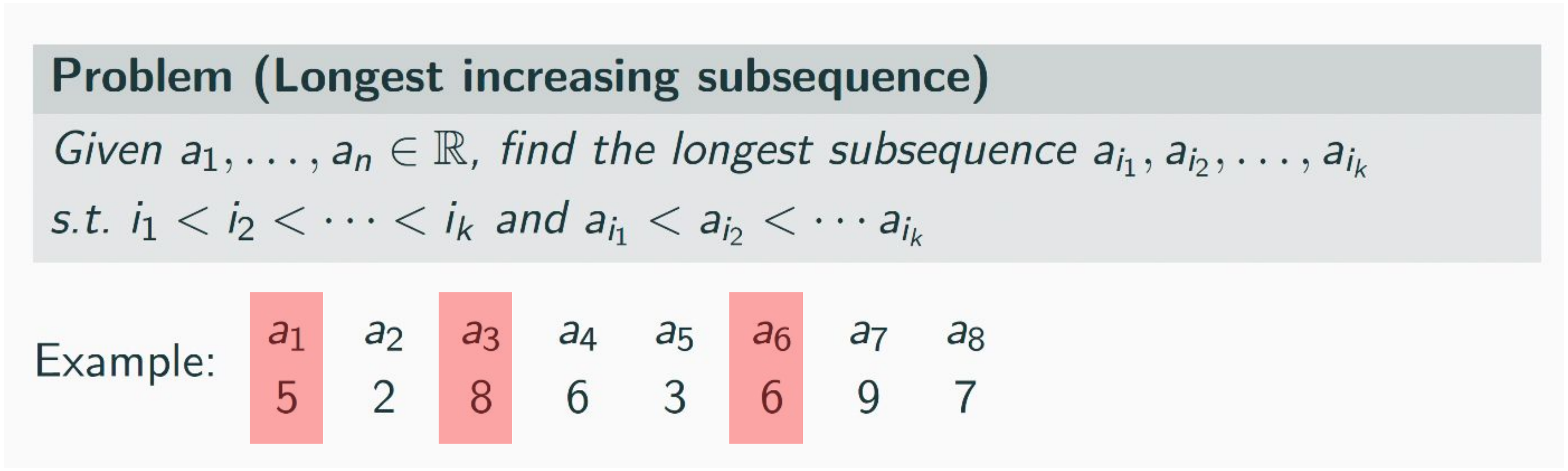
Given $a_1, \dots, a_n \in \mathbb{R}$, find the longest subsequence $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ s.t. $i_1 < i_2 < \cdots < i_k$ and $a_{i_1} < a_{i_2} < \cdots a_{i_k}$

Example:

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
5	2	8	6	3	6	9	7

What would be a greedy algorithm for this problem?

- Start with an empty subsequence LIS = []
- Iterate over all positions of $A = a_1,\ldots,a_n$
- If a_i is greater than the last element of LIS, then LIS += a_i



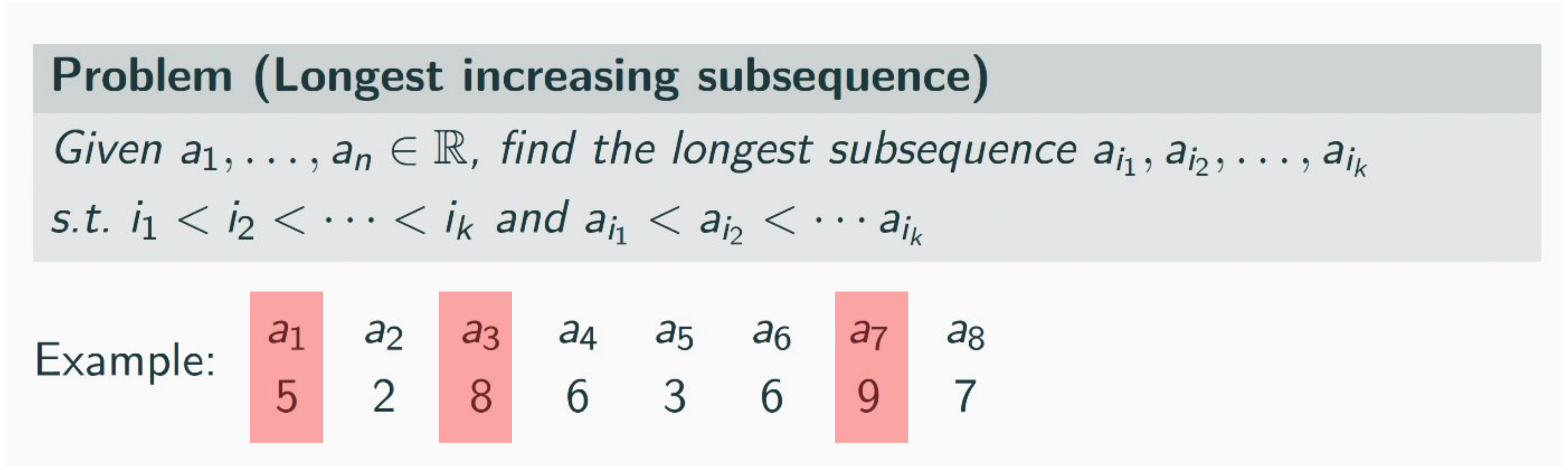
Problem (Longest increasing subsequence)

Given $a_1, \ldots, a_n \in \mathbb{R}$, find the longest subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ s.t. $i_1 < i_2 < \cdots < i_k$ and $a_{i_1} < a_{i_2} < \cdots a_{i_k}$

Example:	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
	5	2	8	6	3	6	9	7

What would be a greedy algorithm for this problem?

- Start with an empty subsequence LIS = []
- Iterate over all positions of $A = a_1,\ldots,a_n$
- If a_i is greater than the last element of LIS, then LIS += a_i



Problem (Longest increasing subsequence)

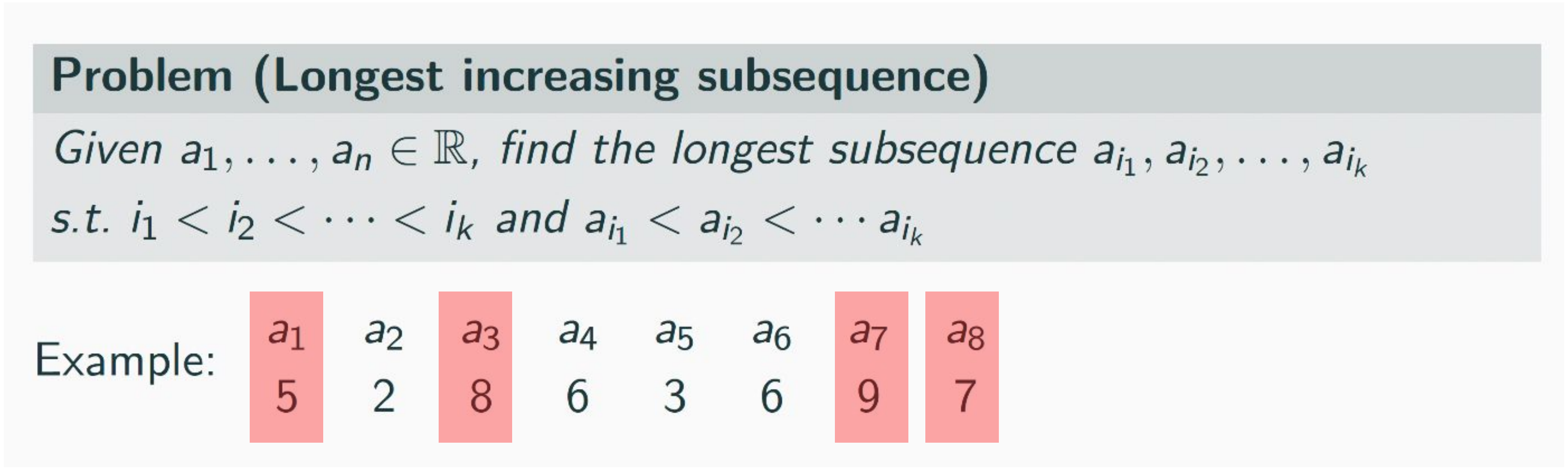
Given $a_1, \ldots, a_n \in \mathbb{R}$, find the longest subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ s.t. $i_1 < i_2 < \cdots < i_k$ and $a_{i_1} < a_{i_2} < \cdots a_{i_k}$

Example:

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
5	2	8	6	3	6	9	7

What would be a greedy algorithm for this problem?

- Start with an empty subsequence LIS = []
- Iterate over all positions of $A = a_1,\ldots,a_n$
- If a_i is greater than the last element of LIS, then LIS += a_i



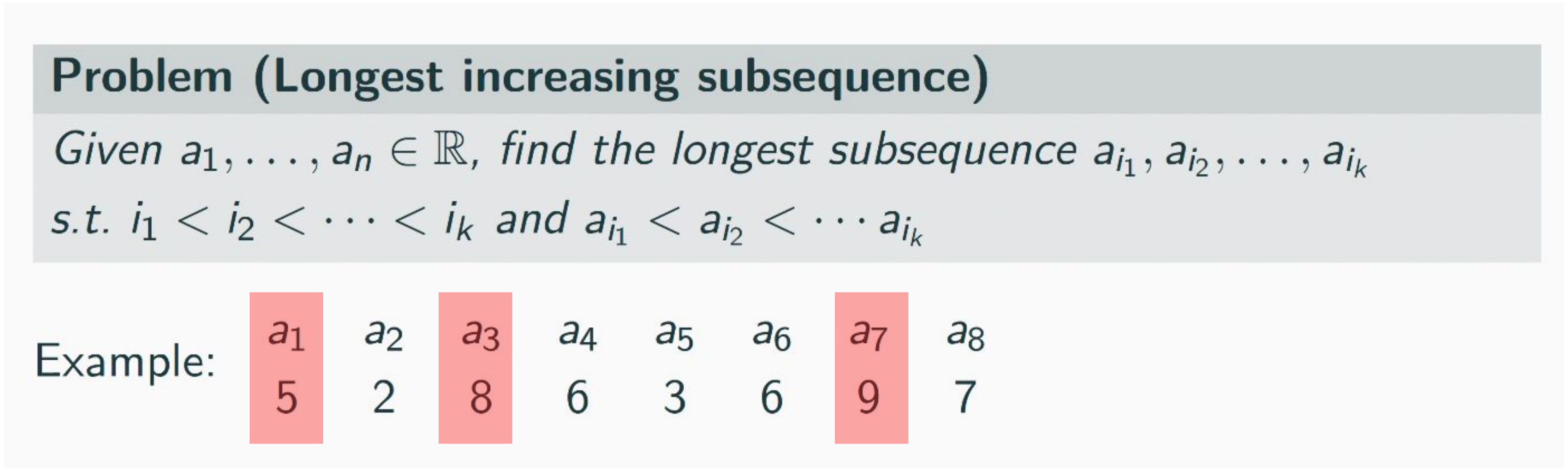
Problem (Longest increasing subsequence)

Given $a_1, \ldots, a_n \in \mathbb{R}$, find the longest subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ s.t. $i_1 < i_2 < \cdots < i_k$ and $a_{i_1} < a_{i_2} < \cdots a_{i_k}$

Example:	**a_1**	a_2	**a_3**	a_4	a_5	a_6	**a_7**	**a_8**
	5	2	**8**	6	3	6	**9**	**7**

What would be a greedy algorithm for this problem?

- Start with an empty subsequence LIS = []
- Iterate over all positions of A = $a_1,\ldots,a_n$
- If a_i is greater than the last element of LIS, then LIS += a_i



Problem (Longest increasing subsequence)

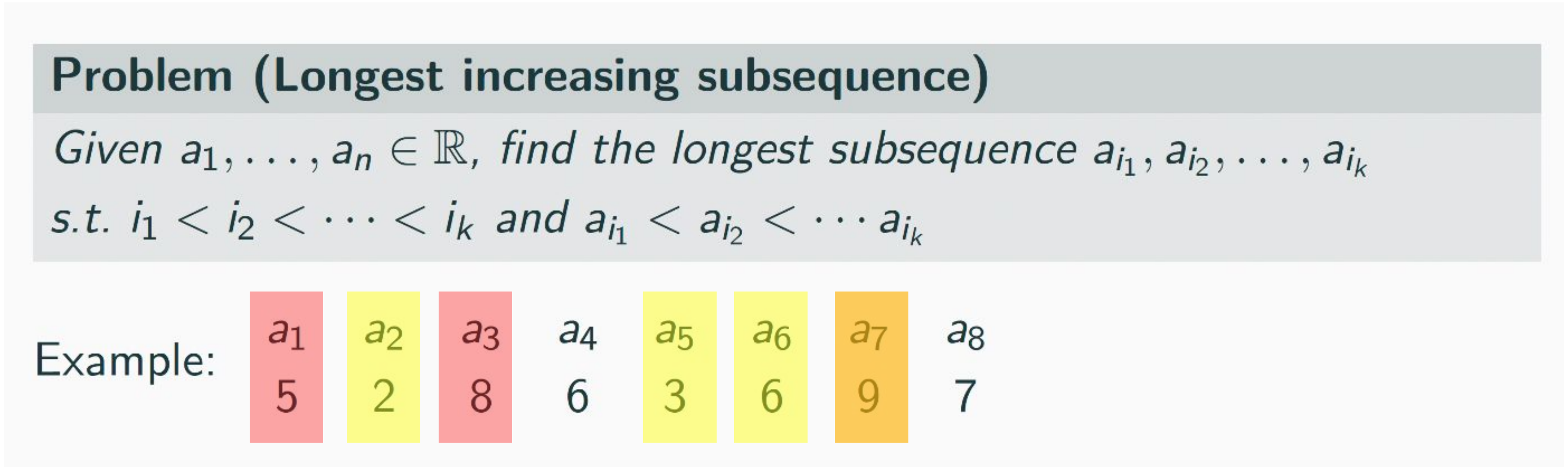
Given $a_1, \ldots, a_n \in \mathbb{R}$, find the longest subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ s.t. $i_1 < i_2 < \cdots < i_k$ and $a_{i_1} < a_{i_2} < \cdots a_{i_k}$

Example:

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
5	2	8	6	3	6	9	7

What would be a greedy algorithm for this problem?

- Start with an empty subsequence LIS = []
- Iterate over all positions of $A = a_1,\ldots,a_n$
- If a_i is greater than the last element of LIS, then LIS += a_i



Problem (Longest increasing subsequence)

Given $a_1, \ldots, a_n \in \mathbb{R}$, find the longest subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ s.t. $i_1 < i_2 < \cdots < i_k$ and $a_{i_1} < a_{i_2} < \cdots a_{i_k}$

Example:

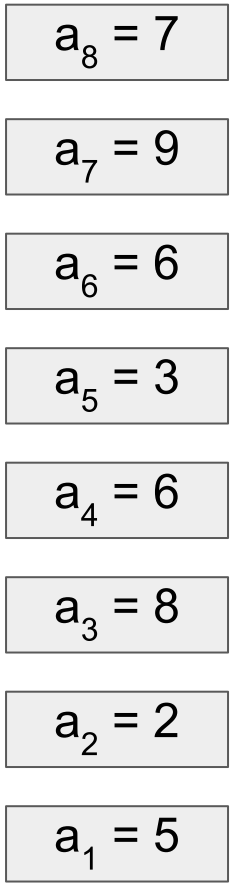
a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
5	2	8	6	3	6	9	7

- The greedy algorithm doesn't provide us with the optimal answer
- How we can modify the algorithm to fix it?

Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

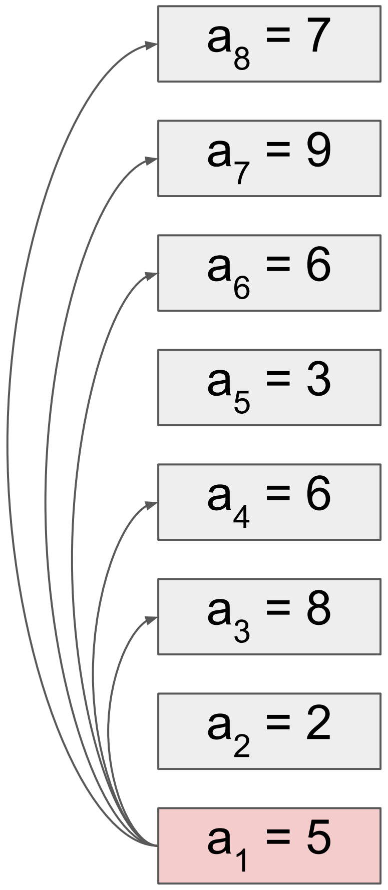
- (a_i, a_j) if $i < j$ and $a_i < a_j$



Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

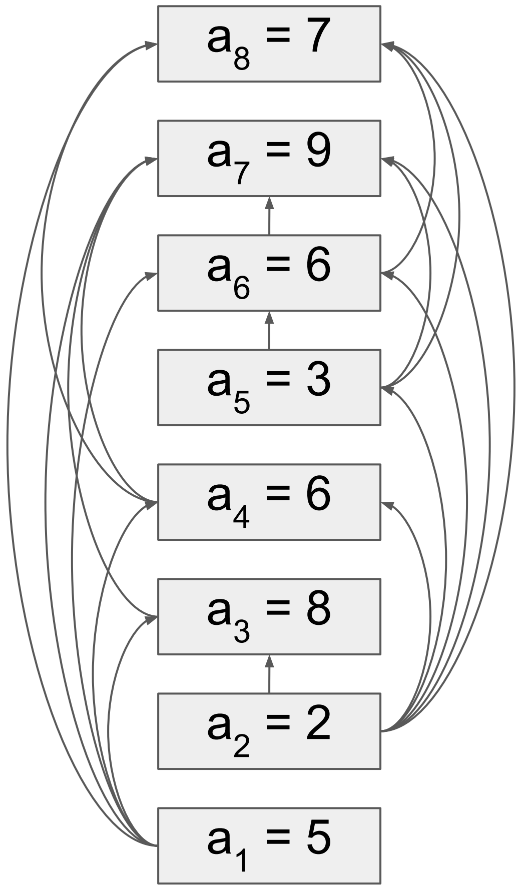
- (a_i, a_j) if $i < j$ and $a_i < a_j$



Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

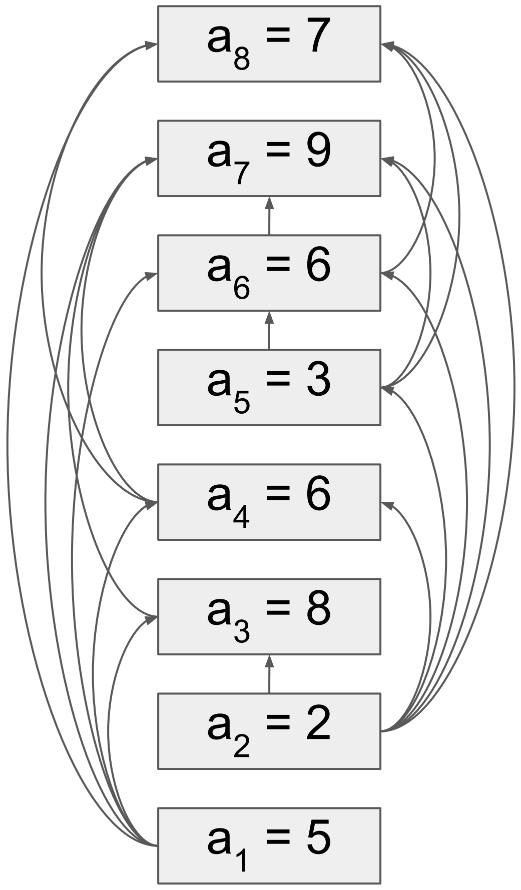
- (a_i, a_j) if $i < j$ and $a_i < a_j$



Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

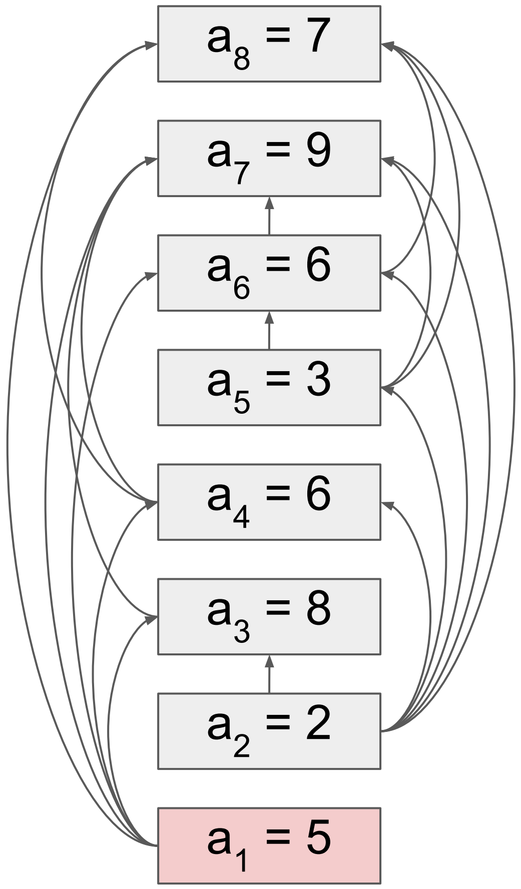
- (a_i, a_j) if $i < j$ and $a_i < a_j$
- Find the longest path in the DAG:
- Use L(j) to denote the length of the longest path (longest increasing subsequence) ending with a_j



Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

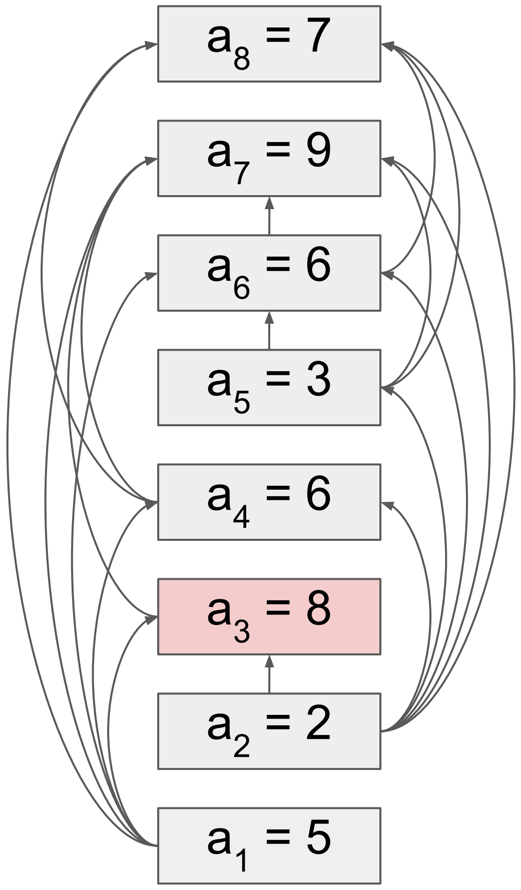
- (a_i, a_j) if $i < j$ and $a_i < a_j$
- Find the longest path in the DAG:
- Use L(j) to denote the length of the longest path (longest increasing subsequence) ending with a_j
  - $L(a_1) = ?$



Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

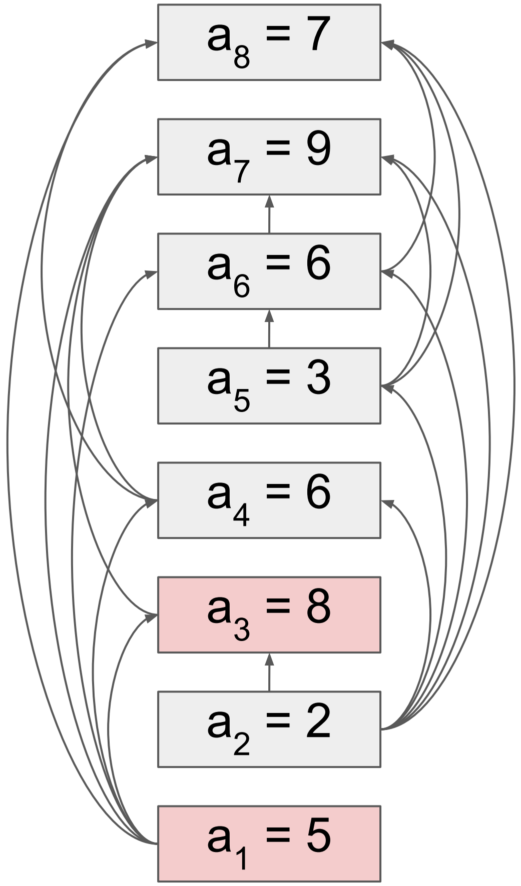
- (a_i, a_j) if $i < j$ and $a_i < a_j$
- Find the longest path in the DAG:
- Use L(j) to denote the length of the longest path (longest increasing subsequence) ending with a_j
    - $L(a_1) = 1$
    - $L(a_2) = 1$
    - $L(a_3) = ?$



Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

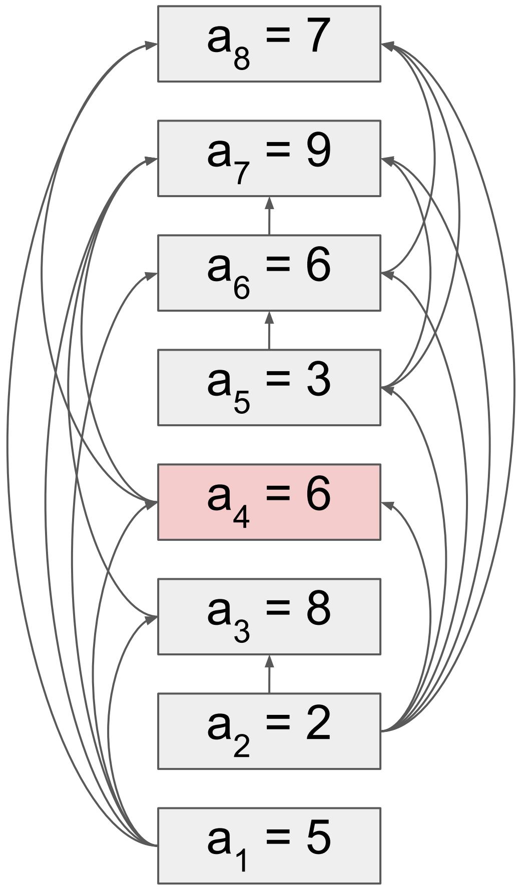
- (a_i, a_j) if $i < j$ and $a_i < a_j$
- Find the longest path in the DAG:
- Use $L(j)$ to denote the length of the longest path (longest increasing subsequence) ending with a_j
  - $L(a_1) = 1$
  - $L(a_2) = 1$
  - $L(a_3) = 2$
  -



Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

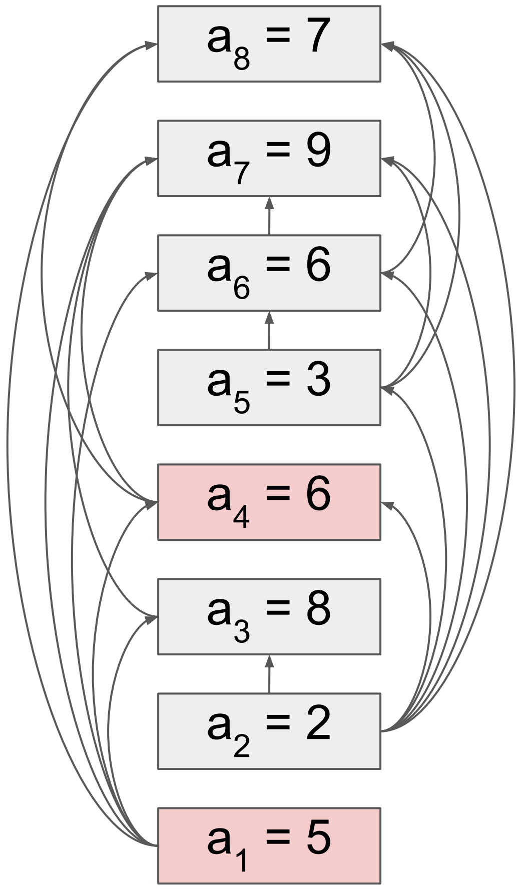
- (a_i, a_j) if $i < j$ and $a_i < a_j$
- Find the longest path in the DAG:
- Use $L(j)$ to denote the length of the longest path (longest increasing subsequence) ending with a_j
  - $L(a_1) = 1$
  - $L(a_2) = 1$
  - $L(a_3) = 2$
  - $L(a_4) = ?$



Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

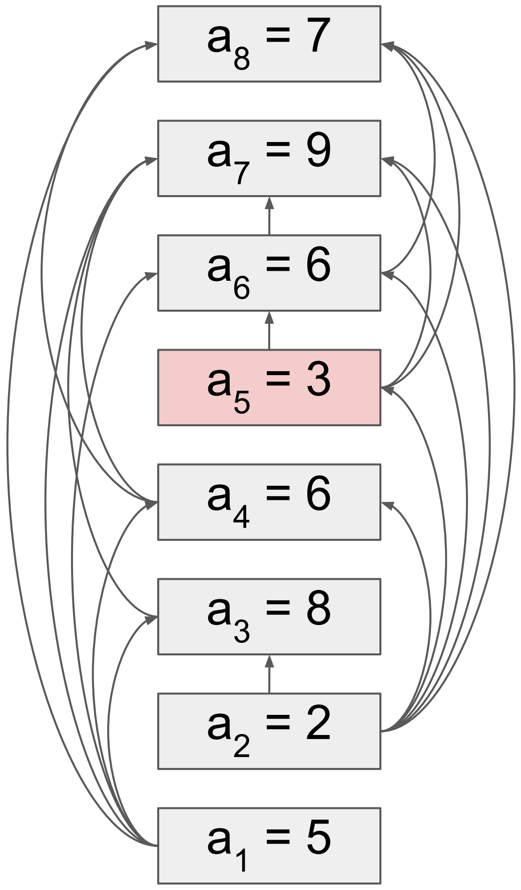
- (a_i, a_j) if $i < j$ and $a_i < a_j$
- Find the longest path in the DAG:
- Use $L(j)$ to denote the length of the longest path (longest increasing subsequence) ending with a_j
    - $L(a_1) = 1$
    - $L(a_2) = 1$
    - $L(a_3) = 2$
    - $L(a_4) = 2$



Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

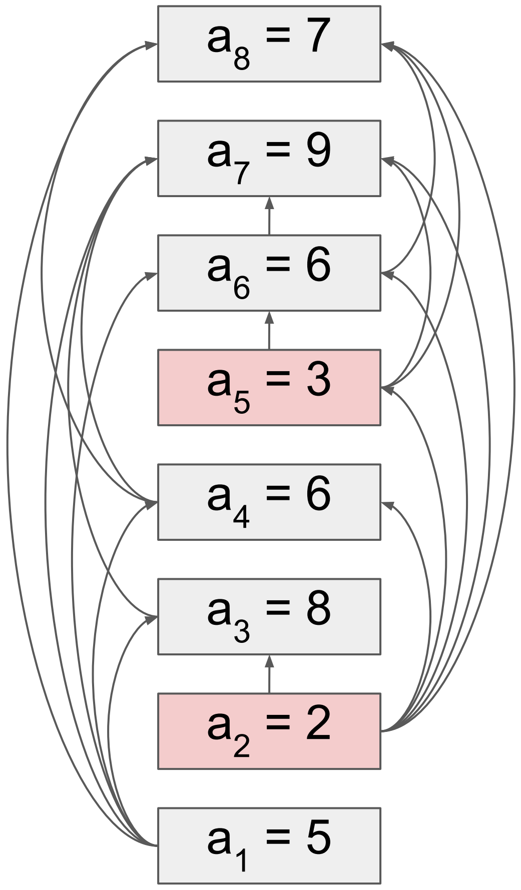
- (a_i, a_j) if $i < j$ and $a_i < a_j$
- Find the longest path in the DAG:
- Use L(j) to denote the length of the longest path (longest increasing subsequence) ending with a_j
  - $L(a_1) = 1$
  - $L(a_2) = 1$
  - $L(a_3) = 2$
  - $L(a_4) = 2$
  - $L(a_5) = ?$



Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

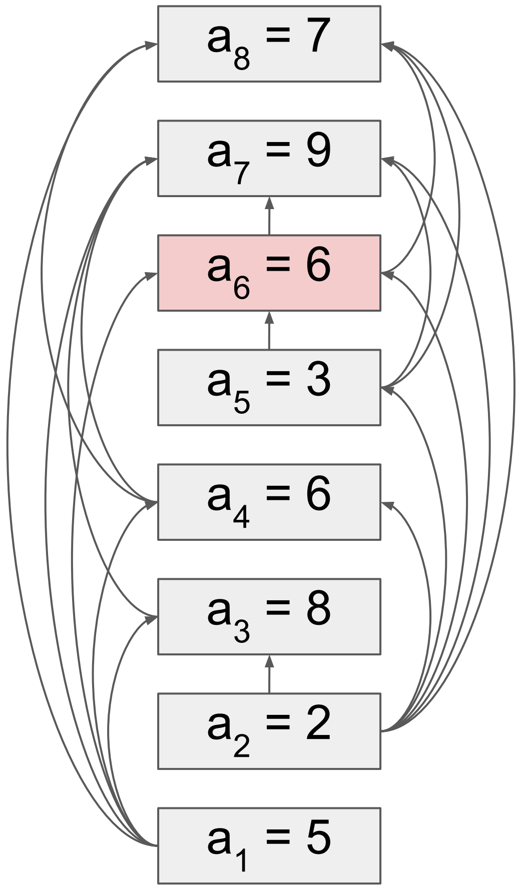
- (a_i, a_j) if $i < j$ and $a_i < a_j$
- Find the longest path in the DAG:
- Use L(j) to denote the length of the longest path (longest increasing subsequence) ending with a_j
    - $L(a_1) = 1$
    - $L(a_2) = 1$
    - $L(a_3) = 2$
    - $L(a_4) = 2$
    - $L(a_5) = 2$



Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

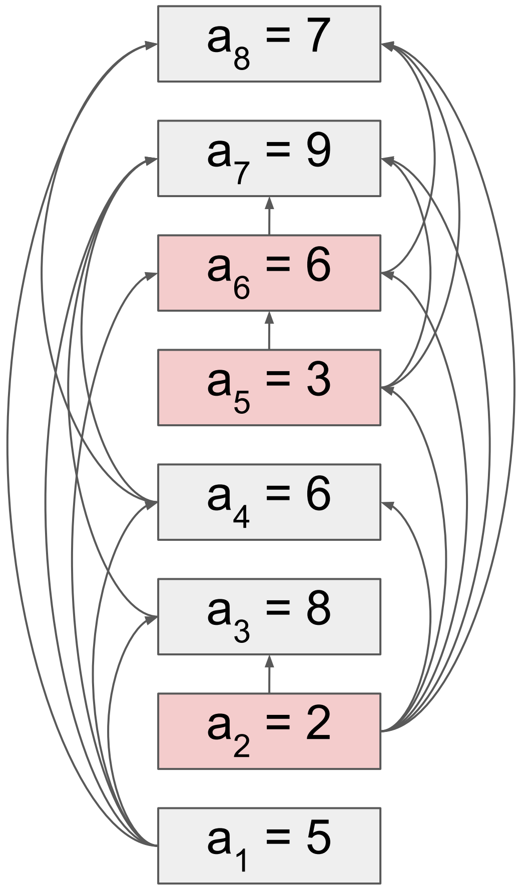
- (a_i, a_j) if $i < j$ and $a_i < a_j$
- Find the longest path in the DAG:
- Use L(j) to denote the length of the longest path (longest increasing subsequence) ending with a_j
    - $L(a_1) = 1$
    - $L(a_2) = 1$
    - $L(a_3) = 2$
    - $L(a_4) = 2$
    - $L(a_5) = 2$
    - $L(a_6) = ?$



Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

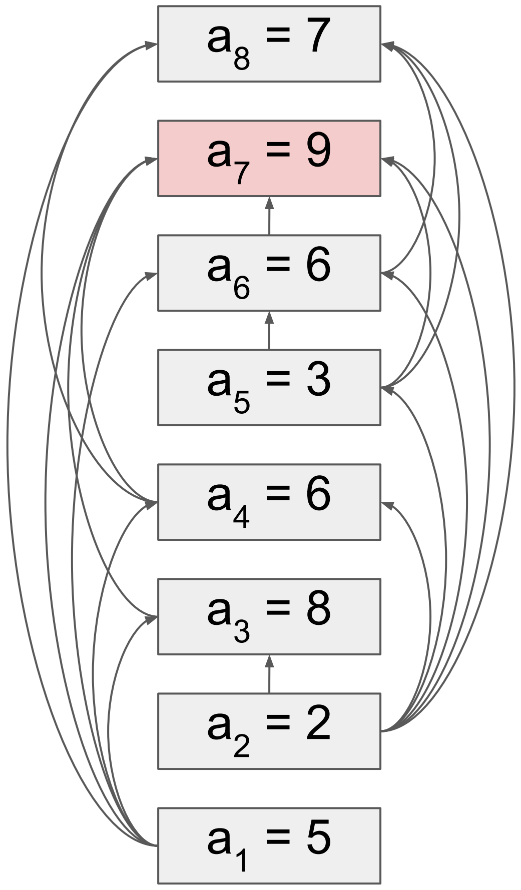
- (a_i, a_j) if $i < j$ and $a_i < a_j$
- Find the longest path in the DAG:
- Use $L(j)$ to denote the length of the longest path (longest increasing subsequence) ending with a_j
    - $L(a_1) = 1$
    - $L(a_2) = 1$
    - $L(a_3) = 2$
    - $L(a_4) = 2$
    - $L(a_5) = 2$
    - $L(a_6) = 3$



Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

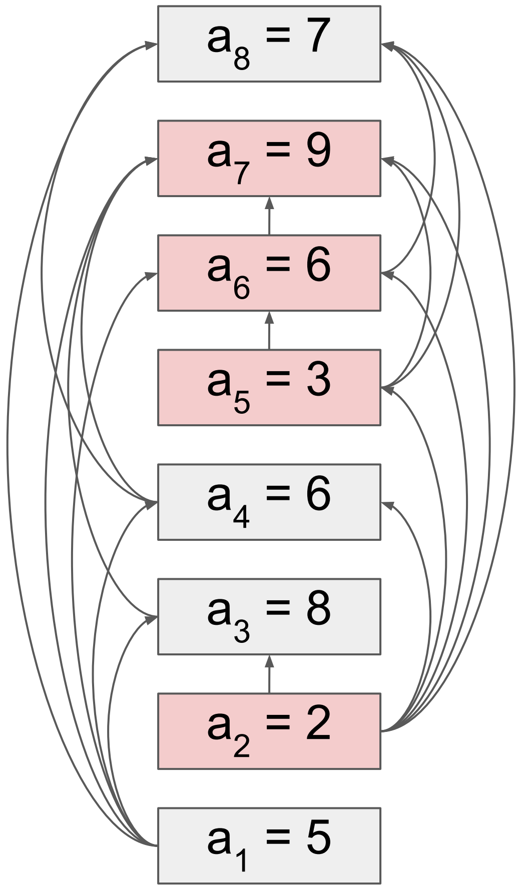
- (a_i, a_j) if $i < j$ and $a_i < a_j$
- Find the longest path in the DAG:
- Use L(j) to denote the length of the longest path (longest increasing subsequence) ending with a_j
    - $L(a_1) = 1$
    - $L(a_2) = 1$
    - $L(a_3) = 2$
    - $L(a_4) = 2$
    - $L(a_5) = 2$
    - $L(a_6) = 3$
    - $L(a_7) = ?$



Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

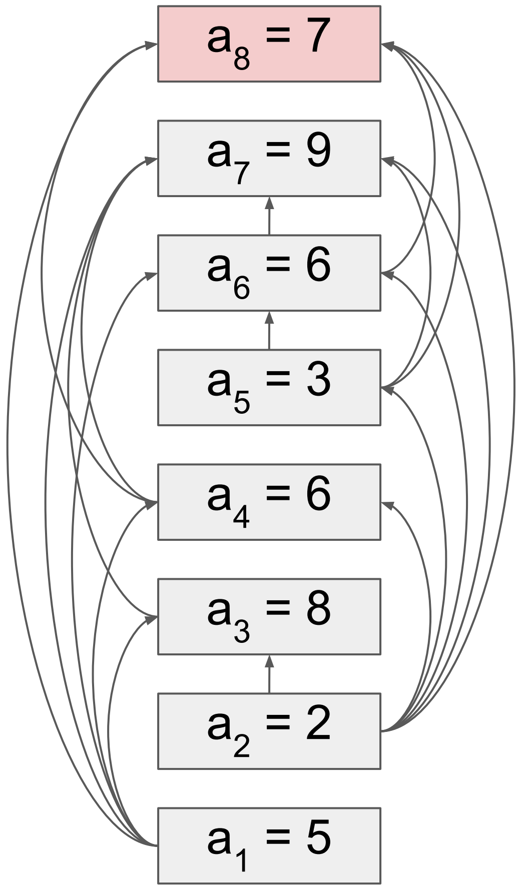
- (a_i, a_j) if $i < j$ and $a_i < a_j$
- Find the longest path in the DAG:
- Use L(j) to denote the length of the longest path (longest increasing subsequence) ending with a_j
    - $L(a_1) = 1$
    - $L(a_2) = 1$
    - $L(a_3) = 2$
    - $L(a_4) = 2$
    - $L(a_5) = 2$
    - $L(a_6) = 3$
    - $L(a_7) = 4$



Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

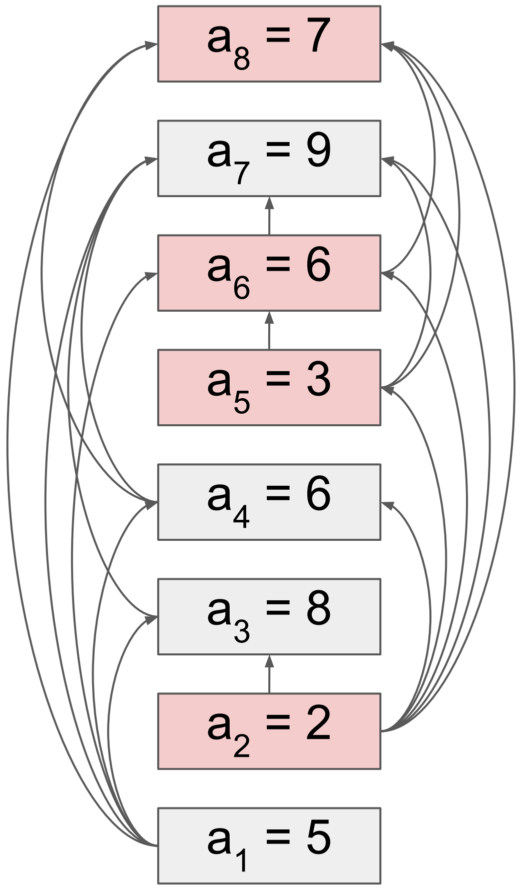
- (a_i, a_j) if $i < j$ and $a_i < a_j$
- Find the longest path in the DAG:
- Use L(j) to denote the length of the longest path (longest increasing subsequence) ending with a_j
    - $L(a_1) = 1$
    - $L(a_2) = 1$
    - $L(a_3) = 2$
    - $L(a_4) = 2$
    - $L(a_5) = 2$
    - $L(a_6) = 3$
    - $L(a_7) = 4$
    - $L(a_8) = ?$



Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

- (a_i, a_j) if $i < j$ and $a_i < a_j$
- Find the longest path in the DAG:
- Use $L(j)$ to denote the length of the longest path (longest increasing subsequence) ending with a_j
    - $L(a_1) = 1$
    - $L(a_2) = 1$
    - $L(a_3) = 2$
    - $L(a_4) = 2$
    - $L(a_5) = 2$
    - $L(a_6) = 3$
    - $L(a_7) = 4$
    - $L(a_8) = 4$

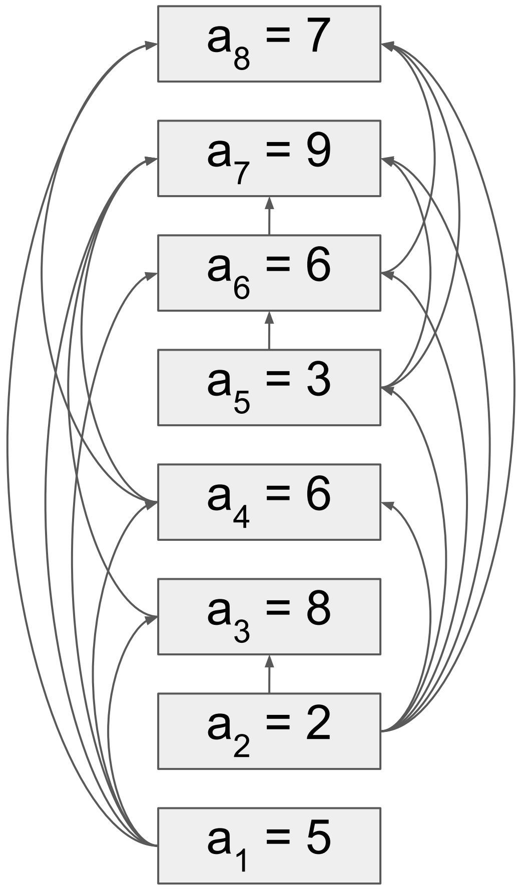


Solving LIS using directed acyclic graph

We can model the longest increasing sequencing using a directed acyclic graph

- (a_i, a_j) if $i < j$ and $a_i < a_j$
- Find the longest path in the DAG:
- Use $L(j)$ to denote the length of the longest path (longest increasing subsequence) ending with a_j

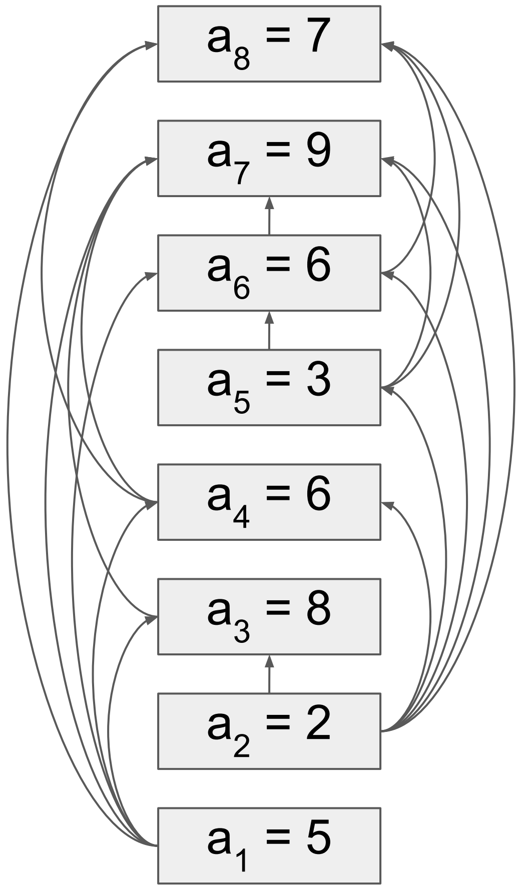
```
def LIS_DAG($DAG\ G = (V, E)$ for $a_1, \ldots, a_n$):
    for $j = 1, \ldots, n$:
        $L(j) = \begin{cases} 1 + \max\{L(i) : (i, j) \in E\} \\ 1 \text{ if no such edge} \end{cases}$;
    return $\max_j L(j)$;
```



Solving LIS using directed acyclic graph

```
def LIS_DAG(GAG $G = (V, E)$ for $a_1, \ldots, a_n$):
    for $j = 1, \ldots, n$:
        $L(j) = \begin{cases} 1 + \max\{L(i) : (i,j) \in E\} \\ 1 \text{ if no such edge} \end{cases}$
    return $\max_j L(j)$;
```

a_j	5	2	8	6	3	6	9	7
j	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4



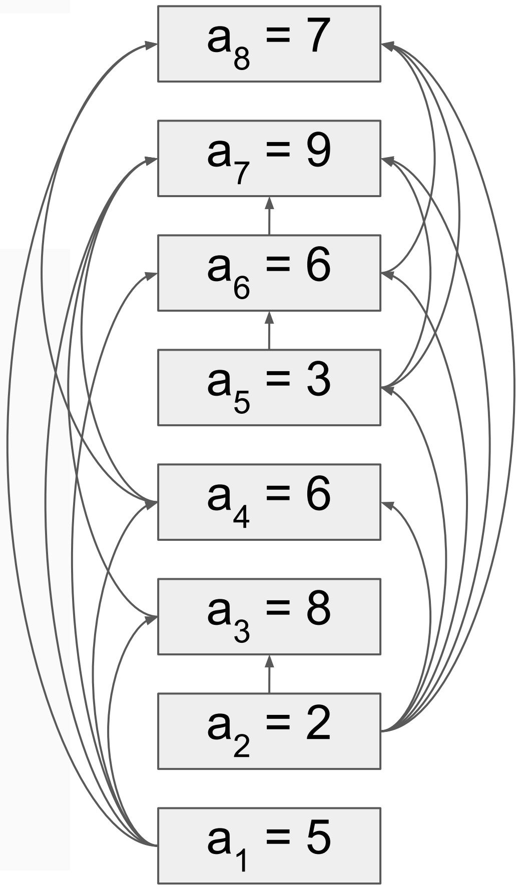
A more direct approach

Do we really need to work on a DAG?

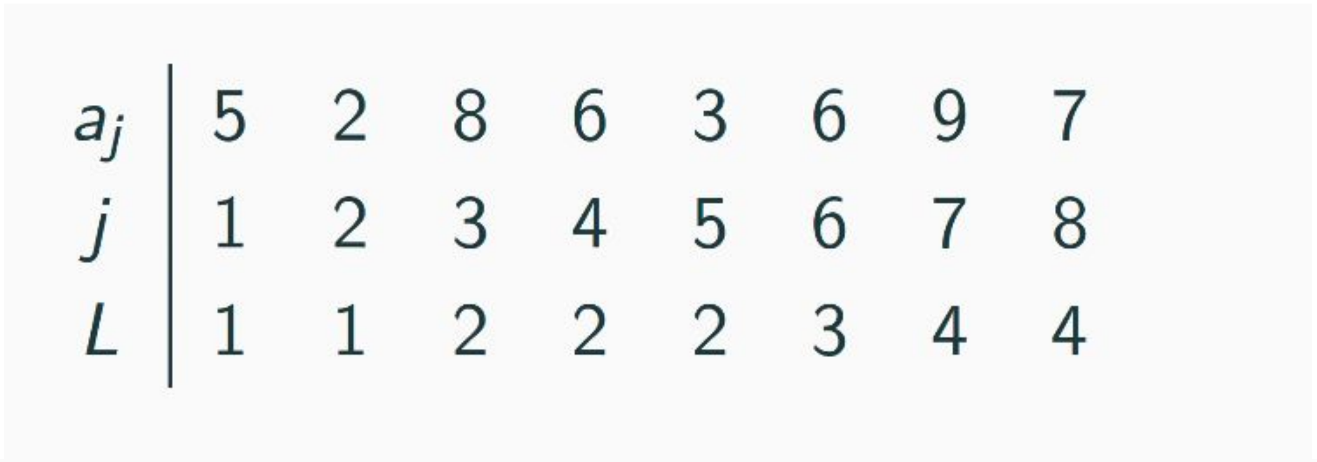
```
def LIS_DAG(GAG $G = (V, E)$ for $a_1, \dots, a_n$):
    for $j = 1, \dots, n$:
        $L(j) = \begin{cases} 1 + \max\{L(i) : (i,j) \in E\} \\ 1 \text{ if no such edge} \end{cases}$;
    return $\max_j L(j)$;
```

A more direct approach:

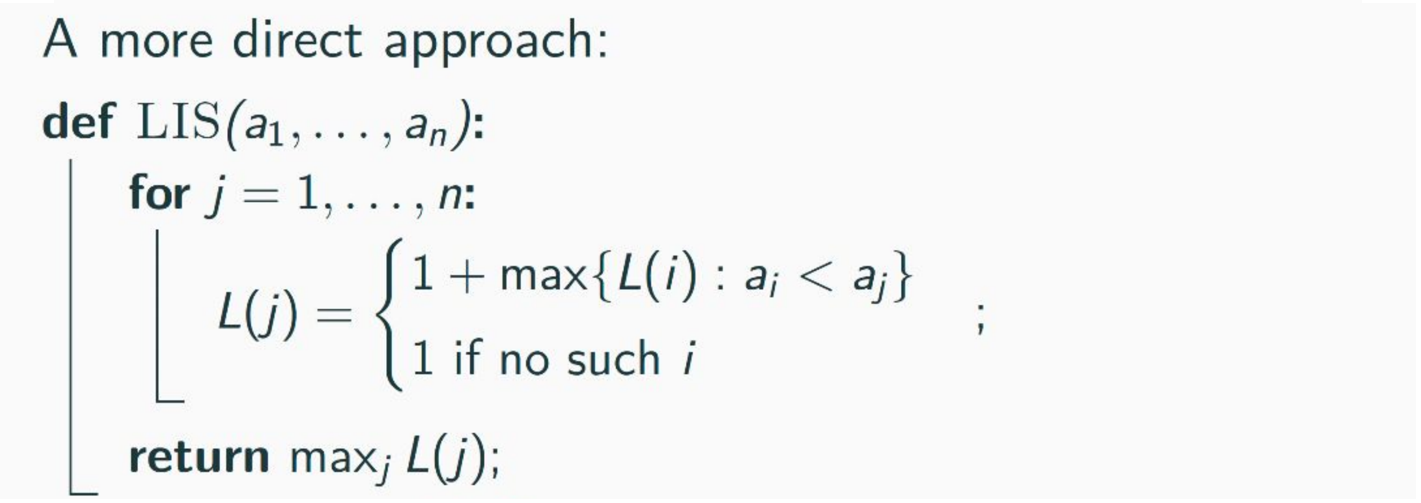
```
def LIS($a_1, \dots, a_n$):
    for $j = 1, \dots, n$:
        $L(j) = \begin{cases} 1 + \max\{L(i) : a_i < a_j\} \\ 1 \text{ if no such } i \end{cases}$;
    return $\max_j L(j)$;
```



A more direct approach



a_j	5	2	8	6	3	6	9	7
j	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4



A more direct approach:

```
def LIS($a_1, \ldots, a_n$):
    for $j = 1, \ldots, n$:
        $L(j) = \begin{cases} 1 + \max\{L(i) : a_i < a_j\} \\ 1 \text{ if no such } i \end{cases}$ ;
    return $\max_j L(j)$;
```

A more direct approach

a_j	5	2	8	6	3	6	9	7
j	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4

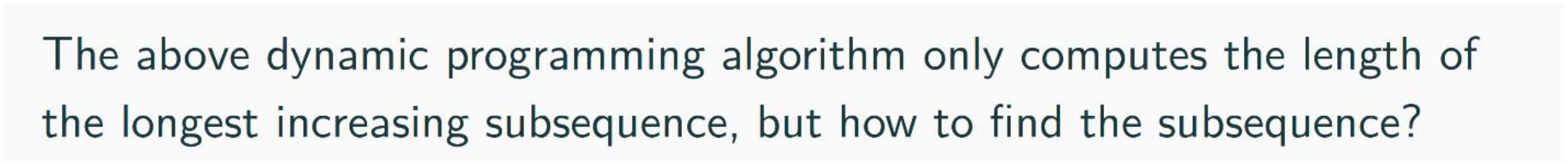
A more direct approach:

```
def LIS($a_1, \ldots, a_n$):
    for $j = 1, \ldots, n$:
        $L(j) = \begin{cases} 1 + \max\{L(i) : a_i < a_j\} \\ 1 \text{ if no such } i \end{cases}$ ;
    return $\max_j L(j)$;
```

Running time: $O(n^2)$

Costs more than greedy: need to check more subproblems

Reconstruction of LIS



The above dynamic programming algorithm only computes the length of the longest increasing subsequence, but how to find the subsequence?

Reconstruction of LIS

The above dynamic programming algorithm only computes the length of the longest increasing subsequence, but how to find the subsequence? We use an additional table to keep track of the subsequence

```
def $\mathrm{LIS}(a_1, \dots, a_n)$:
    for $j = 1, \dots, n$:
        $L(j) = 1$, $\mathrm{prev}(j) = \cdot$;
        for $i = 1, \dots, j$:
            if $a_i < a_j$ and $L(i) + 1 > L(j)$:
                $L(j) = L(i) + 1$, $\mathrm{prev}(j) = i$;
    return $\max_j L(j)$;
```

Reconstruction of LIS

The above dynamic programming algorithm only computes the length of the longest increasing subsequence, but how to find the subsequence?

We use an additional table to keep track of the subsequence

```
def LIS($a_1, \ldots, a_n$):
    for $j = 1, \ldots, n$:
        $L(j) = 1$, $\mathrm{prev}(j) = \cdot$;
        for $i = 1, \ldots, j$:
            if $a_i < a_j$ and $L(i) + 1 > L(j)$:
                $L(j) = L(i) + 1$, $\mathrm{prev}(j) = i$;
    return $\max_j L(j)$;
```

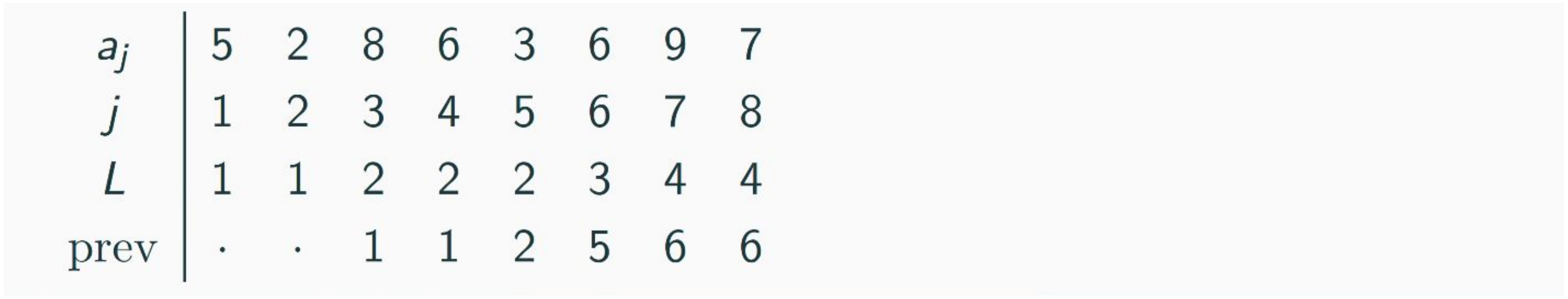
a_j	5	2	8	6	3	6	9	7
j	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4
prev	·	·	1	1	2	5	6	6

Reconstruction of LIS

- L(j) is the length of the longest increasing subsequence ending with a_j
- prev(j) is the index of the previous element in the LIJ ending with a_j

```
max_idx = 8 # the index of the element with max L(j)

lic = [a[max_idx]]
pointer = max_idx
while prev[pointer] != ‘.’:
    pointer = prev[pointer]
    lic = a[pointer] + lic
print(lic)
```



a_j	5	2	8	6	3	6	9	7
j	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4
prev	·	·	1	1	2	5	6	6

Reconstruction of LIS

- L(j) is the length of the longest increasing subsequence ending with a_j
- prev(j) is the index of the previous element in the LIJ ending with a_j

```
max_idx = 8 # the index of the element with max L(j)

lic = [a[max_idx]]
pointer = max_idx
while prev[pointer] != ‘.’:
    pointer = prev[pointer]
    lic = a[pointer] + lic
print(lic)
```

a_j	5	2	8	6	3	6	9	7
j	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4
prev	·	·	1	1	2	5	6	6

```
pointer = 8
a[pointer] = 7
lic = [7]
prev[pointer] = 6
```

Reconstruction of LIS

- L(j) is the length of the longest increasing subsequence ending with a_j
- prev(j) is the index of the previous element in the LIJ ending with a_j

```
max_idx = 8 # the index of the element with max L(j)

lic = [a[max_idx]]
pointer = max_idx
while prev[pointer] != ‘.’:
    pointer = prev[pointer]
    lic = a[pointer] + lic
print(lic)
```

a_j	5	2	8	6	3	6	9	7
j	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4
prev	·	·	1	1	2	5	6	6

```
pointer = 6
a[pointer] = 6
lic = [6, 7]
prev[pointer] = 5
```

Reconstruction of LIS

- L(j) is the length of the longest increasing subsequence ending with a$_j$
- prev(j) is the index of the previous element in the LIJ ending with a$_j$

```
max_idx = 8 # the index of the element with max L(j)

lic = [a[max_idx]]
pointer = max_idx
while prev[pointer] != ‘.’:
    pointer = prev[pointer]
    lic = a[pointer] + lic
print(lic)
```

a_j	5	2	8	6	3	6	9	7
j	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4
prev	·	·	1	1	2	5	6	6

```
pointer = 5
a[pointer] = 3
lic = [3, 6, 7]
prev[pointer] = 2
```

Reconstruction of LIS

- L(j) is the length of the longest increasing subsequence ending with a_j
- prev(j) is the index of the previous element in the LIJ ending with a_j

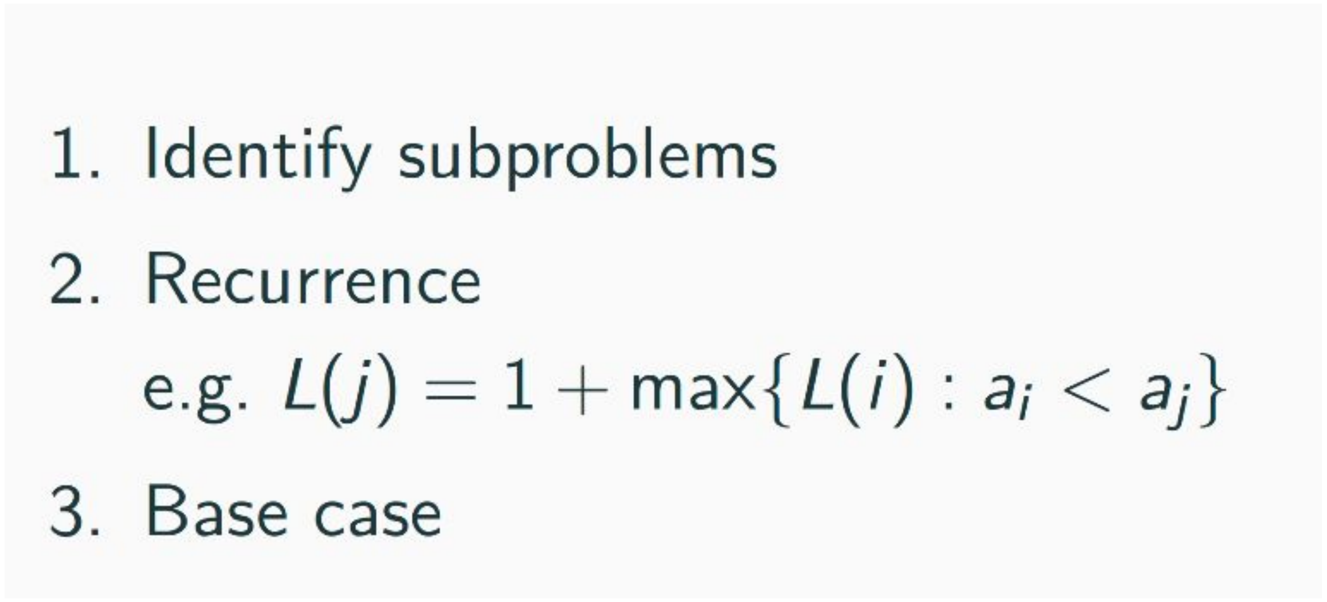
```
max_idx = 8 # the index of the element with max L(j)

lic = [a[max_idx]]
pointer = max_idx
while prev[pointer] != ‘.’:
    pointer = prev[pointer]
    lic = a[pointer] + lic
print(lic)
```

a_j	5	2	8	6	3	6	9	7
j	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4
prev	·	·	1	1	2	5	6	6

```
pointer = 2
a[pointer] = 2
lic = [2, 3, 6, 7]
prev[pointer] = ‘.’
stop
```

Key steps of DP

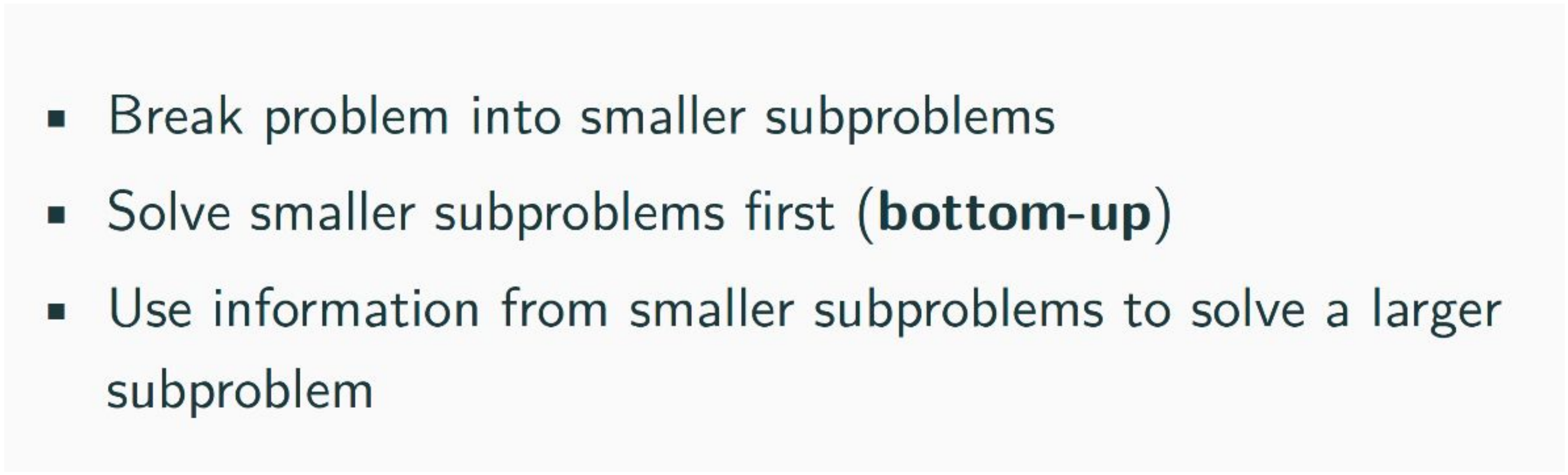


1. Identify subproblems
2. Recurrence

   e.g. $L(j) = 1 + \max\{L(i) : a_i < a_j\}$
3. Base case

Edit distance

Main steps of DP

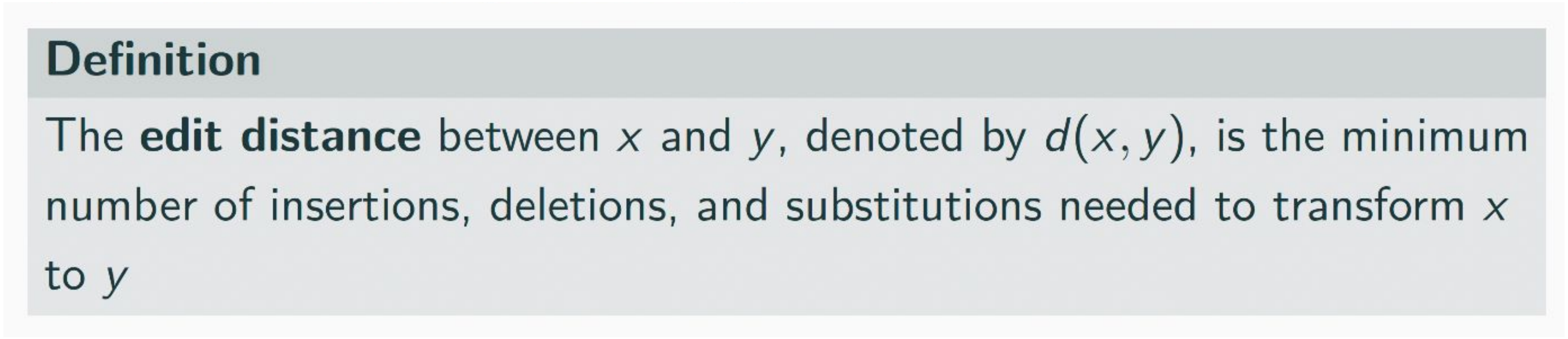


- Break problem into smaller subproblems
- Solve smaller subproblems first (**bottom-up**)
- Use information from smaller subproblems to solve a larger subproblem

Key steps of DP

- Identify subproblems
- Formulate a recurrent way to solve subproblems
- Identify base case for the recurrence

Edit distance



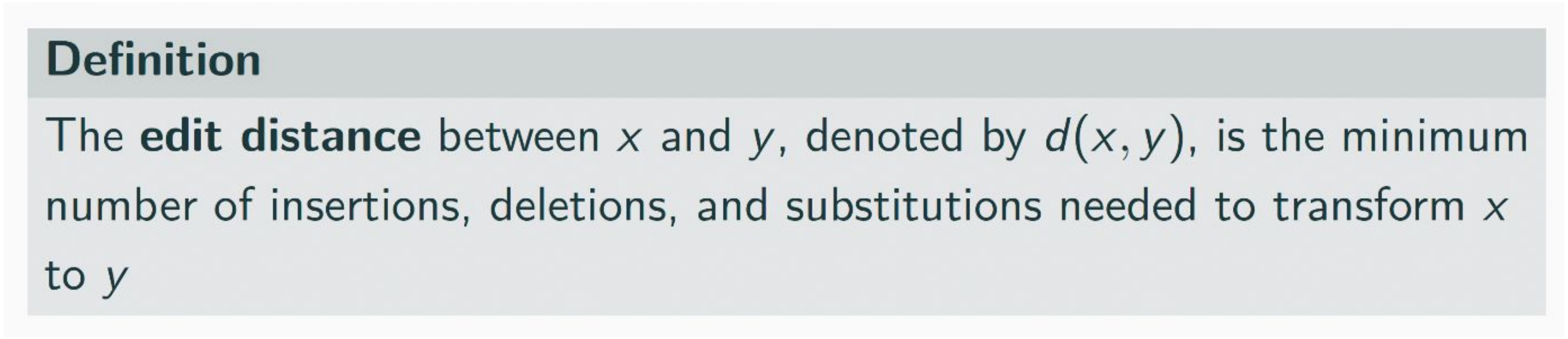
Definition

The **edit distance** between x and y, denoted by $d(x, y)$, is the minimum number of insertions, deletions, and substitutions needed to transform x to y

```
X = PLACE

Y = SPACE
```

Edit distance

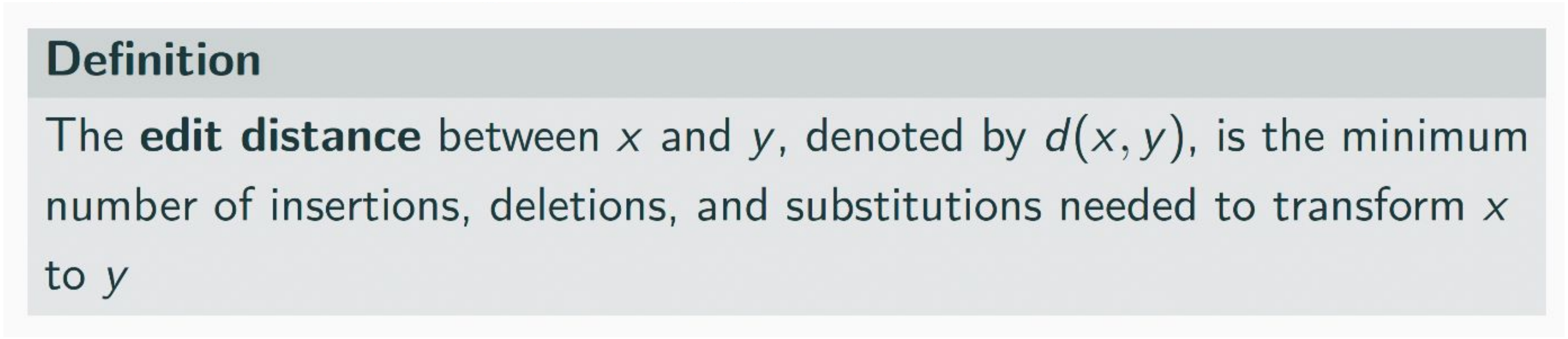

Definition

The **edit distance** between x and y, denoted by $d(x, y)$, is the minimum number of insertions, deletions, and substitutions needed to transform x to y

```
X = PLACE
    ..|||
Y = SPACE
```

```
dist = 2 (two mismatches)
```

Edit distance



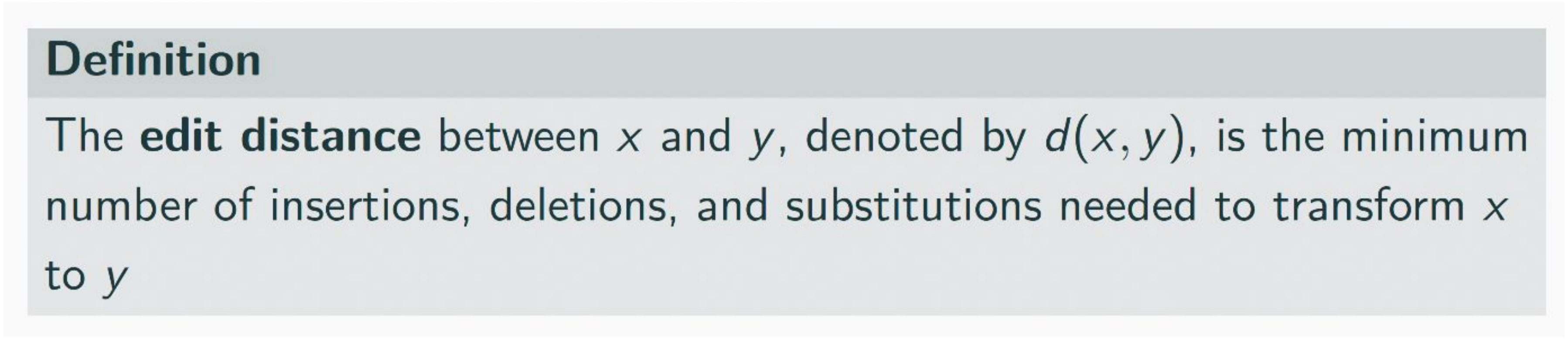
Definition

The **edit distance** between x and y, denoted by $d(x, y)$, is the minimum number of insertions, deletions, and substitutions needed to transform x to y

```
X = PLACE                    X = TOAD
    ..|||
Y = SPACE                    Y = TRADE

dist = 2 (two mismatches)
```

Edit distance



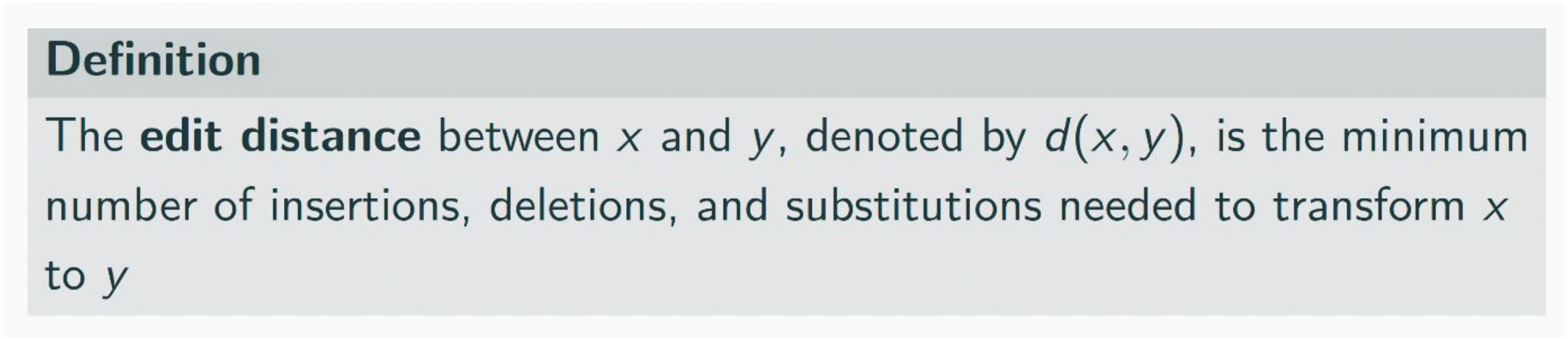
Definition

The **edit distance** between x and y, denoted by $d(x, y)$, is the minimum number of insertions, deletions, and substitutions needed to transform x to y

```
X = PLACE              X = TOAD-
    ..|||                  |.||
Y = SPACE              Y = TRADE

dist = 2 (two mismatches)   dist = 2 (1 mismatch, 1
                            insertion)
```

Edit distance



Definition

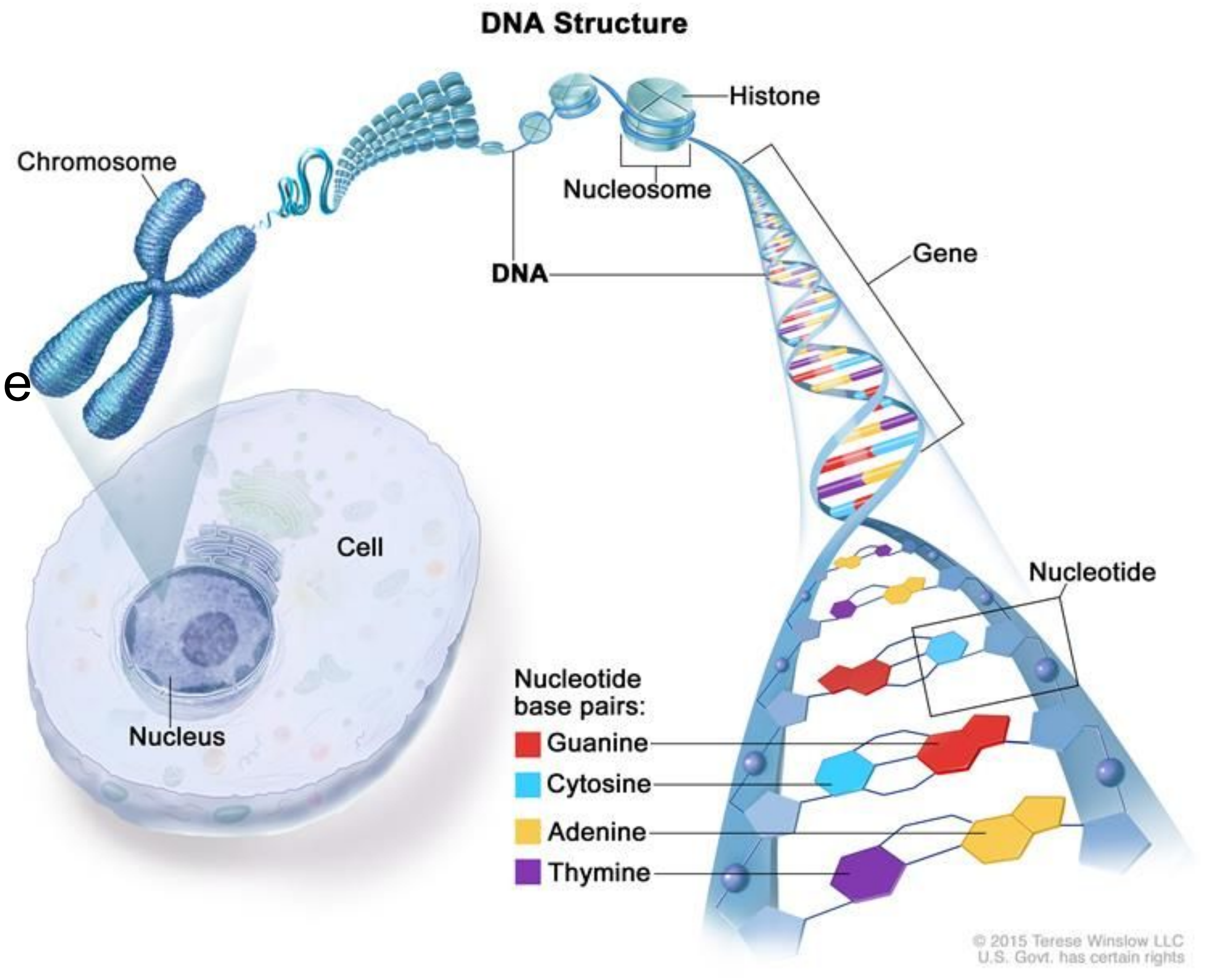
The **edit distance** between x and y, denoted by $d(x, y)$, is the minimum number of insertions, deletions, and substitutions needed to transform x to y

What are applications of edit distance?

DNA alphabet

Genetic information written in a DNA molecule is encoded via four chemical compounds:

- Adenine (A)
- Cytosine (C)
- Guanine (G)
- Thymine (T)

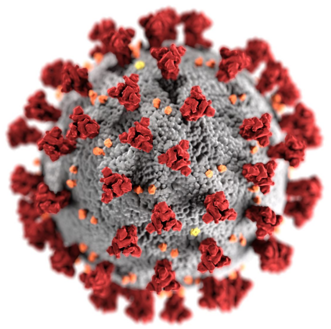


A DNA molecule can be viewed as a string (**DNA sequence**) over an alphabet consisting of four symbols: {A, C, G, T}

Genome = DNA molecules of a given species

Examples of DNA sequence lengths

Species	*T2 phage*	*Escherichia coli*	*Drosophila melanogaster*	*Homo sapiens*	*Paris japonica*
Genome Size	170,000 bp	4.6 million bp	130 million bp	3.2 billion bp	150 billion bp
Common Name	Virus	Bacteria	Fruit fly	Human	Canopy Plant



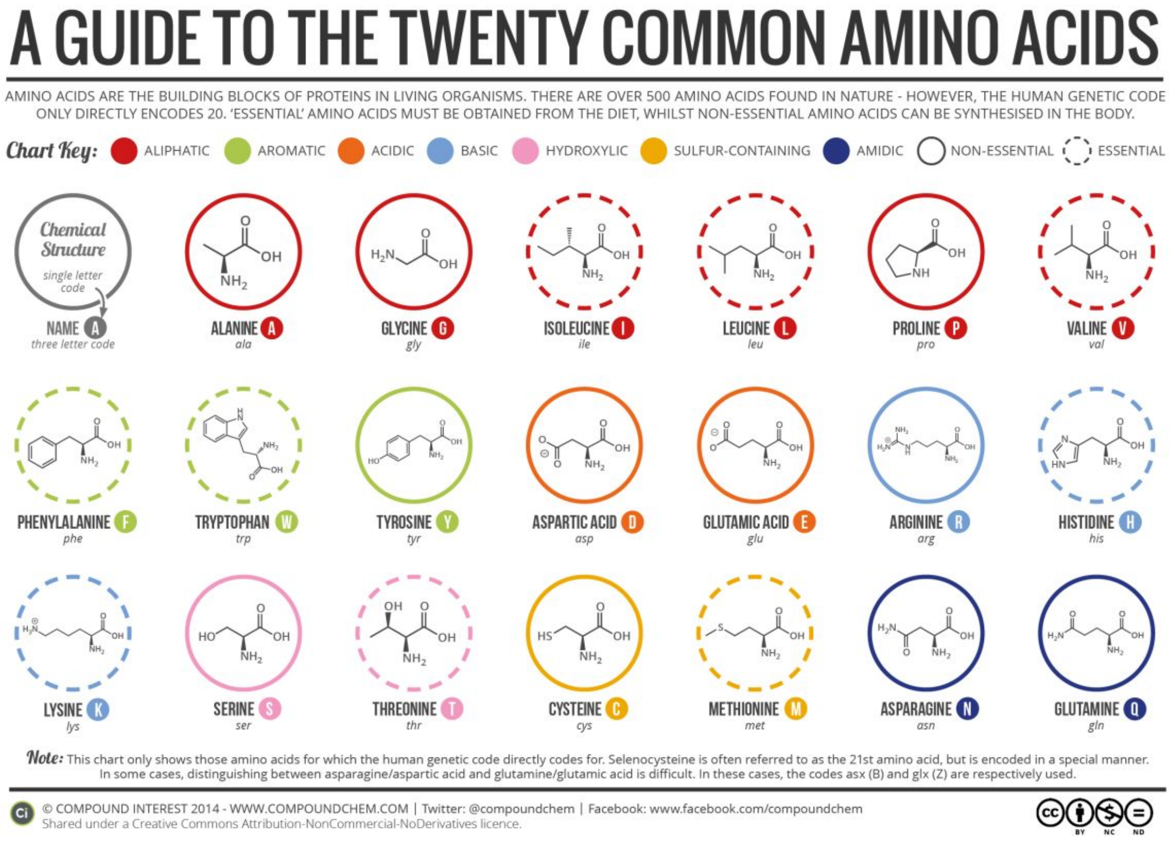
SARS-CoV-2: 30,000

Mammalian genomes are ~2-3 billion nucleotide long

Amino acid alphabet

Proteins consist of 20 chemical compounds (amino acids)

Proteins can be written as strings over an alphabet consisting of 20 letters shown on the right


A GUIDE TO THE TWENTY COMMON AMINO ACIDS

AMINO ACIDS ARE THE BUILDING BLOCKS OF PROTEINS IN LIVING ORGANISMS. THERE ARE OVER 500 AMINO ACIDS FOUND IN NATURE - HOWEVER, THE HUMAN GENETIC CODE ONLY DIRECTLY ENCODES 20. 'ESSENTIAL' AMINO ACIDS MUST BE OBTAINED FROM THE DIET, WHILST NON-ESSENTIAL AMINO ACIDS CAN BE SYNTHESISED IN THE BODY.

Note: This chart only shows those amino acids for which the human genetic code directly codes for. Selenocysteine is often referred to as the 21st amino acid, but is encoded in a special manner. In some cases, distinguishing between asparagine/aspartic acid and glutamine/glutamic acid is difficult. In these cases, the codes asx (B) and glx (Z) are respectively used.