CMPSC 465
Data Structures and Algorithms
Spring 2022

Instructor: Chunhao Wang

Dynamic Programming

Dynamic Programming

Prelude

Dynamic programming vs. Greedy algorithms

- Similarity: optimal substructure

Dynamic programming vs. Greedy algorithms

- Similarity: optimal substructure
- Difference: greedy choice property

Dynamic programming vs. Greedy algorithms

- Similarity: optimal substructure
- Difference: greedy choice property

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

Dynamic programming vs. Greedy algorithms

- 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

Dynamic programming vs. Greedy algorithms

- 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**

General steps for Dynamic Programming

- Break problem into smaller subproblems

General steps for Dynamic Programming

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

General steps for Dynamic Programming

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

Warm-up: Longest increasing subsequence

Problem (Longest increasing subsequence)

Warm-up: Longest increasing subsequence

Problem (Longest increasing subsequence)

Given $a_1, \ldots, a_n \in \mathbb{R}$,

Warm-up: Longest increasing subsequence

Problem (Longest increasing subsequence)

Given $a_1, \ldots, a_n \in \mathbb{R}$*, find the longest subsequence* $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$

Warm-up: 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* $a_{i_1} < a_{i_2} < \cdots a_{i_k}$

Warm-up: Longest increasing subsequence

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

Warm-up: 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

Warm-up: 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

$i_1 = 2, i_2 = 5, i_3 = 6, i_4 = 7$

Solving LIS using DAG

We can model the longest increasing subsequence problem as a directed acyclic graph

Solving LIS using DAG

We can model the longest increasing subsequence problem as a directed acyclic graph

$$
\begin{aligned}
a_8 &= 7\\
a_7 &= 9\\
a_6 &= 6\\
a_5 &= 3\\
a_4 &= 6\\
a_3 &= 8\\
a_2 &= 2\\
a_1 &= 5
\end{aligned}
$$

Solving LIS using DAG

We can model the longest increasing subsequence problem as a directed acyclic graph

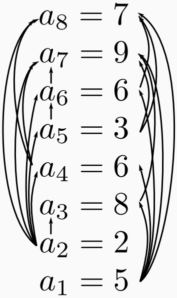
- There is a link $i \to j$ if $a_i < a_j$

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

Solving LIS using DAG

We can model the longest increasing subsequence problem as a directed acyclic graph

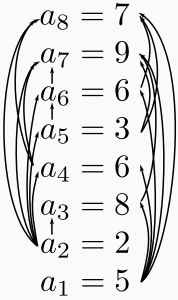
- There is a linke $i \to j$ if $a_i < a_j$



Solving LIS using DAG

We can model the longest increasing subsequence problem as a directed acyclic graph

- There is a linke $i \to j$ if $a_i < a_j$
- Find the longest path in the DAG:

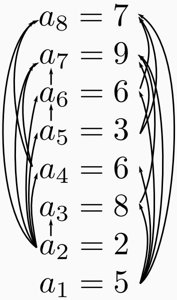


Solving LIS using DAG

We can model the longest increasing subsequence problem as a directed acyclic graph

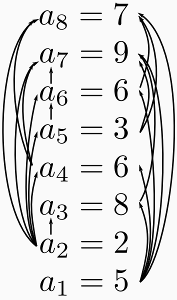
- There is a linke $i \to j$ if $a_i < a_j$
- Find the longest path in the DAG:

Use $L(j)$ to denote the length of the longest path (longest increasing subsequence) ending with a_j



Solving LIS using DAG

We can model the longest increasing subsequence problem as a directed acyclic graph



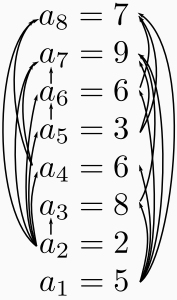
- There is a linke $i \to j$ if $a_i < a_j$
- Find the longest path in the DAG:

  Use $L(j)$ to denote the length of the longest path (longest increasing subsequence) ending with a_i

def $\mathrm{LIS\_DAG}$(*GAG* $G = (V, E)$ *for* $a_1, \ldots, a_n$)**:**

Solving LIS using DAG

We can model the longest increasing subsequence problem as a directed acyclic graph



- There is a linke $i \to j$ if $a_i < a_j$
- Find the longest path in the DAG:

  Use $L(j)$ to denote the length of the longest path (longest increasing subsequence) ending with a_i

def $\mathrm{LIS\_DAG}$(*GAG* $G = (V, E)$ *for* $a_1, \ldots, a_n$)**:**

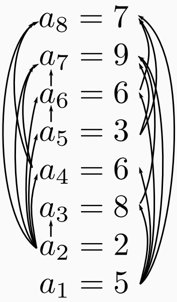
  for $j = 1, \ldots, n$**:**

Solving LIS using DAG

We can model the longest increasing subsequence problem as a directed acyclic graph

- There is a linke $i \to j$ if $a_i < a_j$
- Find the longest path in the DAG:

  Use $L(j)$ to denote the length of the longest path (longest increasing subsequence) ending with a_i



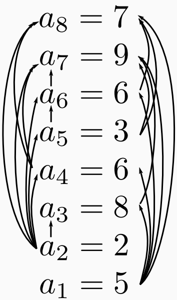
def $\mathrm{LIS\_DAG}$(*GAG* $G = (V, E)$ *for* $a_1, \ldots, a_n$):

  for $j = 1, \ldots, n$:

  $$L(j) = \begin{cases} 1 + \max\{L(i) : (i, j) \in E\} \\ 1 \text{ if no such edge} \end{cases};$$

Solving LIS using DAG

We can model the longest increasing subsequence problem as a directed acyclic graph



- There is a linke $i \to j$ if $a_i < a_j$
- Find the longest path in the DAG:

Use $L(j)$ to denote the length of the longest path (longest increasing subsequence) ending with a_i

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 $\max_j L(j)$;

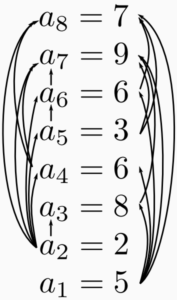
Running example

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 $\max_j L(j)$;



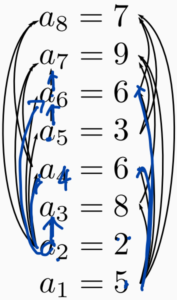
Running example

def $\text{LIS\_DAG}(GAG\ G = (V, E)\ \mathit{for}$
$a_1, \ldots, a_n)$**:**

  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_i	5	2	8	6	3	6	9	7
i	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4

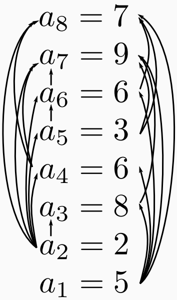
$1 + L(6)$

$1+L(2)$ $1+L(1)$ $1+L(2)$ $1+L(3)$ $1+L(6)$

Running example

def $\mathrm{LIS\_DAG}$*(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_i	5	2	8	6	3	6	9	7
i	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?

A more direct approach

Do we really need to work on a DAG?

def $\mathrm{LIS\_DAG}$*(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 more direct approach

Do we really need to work on a DAG?

def $\mathrm{LIS\_DAG}$*(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 more direct approach:

def LIS*(GAG $G = (V, E)$ for $a_1, \ldots, a_n$)***:**

  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

Do we really need to work on a DAG?

def $\mathrm{LIS\_DAG}$*(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 more direct approach:

def LIS*(GAG* $G = (V, E)$ *for* $a_1, \ldots, a_n$*)***:**

  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: $O(n^2)$

A more direct approach

Do we really need to work on a DAG?

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 more direct approach:

def LIS(*GAG* $G = (V, E)$ *for* $a_1, \ldots, a_n$)**:**

  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 $\max_j L(j)$;

$\Theta(n^2)$

Running time: $O(n^2)$

Costs more than greedy: need to check more subproblems

The actual subsequence

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

The actual subsequence

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

The actual subsequence

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(~~DAG $G = (V, E)$ for~~ $a_1, \ldots, a_n$)**:**

  for $j = 1, \ldots, n$**:**

    $L(j) = 1$, $\boxed{\mathrm{prev}(j)} = \cdot$;

    for $i = 1, \ldots, j$**:**

      if $L(i) > L(j)$**:**

        $\boxed{L(j) = L(i) + 1}$, $\mathrm{prev}(j) = i$;

  return $\max_j L(j)$;

The actual subsequence

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(*GAG* $G = (V, E)$ *for* $a_1, \dots, a_n$)**:**

  for $j = 1, \dots, n$**:**

    $L(j) = 1$, $\text{prev}(j) = \boxed{\cdot}$;

    for $i = 1, \dots, j$**:**

      if $L(i) > L(j)$**:**

        $L(j) = L(i) + 1$, $\text{prev}(j) = i$;

  ~~**return** $\max_j L(j)$;~~

$L(1)+1$ $L(1)+1$ $L(2)+1$ $L(5)+1$

a_i	5	2	8	6	3	6	9	7
i	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4
prev	•	•	1	1	2	5	6	6

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(*GAG* $G = (V, E)$ *for* $a_1, \ldots, a_n$)**:**

```
for j = 1, ..., n:
    L(j) = 1, prev(j) = ·;
    for i = 1, ..., j:
        if L(i) > L(j):
            L(j) = L(i) + 1, prev(j) = i;
return max_j L(j);
```

$j = 3$

$L(3) = \cancel{1}$

$L(3) = 1 + 1 = 2$

$pre(3) = 1$

$L(2)+1$

a_i	5	2	8	6	3	6	9	7
i	1	2	3	4	5	6	7	8
L	1	1	2	2	2	3	4	4
prev	·	·	1	1	2	5	6	6

Key steps to design DP algorithms

1. Identify subproblems

Key steps to design DP algorithms

1. Identify subproblems
2. Recurrence

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

Key steps to design DP algorithms

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