CS 341: ALGORITHMS

Lecture 7: dynamic programming I

Readings: see website

Trevor Brown

https://student.cs.uwaterloo.ca/~cs341

trevor.brown@uwaterloo.ca
FINISHING UP GREEDY
INTERVAL COLOURING
**PROBLEM: INTERVAL COLOURING**

*Instance:* A set $\mathcal{A} = \{A_1, \ldots, A_n\}$ of intervals.

For $1 \leq i \leq n$, $A_i = [s_i, f_i)$, where $s_i$ is the **start time** of interval $A_i$ and $f_i$ is the **finish time** of $A_i$.

*Feasible solution:* A $c$-colouring is a mapping $\text{col} : \mathcal{A} \rightarrow \{1, \ldots, c\}$ that assigns each interval a **colour** such that two intervals receiving the same colour are always disjoint.

*Find:* A $c$-colouring of $\mathcal{A}$ with the **minimum number of colours**.

Example:

- 7 intervals, 7 colours.
  - Feasible, but not optimal.
MORE EXAMPLES

Example

Not feasible!

Example

7 intervals, 6 colours. Feasible, but not optimal

Example

Same color, but disjoint. OK!

Example

7 intervals, 2 colours. Optimal
Greedy Strategies for Interval Colouring

As usual, we consider the intervals one at a time.

At a given point in time, suppose we have coloured the first $i < n$ intervals using $d$ colours.

We will colour the $(i + 1)$st interval with any permissible colour. If it cannot be coloured using any of the existing $d$ colours, then we introduce a new colour and $d$ is increased by 1.

Question: In what order should we consider the intervals?
We will colour the \((i + 1)\)st interval with any permissible colour. If it cannot be coloured using any of the existing \(d\) colours, then we introduce a new colour and \(d\) is increased by 1.

Consider intervals in the order they are given in the input: \(A_1 \ldots A_{10}\)
EXAMPLE:
ORDER MATTERS!
**EXAMPLE:**

**ORDER MATTERS!**

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_2$</td>
<td>1</td>
</tr>
<tr>
<td>$A_3$</td>
<td>1</td>
</tr>
<tr>
<td>$A_4$</td>
<td>1</td>
</tr>
<tr>
<td>$A_5$</td>
<td>1</td>
</tr>
<tr>
<td>$A_6$</td>
<td>1</td>
</tr>
<tr>
<td>$A_7$</td>
<td>1</td>
</tr>
<tr>
<td>$A_8$</td>
<td>1</td>
</tr>
<tr>
<td>$A_9$</td>
<td>1</td>
</tr>
<tr>
<td>$A_{10}$</td>
<td>1</td>
</tr>
</tbody>
</table>

$x$-axis

0 2 4 6 8 10 12 14 16 18 20
**EXAMPLE:**

**ORDER MATTERS!**

<table>
<thead>
<tr>
<th>A_1</th>
<th>A_2</th>
<th>A_3</th>
<th>A_4</th>
<th>A_5</th>
<th>A_6</th>
<th>A_7</th>
<th>A_8</th>
<th>A_9</th>
<th>A_10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**x-axis**

0  2  4  6  8  10  12  14  16  18  20
EXAMPLE:
ORDER
MATTERS!
EXAMPLE: ORDER MATTERS!
**EXAMPLE:**

**ORDER MATTERS!**

| $A_1$ | 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| $A_2$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| $A_3$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| $A_4$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| $A_5$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| $A_6$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| $A_7$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| $A_8$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| $A_9$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| $A_{10}$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

*x-axis*
EXAMPLE:
ORDER MATTERS!

```
1
A2
1
A3
A4
2
A5
2
A6
3
A7
2
A8
A9
A10
```
**EXAMPLE: ORDER MATTERS!**

<table>
<thead>
<tr>
<th>A_1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_2</td>
<td>1</td>
</tr>
<tr>
<td>A_3</td>
<td>2</td>
</tr>
<tr>
<td>A_4</td>
<td>1</td>
</tr>
<tr>
<td>A_5</td>
<td>2</td>
</tr>
<tr>
<td>A_6</td>
<td>3</td>
</tr>
<tr>
<td>A_7</td>
<td>2</td>
</tr>
<tr>
<td>A_8</td>
<td>4</td>
</tr>
<tr>
<td>A_9</td>
<td></td>
</tr>
<tr>
<td>A_{10}</td>
<td></td>
</tr>
</tbody>
</table>

**x-axis**

0 2 4 6 8 10 12 14 16 18 20
EXAMPLE:
ORDER MATTERS!
EXAMPLE:
ORDER
MATTERS!

Used 4 colours

Can we do better?
**EXAMPLE:**

**ORDER MATTERS!**

Pre-sort intervals by **increasing start time**!
**EXAMPLE:**

ORDER MATTERS!

Pre-sort intervals by increasing start time!
EXAMPLE: ORDER MATTERS!
EXAMPLE:
ORDER
MATTERS!

<table>
<thead>
<tr>
<th>A_1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_2</td>
<td>2</td>
</tr>
<tr>
<td>A_3</td>
<td></td>
</tr>
<tr>
<td>A_4</td>
<td></td>
</tr>
<tr>
<td>A_5</td>
<td></td>
</tr>
<tr>
<td>A_6</td>
<td></td>
</tr>
<tr>
<td>A_7</td>
<td></td>
</tr>
<tr>
<td>A_8</td>
<td></td>
</tr>
<tr>
<td>A_9</td>
<td></td>
</tr>
<tr>
<td>A_{10}</td>
<td></td>
</tr>
</tbody>
</table>

x-axis
EXAMPLE: ORDER MATTERS!
**EXAMPLE:**

**ORDER MATTERS!**

| A₁ | 1 |
| A₂ | 2 |
| A₃ | 3 |
| A₄ |  |
| A₅ |  |
| A₆ |  |
| A₇ |  |
| A₈ |  |
| A₉ |  |
| A₁₀|   |

---

**x-axis**

0  2  4  6  8  10  12  14  16  18  20
EXAMPLE: ORDER MATTERS!
EXAMPLE: ORDER MATTERS!
EXAMPLE: ORDER MATTERS!
**EXAMPLE:**

**ORDER MATTERS!**

<table>
<thead>
<tr>
<th>A_1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_2</td>
<td>2</td>
</tr>
<tr>
<td>A_3</td>
<td>3</td>
</tr>
<tr>
<td>A_4</td>
<td></td>
</tr>
<tr>
<td>A_5</td>
<td></td>
</tr>
<tr>
<td>A_6</td>
<td></td>
</tr>
<tr>
<td>A_7</td>
<td></td>
</tr>
<tr>
<td>A_8</td>
<td></td>
</tr>
<tr>
<td>A_9</td>
<td></td>
</tr>
<tr>
<td>A_{10}</td>
<td></td>
</tr>
</tbody>
</table>

**x-axis**

0 2 4 6 8 10 12 14 16 18 20
EXAMPLE: ORDER MATTERS!
EXAMPLE: ORDER MATTERS!

Used 3 colours

Turns out to be optimal…
Consider interval $A_i = (s_i, f_i)$. If $s_i \geq \text{finish}[c]$, then we can give $A_i$ colour $c$ without breaking feasibility.

If we didn’t reuse a colour, use a new colour.

We reused a colour

Check if we can reuse any colour $c$ in $1..d$
EXAMPLE: RUNNING GREEDY

Initial state

A1
A2
A3
A4
A5
A6
A7
A8
A9
A10

x-axis
**Example: Running Greedy**

**Code before the loop:** just assign colour 1

<table>
<thead>
<tr>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
<th>A6</th>
<th>A7</th>
<th>A8</th>
<th>A9</th>
<th>A10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $i=1$
- $d=1$
- $\text{finish}[1] = 32$

[Diagram showing a timeline with grid and bars indicating the progression of the algorithm]
Is \( \text{finish}[1] \leq s_2 \) ?

No. We cannot reuse colour 1.

Cannot reuse any colour. Create a new one!

EXAMPLE:
RUNNING GREEDY

While loop over \( c \).
Check if we can reuse a color in \( 1..d \).
EXAMPLE: RUNNING GREEDY

While loop over \( c \). Check if we can reuse a color in \( 1..d \)

\[
\text{Is } \text{finish}[1] \leq s_2? \\
\text{No. We cannot reuse colour 1.} \\
\text{Cannot reuse any colour. Create a new one!}
\]
While loop over c. Check if we can reuse a color in 1..d.

**EXAMPLE:** RUNNING GREEDY

<table>
<thead>
<tr>
<th>A1</th>
<th>1</th>
<th>finish[1] =</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td>2</td>
<td>finish[2] =</td>
</tr>
<tr>
<td>A3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is $finish[1] \leq s_3$?
No. We cannot reuse colour 1.

Is $finish[2] \leq s_3$?
No. We cannot reuse colour 2.

Cannot reuse any colour. Create new one.
**EXAMPLE:**

**RUNNING GREEDY**

While loop over $c$. Check if we can reuse a color in $1..d$

<table>
<thead>
<tr>
<th></th>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_3$</th>
<th>$A_4$</th>
<th>$A_5$</th>
<th>$A_6$</th>
<th>$A_7$</th>
<th>$A_8$</th>
<th>$A_9$</th>
<th>$A_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i=3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**EXAMPLE: RUNNING GREEDY**

<table>
<thead>
<tr>
<th>$A_i$</th>
<th>$c_1$</th>
<th>$c_2$</th>
<th>$c_3$</th>
<th>$c_4$</th>
<th>$c_5$</th>
<th>$c_6$</th>
<th>$c_7$</th>
<th>$c_8$</th>
<th>$c_9$</th>
<th>$c_{10}$</th>
<th>$c_{11}$</th>
<th>$c_{12}$</th>
<th>$c_{13}$</th>
<th>$c_{14}$</th>
<th>$c_{15}$</th>
<th>$c_{16}$</th>
<th>$c_{17}$</th>
<th>$c_{18}$</th>
<th>$c_{19}$</th>
<th>$c_{20}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_2$</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_3$</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_6$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_7$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_8$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_9$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{10}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Is $finish[1] \leq s_4$?**

Yes. We can reuse colour 1.
EXAMPLE: RUNNING GREEDY

While loop over \( c \).
Check if we can reuse a color in \( 1..d \).

Is \( \text{finish}[1] \leq s_4 \) ?
Yes. We can reuse colour 1.

\[
\begin{array}{c|c}
\text{A}_1 & 1 \\
\text{A}_2 & 2 \\
\text{A}_3 & 3 \\
\text{A}_4 & \text{A}_5 \\
\text{A}_6 & \text{A}_7 \\
\text{A}_8 & \text{A}_9 \\
\text{A}_{10} & \\
\end{array}
\]
EXAMPLE: RUNNING GREEDY

While loop over $c$. Check if we can reuse a color in $1..d$

Is $finish[1] \leq s_5$?
No. We cannot reuse colour 1.

Is $finish[2] \leq s_5$?
No. We cannot reuse colour 2.

Is $finish[3] \leq s_5$?
Yes. We can reuse colour 3.
**EXAMPLE:** RUNNING GREEDY

<table>
<thead>
<tr>
<th></th>
<th>A_1</th>
<th>A_2</th>
<th>A_3</th>
<th>A_4</th>
<th>A_5</th>
<th>A_6</th>
<th>A_7</th>
<th>A_8</th>
<th>A_9</th>
<th>A_10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

While loop over c. Check if we can reuse a color in 1..d.

Is $\text{finish}[1] \leq s_5$?  
No. We **cannot** reuse colour 1.

Is $\text{finish}[2] \leq s_5$?  
No. We **cannot** reuse colour 2.

Is $\text{finish}[3] \leq s_5$?  
Yes. We **can** reuse colour 3.

And so on, and so forth…
Correctness of the Algorithm

The correctness of this greedy algorithm can be proven inductively as well as by a “slick” method—we give the “slick” proof:
Let $D$ denote the number of colours used by the algorithm.
Let $F_D$ be the first interval that has colour $D$
Let $F_D$ be the \textbf{first} interval that has \textcolor{red}{colour} $D$

We prove $F_D$ overlaps $D-1$ other intervals at a single point in time
Let $F_D$ be the first interval that has colour $D$.
Let $L_c$ be the last interval that has colour $c$ and starts before $F_D$.

We prove $F_D$ overlaps every interval $L_c$ for all $c < D$.

<table>
<thead>
<tr>
<th>$L_1$</th>
<th>$L_2$</th>
<th>$L_3$</th>
<th>$L_4$</th>
<th>$L_5$</th>
<th>$L_{D-1}$</th>
<th>$F_D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>...</td>
<td>D-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D</td>
</tr>
</tbody>
</table>

Let’s argue $L_1$ overlaps $F_D$.
Note $L_1$ must exist (otherwise greedy would just use colour 1 for $F_D$).
And $\text{finish}[L_1]$ must be after $F_D$ starts (same reason).

Same argument applies to $L_2, \ldots, L_{D-1}$.

So, $F_D$ overlaps $D - 1$ intervals!

Moreover, every interval in $\{L_1, \ldots, L_{D-1}\}$ contains the starting time of $F_D$.

So, we must use $D$ colours!
Preprocess(A[1..n])
sort A by increasing start time
let s[1..n] be the start times in A
let f[1..n] be the finish times in A
return GreedyIntervalColouring(s, f)

GreedyIntervalColouring(s[1..n], f[1..n])
d = 1
colour[1] = 1
finish[1] = f[1]

for i = 2..n
    reused = false
    for c = 1..d
        if finish[c] <= s[i] then
            colour[i] = c
            finish[c] = f[i]
            reused = true
            break
    if not reused then
        d++
        colour[i] = d
        finish[d] = f[i]

return d

\(O(n \log n)\)

\(O(n)\) iterations

\(O(d)\) iterations...

\(O(n) \log n + nd\)

Could be \(O(n \log n)\) if only a constant number of colours are needed
(or even \(\log n\) colours!)

Could be \(O(n^2)\) if \(n\) colours are needed

Most accurate complexity statement is \(\Theta(n \log n + nD)\) where \(D\) is # colours used

What inefficiencies exist in this algorithm? Could we make it faster with clever data structure usage?
IMPROVING THIS ALGORITHM

• Current greedy algorithm:
  • For each interval $A_i$, compare its start time $s_i$ with the $finish[c]$ times of all colours introduced so-far
  • Why? Looking for some $finish[c]$ time that is earlier than $s_i$
  • We are doing linear search... Can we do better?
  • Use a priority queue to keep track of the earliest $finish[c]$ at all times in the algorithm
    • Then we only need to look at minimum element
### EXAMPLE: HEAP-BASED ALGORITHM

**Min element:** NULL

**Heap**

<table>
<thead>
<tr>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
<th>A₄</th>
<th>A₅</th>
<th>A₆</th>
<th>A₇</th>
<th>A₈</th>
<th>A₉</th>
<th>A₁₀</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image-url" alt="Heap Diagram" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**x-axis**

0  2  4  6  8  10  12  14  16  18  20
EXAMPLE: 
HEAP-BASED 
ALGORITHM

Min element: NULL

Heap

Iteration i=1
Check heap minimum
Empty, so a new colour is needed
**EXAMPLE: HEAP-BASED ALGORITHM**

<table>
<thead>
<tr>
<th>A1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td></td>
</tr>
<tr>
<td>A7</td>
<td></td>
</tr>
<tr>
<td>A8</td>
<td></td>
</tr>
<tr>
<td>A9</td>
<td></td>
</tr>
<tr>
<td>A10</td>
<td></td>
</tr>
</tbody>
</table>

**Min element:** finish at time 3

**Heap:** finish at time 3

- **Iteration i=1:** Check heap minimum
- **Empty, so a new colour is needed**

- **Check heap minimum**
- **Empty, so a new colour is needed**

- **Check heap minimum**
- **Empty, so a new colour is needed**

- **Check heap minimum**
- **Empty, so a new colour is needed**

- **Check heap minimum**
- **Empty, so a new colour is needed**

- **Check heap minimum**
- **Empty, so a new colour is needed**

- **Finish at time 3**
### Example: Heap-Based Algorithm

**Min element:** finish at time 3

**Heap:** finish at time 3

<table>
<thead>
<tr>
<th>Iteration i=2</th>
<th>Check heap minimum</th>
<th>Check if finish time 3 is before $s_2$</th>
<th>No. New colour!</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**x-axis**

0  2  4  6  8  10  12  14  16  18  20

**Check heap minimum**

**Check if finish time 3 is before $s_2$**

**No. New colour!**
EXAMPLE: HEAP-BASED ALGORITHM

Min element: finish at time 3
Heap: finish at time 3
finish at time 7

Iteration i=2 | Check heap minimum | Check if finish time 3 is before \( s_2 \) | No. New colour!

<table>
<thead>
<tr>
<th>A_1</th>
<th>1</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>A_2</th>
<th>2</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A_4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A_6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A_8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A_10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Min element: finish at time 3

Finish at time 3

Finish at time 7

x-axis
EXAMPLE: HEAP-BASED ALGORITHM

Min element: finish at time 3
Heap finish at time 3
finish at time 7

<table>
<thead>
<tr>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
<th>A6</th>
<th>A7</th>
<th>A8</th>
<th>A9</th>
<th>A10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Iteration i=3
Check heap minimum
Check if finish time 3 is before \( s_3 \)
No. New colour!

x-axis
### Example: Heap-Based Algorithm

#### Min element:
- Finish at time 3

#### Heap:
- Finish at time 3
- Finish at time 7
- Finish at time 5

<table>
<thead>
<tr>
<th>Iteration i=3</th>
<th>Check heap minimum</th>
<th>Check if finish time 3 is before $s_3$</th>
<th>No. New colour!</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_2 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_3 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_{10}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**x-axis**

0  2  4  6  8 10 12 14 16 18 20
**EXAMPLE: HEAP-BASED ALGORITHM**

<table>
<thead>
<tr>
<th>Heap</th>
<th>Min element:</th>
<th>finish at time 3</th>
<th>finish at time 3</th>
<th>finish at time 7</th>
<th>finish at time 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_{10}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Iteration i=4**

Check heap minimum

Check if finish time 3 is before \( s_4 \)

Yes. **Reuse** colour, **deleteMin** and **insert** new finish time into heap!

```
<table>
<thead>
<tr>
<th></th>
<th>A_1</th>
<th>A_2</th>
<th>A_3</th>
<th>A_4</th>
<th>A_5</th>
<th>A_6</th>
<th>A_7</th>
<th>A_8</th>
<th>A_9</th>
<th>A_{10}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
</tbody>
</table>
```

x-axis
EXAMPLE: HEAP-BASED ALGORITHM

Min element: finish at time 5

Heap

finish at time 7 finish at time 5

Check heap minimum

Check if finish time 3 is before $s_4$

Yes. **Reuse** colour, deleteMin and insert new finish time into heap!
EXAMPLE: HEAP-BASED ALGORITHM

Min element: finish at time 5

Heap

<table>
<thead>
<tr>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
<th>A6</th>
<th>A7</th>
<th>A8</th>
<th>A9</th>
<th>A10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Iteration i=4
Check heap minimum
Check if finish time 3 is before $s_4$

Yes. **Reuse** colour, **deleteMin** and insert new finish time into heap!
EXAMPLE:
HEAP-BASED ALGORITHM

Min element: finish at time 5

Heap

<table>
<thead>
<tr>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
<th>A6</th>
<th>A7</th>
<th>A8</th>
<th>A9</th>
<th>A10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Iteration i=5

Check heap minimum

Check if finish time 5 is before $s_5$

Yes. **Reuse** colour, **deleteMin** and insert new finish time into heap!
EXAMPLE: HEAP-BASED ALGORITHM

Min element: finish at time 7

Heap finish at time 9

And so on, and so forth...

- Iteration i=5
- Check heap minimum
- Check if finish time 5 is before $s_5$
- Yes. Reuse colour, deleteMin and insert new finish time into heap!
\[
O(\log S) \text{ where } S = \text{size(priority queue)}
\]
DYNAMIC PROGRAMMING

What?
We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word "research"… He would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical.

I felt I had to do something to shield Wilson … from the fact that I was really doing mathematics… What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming." I wanted to get across the idea that this was “dynamic,” this was multistage, this was time-varying. I thought, let's kill two birds with one stone.

I thought dynamic programming was a good name. It was something not even a Congressman could object to.
COMPUTING FIBONACCI NUMBERS INEFFICIENTLY
A TOY EXAMPLE TO COMPARE D&C TO DYNAMIC PROGRAMMING

1. BadFib(n)
2. if n == 0 or n == 1 then return n
3. return BadFib(n-1) + BadFib(n-2)

FIBONACCI PIGEONS
RUNTIME

• In unit cost model
  • (UNREALISTIC!)

• \( T(n) = T(n - 1) + T(n - 2) + O(1) \)
  • \( T(n) \geq 2T(n - 2) + O(1) \)
  • \( T(n) \leq 2T(n - 1) + O(1) \)

• \( n/2 \) levels of recursion for the first expression
• \( n \) levels for the second expression
• Work doubles at each level
• \( T(n) \) is certainly in \( \Omega(2^{n/2}) \) and \( O(2^n) \)

```
1  BadFib(n)
2  if n == 0 or n == 1 then return n
3  return BadFib(n-1) + BadFib(n-2)
```

This \( O(1) \) would change in the bit complexity model.
WHY IS THIS SO SLOW?

- Subproblems have LOTS of overlap!
- Every subtree on the right appears on the left
- ... recursively ...
- Each subtree is computed exponentially often in its depth

This overlap suggests dynamic programming may be able to help!
Designing Dynamic Programming Algorithms for Optimization Problems

(Optimal) Recursive Structure
Examine the structure of an optimal solution to a problem instance $I$, and determine if an optimal solution for $I$ can be expressed in terms of optimal solutions to certain subproblems of $I$.

Define Subproblems
Define a set of subproblems $S(I)$ of the instance $I$, the solution of which enables the optimal solution of $I$ to be computed. $I$ will be the last or largest instance in the set $S(I)$. 
Designing Dynamic Programming Algorithms (cont.)

Recurrence Relation
Derive a **recurrence relation** on the optimal solutions to the instances in $S(I)$. This recurrence relation should be completely specified in terms of optimal solutions to (smaller) instances in $S(I)$ and/or base cases.

Compute Optimal Solutions
Compute the optimal solutions to all the instances in $S(I)$. Compute these solutions using the recurrence relation in a **bottom-up** fashion, filling in a table of values containing these optimal solutions. Whenever a particular table entry is filled in using the recurrence relation, the optimal solutions of relevant subproblems can be looked up in the table (they have been computed already). The final table entry is the solution to $I$. 
SOLVING FIB USING DYNAMIC PROGRAMMING

• (Optimal) Recursive Structure
  • Solution to $n$-th Fibonacci number $f(n)$ can be expressed as the addition of smaller Fibonacci numbers
  • No notion of optimality for this particular problem

• Define Subproblems
  • The set subproblems that will be combined to obtain $Fib(n)$ is \{Fib$(n - 1)$, Fib$(n - 2)$\}
  • $S(I) = \{Fib(0), Fib(1), ..., Fib(n)\}$

• Recurrence Relation
  \[
  f(n) = \begin{cases} 
  f(n - 1) + f(n - 2) : i \geq 2 \\
  1 : i = 1 \\
  0 : i = 0 
  \end{cases}
  \]

• Computing (Optimal) Solutions
  • Create table $f[1..n]$ and compute its entries “bottom-up”
FILLING THE TABLE “BOTTOM-UP”

• Key idea:
  • When computing a table entry
  • Must have already computed the entries it depends on!

• Dependencies
  • Extract directly from recurrence
  • Entry n depends on n-1 and n-2

• Computing entries in order 1..n guarantees n-1 and n-2 are already computed when we compute n
**DP SOLUTION**

- **Space saving** optimization:
  - We never look at $f[i-3]$ or earlier
  - Can make do with a few variables instead of a table

```java
1  FibDP(n)
2    f = new array of size n
3
4      f[0] = 0
5      f[1] = 1
6
7      for i = 2..n
8          f[i] = f[i-1] + f[i-2]
9
10    return f[n]
```

This is still considered to be dynamic programming... We’ve just optimized out the table.

Contains $f[n]$
CORRECTNESS

• **Step 1**
  - Prove that when computing a table entry, dependent entries are already computed

• **Step 2** (similar to D&C)
  - Suppose subproblems are solved correctly (optimally)
  - Prove these (optimal) subsolutions are combined into an optimal solution

```
FibDP(n)

f = new array of size n
f[0] = 0
f[1] = 1
for i = 2..n
  f[i] = f[i-1] + f[i-2]
return f[n]
```

- Suppose $f[i-1]$ and $f[i-2]$ are the (i-1)th and (i-2)th Fib #s
- Then prove $f[i] = \text{the n-th Fib #}$
MODEL OF COMPUTATION FOR RUNTIME

- Unit cost model is **not very realistic** for this problem, because Fibonacci numbers grow quickly
  - \( F[10] = 55 \)
  - \( F[100] = 354224848179261915075 \)
  - \( F[300] = 2222322446294204455297398934619099672066666939096499764990979600 \)
- Value of \( F[n] \) is exponential in \( n \): \( f_n \in \Theta(\phi^n) \) where \( \phi \approx 1.6 \)
  - \( \phi^n \) needs \( \log(\phi^n) \) bits to store it
  - So \( F[n] \) needs \( \Theta(n) \) bits to store!

But let’s use unit cost anyway for simplicity
RUNNING TIME (UNIT COST)

- \( T(n) \in \Theta(n) \)

```java
1  FibDP(n)
2     f = new array of size n
3 4
5     f[0] = 0
6     f[1] = 1
7     for i = 2..n
8         f[i] = f[i-1] + f[i-2]
9 10    return f[n]
```
A BRIEF ASIDE

• Is this **linear runtime**?
• NO! This is “a linear function of n”
• When we say “linear runtime” we mean “a linear function of the input size”
• What is the input size $S$?
  • The input is the number $n$.
  • How many bits does it take to store $n$?
    $\Theta(\log n)$
  • So $S = \log n$ bits

Express $T(n)$ as a function of the input size $S$ (in bits)

$T(n) \in \Theta(n)$
$2^S = 2^{\log n} = n$
So $T(n) \in \Theta(2^S)$

This algorithm is **exponential** in the input size!

... but still exponentially faster than $2^{n/2}$
ROD CUTTING
A “REAL” DYNAMIC PROGRAMMING EXAMPLE

• Input:
  • \( n \): length of rod
  • \( p_1, \ldots, p_n \): \( p_i = \) price of a rod of length \( i \)

• Output:
  • Max income possible by cutting the rod of length \( n \) into any number of integer pieces (maybe no cuts)

<table>
<thead>
<tr>
<th>length ( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>price ( p_i )</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

\( n = 4 \)

All ways of cutting a rod of length 4

Example output: 10
DYNAMIC PROGRAMMING APPROACH

• High level idea (can just think recursively to start)
  • Given a rod of length n
  • Either make no cuts, or make a cut and **recurse** on the remaining parts

• **Where** should we cut?
DYNAMIC PROGRAMMING APPROACH

• Try **all ways** of making that cut
  • I.e., try a cut at positions 1, 2, ..., \( n - 1 \)
  • In each case, recurse on two rods \([0, i]\) and \([i, n]\)

• Take the max income over **all possibilities** (each \( i \) / no cut)

\[
\begin{align*}
i = 1 & \quad \text{[rod]} \\
i = 2 & \quad \text{[rod]} \\
i = 3 & \quad \text{[rod]} \\
\cdots & \\
i = n - 1 & \quad \text{[rod]}
\end{align*}
\]

Optimal substructure:
Max income from two rods w/sizes \( i \) and \( n - i \)

... is max income we can get from the rod size \( i \)

+ max income we can get from the rod size \( n - i \)
WE STOPPED HERE
RECURRENCE RELATION

- Define $M(k) = \text{maximum income for rod of length } k$
- If we do **not** cut the rod, max income is $p_k$
- If we **do** cut a rod at $i$
  - max income is $M(i) + M(k - i)$
  - Want to maximize this *over all* $i$
    - $\max_i\{M(i) + M(k - i)\}$ (for $0 < i < k$)
- $M(k) = \max\{p_k, \max_{1 \leq i \leq k-1}\{M(i) + M(k - i)\}\}$
COMPUTING SOLUTIONS BOTTOM-UP

- **Recurrence:** \( M(k) = \max\{p_k, \max_{1 \leq i \leq k-1}(M(i) + M(k - i))\} \)

- Compute **table** of solutions: \( M[1..n] \)

- **Dependencies:** entry \( k \) depends on
  - \( M[i] \rightarrow M[1..(k - 1)] \)
  - \( M[k - i] \rightarrow M[1..(k - 1)] \)

- All of these dependencies are \(< k\)

- So we can fill in the table entries in order \(1..n\)
Recurrence: \( M(k) = \max \{ p_k, \max_{1 \leq i \leq k-1} (M(i) + M(k - i)) \} \)

Recall, semantically, \( M(k) \) = maximum income for rod of length \( k \)

Time complexity (unit cost)? \( \Theta(n^2) \)

Is this a “quadratic time” algorithm?
MISCELLANEOUS TIPS

• Building a table of results bottom-up is what makes an algorithm DP

• There is a similar concept called **memoization**
  • But, for the purposes of this course, we want to see bottom-up table filling!

• Base cases are **critical**
  • They often completely determine the answer
  • Try setting f[0]=f[1]=0 in FibDP…