CS 341: ALGORITHMS

Lecture 7: dynamic programming I

Readings: see website

Trevor Brown

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

trevor.brown@uwaterloo.ca

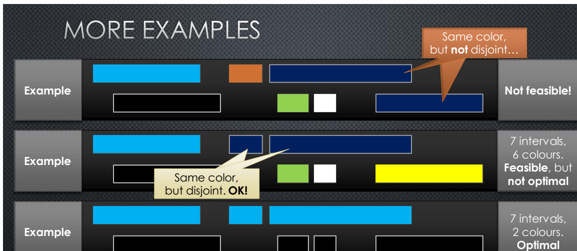
FINISHING UP GREEDY

INTERVAL **COLOURING**

PROBLEM: INTERVAL COLOURING

Instance: *A set $\mathcal{A} = \{A_1, \ldots, A_n\}$ of* **intervals***.*
For $1 \le i \le n$, $A_i = [s_i, f_i)$, 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 $col : \mathcal{A} \to \{1, \ldots, c\}$ 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.*

MORE EXAMPLES



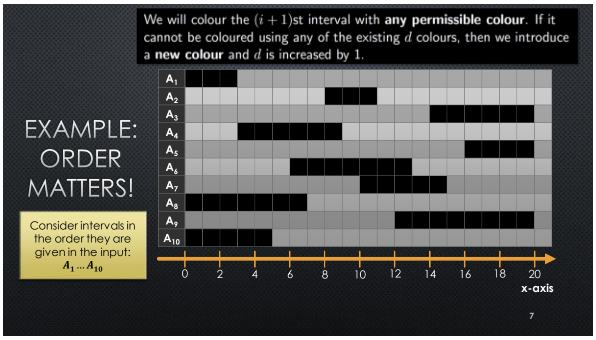
Greedy Strategies for Interval Colouring

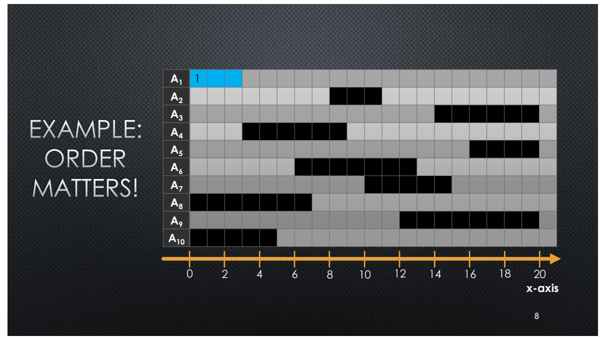
As usual, we consider the intervals one at a time.

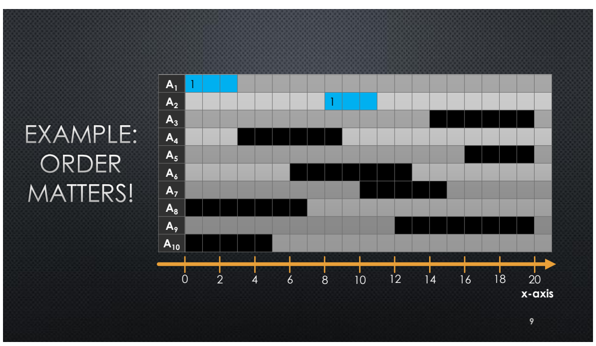
At a given point in time, suppose we have coloured the first $i < n$ intervals using d colours.

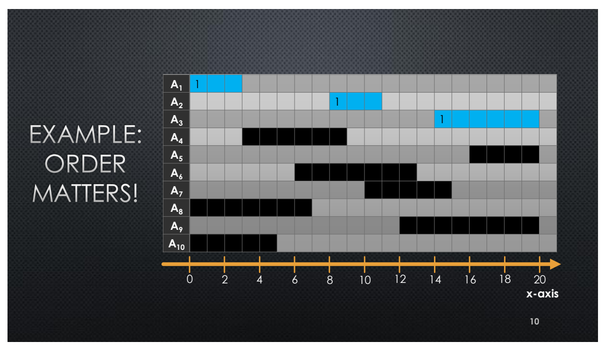
We will colour the $(i+1)$st interval with **any permissible colour**. If it cannot be coloured using any of the existing d colours, then we introduce a **new colour** and d is increased by 1.

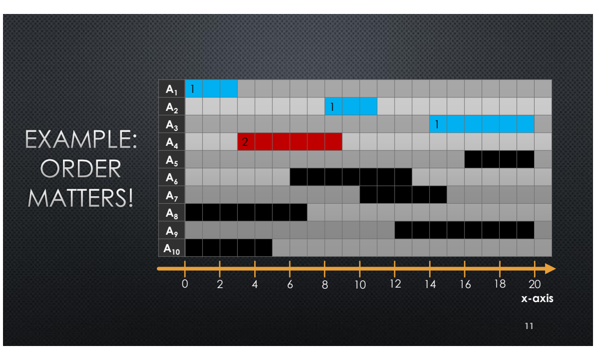
Question: In **what order** should we consider the intervals?

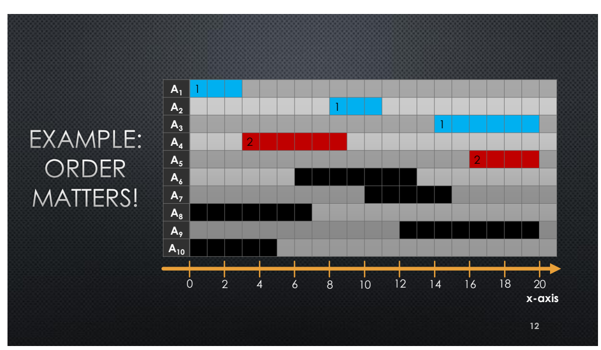

We will colour the $(i+1)$st interval with **any permissible colour**. If it cannot be coloured using any of the existing d colours, then we introduce a **new colour** and d is increased by 1.
EXAMPLE: ORDER MATTERS!
Consider intervals in the order they are given in the input: $A_1 \dots A_{10}$
A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10}
0 2 4 6 8 10 12 14 16 18 20
x-axis
7

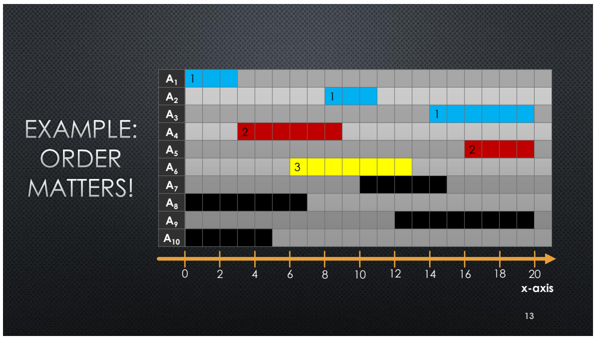

EXAMPLE: ORDER MATTERS!
A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10}
1
0 2 4 6 8 10 12 14 16 18 20
x-axis
8

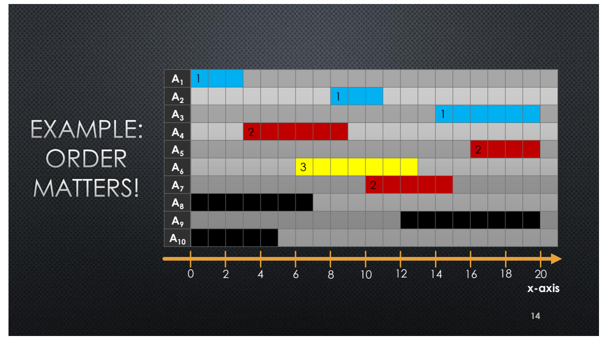

EXAMPLE: ORDER MATTERS!
A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10}
1
1
0 2 4 6 8 10 12 14 16 18 20
x-axis
9

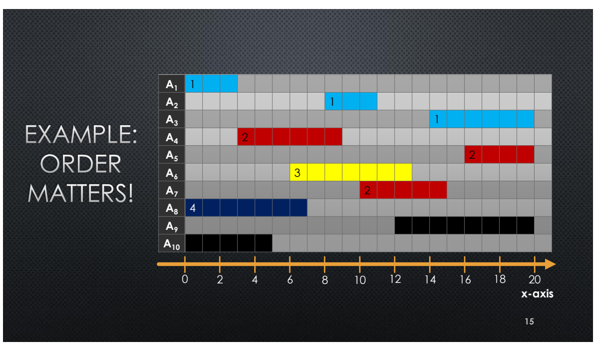

EXAMPLE: ORDER MATTERS!
A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10}
1
1
1
0 2 4 6 8 10 12 14 16 18 20
x-axis
10

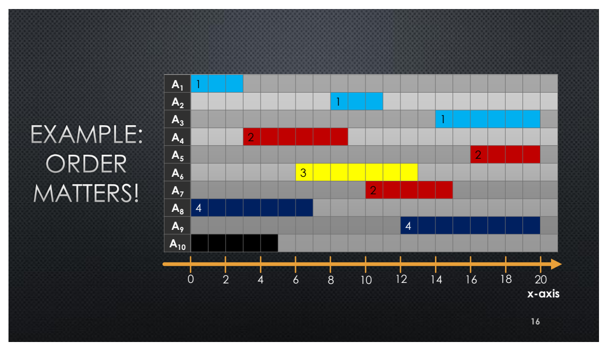

EXAMPLE: ORDER MATTERS!
A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10}
1
1
1
2
0 2 4 6 8 10 12 14 16 18 20
x-axis
11

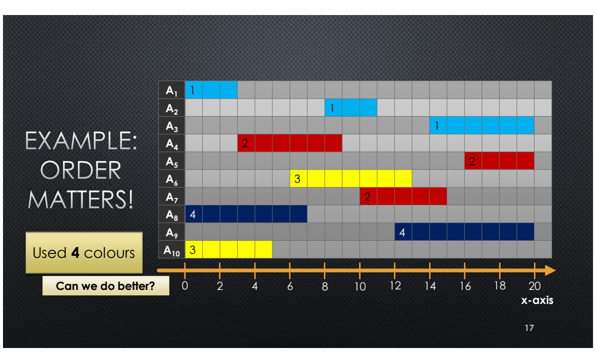

EXAMPLE: ORDER MATTERS!
A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10}
1
1
1
2
2
0 2 4 6 8 10 12 14 16 18 20
x-axis
12

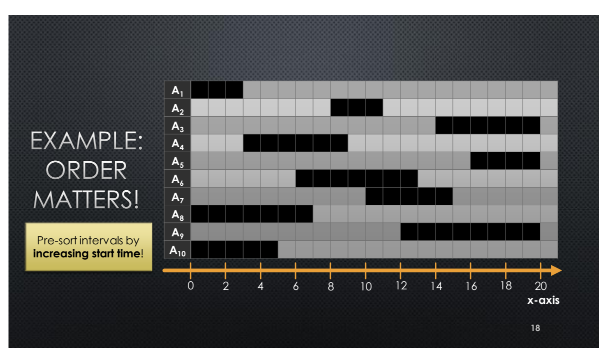

EXAMPLE:
ORDER
MATTERS!
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
0 2 4 6 8 10 12 14 16 18 20
x-axis
13

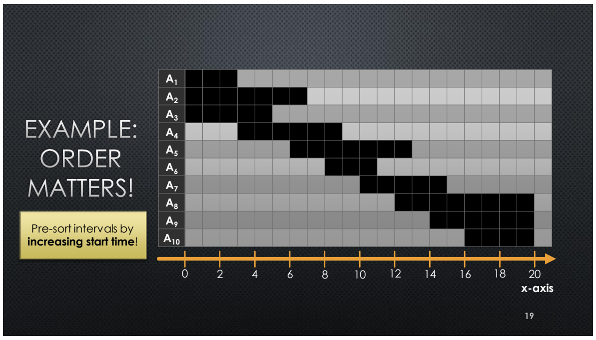

EXAMPLE:
ORDER
MATTERS!
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
0 2 4 6 8 10 12 14 16 18 20
x-axis
14

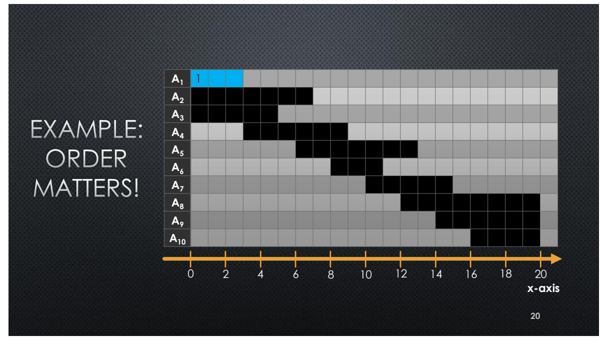

EXAMPLE:
ORDER
MATTERS!
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
0 2 4 6 8 10 12 14 16 18 20
x-axis
15

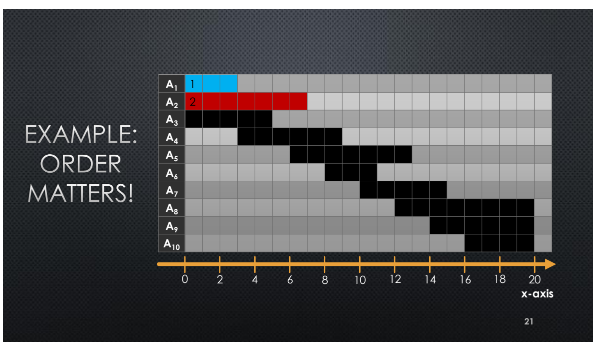

EXAMPLE:
ORDER
MATTERS!
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
0 2 4 6 8 10 12 14 16 18 20
x-axis
16

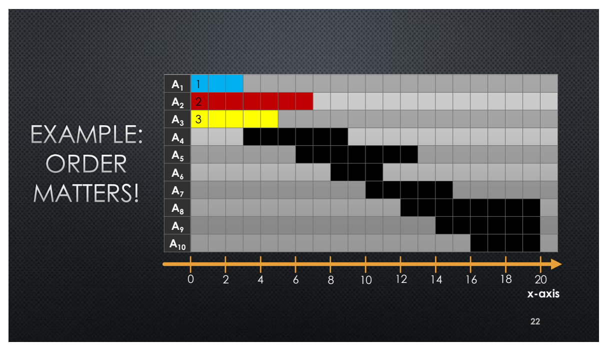

EXAMPLE:
ORDER
MATTERS!
Used 4 colours
Can we do better?
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
0 2 4 6 8 10 12 14 16 18 20
x-axis
17

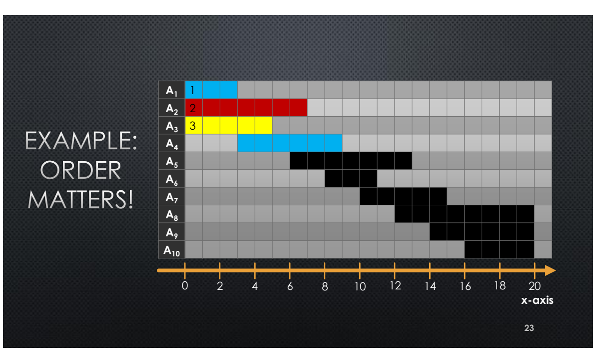

EXAMPLE:
ORDER
MATTERS!
Pre-sort intervals by increasing start time!
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
0 2 4 6 8 10 12 14 16 18 20
x-axis
18

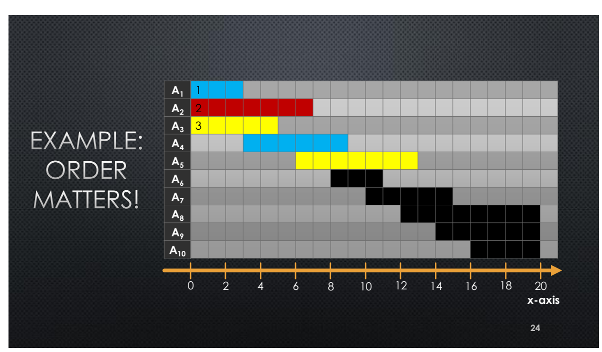

EXAMPLE: ORDER MATTERS!
Pre-sort intervals by increasing start time!
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
0 2 4 6 8 10 12 14 16 18 20
x-axis
19

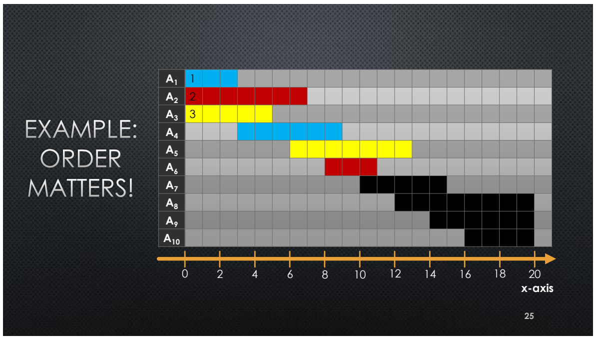

EXAMPLE: ORDER MATTERS!
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
1
0 2 4 6 8 10 12 14 16 18 20
x-axis
20

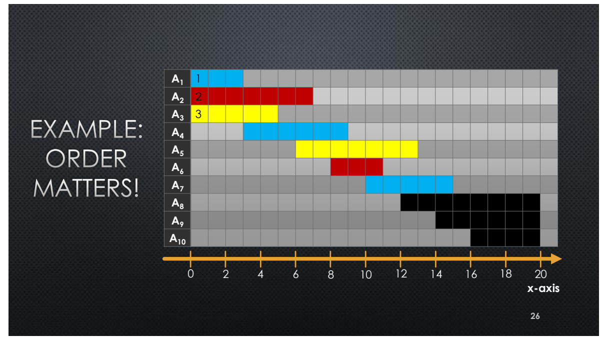

EXAMPLE: ORDER MATTERS!
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
1
2
0 2 4 6 8 10 12 14 16 18 20
x-axis
21

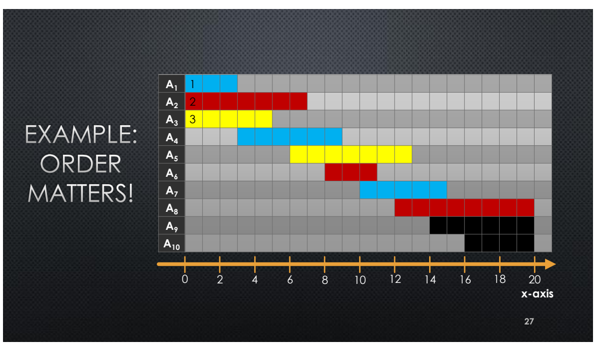

EXAMPLE: ORDER MATTERS!
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
1
2
3
0 2 4 6 8 10 12 14 16 18 20
x-axis
22

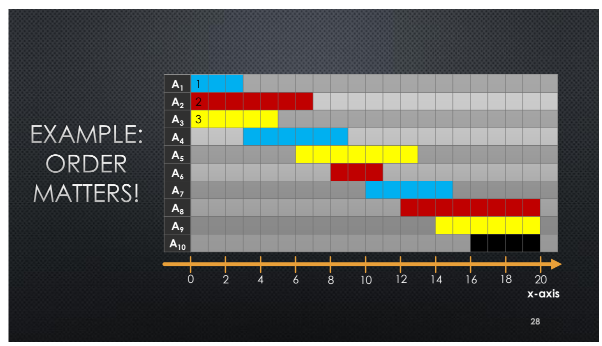

EXAMPLE: ORDER MATTERS!
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
1
2
3
0 2 4 6 8 10 12 14 16 18 20
x-axis
23

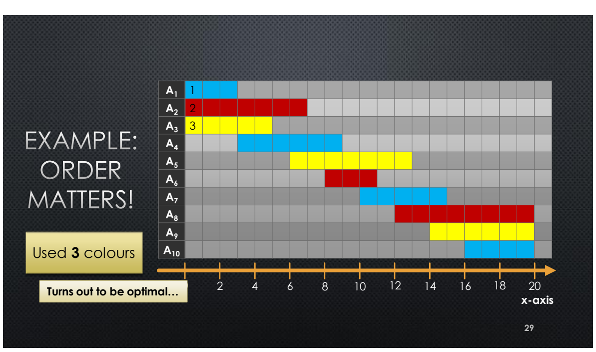

EXAMPLE: ORDER MATTERS!
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
1
2
3
0 2 4 6 8 10 12 14 16 18 20
x-axis
24

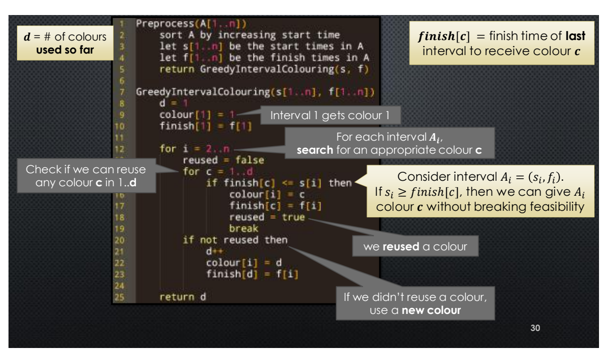

EXAMPLE: ORDER MATTERS!
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
1
2
3
0 2 4 6 8 10 12 14 16 18 20
x-axis
25

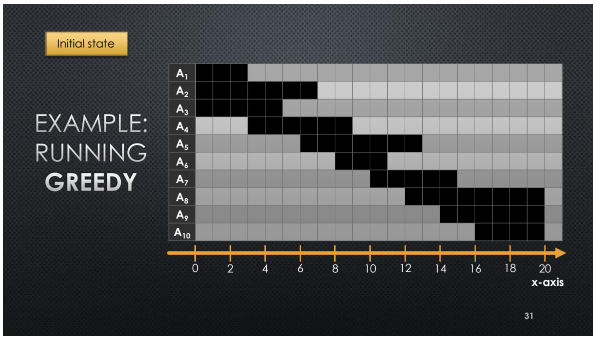

EXAMPLE: ORDER MATTERS!
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
1
2
3
0 2 4 6 8 10 12 14 16 18 20
x-axis
26

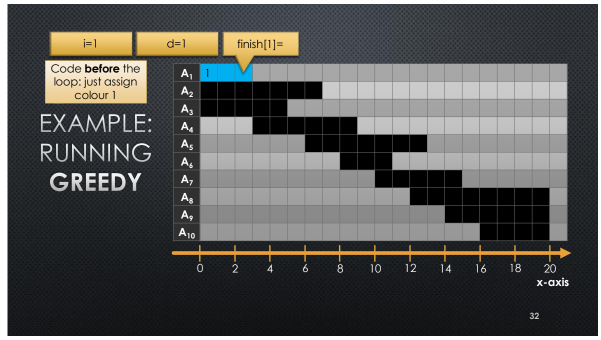

EXAMPLE: ORDER MATTERS!
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
1
2
3
0 2 4 6 8 10 12 14 16 18 20
x-axis
27

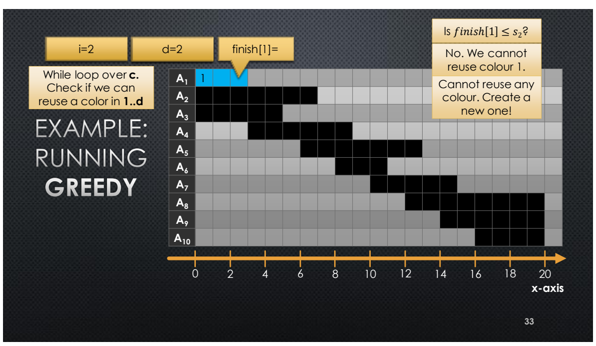

EXAMPLE: ORDER MATTERS!
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
1
2
3
0 2 4 6 8 10 12 14 16 18 20
x-axis
28

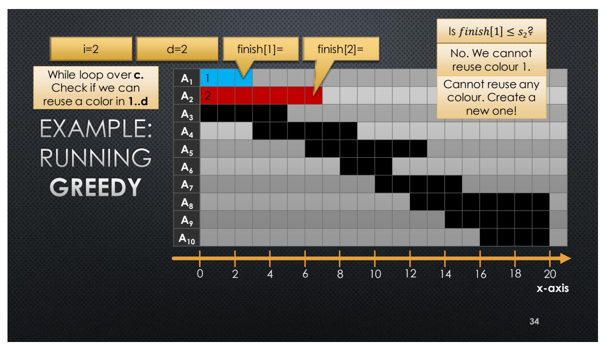

EXAMPLE: ORDER MATTERS!
Used 3 colours
Turns out to be optimal...
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
1
2
3
2 4 6 8 10 12 14 16 18 20
x-axis
29

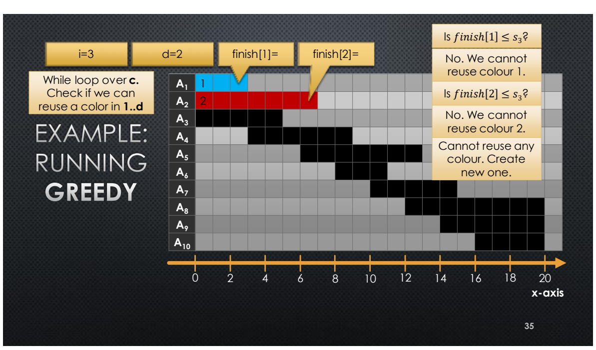

d = # of colours used so far
Preprocess(A[1..n])
    sort A by increasing start time
    let s[1..n] be the start times in A
    let f[1..n] be the finish times in A
    return GreedyIntervalColouring(s, f)
GreedyIntervalColouring(s[1..n], f[1..n])
    d = 1
    colour[1] = 1
    finish[1] = f[1]
    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
finish[c] = finish time of last interval to receive colour c
Interval 1 gets colour 1
For each interval A_i, search for an appropriate colour c
Check if we can reuse any colour c in 1..d
Consider interval $A_i = (s_i, f_i)$. If $s_i \geq finish[c]$, then we can give A_i colour c without breaking feasibility
we reused a colour
If we didn't reuse a colour, use a new colour
30

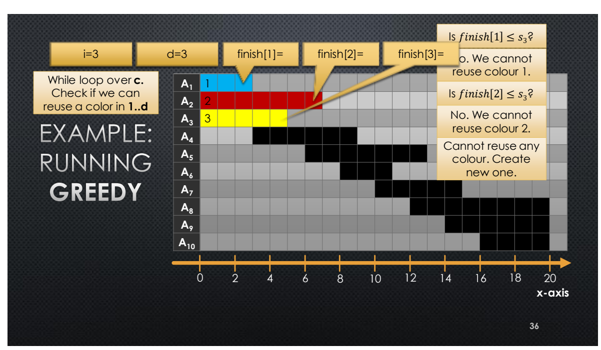

Initial state
EXAMPLE: RUNNING GREEDY
A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10}
0 2 4 6 8 10 12 14 16 18 20
x-axis
31

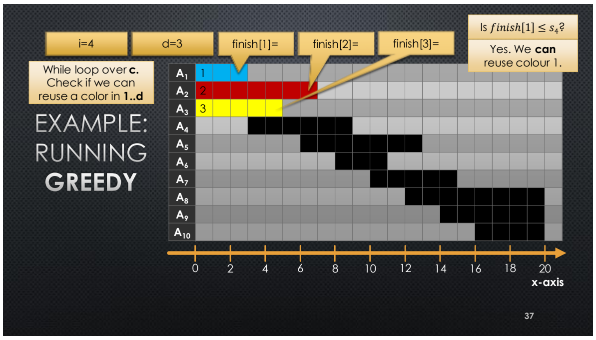

i=1
d=1
finish[1]=
Code **before** the loop: just assign colour 1
EXAMPLE: RUNNING GREEDY
1
A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10}
0 2 4 6 8 10 12 14 16 18 20
x-axis
32

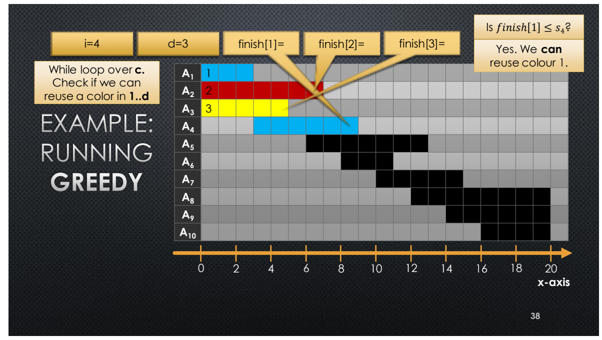

i=2
d=2
finish[1]=
Is $finish[1] \le s_2$?
No. We cannot reuse colour 1.
Cannot reuse any colour. Create a new one!
While loop over **c.** Check if we can reuse a color in **1..d**
EXAMPLE: RUNNING GREEDY
1
A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10}
0 2 4 6 8 10 12 14 16 18 20
x-axis
33

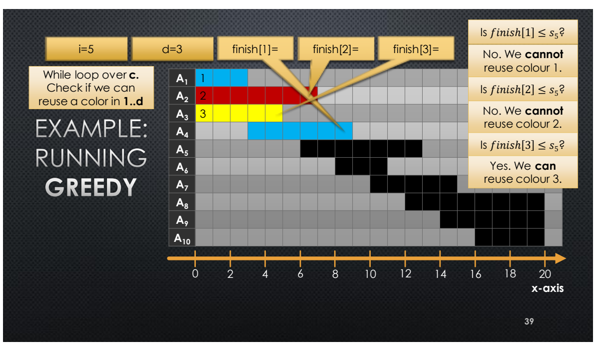

i=2
d=2
finish[1]=
finish[2]=
Is $finish[1] \le s_2$?
No. We cannot reuse colour 1.
Cannot reuse any colour. Create a new one!
While loop over **c.** Check if we can reuse a color in **1..d**
EXAMPLE: RUNNING GREEDY
1
2
A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10}
0 2 4 6 8 10 12 14 16 18 20
x-axis
34

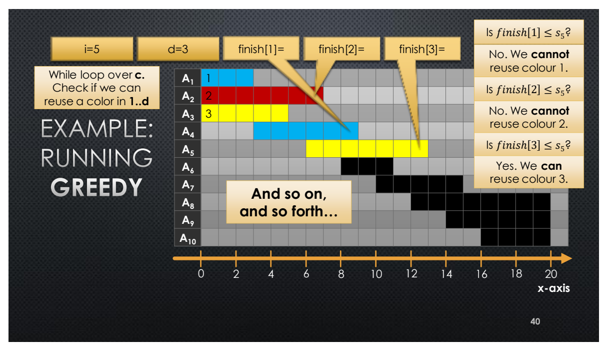

i=3
d=2
finish[1]=
finish[2]=
Is $finish[1] \le s_3$?
No. We cannot reuse colour 1.
Is $finish[2] \le s_3$?
No. We cannot reuse colour 2.
Cannot reuse any colour. Create new one.
While loop over **c.** Check if we can reuse a color in **1..d**
EXAMPLE: RUNNING GREEDY
1
2
A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10}
0 2 4 6 8 10 12 14 16 18 20
x-axis
35

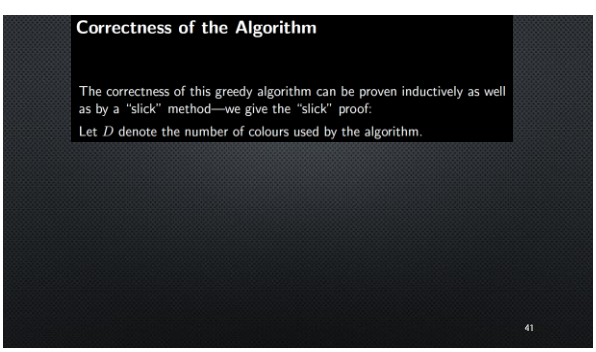

i=3
d=3
finish[1]=
finish[2]=
finish[3]=
Is $finish[1] \le s_3$?
o. We cannot reuse colour 1.
Is $finish[2] \le s_3$?
No. We cannot reuse colour 2.
Cannot reuse any colour. Create new one.
While loop over **c.** Check if we can reuse a color in **1..d**
EXAMPLE: RUNNING GREEDY
1
2
3
A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10}
0 2 4 6 8 10 12 14 16 18 20
x-axis
36

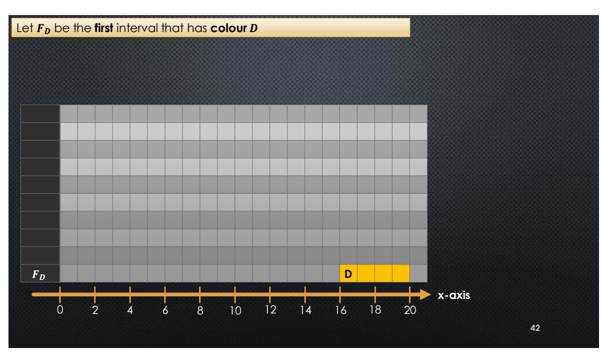

i=4
d=3
finish[1]=
finish[2]=
finish[3]=
Is $finish[1] \le s_4$?
Yes. We can reuse colour 1.
While loop over c. Check if we can reuse a color in 1..d
EXAMPLE: RUNNING GREEDY
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
1 2 3
0 2 4 6 8 10 12 14 16 18 20
x-axis
37

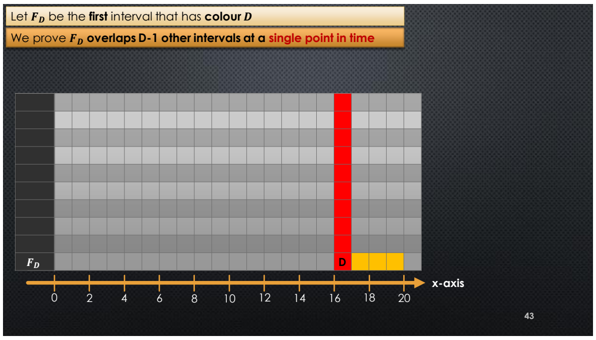

i=4
d=3
finish[1]=
finish[2]=
finish[3]=
Is $finish[1] \le s_4$?
Yes. We can reuse colour 1.
While loop over c. Check if we can reuse a color in 1..d
EXAMPLE: RUNNING GREEDY
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
1 2 3
0 2 4 6 8 10 12 14 16 18 20
x-axis
38

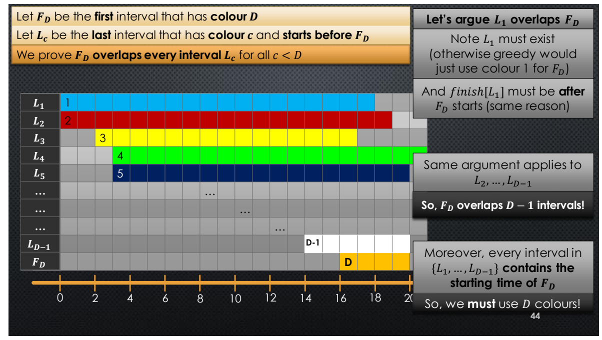

i=5
d=3
finish[1]=
finish[2]=
finish[3]=
Is $finish[1] \le s_5$?
No. We cannot reuse colour 1.
Is $finish[2] \le s_5$?
No. We cannot reuse colour 2.
Is $finish[3] \le s_5$?
Yes. We can reuse colour 3.
While loop over c. Check if we can reuse a color in 1..d
EXAMPLE: RUNNING GREEDY
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
1 2 3
0 2 4 6 8 10 12 14 16 18 20
x-axis
39

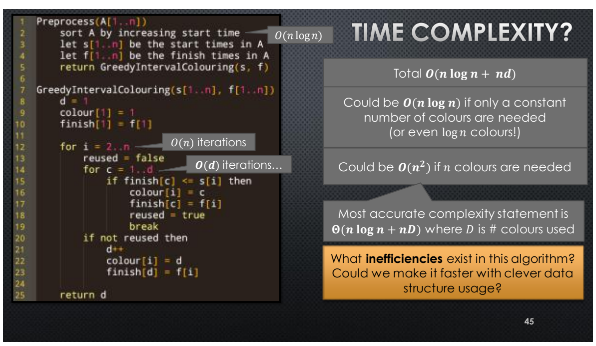

i=5
d=3
finish[1]=
finish[2]=
finish[3]=
Is $finish[1] \le s_5$?
No. We cannot reuse colour 1.
Is $finish[2] \le s_5$?
No. We cannot reuse colour 2.
Is $finish[3] \le s_5$?
Yes. We can reuse colour 3.
While loop over c. Check if we can reuse a color in 1..d
EXAMPLE: RUNNING GREEDY
And so on, and so forth…
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
1 2 3
0 2 4 6 8 10 12 14 16 18 20
x-axis
40

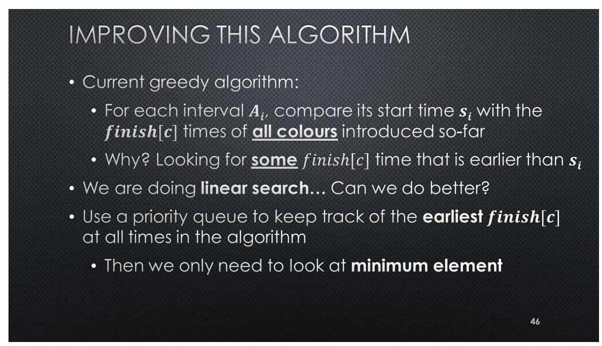

Correctness of the Algorithm
The correctness of this greedy algorithm can be proven inductively as well as by a "slick" method—we give the "slick" proof:
Let D denote the number of colours used by the algorithm.
41

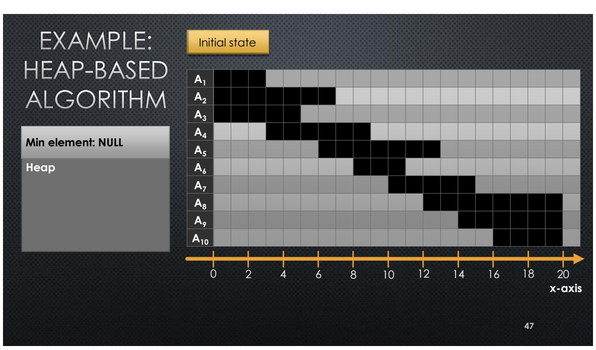

Let F_D be the first interval that has colour D
F_D
D
0 2 4 6 8 10 12 14 16 18 20
x-axis
42

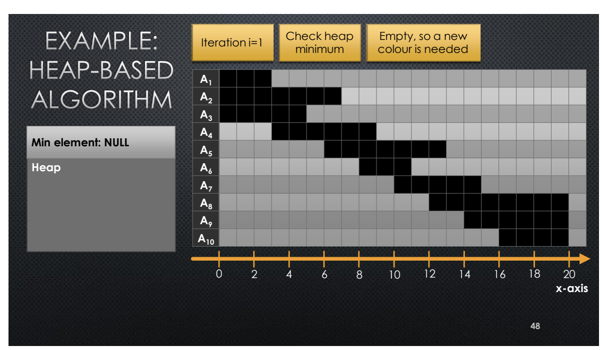

Let F_D be the **first** interval that has **colour** D
We prove F_D **overlaps D-1 other intervals at a single point in time**
F_D
D
0 2 4 6 8 10 12 14 16 18 20
x-axis
43

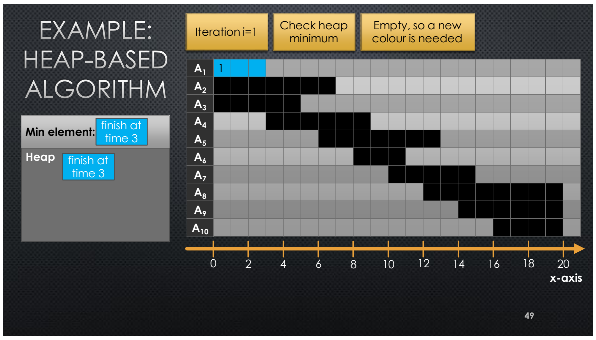

Let F_D be the **first** interval that has **colour** D
Let L_c be the **last** interval that has **colour** c and **starts before** F_D
We prove F_D **overlaps every interval** L_c for all $c < D$
L_1
L_2
L_3
L_4
L_5
…
…
…
L_{D-1}
F_D
1
2
3
4
5
D-1
D
0 2 4 6 8 10 12 14 16 18 20
Let's argue L_1 overlaps F_D
Note L_1 must exist (otherwise greedy would just use colour 1 for F_D)
And $finish[L_1]$ 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!
44

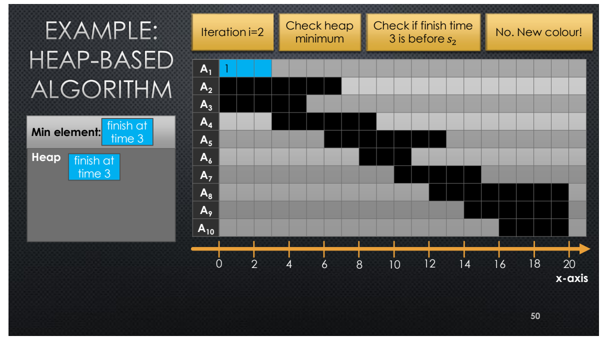

Preprocess(A[1..n])
sort A by increasing start time
let s[1..n] be the start times in A
let f[1..n] be the finish times in A
return GreedyIntervalColouring(s, f)
GreedyIntervalColouring(s[1..n], f[1..n])
d = 1
colour[1] = 1
finish[1] = f[1]
for i = 2..n
reused = false
for c = 1..d
if finish[c] <= s[i] 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…
TIME COMPLEXITY?
Total $O(n \log n + nd)$
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?
45

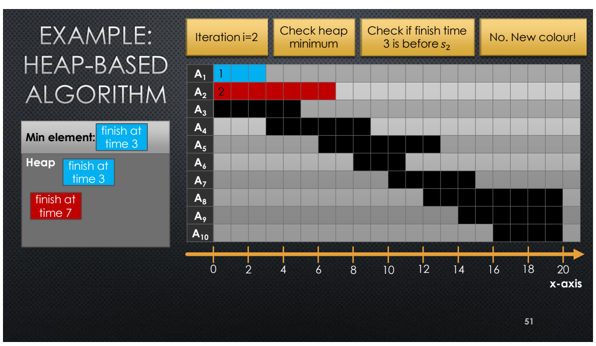

IMPROVING THIS ALGORITHM
Current greedy algorithm:
For each interval A_i, compare its start time s_i with the $finish[c]$ times of **all colours** introduced so-far
Why? Looking for **some** $finish[c]$ time that is earlier than s_i
We are doing **linear search…** Can we do better?
Use a priority queue to keep track of the **earliest $finish[c]$** at all times in the algorithm
Then we only need to look at **minimum element**
46

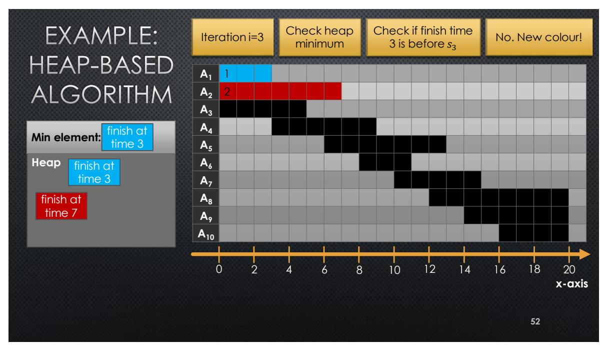

EXAMPLE: HEAP-BASED ALGORITHM
Initial state
Min element: NULL
Heap
A_1
A_2
A_3
A_4
A_5
A_6
A_7
A_8
A_9
A_{10}
0 2 4 6 8 10 12 14 16 18 20
x-axis
47

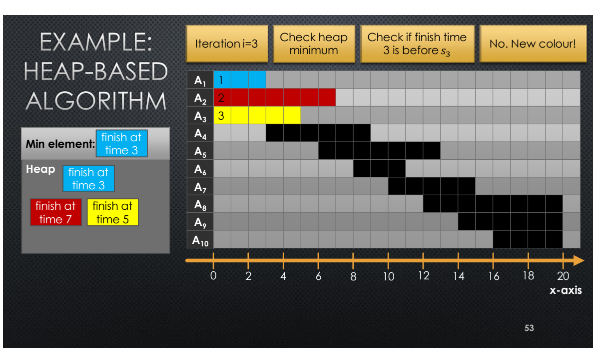

EXAMPLE: HEAP-BASED ALGORITHM
Iteration i=1
Check heap minimum
Empty, so a new colour is needed
Min element: NULL
Heap
A_1
A_2
A_3
A_4
A_5
A_6
A_7
A_8
A_9
A_{10}
0 2 4 6 8 10 12 14 16 18 20
x-axis
48

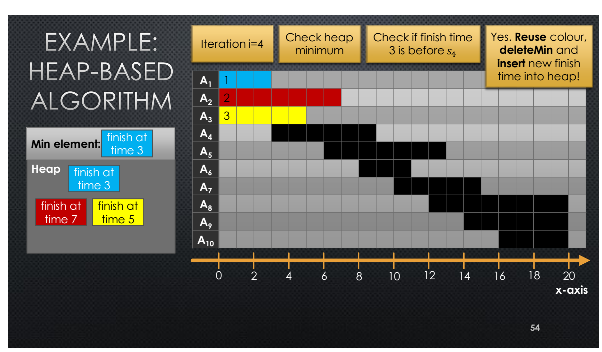

EXAMPLE: HEAP-BASED ALGORITHM
Iteration i=1
Check heap minimum
Empty, so a new colour is needed
Min element: finish at time 3
Heap
finish at time 3
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
1
0 2 4 6 8 10 12 14 16 18 20
x-axis
49

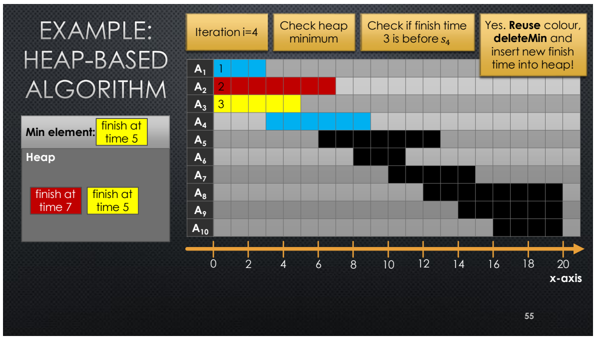

EXAMPLE: HEAP-BASED ALGORITHM
Iteration i=2
Check heap minimum
Check if finish time 3 is before s_2
No. New colour!
Min element: finish at time 3
Heap
finish at time 3
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
1
0 2 4 6 8 10 12 14 16 18 20
x-axis
50

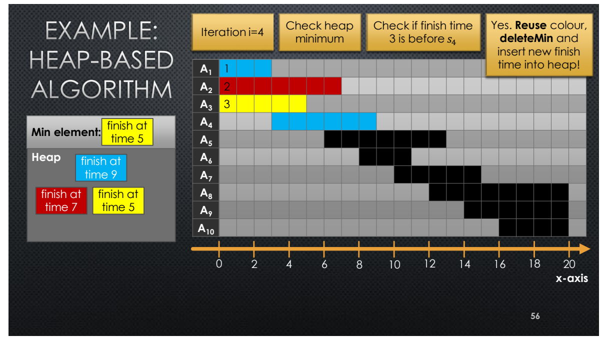

EXAMPLE: HEAP-BASED ALGORITHM
Iteration i=2
Check heap minimum
Check if finish time 3 is before s_2
No. New colour!
Min element: finish at time 3
Heap
finish at time 3
finish at time 7
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
1
2
0 2 4 6 8 10 12 14 16 18 20
x-axis
51

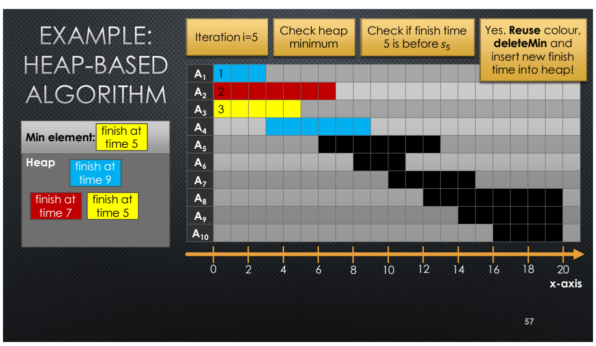

EXAMPLE: HEAP-BASED ALGORITHM
Iteration i=3
Check heap minimum
Check if finish time 3 is before s_3
No. New colour!
Min element: finish at time 3
Heap
finish at time 3
finish at time 7
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
1
2
0 2 4 6 8 10 12 14 16 18 20
x-axis
52

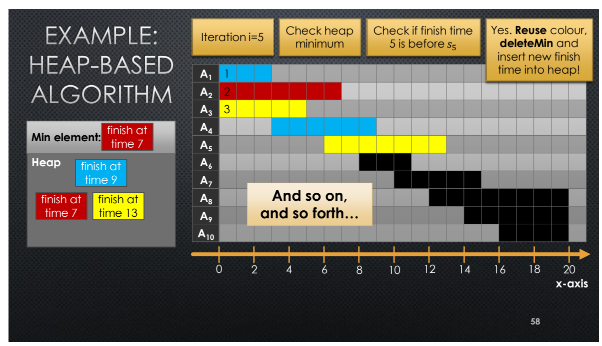

EXAMPLE: HEAP-BASED ALGORITHM
Iteration i=3
Check heap minimum
Check if finish time 3 is before s_3
No. New colour!
Min element: finish at time 3
Heap
finish at time 3
finish at time 7
finish at time 5
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
1
2
3
0 2 4 6 8 10 12 14 16 18 20
x-axis
53


EXAMPLE: HEAP-BASED ALGORITHM
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!
Min element: finish at time 3
Heap
finish at time 3
finish at time 7
finish at time 5
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10
1
2
3
0 2 4 6 8 10 12 14 16 18 20
x-axis
54










```
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
    h = new minPQ
    h.insert([f[1],colour[1]])

    for i = 2..n
        (fc, c) = h.min()
        if fc <= s[i] then
            h.deleteMin()
            colour[i] = c
        else
            d++
            colour[i] = d
        h.insert([f[i], colour[i]])

    return d
```

$O(\log S)$ where S = size(priority queue)

$O(1)$

$O(1)$

$O(\log D)$

$O(\log D)$

Total $\Theta(n \log n) + \Theta(n \log D)$

Since $n \geq D$, $\Theta(n \log n)$

59

DYNAMIC PROGRAMMING

What?

60

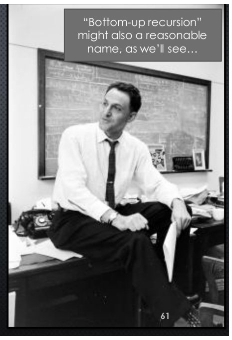
—*Richard Bellman, Eye of the Hurricane: An Autobiography (1984, excerpts from page 159)*

Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research.

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.



"Bottom-up recursion" might also a reasonable name, as we'll see…

61

COMPUTING FIBONACCI NUMBERS **INEFFICIENTLY**

A **TOY** EXAMPLE TO COMPARE D&C TO DYNAMIC PROGRAMMING

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

62

RUNTIME

- **In unit cost model**
  - **(UNREALISTIC!)**

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

- $T(n) = T(n-1) + T(n-2) + O(1)$
  - $T(n) \geq 2T(n-2) + O(1)$
  - $T(n) \leq 2T(n-1) + O(1)$
- 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 $\boldsymbol{\Omega(2^{n/2})}$ **and** $\boldsymbol{O(2^n)}$

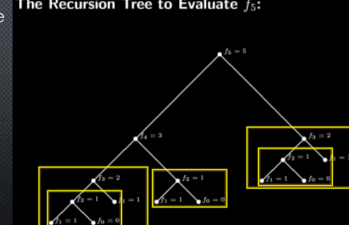
This O(1) would change in the bit complexity model

63

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

The Recursion Tree to Evaluate f_5:



This **overlap** suggests dynamic programming may be able to help!

64

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 $\mathcal{S}(I)$ 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 $\mathcal{S}(I)$.

65

Designing Dynamic Programming Algorithms (cont.)

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

Compute Optimal Solutions
Compute the optimal solutions to all the instances in $\mathcal{S}(I)$. 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.

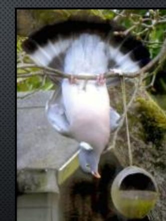
66

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 $\{Fib(n-1), Fib(n-2)\}$
  - $S(I) = \{Fib(0), Fib(1), \ldots, Fib(n)\}$
- Recurrence Relation $f(n) = \begin{cases} f(n-1) + f(n-2) : i \geq 2 \\ 1 \quad : i = 1 \\ 0 \quad : i = 0 \end{cases}$
- 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

```
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]
```

```
FibDP(n)
    fi2 = 0
    fi1 = 1

    for i = 2..n
        temp = fi

        fi = fi1 + fi2

        fi2 = fi1
        fi1 = temp

    return fi
```

represents f[i-2]

represents f[i-1]

Save f[i] before overwriting it (so its value can be stored in f[i-1] later)

Contains f[n]

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

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

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 a(n optimal) solution

Order 0..n means i-1 and i-2 are already computed when we compute i

- Suppose f[i-1] and f[i-2] are the (i-1)th and (i-2)th Fib #s
- Then prove f[i] = the n-th Fib #

```
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]
```


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]=222232244629420445529739893461909967206666939096499764990979600
  - Value of F[n] is exponential in n: $f_n \in \Theta(\phi^n)$ *where* $\phi \cong 1.6$
  - ϕ^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)

- $T(n) \in \boldsymbol{\Theta(n)}$

```
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]
```

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? $O(\log n)$
  - **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 $2^{n/2}$

73

ROD CUTTING

A "REAL" DYNAMIC PROGRAMMING EXAMPLE

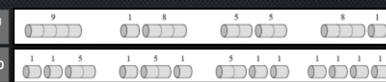
- Input:
  - n: length of rod
  - $p_1, \dots, p_n$: p_i = price of a rod of length i
- Output:
  - Max **income** possible by cutting the rod of length n into any number of **integer** pieces (maybe **no** cuts)

$n = 4$

length i	1	2	3	4
price p_i	1	5	8	9

All ways of cutting a rod of length 4

Example output: 10



74

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?

75

DYNAMIC PROGRAMMING APPROACH

- Try **all ways** of making that cut
  - I.e., try a cut at positions $1, 2, \dots, n-1$
  - In each case, recurse on two rods $[0, i]$ and $[i, n]$
- Take the max income over **all possibilities** (each i / no cut)

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 $n - i$

76

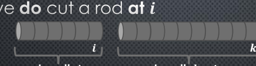
WE STOPPED HERE

77

RECURRENCE RELATION

Critical step! Must define what M(k) means, semantically!

- Define $M(k)$ = 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**

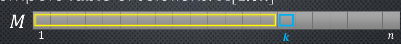


  - max income is $M(i) + M(k - i)$
  - Want to maximize this **over all i**
    - $max_i\{M(i) + M(k - i)\}$ (for $0 < i < k$)
- $M(k) = \max\{p_k, \max_{1 \le i \le k-1}\{M(i) + M(k - i)\}\}$

78

COMPUTING SOLUTIONS BOTTOM-UP

- **Recurrence:** $M(k) = max\{p_k, \max_{1 \le i \le k-1}\{M(i) + M(k-i)\}\}$
- Compute **table** of solutions: $M[1..n]$



- Dependencies: **entry** k depends on
  - $M[i]$ → $M[1..(k-1)]$
  - $M[k-i]$ → $M[1..(k-1)]$
  - All of these dependencies are $< k$
- So we can fill in the table entries in order $1..n$

79

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

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

```
RodCutting(n, p[1..n])
    M = new array[1..n]

    // compute each entry M[k]
    for k = 1..n
        M[k] = p[k] // current best = no cuts

        // try each cut in 1..(k-1)
        for i = 1..(k-1)
            M[k] = max(M[k], M[i] + M[k-i])

    return M[n]
```

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

Is this a "quadratic time" algorithm?

80

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…

81