CS 341: ALGORITHMS

Lecture 7: finishing greedy

Readings: see website

Trevor Brown
https://student.cs.uwaterloo.ca/~cs341
trevor.brown@uwaterloo.ca

LAST TIME: EXCHANGE ARGUMENT FOR INTERVAL SELECTION

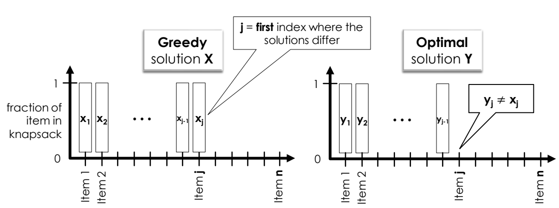
ASSUMED: PROFIT / WEIGHT RATIOS ARE **DISTINCT**

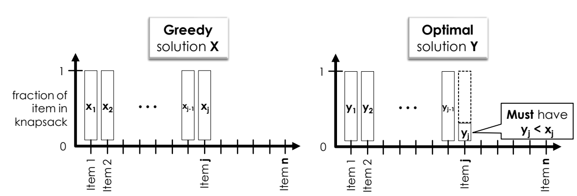
WHAT IF PROFIT/WEIGHT RATIOS ARE **NOT DISTINCT**?

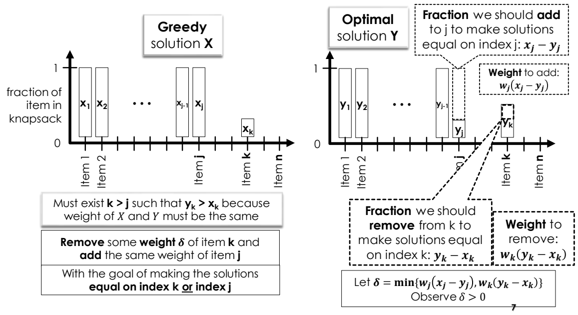
OR, MORE GENERALLY, **WHAT IF THERE ARE MANY OPT SOLUTIONS?**

WHAT IF THERE ARE **MANY** OPTIMAL SOLUTIONS

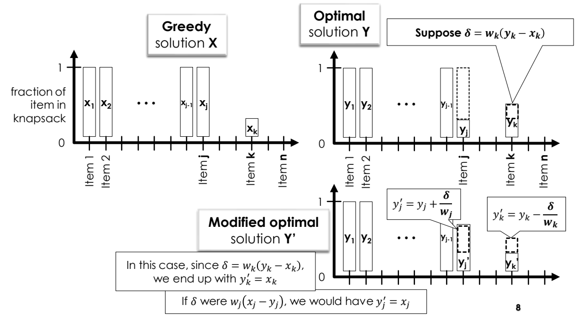
- Can't just assume X != Y and obtain a contradiction!
- **Key idea:** focus on **one particular optimal solution**
  - Let Y be an optimal solution that **matches X on a maximal number of indices**
  - **Observe**: if X is really optimal, then $Y = X$
- Suppose X != Y for contra
  - We will modify Y, preserving its optimality, but making it match X on **one more index** (a contradiction!)



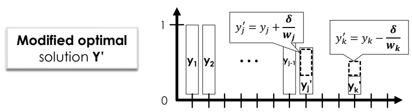




7



8



To show Y' is feasible, we show $weight(Y') \le M$ and $y_k' \ge 0, y_j' \le 1$

Weight	We move δ weight from item k to item j This does not change the total weight! So $weight(Y') = weight(Y) = M$

9

FEASIBILITY OF Y'

- Showing $y_k' \ge 0$
  - By definition, $y_k' = y_k - \frac{\delta}{w_k} \ge 0$ **iff** $\boldsymbol{\delta \le y_k w_k}$
  - But δ is the **minimum** of $w_j(x_j - y_j)$ and $w_k(y_k - x_k)$
  - And $w_k(y_k - x_k) \le w_k y_k$ **so** $\boldsymbol{\delta \le y_k w_k}$
- Showing $y_j' \le 1$
  - $y_j' = y_j + \frac{\delta}{w_j} \le 1$ **iff** $\boldsymbol{\delta \le w_j(1 - y_j)}$ (rearranging)
  - $\delta \le \boldsymbol{w_j(x_j - y_j)}$ (definition of δ)
  - and $w_j(x_j - y_j) \le \boldsymbol{w_j(1 - y_j)}$ (by feasibility of X, i.e., $x_j \le 1$)

10

PROFIT OF Y'

(Fraction of item j **added**) × (profit for entire item)

- $profit(Y') = profit(Y) + \frac{\delta}{w_j}p_j - \frac{\delta}{w_k}p_k = profit(Y) + \delta\left(\frac{p_j}{w_j} - \frac{p_k}{w_k}\right)$
- Since j is before k, and we consider items with more profit per unit weight first, we have $\frac{p_j}{w_j} \ge \frac{p_k}{w_k}$.
- Since $\delta > 0$ and $\frac{p_j}{w_j} \ge \frac{p_k}{w_k}$, we have $\delta\left(\frac{p_j}{w_j} - \frac{p_k}{w_k}\right) \ge 0$
- Since Y is optimal, this **cannot be positive**
- So Y' is a new optimal solution that **matches X on one more index than Y**
- Contradiction: Y matched X on a **maximal** number of indices!

11

SUMMARIZING EXCHANGE ARGUMENTS

- If there is a **unique optimal solution**
  - Let O != G be an optimal solution that beats greedy
  - Show how to change O to obtain a better solution
- If there is **more than one optimal solution**
  - Let O != G be an optimal solution that matches greedy on as many choices as possible
  - Show how to change O to obtain an optimal solution O' that matches greedy for even more choice(s)

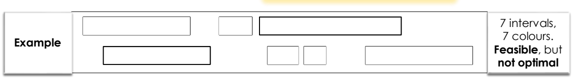
12

FINISHING UP GREEDY

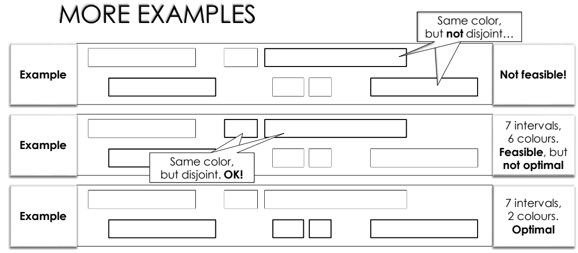
INTERVAL **COLOURING**

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* $col : \mathcal{A} \rightarrow \{1, \ldots, c\}$ *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.*



MORE EXAMPLES



Greedy Strategies for Interval Colouring

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

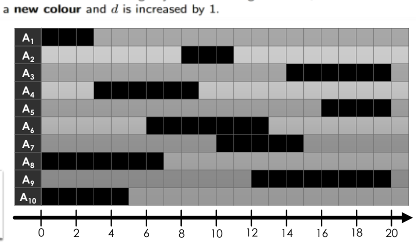
At a given point in time, suppose we have coloured the first $i < n$ intervals using d colours.

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?

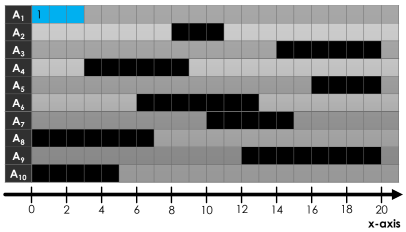
EXAMPLE: ORDER MATTERS!

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.



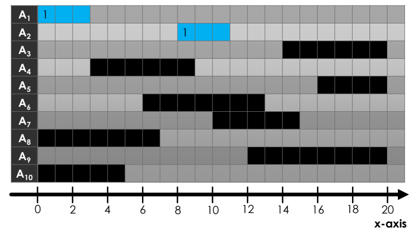
Consider intervals in the order they are given in the input: $A_1 \ldots A_{10}$

EXAMPLE: ORDER MATTERS!



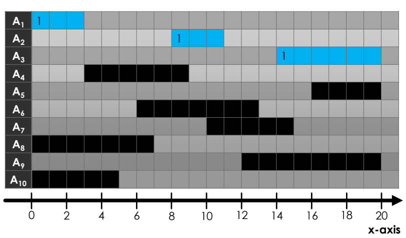
19

EXAMPLE: ORDER MATTERS!



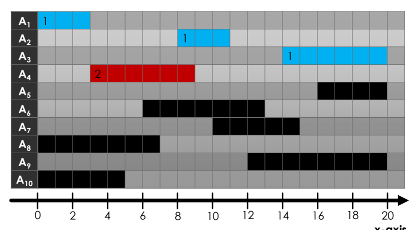
20

EXAMPLE: ORDER MATTERS!



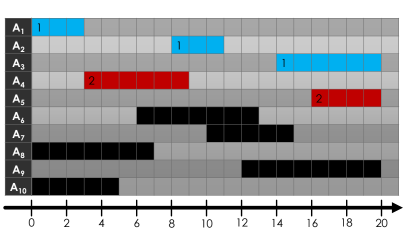
21

EXAMPLE: ORDER MATTERS!



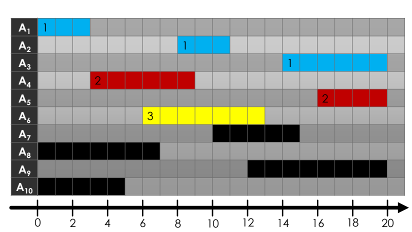
22

EXAMPLE: ORDER MATTERS!



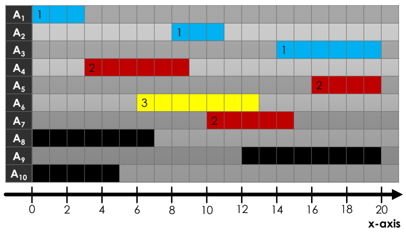
23

EXAMPLE: ORDER MATTERS!



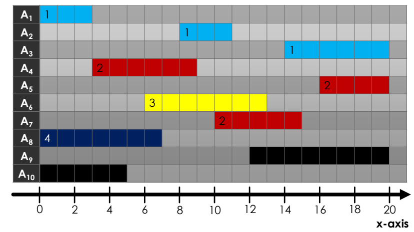
24

EXAMPLE: ORDER MATTERS!



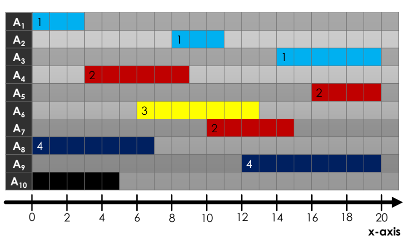
25

EXAMPLE: ORDER MATTERS!



26

EXAMPLE: ORDER MATTERS!

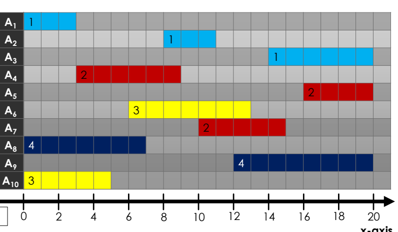


27

EXAMPLE: ORDER MATTERS!

Used **4** colours

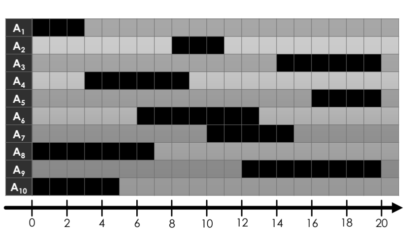
Can we do better?



28

EXAMPLE: ORDER MATTERS!

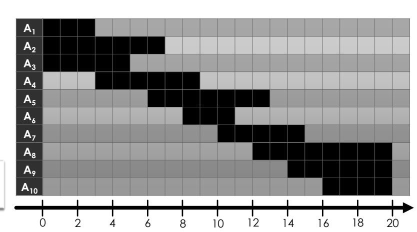
Pre-sort intervals by **increasing start time**!



29

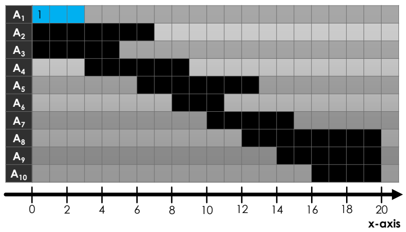
EXAMPLE: ORDER MATTERS!

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



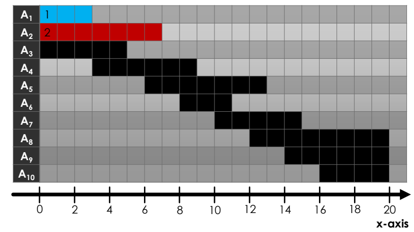
30

EXAMPLE: ORDER MATTERS!



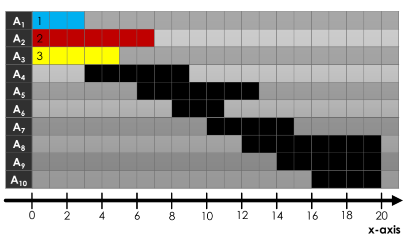
31

EXAMPLE: ORDER MATTERS!



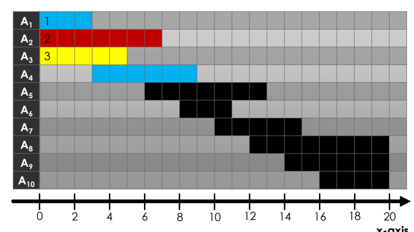
32

EXAMPLE: ORDER MATTERS!



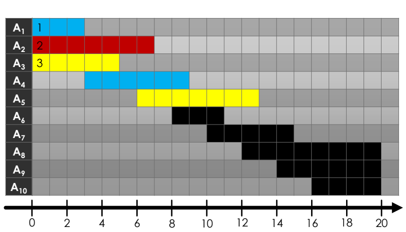
33

EXAMPLE: ORDER MATTERS!



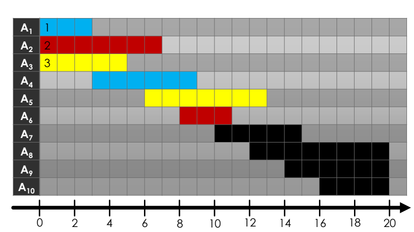
34

EXAMPLE: ORDER MATTERS!



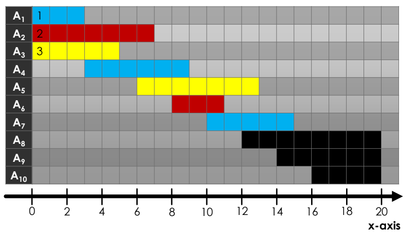
35

EXAMPLE: ORDER MATTERS!



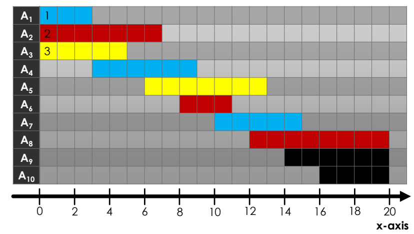
36

EXAMPLE: ORDER MATTERS!



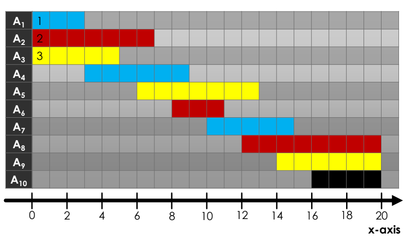
37

EXAMPLE: ORDER MATTERS!



38

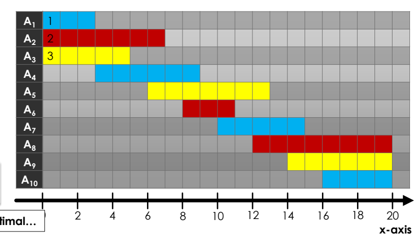
EXAMPLE: ORDER MATTERS!



39

EXAMPLE: ORDER MATTERS!

Used **3** colours



Turns out to be optimal…

40

d = # of colours **used so far**

$\boldsymbol{finish}[c]$ = finish time of **last** interval to receive colour c

```
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
```

Interval 1 gets colour 1

For each interval $\boldsymbol{A_i}$, **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 $\boldsymbol{c}$ without breaking feasibility

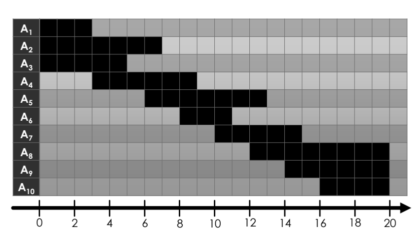
we **reused** a colour

If we didn't reuse a colour, use a **new colour**

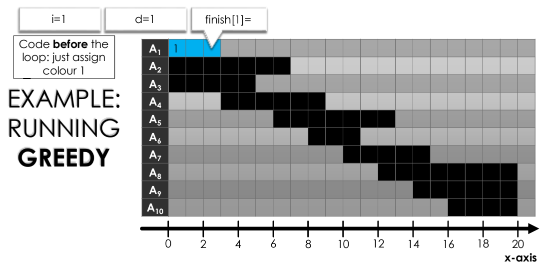
41

Initial state

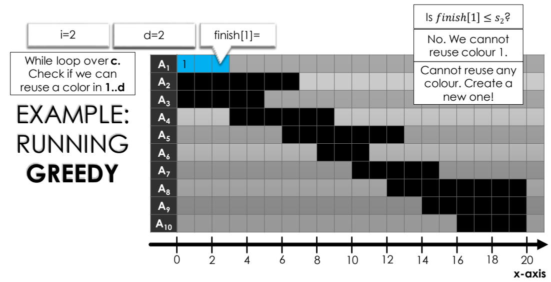
EXAMPLE: RUNNING **GREEDY**



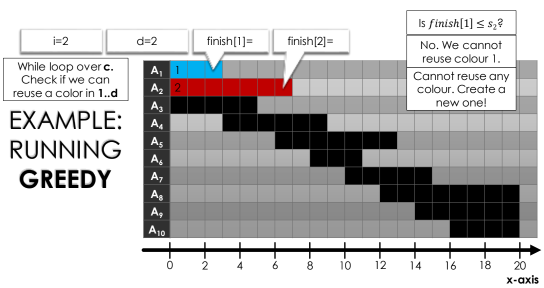
42



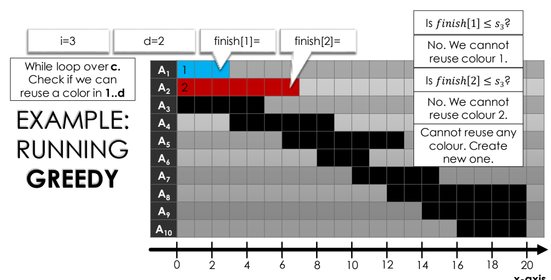
43



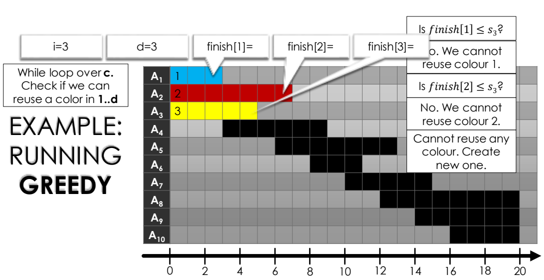
44



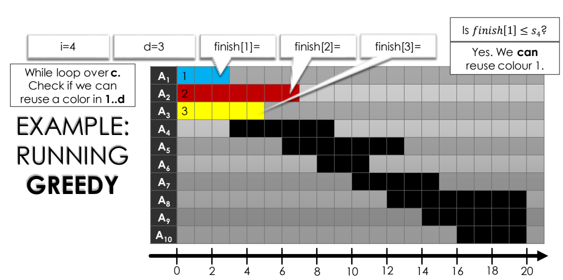
45



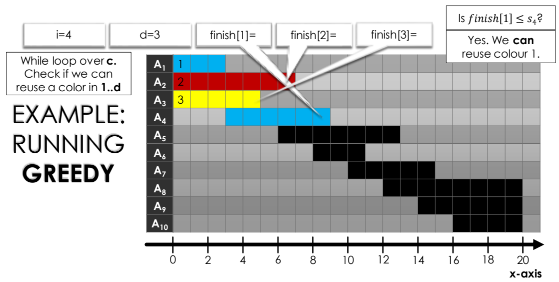
46



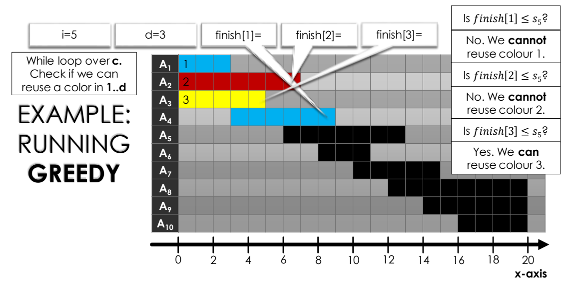
47



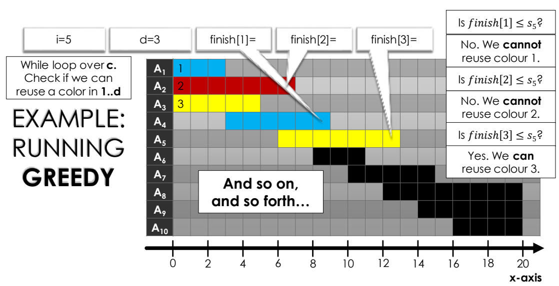
48



49



50



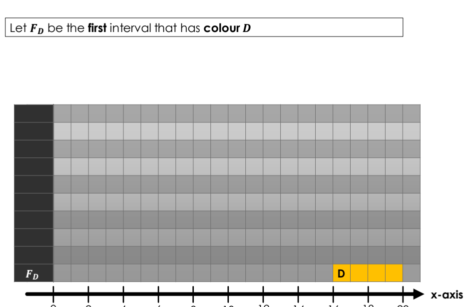
51

Correctness of the Algorithm

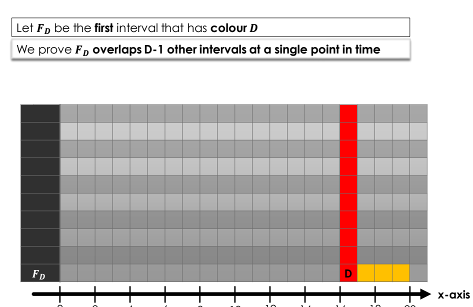
The correctness of this greedy algorithm can be proven inductively as well as by a "slick" method—we give the "slick" proof:

Let D denote the number of colours used by the algorithm.

52



53



54

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 **start[F_D] is properly contained in every such interval L_c**

Let's argue this for L_1

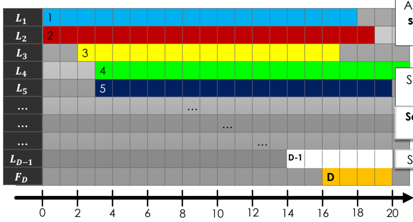
Note L_1 must exist (otherwise greedy would just use colour 1 for F_D)

And $finish[L_1]$ must be **after start[F_D]** or colour 1 would be eligible for reuse!

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

So, F_D overlaps $D-1$ intervals at a single time start[F_D]!

So, we **must** use D colours!



TIME COMPLEXITY?

```
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

$\boldsymbol{O(d)}$ iterations…

Total $\boldsymbol{O(n \log n + nd)}$

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

Could be $\boldsymbol{O(n^2)}$ if n colours are needed

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

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

IMPROVING THIS ALGORITHM

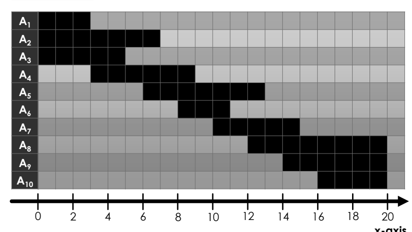
- Current greedy algorithm:
  - For each interval $\boldsymbol{A_i}$, compare its start time $\boldsymbol{s_i}$ with the $\boldsymbol{finish}[c]$ times of **all colours** introduced so-far
  - Why? Looking for **some** $finish[c]$ time that is earlier than $\boldsymbol{s_i}$
- We are doing **linear search…**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

EXAMPLE: HEAP-BASED ALGORITHM

Initial state

Min element: NULL

Heap



EXAMPLE: HEAP-BASED ALGORITHM

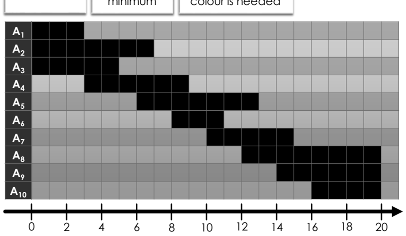
Iteration i=1

Check heap minimum

Empty, so a new colour is needed

Min element: NULL

Heap



EXAMPLE: HEAP-BASED ALGORITHM

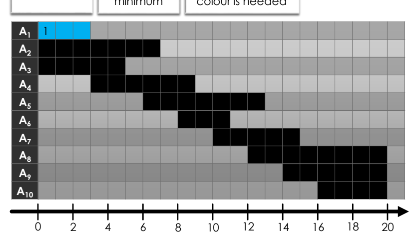
Iteration i=1

Check heap minimum

Empty, so a new colour is needed

Min element: finish at time 3

Heap finish at time 3

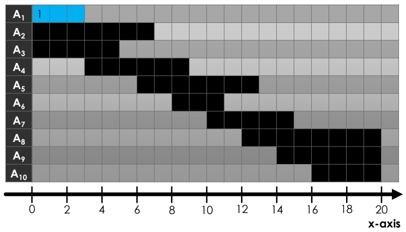


EXAMPLE: HEAP-BASED ALGORITHM

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



61

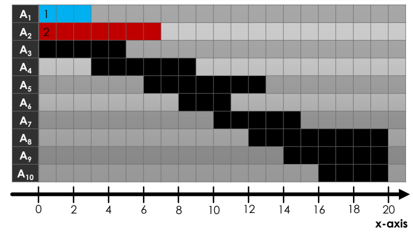
EXAMPLE: HEAP-BASED ALGORITHM

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



62

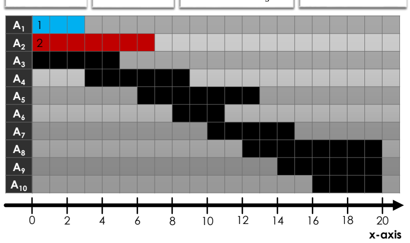
EXAMPLE: HEAP-BASED ALGORITHM

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



63

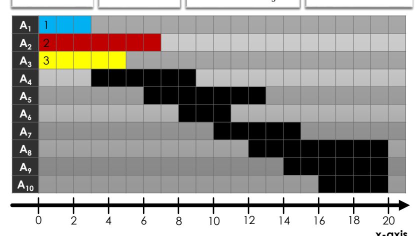
EXAMPLE: HEAP-BASED ALGORITHM

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



64

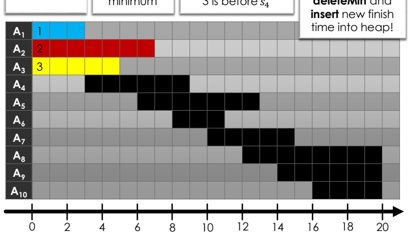
EXAMPLE: HEAP-BASED ALGORITHM

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



65

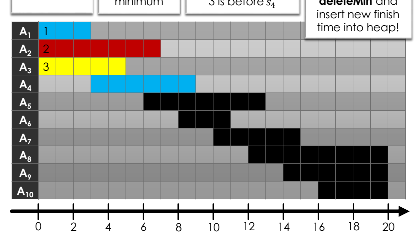
EXAMPLE: HEAP-BASED ALGORITHM

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 5

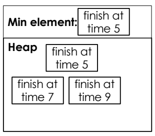
Heap finish at time 5

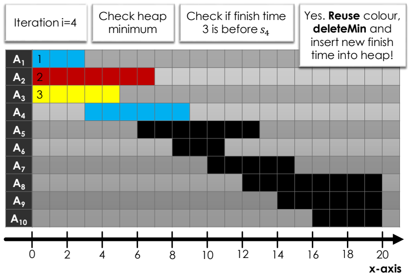
finish at time 7



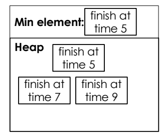
66

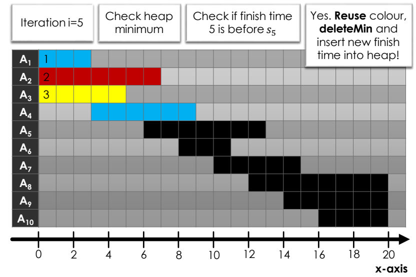
EXAMPLE: HEAP-BASED ALGORITHM



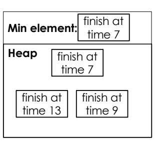


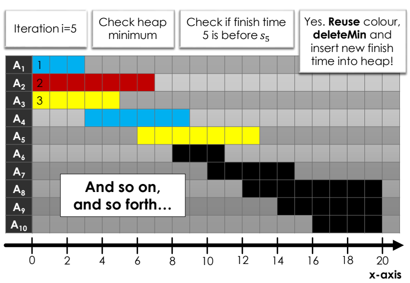
EXAMPLE: HEAP-BASED ALGORITHM





EXAMPLE: HEAP-BASED ALGORITHM






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

DYNAMIC PROGRAMMING

What?

—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.

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)
```

RUNTIME

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

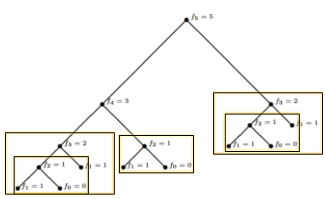
```
BadFib(n)
    if n == 0 or n == 1 then return n
    return BadFib(n-1) + BadFib(n-2)
```

- $T(n) = T(n-1) + T(n-2) + O(1)$ ← This O(1) would change in the bit complexity model
  - $T(n) \geq 2T(n-2) + O(1)$
  - $T(n) \leq 2T(n-1) + O(1)$
- 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 $\mathbf{\Omega(2^{n/2})}$ **and** $\mathbf{O(2^n)}$

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!

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 $\mathcal{S}(I)$.

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.

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 $\{Fib(n-1), Fib(n-2)\}$
  - $S(I) = \{Fib(0), Fib(1), \dots, Fib(n)\}$
- 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

```
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**

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

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

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)

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

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 $\boldsymbol{S}$?
  - 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

UNLIKELY THAT WE GET HERE

ROD CUTTING

A "REAL" DYNAMIC PROGRAMMING EXAMPLE

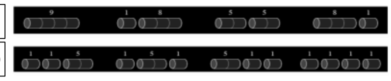
- Input:
  - n: length of rod
  - $p_1, \dots, p_n$: p_i = price of a rod of length i
- Output:
  - Max **income** possible by cutting the rod of length n into any number of **integer** pieces (maybe **no** cuts)

$n = 4$

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

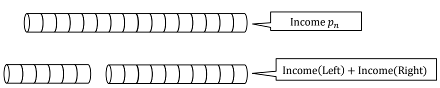
All ways of cutting a rod of length 4

Example output: 10



DYNAMIC PROGRAMMING APPROACH

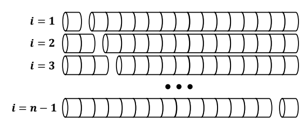
- High level idea **(can just think recursively to start)**
  - Given a rod of length n
  - Either make no cuts, or make a cut and **recurse** on the remaining parts



  - **Where** should we cut?

DYNAMIC PROGRAMMING APPROACH

- Try **all ways** of making that cut
  - I.e., try a cut at positions $1, 2, \dots, n-1$
  - 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 $n - i$

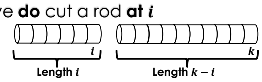
... is max income we can get from the rod size i

+ max income we can get from the rod size $n - i$

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 $\boldsymbol{p_k}$
- If we **do** cut a rod **at $\boldsymbol{i}$**



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

COMPUTING SOLUTIONS BOTTOM-UP

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

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

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)?	$\Theta(n^2)$

Aside: Is this a "quadratic time" algorithm?

Exercise: devise an even simpler DP solution (hint: try "recursing" only once)

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…