CS 4102, Algorithms
Spring 2022

5th Lecture in Unit C

Wrap up DP and Gerrymandering,
then on to Greedy Algorithms!

Dynamic Programming and Gerrymandering

- Let's review / wrap-up a few things from the Gerrymandering lecture
  - Time-complexity
  - Pseudo-polynomial time-complexity for algorithms

Run Time

$$S(j,k,x,y) = S(j-1,k-1,x-R(p_j),y) \vee S\left(j-1,k,x,y-R(p_j)\right)$$

Initialize $S(0,0,0,0) = \text{True}$

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

$\frac{n}{2}$ for $k = 1,\dots,\min(j,\frac{n}{2})$:

nm for $x = 0,\dots,jm$:

nm for $y = 0,\dots,jm$:

$S(j,k,x,y) =$

$S(j-1,k-1,x-R(p_j),y) \vee S\left(j-1,k,x,y-R(p_j)\right)$

Search for True entry at $S(n,\frac{n}{2},>\frac{mn}{4},>\frac{mn}{4})$

$\Theta(n^4m^2)$

$\Theta(n^4 m^2)$

- This looks big! Yes, and it's interesting too! ☺
- Inputs:
  - List (size n) of precincts and counts of voters for Regular Party, $R(p_i)$
  - Number of voters (integer m)
- n is a size of one of the inputs
  - If n doubles, twice as many items in the list that's our input
- But m is an input value (not a size)
  - If m doubles, it's still one integer, one input item
  - But the amount of work grows
  - The complexity depends on the size of this single integer

Size of a Numeric Input-Value

Question: How do we measure the size of an integer?

Answer: the number of bits to represent it.

Example:
The value 4 (decimal) in binary is 100, so the size of “value 4” is 3.

If the size grows by 1, that’s 4 bits. With 4 bits, the value could be 1000 or 8 decimal.

Wait, what? Size of input grows by 1, and the value doubles (4 to 8).
That sounds like exponential! 2^n vs. 2^{n+1}

Pseudo-Polynomial Time

Yes, the *inputSize* (in bits) of value m is $\log_2 m$

$$inputSize = \log_2 m$$
$$m = 2^{inputSize}$$
So $m^2 = (2^{inputSize})^2 = 2^{2 \cdot inputSize}$

Gerrymandering's run-time is <u>exponential</u> because of *size* of input $\boldsymbol{m}$

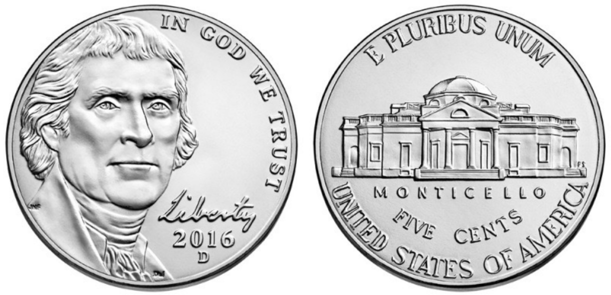
- Because run-time $\Theta(n^4 m^2)$ written in terms of the *value* of m, not the *size* of m
- Input size is really $n + |m| = n + \log m$

This is called **pseudo-polynomial time** (https://en.wikipedia.org/wiki/Pseudo-polynomial_time)
We may see others like this! E.g. Coin-changing DP $\Theta(k \cdot x)$ and Knapsack DP $\Theta(n \cdot C)$

CS4102 Algorithms

Spring 2022 –Hott and Horton

It’s time to “change” things up and start our unit on Greedy Algorithms! ☺



Where We're Going

- Terminology about optimization problems and greedy algorithms
- Example 1: Coin Changing
  - Contrast with dynamic programming approach
  - Proofs of correctness
- Example 2: Interval Scheduling
- …
- **Textbook readings:** CLRS Chapter 16 (go for it!)

Remember Our First Example?

Finding the correct change with minimum number of coins

- **Problem**:
  - Give back the right amount of change, and…
  - Return the fewest number of coins!
- **Inputs**: the dollar-amount to return
  - Also, the set of possible coins
- **Output**: a set of coins

- Let’s re-visit this with more detail

Coin Changing

Imagine a world without computerized cash registers!

The problem: Given an unlimited quantities of pennies, nickels, dimes, and quarters (worth value 1, 5, 10, 25 respectively), determine a set of coins (the *change*) for a given value x using the fewest number of coins.

How Would You Solve This?

- Would this be your algorithm?
  - Generate each possible set of coins that sum to x.
  - Determine which of these sets has the fewest coins.
- No, this is probably *not at all* what you thought of doing!
  - It’s correct. But it’s a *brute force* approach.
- What would you do?
  - Take a moment and try to describe your approach as an algorithm.

Change Making Algorithm

- Given: target value x, list of coins $C = [c_1, \ldots, c_k]$
  (in this case $C = [1, 5, 10, 25]$)
- Repeatedly select the largest coin less than the remaining target value:

while ($x > 0$)
    let $c = \max(c_i \in \{c_1, \ldots, c_k\} \mid c_i \leq x)$
    add c to solution
    $x = x - c$

Observation: We can rewrite this to take $\lfloor n/c \rfloor$ copies of the next largest coin at each step, and reduce x by $(c \cdot \lfloor n/c \rfloor)$
Avoid call to max() by choosing next c_i from largest to smallest.
C must be sorted.

Let’s reflect on this

- What’s its time-complexity?
  - Looks like it’s $O(x)$ in the worst-case. (Why do I say that?)
    - Maybe it’s $O(kx)$ if I really have to do a max() operation at each step
  - We’ll come back to this.
- Does this always work? I.e. how can we prove it to be correct?
  - Intuitively you know it’s true for US coins, right?

Some Terminology Before We Continue…

- **Optimization problems: terminology**
  - A solution must meet certain constraints:
    A solution is *feasible*

  Example: All edges in solution are in graph, form a simple path.
  - Solutions judged on some criteria:

    Objective function

  Example: Sum of edge weights in path is smallest
  - One (or more) feasible solutions that scores highest (by the objective function) is called the *optimal solution(s)*
- Both **dynamic programming** and the **greedy approach** are often good choices for optimization problems.

Greedy Strategy: An Overview

- Greedy strategy:
  - Build solution by stages, adding one item to the partial solution we've found before this stage
  - At each stage, make *locally optimal choice* based on the ***greedy choice*** (sometimes called the *greedy rule* or the *selection function)*
    - Locally optimal, i.e. best given what info we have now
  - Irrevocable: a choice can't be un-done
  - Sequence of locally optimal choices leads to globally optimal solution (hopefully)
    - Must prove this for a given problem!
    - Sometimes basis for *approximation algorithms* or *heuristic algorithms* used to get something close to optimal solution.

We've Seen Greedy Graph Algorithms

- Dijkstra's Shortest Path and Prim's MST are greedy!
- Build solution by adding item to partial solution
  - Dijkstra's: add edge to connect *k*th vertex, where the edges for the *k-1* already selected show the shortest paths to those *k-1* vertices
- Greedy choice
  - Dijkstra's: for all vertices connected to one of the *k-1* vertices processed, choose *w* where *dist(s,w)* is the minimum
- We did have to prove that this sequence of locally optimal choices leads to globally optimal solution

Back to Coin Changing: Correctness?

- Can you think of how you might argue this strategy (algorithm) always choose the optimal solution for coin-changing?

- Maybe argue along these lines:
  – If an algorithm did something different than what our algorithm does, then it won't choose optimal solution.
  – We'll see proof later in slides.

Warm Up, take 2

Given access to unlimited quantities of pennies, nickels, dimes, toms, and quarters (worth value 1, 5, 10, 11, 25 respectively), give 90 cents change using the **fewest** number of coins.

Greedy method's solution

90 cents

Greedy solution not optimal!

90 cents

Warm Up, take 2

Given access to unlimited quantities of pennies, nickels, dimes, toms, and quarters (worth value 1, 5, 10, 11, 25 respectively), give 90 cents change using the **fewest** number of coins.

We can solve coin changing with dynamic programming, too.

Will that work for this set of coins?

Dynamic Programming

- Requires Optimal Substructure
  – Optimal solution to a problem contains optimal solutions to subproblems
- Idea:
  1. Identify the recursive structure of the problem
     - What is the “last thing” done?
  2. Save the solution to each subproblem in memory
  3. Select a good order for solving subproblems
     - “Top Down”: Solve each recursively
     - “Bottom Up”: Iteratively solve smallest to largest

Identify Recursive Structure

$\text{Change}(x)$: minimum number of coins needed to give change for x cents

Possibilities for last coin	Coins needed	
	$\text{Change}(x-25)+1$	if $x \geq 25$
	$\text{Change}(x-11)+1$	if $x \geq 11$
	$\text{Change}(x-10)+1$	if $x \geq 10$
	$\text{Change}(x-5)+1$	if $x \geq 5$
	$\text{Change}(x-1)+1$	if $x \geq 1$

Of course we need to define a data structure to remember partial results and fill it in some order.
We won’t address that here.

Identify Recursive Structure

$\text{Change}(x)$: minimum number of coins needed to give change for x cents

$$\text{Change}(x) = \min \begin{cases} \text{Change}(x-25)+1 & \text{if } x \geq 25 \\ \text{Change}(x-11)+1 & \text{if } x \geq 11 \\ \text{Change}(x-10)+1 & \text{if } x \geq 10 \\ \text{Change}(x-5)+1 & \text{if } x \geq 5 \\ \text{Change}(x-1)+1 & \text{if } x \geq 1 \end{cases}$$

Correctness: The optimal solution must be contained in one of these configurations

Size of input x is how many bits to store x.
Size $n = \lg x$ so $x = 2^{\lg x}$, so $O(k\,2^n)$

Base Case: $\text{Change}(0) = 0$

Running time: $O(kx)$

k is number of possible coins

Is this <u>efficient</u>? Isn't it polynomial?

No, this is <u>pseudo-polynomial</u> time

Greedy Change Making

Greedy approach: Only consider a single case/subproblem, which gives an <u>asymptotically-better</u> algorithm. When can we use the greedy approach?

$c_k]$

coin less than the

remain value:

while($x > 0$)
  let $c = \max(c_i \in \{c_1, \dots, c_k\} \mid c_i \leq x)$
  add c to solution
  $x = x - c$

Observation: We can rewrite this to take $\lfloor n/c \rfloor$ copies of the largest coin at each step. Then loop over values c_i from largest to smallest. Then if C is sorted….

Running time: $O(k)$ **Polynomial-time!**

k is number of possible coins

Greedy vs DP

- Dynamic Programming:
  - Require Optimal Substructure
  - Several choices for which small subproblem
- Greedy:
  - Require Optimal Substructure
  - Must only consider one choice for small subproblem

Log Cutting:
Maximum profit for each last cut

Longest Common Subsequence:
Max length with same last character or with one or the other

Seam Carving:
Min energy seam that could connect with this pixel

Greedy Algorithms

- Require Optimal Substructure
  - Optimal solution to a problem contains optimal solutions to subproblems
  - Only one subproblem to consider!
- Idea:
  1. Identify a greedy choice property
     - How to make a choice guaranteed to be included in some optimal solution
  2. Repeatedly apply the choice property until no subproblems remain

Change Making Choice Property

- Our algorithm's **Greedy choice:**
  Choose largest coin less than or equal to target value
- Leads to optimal solution?
  - For standard U.S. coins: Yes, coin chosen must be part of some optimal solution. We can prove it!
  - For "unusual" sets of coins? We saw a counter-example.
  - For U.S. postage stamps? Hmm…

Correctness of Greedy Algorithm

Optimal solution must satisfy following properties:

- At most 4 pennies
- At most 1 nickel
- At most 2 dimes
- Cannot contain 2 dimes and 1 nickel

Correctness of Greedy Algorithm

Claim: argue that at every step, greedy choice is part of some optimal solution

- **Case 1:** Suppose $x < 5$
  - Optimal solution must contain a penny (no other option available)
  - **Greedy choice:** penny
- **Case 2:** Suppose $5 \leq x < 10$
  - Optimal solution must contain a nickel
    - Suppose otherwise. Then optimal solution can only contain pennies (there are no other options), so it must contain $x > 4$ pennies (contradiction)
  - **Greedy choice:** nickel
- **Case 3:** Suppose $10 \leq x < 25$
  - Optimal solution must contain a dime
    - Suppose otherwise. By construction, the optimal solution can contain at most 1 nickel, so there must be at least 6 pennies in the optimal solution (contradiction)
  - **Greedy choice:** dime

Correctness of Greedy Algorithm

Claim: argue that at every step, greedy choice is part of <u>some</u> optimal solution

- **Case 4:** Suppose $25 \leq x$
  - Optimal solution <u>must</u> contain a quarter
    - Suppose otherwise. There are two possibilities for the optimal solution:
      - If it contains 2 dimes, then it can contain 0 nickels, in which case it contains at least 5 pennies (contradiction)
      - If it contains fewer than 2 dimes, then it can contain at most 1 nickel, so it must also contain at least 10 pennies (contradiction)
  - **Greedy choice:** quarter

Conclusion: in <u>every</u> case, the greedy choice is consistent with <u>some</u> optimal solution

Correctness of Greedy Algorithm

What about that 11-cent coin, the "tom"? How's that break this proof?

- **Claim:** argue that at every step, greedy choice is part of some optimal solution
- **Case 1:** Suppose $n < 5$
  - Optimal solution must contain a penny (no other option available)
  - **Greedy choice:** penny
- **Case 2:** Suppose $5 \leq n < 10$
  - Optimal solution must contain a nickel
    - Suppose otherwise. Then optimal solution can only contain p
  - **Greedy choice:** nickel

This argument no longer holds. Sometimes, it's better to take the dime; other times, it's better to take the 11-cent piece.
For 15: 1 tom + 4 pennies vs. 1 dime + 1 nickel.
For 12: 1 tom + 1 penny vs. 1 dime + 2 pennies

- **Revised Case 3:** Suppose $11 \leq x < 25$
  - Optimal solution must contain a ~~dime~~ tom
    - Suppose otherwise. By construction, the optimal solution can contain at most 1 nickel, so there must be at least 6 pennies in the optimal solution (contradiction).
  - **Greedy choice:** ~~dime~~ tom

Wrap-up on Greedy basics

- An approach to solving ***optimization problems***
  - Finds ***optimal solution*** among set of ***feasible solutions***
- Problem must have ***optimal substructure property***
- Works in stages, applying ***greedy choice*** at each stage
  - Makes locally optimal choice, with goal of reaching overall optimal solution for entire problem
- Proof needed to show correctness

Need more on Optimal Substructure Property?

- Detailed discussion on p. 379 of CLRS (chapter on Dynamic Programming)
  - If A is an optimal solution to a problem, then the components of A are optimal solutions to subproblems
- Another example: Shortest Path in graph problem
  - Say P is min-length path from CHO to LA and includes DAL
  - Let P_1 be component of P from CHO to DAL, and P_2 be component of P from DAL to LA
  - P_1 must be shortest path from CHO to DAL, and P_2 must be shortest path from DAL to LA
  - Why is this true? Can you prove it? Yes, by contradiction. (Try this at home!)

Next: *Interval Scheduling*

– CLRS Section 16.1

Interval Scheduling

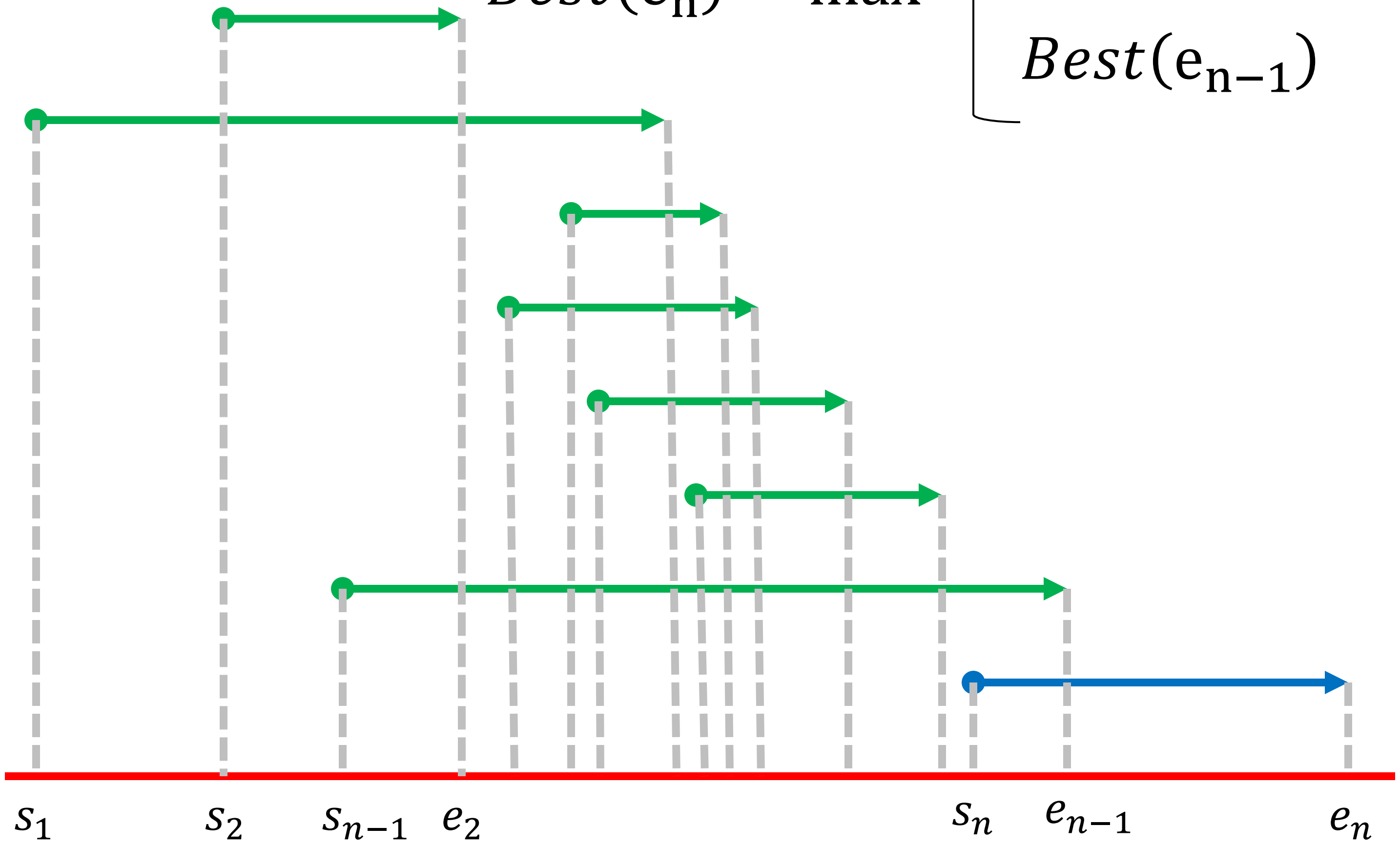
- Input: List of events with their start and end times (sorted by end time)
- Output: largest set of non-conflicting events (start time of each event is after the end time of all preceding events)

[1, 2.25]	Lunch Zoom with friends
[2, 3.25]	CS4102 Live Zoom session
[3, 4]	Streaming department talk
[4, 5.25]	Virtual Office hours
[4.5, 6]	Zoom discussion section
[5, 7.5]	Super Smash Brothers online game night
[7.75, 11]	UVA Basketball Championship re-watch party

Interval Scheduling DP

$Best(t)$ = max # events that can be scheduled before time t

$$Best(\mathrm{e_n}) = \max \begin{cases} Best(\mathrm{s_n}) + 1 & \text{Include event } n \\ Best(\mathrm{e_{n-1}}) & \text{Exclude event } n \end{cases}$$



Greedy Interval Scheduling

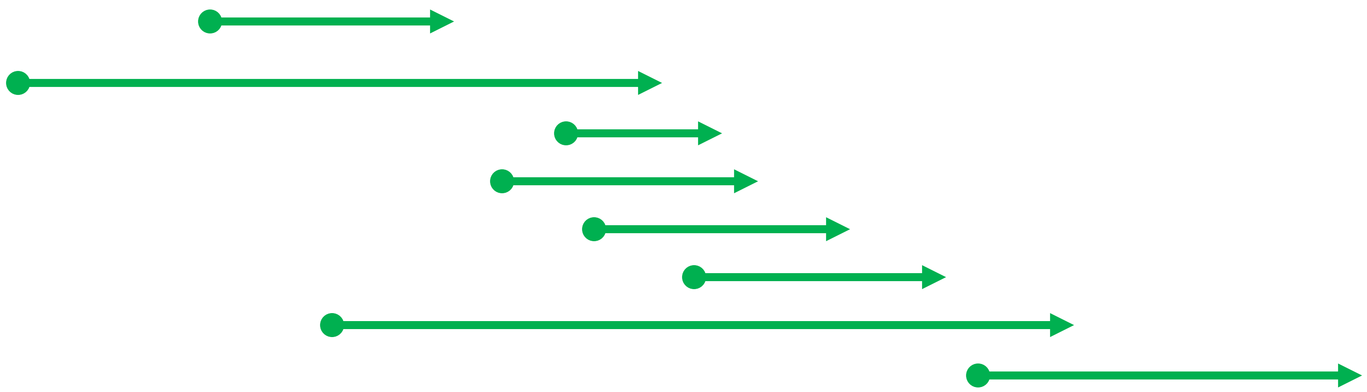
- Step 1: Identify a greedy choice property

Interval Scheduling Algorithm

Find event ending earliest, add to solution,

Remove it and all conflicting events,

Repeat until all events removed, return solution

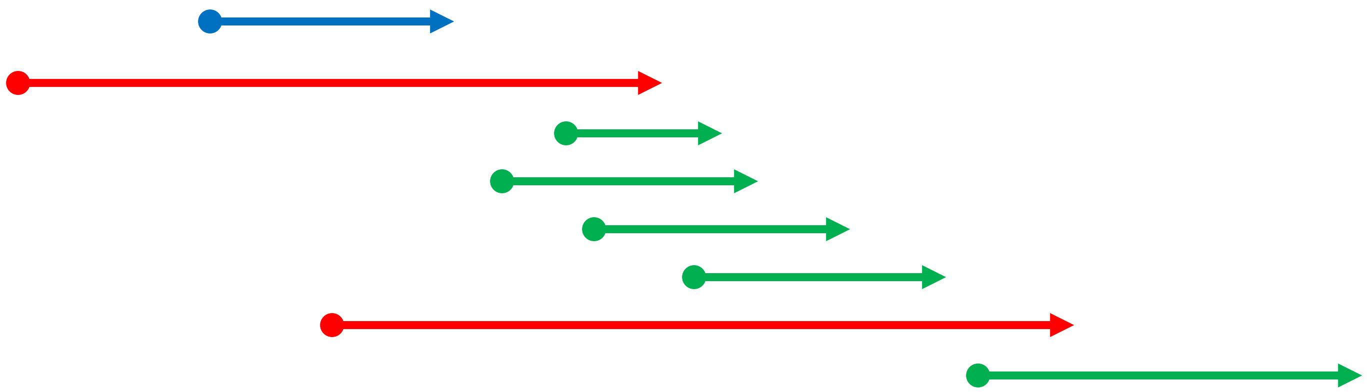


Interval Scheduling Algorithm

Find event ending earliest, add to solution,

Remove it and all conflicting events,

Repeat until all events removed, return solution

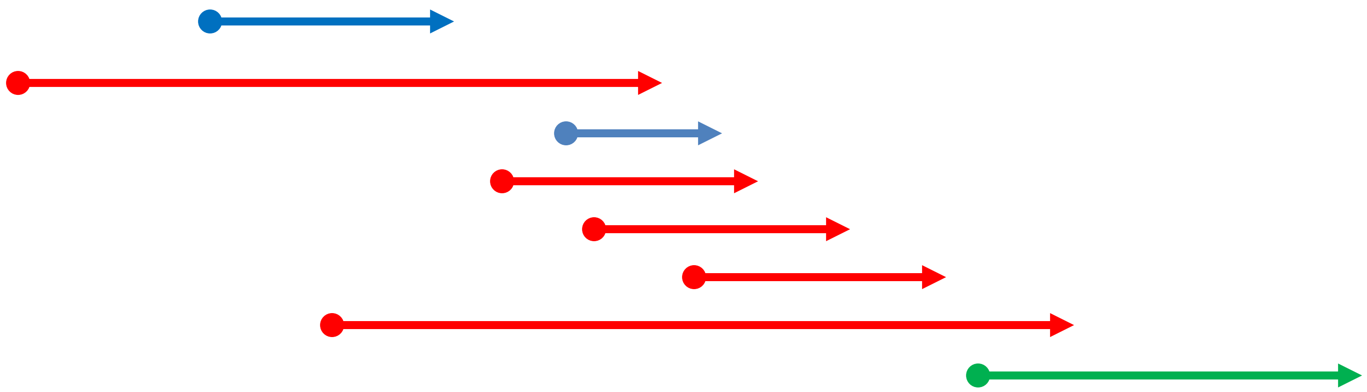


Interval Scheduling Algorithm

Find event ending earliest, add to solution,

Remove it and all conflicting events,

Repeat until all events removed, return solution

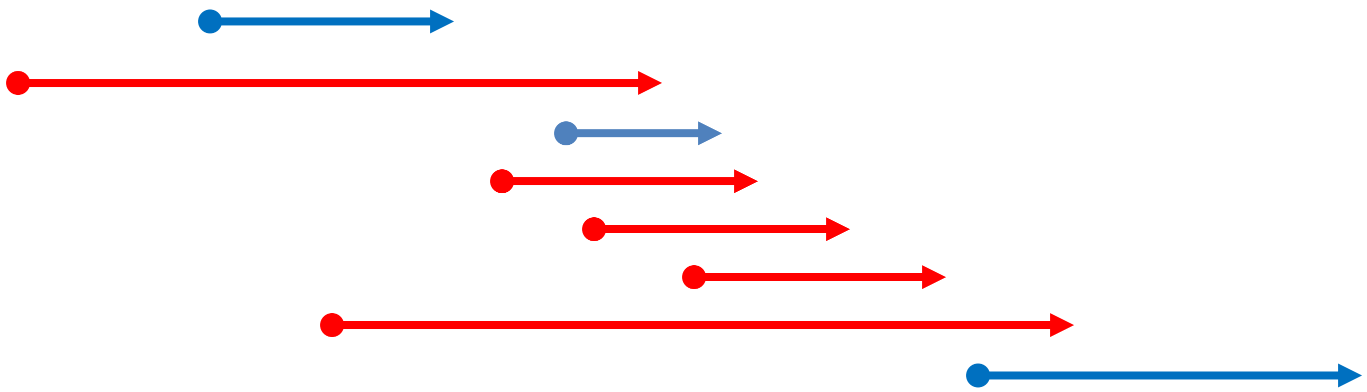


Interval Scheduling Algorithm

Find event ending earliest, add to solution,

Remove it and all conflicting events,

Repeat until all events removed, return solution



Interval Scheduling Run Time

Find event ending earliest, add to solution,

Remove it and all conflicting events,

Repeat until all events removed, return solution

```
Sort intervals by finish time

StartTime = 0
for each interval (in order of finish time):
        if begin of interval > StartTime:
                add interval to solution
                StartTime = end of interval
```

Exchange argument

- Shows correctness of a greedy algorithm
- Idea:
  - Show exchanging an item from an arbitrary optimal solution with your greedy choice makes the new solution no worse
  - How to show my sandwich is at least as good as yours:
    - Show: “I can remove any item from your sandwich, and it would be no worse by replacing it with the same item from my sandwich”

Exchange Argument for Earliest End Time

Exchange Argument for Earliest End Time

- Claim: earliest ending interval is always part of <u>some</u> optimal solution

- Let $OPT_{i,j}$ be an optimal solution for time range $[i,j]$
- Let a^* be the first interval in $[i,j]$ to finish overall
- If $a^* \in OPT_{i,j}$ then claim holds
- Else if $a^* \notin OPT_{i,j}$, let a be the first interval to end in $OPT_{i,j}$
  - By definition a^* ends before a, and therefore does not conflict with any other events in $OPT_{i,j}$
  - Therefore $OPT_{i,j} - \{a\} + \{a^*\}$ is also an optimal solution
  - Thus claim holds