CS4102 Algorithms

Fall 2021 – Floryan and Horton

Greedy Algorithms

CLRS Readings

- Chapter 16, Greedy Algorithms
  - Intro, page 414
  - Section 16.2, Elements of the Greedy Strategy, Knapsack problem
  - Later Section 16.1, Activity Selection problem

Topics

- Greedy Algorithms: Our next algorithmic technique
- How to analyze problems with greedy solutions:
  - Optimal substructure property
  - Greedy choice property
  - Proving correctness of greedy algorithms
- Three example problems
  - Coin Change
  - Activity Selection
  - Knapsack (fractional version)

Optimization Problems

- Greedy algorithms can (sometimes) solve ***optimization problems**:*
  Find the best solution among all ***feasible*** solutions
- An example you know: *Find the shortest path in a weighted graph G from s to v*
  - Form of the solution: a path (and sum of its edge-weights)
- Feasible solutions must meet problem constraints
  - Example: All edges in solution are in graph G and form a simple path from *s* to *v*
- We can get a score for each feasible solution on some criteria:
  We call this the ***objective function***
  - Example: the sum of the edge weights in path
- One (or more) feasible solutions that scores highest (by the objective function) is the ***optimal solution(s)***

Coin Change, Optimal Substructure, and the Greedy Choice Property

Goals!

- First problem with a greedy algorithm solution (***Coin Change***!)

- What is ***optimal substructure***? Why is it useful?
- Making a greedy choice to solve the problem
- What is the ***greedy choice property?***

Everyone Already Knows Many Algorithms!

- Worked retail? You know how to make change!
- Example:
  - My item costs \$4.37. I give you a five dollar bill. What do you give me in change?
  - Answer: two quarters, a dime, three pennies
  - Why? How do we figure that out?

Making Change

- The 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. (Do we have half-dollars? That affects the answer we give.)
- Output: a set of coins

- Note this problem statement is simply a transformation
  - Given input, generate output with certain properties
  - No statement about how to do it.
- Can you describe the algorithm you use?

Optimal Substructure

- This problem has ***optimal substructure***

- ***Optimal Substructure***: If given an optimal solution to the larger problem, it can be seen to be made up of optimal solutions to smaller versions of the same problem.
  - e.g., Optimal solution for giving 15 cents of change contains within it the optimal set of coins to make 5 cents of change (because a dime is part of the solution for 15 cents)
- Another way of stating it:
  If A is an optimal solution to a problem, then the components of A are optimal solutions to subproblems

Optimal Substructure

- This problem has ***optimal substructure***

- ***Lemma 1***: If a problem has optimal substructure, then a greedy algorithm MIGHT solve it (but not necessarily).
- ***Lemma 2***: If a greedy algorithm solves the problem, then it has optimal substructure.

- ***Lesson***: Check for optimal substructure to see if a greedy algorithm MIGHT be applicable. Also gives hints as to what the algorithm might be!!

Optimal Substructure

- This problem has ***<u>optimal substructure</u>***

- Claim (we will prove this):

- If $\mathrm{C} = \{c_1, c_2, \ldots, c_n\}$ is the optimal set of coins to make A cents of change:
- Then $C' = \{c_2, c_3, \ldots, c_n\}$ is the optimal set of coins to make $A - c_1$ cents of change.

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!)

A Change Algorithm

1. Consider the largest coin
2. How many go into the amount left?
3. Add that many of that coin to the output
4. Subtract the amount for those coins from the amount left to return
5. If the amount left is zero, done!
6. If not, consider next largest coin, and go back to Step 2

Evaluating Our Greedy Algorithm

- How much work does it do?
  - Say C is the amount of change, and N is the number of coins in our coin-set
  - Loop at most N times, and inside the loop we do:
    - A division
    - Add something to the output list
    - A subtraction, and a test
  - We say this is O(N), or linear in terms of the size of the coin-set
- Could we do better?
  - Is this an *optimal algorithm*?
  - We need to do a proof somehow to show this

Another Change Algorithm

- Give me another way to do this?

- Brute force:
  - Generate all possible combinations of coins that add up to the required amount
  - From these, choose the one with smallest number
- What would you say about this approach?

- There are other ways to solve this problem
  - *Dynamic programming*: build a table of solutions to small subproblems, work your way up

Algorithm for making change

- This algorithm makes change for an amount *A* using coins of denominations
  $denom[1] > denom[2] > \cdots > denom[n] = 1$.

- Input Parameters: *denom, A*
- Output Parameters: None

-
```
greedy_coin_change(denom, A) {
     i = 1
     while (A > 0) {
          c = A / denom[i]
          println("use " + c + " coins of denomination " + denom[i])
          A = A - c * denom[i]
          i = i + 1
     }
}
```

Making change proof

- One methodology for proving correctness of greedy algorithms:

- A greedy algorithm is correct if the following hold:
  - The problem has ***optimal substructure***
  - The algorithm has the ***greedy choice property*** (see next slide)

Making change proof

- What is the ***greedy choice property***?

- Your algorithm makes some greedy choice and then continues
  - e.g., choose largest coin, then continue

- Prove that the ***one thing*** the greedy algorithm selects MUST be in some optimal solution to the problem.

Making change proof

- Proving the ***<u>greedy choice property</u>***?

- Claim: For making *A* cents of change, some optimal solution MUST contain the largest coin such that $c_i \leq A$

Proof

- Overview of proof:
  - Assume largest coin NOT in some optimal solution
  - Ok, some other coins must be in there instead.
  - 4 Cases:
    - Largest coin that fits is penny (1 cent) //this one is trivial though!
    - Largest coin that fits is nickel (5 cent)
    - Largest coin that fits is dime (10 cent)
    - Largest coin that fits is quarter (25 cent)

Proof

- Largest coin that fits is penny (1 cent)   //this one is trivial though!
  - means A < 5
  - Only penny fits, so penny must be in some optimal solution!

- Largest coin that fits is nickel (5 cent)
  - Assume nickel not in optimal solution. Note A >= 5
  - Pennies are only other option, so 5 or more pennies in optimal solution
  - But I can swap out 5 of those pennies with a nickel
    - Solution decreases by 4 coins!! Contradiction!!

Proof

- Largest coin that fits is Dime (10 cent)
  - Assume dime not in optimal solution. Note A >= 10 and A < 25
  - So the optimal solution contains:
    - >= 2 nickels, some number of pennies (might be 0)
    - 1 nickel, some pennies (at least 5)
    - all pennies (more than 10)
  - In each case above, I can swap a dime in for some combination of nickels or pennies
    - Solution decreases by 1, 5, or 9 coins respectively. Contradiction!

- Largest coin that fits is quarter (25 cent)
  - Assume quarter not in optimal solution. Note A >= 25
  - So the optimal solution contains:

Proof

- Largest coin that fits is quarter (25 cent)
  - Assume quarter not in optimal solution. Note A >= 25
  - So the optimal solution contains:
    - 2 dimes, 1 nickel, some pennies maybe
    - 2 dimes, 0 nickels, 5 or more pennies
    - 1 dime, 3 nickels, 0 or more pennies
    - 1 dime, 2 nickels, 5 or more pennies
    - 1 dime, 1 nickel, 10 or more pennies
    - 1 dime, 0 nickel, 15 or more pennies
    - 0 dime, 5 nickels, 0 or more pennies
    - …
- For each case above, a quarter can be swapped back in for more than 1 coin to make the solution better!! Contradiction!

How would a failed proof work?

- Prove greedy choice property for denominations 1, 6, and 10
- This is going to fail because the algorithm doesn't work. Let's see it!
  - For A = 12, greedy outputs 10,1,1
  - Best answer is 6,6

How would a failed proof work?

- Largest coin that fits is Dime (10 cent)
  - Assume dime not in optimal solution. Note A >= 10
  - So the optimal solution contains:
    - 2 or more six-cent coins, pennies maybe (could be 0)
    - 1 six-cent coin, at least 4 pennies
    - 0 six-cent coins, at least 10 pennies

- For the second two, we can do the exchange, but NOT for the first one. The proof doesn't work!!

Knapsack Problems

Knapsack Problems

- Pages 425-427 in textbook
- **Description:** Thief robbing a store finds n items, each with a profit amount p_i and a weight w_i
  - Wants to steal as valuable a load as possible
  - But can only carry total weight C in their knapsack
  - Which items should they take to maximize profit?
- Form of the solution: an x_i value for each item, showing if (or how much) of that item is taken
- Inputs are: C, n, the p_i and w_i values

Two Types of Knapsack Problem

- 0/1 knapsack problem
  - Each item is discrete: must choose all of it or none of it.
    So each x_i is 0 or 1
  - Greedy approach does not produce optimal solutions
  - But dynamic programming does

- Fractional knapsack problem (AKA continuous knapsack)
  - Can pick up fractions of each item.
    So each x_i is a value between 0 or 1
  - A greedy algorithm finds the optimal solution

Formal Statement of Fractional Knapsack Problem

- Given n objects and a knapsack of capacity C, where object i has weight w_i and earns profit p_i, find values x_i that maximize the total profit

$$\sum_{i=1}^{n} x_i p_i$$

subject to the constraints

$$\sum_{i=1}^{n} x_i w_i \leq C, \quad 0 \leq x_i \leq 1$$

Greedy Approach

- Let's use a **greedy strategy** to solve the fractional knapsack
  - Build solution by stages, adding one item to partial solution found so far
  - 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 choice given what info available 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!
    - Approximation algorithms, heuristic algorithms

A Bit More Terminology

- Problems solvable by both Dynamic Programming and the Greedy approach have the **optimal substructure property:**
  - An optimal solution to a problem contains within it optimal solutions to subproblems
  - This allows us to build a solution one step at a time, because we can solve increasingly smaller problems with confidence

Greedy Approach for Fractional Knapsack?

- Build up a partial solutions:
  - Determine which of the remaining items to add
  - How much can you add (its x_i)
  - Repeat until knapsack is full (or no more items)

- Which item to choose next?
  What's a good **greedy choice** (AKA **greedy selection)**?
- Let's try several obvious options on this example:

n = 3, C = 20

Item	Value	Weight
1	25	18
2	24	15
3	15	10

Possible Greedy Choices for Knapsack

Greedy choice #1: by highest profit value

n = 3, C = 20

Item	Value	Weight
1	25	18
2	24	15
3	15	10

Select item 1 first, then item 2, then item 3. Take as much of each as fits!

1. Item 1 first. Can take all of it, so x_1 is 1. Capacity used is 18 of 20. Profit so far is 25.
2. Item 2 next. Room for only 2 units, so x_2 is 2/15 = 0.133. Capacity used is 20 of 20. Profit so far is 25 + (24 x 0.133) = 28.2.
3. Item 3 would be next, but knapsack full! x_3 is 0. **Total profit is 28.2. x_i = (1, .133, 0)**

Possible Greedy Choices for Knapsack

Greedy choice #2: by lowest weight

n = 3, C = 20

Item	Value	Weight
1	25	18
2	24	15
3	15	10

Select item 3 first, then item 2, then item 1. Take as much of each as fits!

1. Item 3 first. Can take all of it, so x_3 is 1. Capacity used is 10 of 20. Profit so far is 15.
2. Item 2 next. Room for only 10 units, so x_2 is 10/15 = 0.667. Capacity used is 20 of 20. Profit so far is 15 + (24 x 0.667) = 31.
3. Item 1 would be next, but knapsack full! x_1 is 0. **Total profit is 31.0.** **x_i = (0, .667, 1)**

Note it's better than previous greedy choice. Best possible?

Possible Greedy Choices for Knapsack

Greedy choice #3: highest value-to-weight ratio

n = 3, C = 20

Item	Value	Weight	Ratio
1	25	18	1.4
2	24	15	1.6
3	15	10	1.5

Select item 2 first, then item 3, then item 1. Take as much of each as fits!

1. Item 2 first. Can take all of it, so x_2 is 1. Capacity used is 15 of 20. Profit so far is 24.
2. Item 3 next. Room for only 5 units, so x_1 is 5/10 = 0.5. Capacity used is 20 of 20. Profit so far is 24 + (15 x 0.5) = 31.5.
3. Item 1 would be next, but knapsack full! x_1 is 0. **Total profit is 31.5. x_i = (0, 1, 0.5)**

This greedy choice produces optimal solution! Must prove this (but we won't today).

Fractional Knapsack Algorithm

```
FRACTIONAL_KNAPSACK(a, C)
n = a.last
for i = 1 to n
    ratio[i] = a[i].p / a[i].w
sort(a, ratio)
weight = 0
i = 1
while (i ≤ n and weight < C)
    if (weight + a[i].w ≤ C)
        println "select all of object " + a[i].id
        weight = weight + a[i].w
    else
        r = (C – weight) / a[i].w
        println "select " + r + " of object " + a[i].id
        weight = C
    i = i+1
```

Worst-case runtime:
for loop and while loop take $\theta(n)$ time,
sorting takes $\theta(n\lg n)$ time,
so algorithm takes $\theta(n\lg n)$ time

Another Knapsack Example to Try

- Assume for this problem that: $\sum_{i=1}^{n} w_i \leq C$
- Ratios of profit to weight:

  $p_1/w_1 = 5/120 = .0417$

  $p_2/w_2 = 5/150 = .0333$

  $p_3/w_3 = 4/200 = .0200$

  $p_4/w_4 = 8/150 = .0533$

  $p_5/w_5 = 3/140 = .0214$

- What order do we examine items?
- What are the x_i values that result?
- What's the total profit?

Proving a Greedy Algorithm Correct

- For fractional knapsack, we can prove greedy choice of p_i/w_i leads to optimal solution
- In general, given a greedy algorithm, how do approach such a proof?
- Recall we've done this for Dijkstra's SP and Prim's MST
- We can compare the solution our algorithm finds with an optimal solution
  - Show they're the same
  - Or, assume they're not and show a contradiction
  - Remember *exchange argument* for Dijkstra's or for Prim's?

0/1 knapsack

n = 3, C = 4

Item	Value	Weight	Ratio
1	3	1	3
2	5	2	2.5
3	6	3	2

Let’s try this same greedy solution with the 0/1 version
- New example inputs →

1. Item 1 first. So x_1 is 1.
   Capacity used is 1 of 4. Profit so far is 3.
2. Item 2 next. There’s room for it! So x_2 is 1. Capacity used is 3 of 4.
   Profit so far is 3 + 5 = 8.
3. Item 3 would be next, but its weight is 3 and knapsack only has 1 unit left!
   So x_3 is 0. **Total profit is 8.** **x_i = (1, 1, 0)**

But picking items 1 and 3 will fit in knapsack, with total value of 9
- Thus, the greedy solution does not produce an optimal solution to the 0/1 knapsack algorithm
- Greedy choice left unused room, but we can’t take a fraction of an item
- The 0/1 knapsack problem doesn’t have the *greedy choice property*

Activity Selection

Activity-Selection Problem

- Problem: You and your classmates go on Semester at Sea
  - Many exciting activities each morning
  - Each starting and ending at different times
  - Maximize your "education" by doing as many as possible
    - This problem: they're all equally good!
    - Another problem: they have weights (we need DP for that one)
- Welcome to the ***activity selection problem***
  - Also called ***interval scheduling***

The Activities!

Id	Start	End	Activity
1	9:00	10:45	Fractals, Recursion and Crayolas
2	9:15	10:15	Tropical Drink Engineering with Prof. Bloomfield
3	9:30	12:30	Managing Keyboard Fatigue with Swedish Massage
4	9:45	10:30	Applied ChemE: Suntan Oil or Lotion?
5	9:45	11:15	Optimization, Greedy Algorithms, and the Buffet Line
6	10:15	11:00	Hydrodynamics and Surfing
7	10:15	11:30	Computational Genetics and Infectious Diseases
8	10:30	11:45	Turing Award Speech Karaoke
9	11:00	12:00	Pool Tanning for Engineers
10	11:00	12:15	Mechanics, Dynamics and Shuffleboard Physics
11	12:00	12:45	Discrete Math Applications in Gambling

Generalizing Start, End

Id	Start	End	Len	Activity
1	0	6	7	Fractals, Recursion and Crayolas
2	1	4	4	Tropical Drink Engineering with Prof. Bloomfield
3	2	13	12	Managing Keyboard Fatigue with Swedish Massage
4	3	5	3	Applied ChemE: Suntan Oil or Lotion?
5	3	8	6	Optimization, Greedy Algorithms, and the Buffet Line
6	5	7	3	Hydrodynamics and Surfing
7	5	9	5	Computational Genetics and Infectious Diseases
8	6	10	5	Turing Award Speech Karaoke
9	8	11	4	Pool Tanning for Engineers
10	8	12	5	Mechanics, Dynamics and Shuffleboard Physics
11	12	14	3	Discrete Math Applications in Gambling

Greedy Approach

1. Select a first item.
2. Eliminate items that are incompatible with that item.
   (I.e. they overlap, not part of a feasible solution)
3. Apply the ***greedy choice*** (AKA *selection function*) to pick the next item.
4. Go to Step 2

What is a good greedy choice for selecting next item?

Some Possibilities

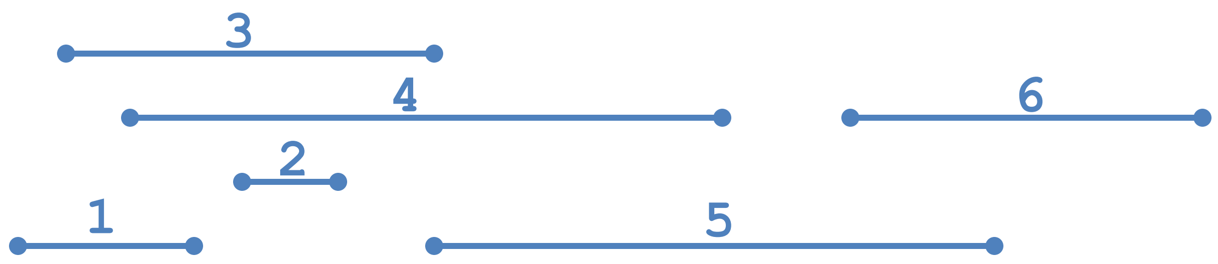
1. Maybe pick the next ***compatible activity*** that starts earliest?
   – “Compatible” here means “doesn’t overlap”
2. Or, pick the shortest one?
3. Or, pick the one that has the least conflicts (i.e. overlaps)?
4. Or…?

Activity-Selection

- Formally:
  - Given a set S of n activities

    s_i = start time of activity i

    f_i = finish time of activity i
  - Find max-size subset A of compatible activities



- Assume (wlog) that $f_1 \leq f_2 \leq \ldots \leq f_n$

Activity Selection: A Greedy Algorithm

- The algorithm using the best **greedy choice** is simple:
  - Sort the activities by <u>finish time</u>
  - Schedule the first activity
  - Then schedule **the next activity in sorted list which starts after previous activity finishes**
  - Repeat until no more activities
- Or in simpler terms:
  - Always pick the compatible activity that finishes earliest

Optimal Substructure Property

- Remember?
- Detailed discussion on p. 379 (in chapter on Dynamic Programming)
  - If A is an optimal solution to a problem, then the components of A are optimal solutions to subproblems
- Reminder: Example 1, Shortest Path
  - 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.
    - Do it! In-class exercise

Activity Selection: Optimal Substructure

- Let k be the minimum activity in the solution A (i.e., the one with the earliest finish time). Then A - $\{k\}$ is an optimal solution to $S' = \{i \in S: s_i \geq f_k\}$
  - In words: once activity #1 is selected, the problem reduces to finding an optimal solution for activity-selection over activities in *S* **compatible** with activity #1
  - Proof: if we could find optimal solution B' to S' with $|B| > |A - \{k\}|$,
    - Then B U $\{k\}$ is compatible
    - And $|B$ U $\{k\}| > |A|$ -- contradiction! We said A is the overall best.
- Note: book's discussion on p. 416 is essentially this, but doesn't assume we choose the 1st activity

Back to Semester at Sea…

Id	Start	End	Len	Activity
2	1	4	4	Tropical Drink Engineering with Prof. Bloomfield
4	3	5	3	Applied ChemE: Suntan Oil or Lotion?
1	0	6	7	Fractals, Recursion and Crayolas
6	5	7	3	Hydrodynamics and Surfing
5	3	8	6	Optimization, Greedy Algorithms, and the Buffet Line
7	5	9	5	Computational Genetics and Infectious Diseases
8	6	10	5	Turing Award Speech Karaoke
9	8	11	4	Pool Tanning for Engineers
10	8	12	5	Mechanics, Dynamics and Shuffleboard Physics
3	2	13	12	Managing Keyboard Fatigue with Swedish Massage
11	12	14	3	Discrete Math Applications in Gambling

Solution: 2, 6, 9, 11

Visualizing these Activities

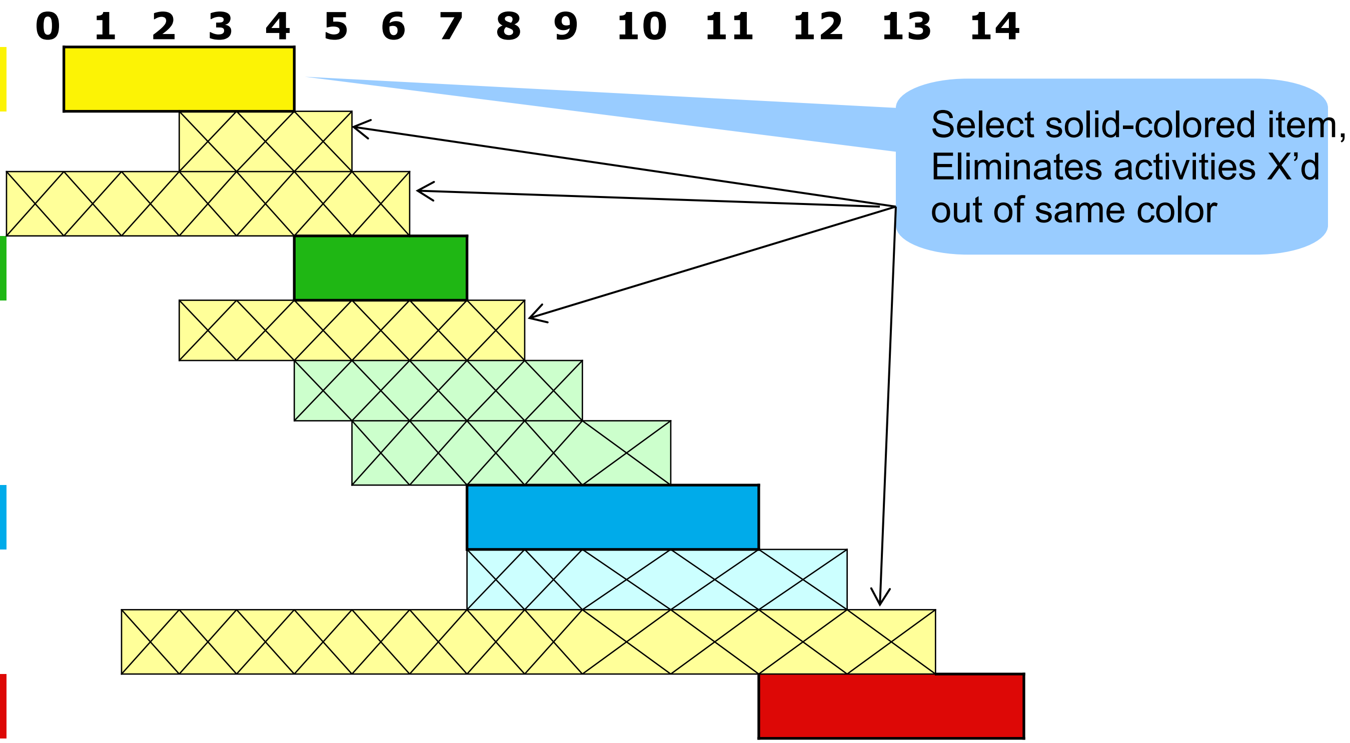
ID	Start	End		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	6																
2	1	4																
3	2	13																
4	3	5																
5	3	8																
6	5	7																
7	5	9																
8	6	10																
9	8	11																
10	8	12																
11	12	14																

Visualizing these Activities in Solution

ID	Start	End		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	6																
2	1	4																
3	2	13																
4	3	5																
5	3	8																
6	5	7																
7	5	9																
8	6	10																
9	8	11																
10	8	12																
11	12	14																

Sorted, Then Showing Selection and Incompatibilities

ID	Start	End
2	**1**	**4**
4	3	5
1	0	6
6	**5**	**7**
5	3	8
7	5	9
8	6	10
9	**8**	**11**
10	8	12
3	2	13
11	**12**	**14**



Book's Recursive Greedy Algorithm

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1  // start with the activity after the last added activity
while m ≤ n and s[m] < f[k]  // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} U RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return Ø
```

- Add dummy activity a_0 with $f_0 = 0$, so that sub-problem S_0 is entire set of activities S
- Initial call: RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n)
- Run time is θ(n), assuming the activities are already sorted by finish times

Non-recursive algorithm

```
greedy-interval (s, f)
   n = s.length
   A = {a1}
   k = 1   # last added
   for m = 2 to n
        if s[m] ≥ f[k]
             A = A U {am}
             k = m
   return A
```

- s is an array of the intervals' start times
- f is an array of the intervals' finish times, <u>sorted</u>
- A is the array of the intervals to schedule
- How long does this take?

55

Does Greedy Always Find Optimal Solution?

- Yes, we can prove that the greedy algorithm always “stays ahead”!

Does Greedy Always Find Optimal Solution?

- Yes, we can prove that the greedy algorithm always “stays ahead”!
  - How?

- Overall idea: Show the i’th interval algorithm chooses always ends earlier than optimal solution

Does Greedy Always Find Optimal Solution?

- ***<u>Lemma 1</u>***: Let $G = \{g_1, g_2, \dots, g_n\}$ be greedy algorithm intervals and $O = \{o_1, o_2, \dots, o_m\}$ be the optimal solution

Show that $\forall_{i \geq 1} \;\; g_i.f \leq o_i.f$ //f is finish time

Does Greedy Always Find Optimal Solution?

Rest of proof:

$\lvert\mathrm{G}\rvert \neq \lvert\mathrm{O}\rvert$	//G not optimal ftpoc
$\lvert G\rvert < \lvert O\rvert$	//definition of optimal
$g_n.f \leq o_n.f$	//by lemma 1
$g_n.f \leq o_{n+1}.s$	//from previous line
//CONTRADICTON	//greedy could have chosen o_{n+1}