CS4102 Algorithms

Spring 2021 – Floryan and Horton

Module 3, Day 2

- Show DP solution to 0/1 knapsack
  - (Solution not in textbook)
- Show greedy solution to Activity Selection problem
  - (CLRS Section 16.1)

Reminder: 0/1 knapsack

n = 3, C = 4

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

Greedy solution for fractional knapsack doesn't work with the 0/1 version

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

- 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*

Reminders about Dynamic Programming

- Requires Optimal Substructure
  - Solution to larger problem contains the solutions to smaller ones
- Strategy:
  1. Identify the recursive structure of the problem
     - What is the “last thing” done?
  2. Formulate a data structure (array, table) that can look-up solution to any sub-problem in constant time
  3. Select a good order for solving subproblems
     - “Bottom Up”: Iteratively solve smallest to largest
     - “Top Down”: Solve each recursively. (We won’t do this for 0/1 knapsack.)

Dynamic programming solution to 0/1

We need to:

- Identify a recursive definition of how a larger solution is built from optimal results for smaller sub-problems.

For 0/1 knapsack, what a sub-problem solution look like?
What can be "smaller"?

- – Smaller capacity for the knapsack
- – Fewer items

Some assumptions and observations

- Given a set S of the objects and a capacity C
  - We assume the optimal solution is O, a subset of S
  - For example, the items in O could be the bolded ones:
    $S = \{ \mathbf{s_1}, s_2, \mathbf{s_3}, \ldots, s_{k-1}, \mathbf{s_k}, \ldots, s_n \}$
  - Note that the last item s_n may or may not be in the solution O
- Let's use subscripts on O_k and S_k when we're talking about the first *k* items
- BTW, we'll assume C and all w_i are integer values
  - And, most books etc. use "W" for what we're calling C

Recursive Structure

What's a recursive definition of how a solution of **size n** is built from optimal results for smaller sub-problems? $S = \{ s_1, s_2, s_3, \ldots, s_{n-1}, s_n \}$

- Let's say $s_n \notin O_n$ (last item **is not** in optimal solution for S_n):
  - Last item didn't add anything to best solution for smaller subproblem
  - We need optimal solution O_{n-1} for the following smaller subproblem S_{n-1}:
    n-1 items using <u>same</u> knapsack capacity C

- Let's say $s_n \in O$ (last item **is** in optimal solution for S_n):
  - Last item contributed w_i to total weight we're carrying
  - We need optimal solution O_{n-1} for the following smaller subproblem S_{n-1} :
    n-1 items using <u>reduced</u> capacity $C\text{-}w_n$

(Note that "getting smaller" decreases number of items and also maybe capacity.)

First Step: Getting Things Started

- For sub-problems, what variables change in size?
  - Maybe C (the capacity) and definitely k (number of items to steal)
- Define what we're calculating: call it **Knap(k, w)**
  - Note: we'll use "w" for the changing capacity value in Knap(), but keep "C" as the overall total capacity for the entire problem. (Sorry if confusing!)
- Whether we do recursion of work bottom-up, we need to know the smallest cases
- Some small or boundary cases:
  - No knapsack capacity (w=0), can't add an item, so Knap(k, 0) = 0
  - Nothing to steal (k=0), so Knap(0, w) = 0

Three cases to calculate Knap(k, w)

- Three cases for calculating Knap(k, w):
  1. There is sufficient capacity to add item s_k to the knapsack, and that creates an optimal solution for k items
  2. There is sufficient capacity to add item s_k to the knapsack, and that does **NOT** create an optimal solution for k items
  3. There is insufficient capacity to add item s_k to the knapsack

- Case 3 is easy to determine; we'll have to compute whether 1 or 2 is optimal
  – How do we know which is optimal? Compute both, pick larger value!

Case 1: Sufficient capacity and Optimal

- There is sufficient capacity to add item s_k to the knapsack, and that creates an optimal solution for k items
- Thus, our solution for the first k items is when we add item s_k to the optimal solution for the first k-1 items
- But by adding item s_k to the knapsack, we have reduced capacity
  - In particular, we only have **$w\text{-}w_k$** for to steal the first **k-1** items
- So the value for **$Knap(k, w) = v_k + Knap(k\text{-}1, w\text{-}w_k)$**

Case 2: Sufficient Capacity but Non-optimal

- There is sufficient capacity to add item s_k to the knapsack, and that does **NOT** create an optimal solution for k items

- Thus, our solution for the first k items is when we do NOT add item s_k to the solution for the first k-1 items
  - Since we are **not** adding item s_k to the knapsack, the solution is the optimal solution to steal the first **k-1** items with the **same capacity**
  - So **Knap(k, w) = Knap(k-1, w)**

Case 3: Insufficient Capacity

- There is insufficient capacity to add item s_k to the knapsack
  - This is because $w - w_k < 0$ (i.e. $w < w_k$)

- Then **Knap(k, w) = Knap(k-1, w)**
  - Since we can't add item s_k to the knapsack, the solution is the same as the first k-1 items with the same capacity
  - Note that this formula is the same as case 2

Putting It All Together

- Recursively define solutions to sub-problems
- Base Case

  Knap(k,0) = 0

  Knap(0,w) = 0

- Recursive Case

Subproblems are smaller!

$$\text{Knap}(k, w) = \max\big(\underbrace{\text{Knap}(k\text{-}1, w)}_{\text{No room for } s_k \text{ or not part optimal solution}},\ \underbrace{\text{Knap}(k\text{-}1, w\text{-}w_k) + v_k}_{s_k \text{ is part of optimal solution}} \big)$$

Reminders about Dynamic Programming

- Requires Optimal Substructure
  – Solution to larger problem contains the solutions to smaller ones
- Strategy:
  1. Identify the recursive structure of the problem
     - What is the “last thing” done?
  2. Formulate a data structure (array, table) that can look-up solution to any sub-problem in constant time
  3. Select a good order for solving subproblems
     - “Bottom Up”: Iteratively solve smallest to largest
     - “Top Down”: Solve each recursively. (We won’t do this for 0/1 knapsack.)

Lookup Table

- We want a data-structure that allows us to lookup a sub-problem value in O(1) time

- Knap(k, w) has two parameters, so two-dimensional array works great.

- Make an array called V[k, w]
  - Store solution to Knap(k, w) at position V[k, w]

Determining the cases

- To determine between cases 1 and 2
  - Simply compute both values, and take the higher

```
if (w-w_k< 0) // not room for item k
      V[k, w] = V[k-1, w] // best result for k-1 items
else {
      val_with_kth = v_k + V[k-1, w-w_k] // Case 1 above
      val_for_k-1 = V[k-1, w] // Case 2 above
      V[k, w] = max( val_with_kth, val_for_k-1 )
}
```

Put Values in Table

- Write a loop that fills in the table one cell at a time
- The table fills in one row at a time, moving rightwards and downwards

V[k,w]	w = 0	w = 1	w = 2	…	w = C
k = 0	0	0	0	0	0
k = 1	0				
k = 2	0				
…	0				
k = n	0				

Pseudo-code

```
Knapsack(v, w, C) {
   for (w = 0 to C) V[0, w] = 0
   for (k = 0 to n) V[k, 0] = 0
   for (k = 1 to n) {          // loop over all rows
           for (w = 1 to C) {  // loop over all columns
             if (w-w_k <  0)    // not room for item k
                V[k, w] = V[k-1, w] // best result for k-1 items
             else {
                val_with_kth = v_k + V[k-1, w-w_k] // Case 1 above
                val_for_k-1 = V[k-1, w]          // Case 2 above
                V[k, w] = max( val_with_kth, val_for_k-1 )
             }
           }
   }
   return V[n,C]
}
```

But our solution is only the value!

- Value V[n, C] is the optimal value

- To find which items were chosen, we can trace backward through the table starting at V[n, C]
  - If V[k, w] = V[k-1, w], then **s_k is not an item in the knapsack** (this was from cases 2 and 3). Look at V[k-1, w] next.
  - Otherwise, **s_k is an item in the knapsack**, and we look at V[k-1, w-w_k] next (this was from case 1)

- More in live session!

Back to Greedy with the Activity Selection Problem

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

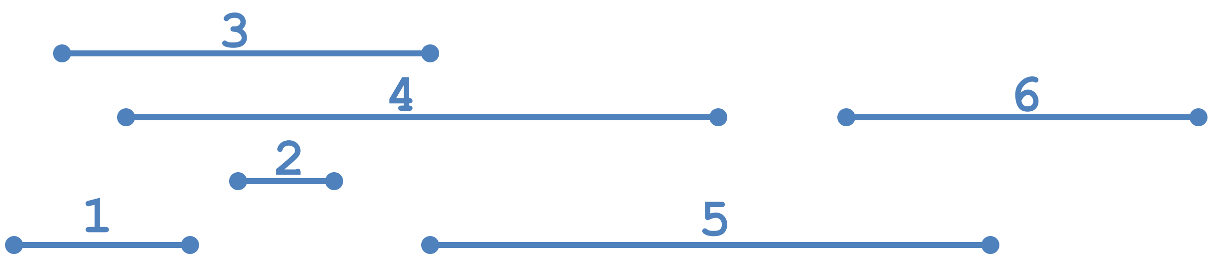
1. Maybe pick the next ***compatible activity*** that starts earliest?
   – "Compatible" here means "doesn't overlap"
2. Or, pick the shortest one?
3. Or, pick the one that has the least conflicts (i.e. overlaps)?
4. Or…?

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

- So 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 $B \cup \{k\}$ is compatible
    - And $|B \cup \{k\}| > |A|$ -- 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

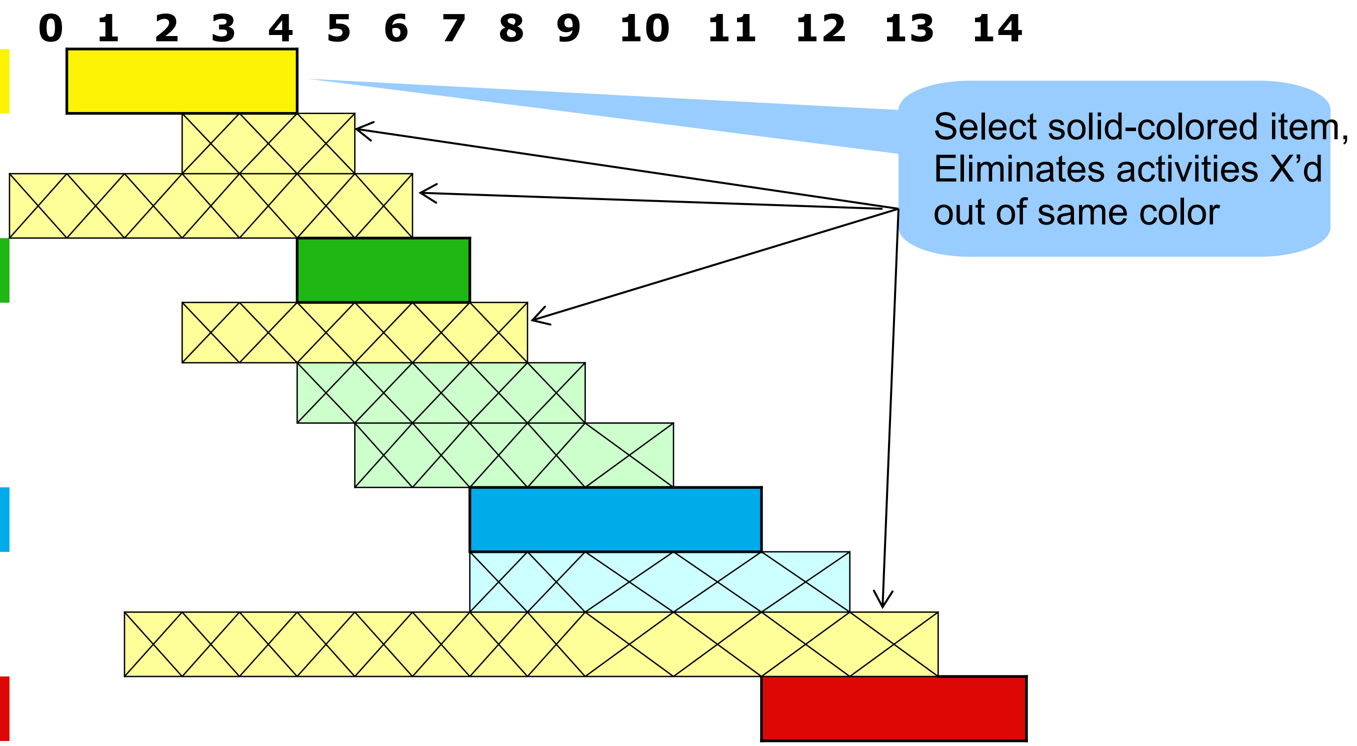
ID	Start	End		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	6																
2	1	4																
3	2	13																
4	3	5																
5	3	8																
6	5	7																
7	5	9																
8	6	10																
9	8	11																
10	8	12																
11	12	14																

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 = {a_1}
   k = 1    # last added
   for m = 2 to n
        if s[m] ≥ f[k]
              A = A U {a_m}
              k = m
   return A
```

- s is an array of the intervals' start times
- f is an array of the intervals' finish times
- A is the array of the intervals to schedule
- How long does this take?

35

Does Greedy Always Find Optimal Solution?

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