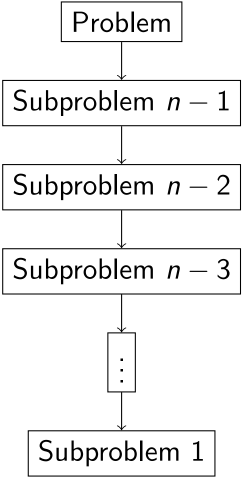
CSE 431/531: Analysis of Algorithms (Summer 2023)
Dynamic Programming

Chen Xu

July 6-18, 2023

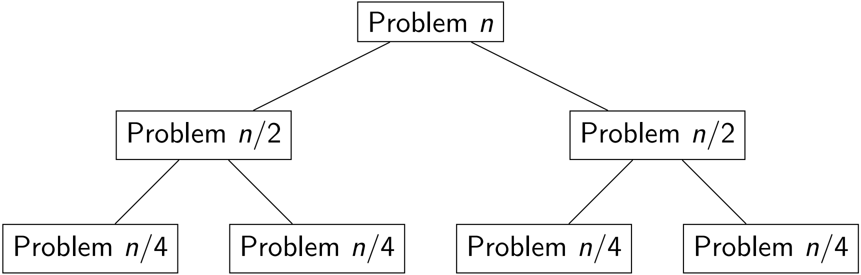
Subproblem structure

- Greedy algorithm:



Subproblem structure

- Divide-and-Conquer(DaC):



- Each node is an **independent** subproblem instance.

Subproblem structure

- Dynamic Programming(DP) also breaks a problem down into simpler sub-problems in a recursive manner. However the recursive structure is **not a fixed structure**. Unlike the tree-like structure of DaC, the subproblems can **overlap**.

The recursive structure of DP

- We usually characterize the subproblem using parameters. i.e. subproblem that solves the sum from the ith element to the jth element that are larger than k. We can denote the solution as $sum[i, j, k]$.
- Note that sometimes the number of parameters of the subproblem may be more than the original problem. We call those parameters **extra information**. For instance, the $sum[i, j, k, 1]$ with the mth element counted and $sum[i, j, k, 0]$ with the mth element not counted.
- Every DP algorithm comes with a recursive structure that associates connections between parameterized subproblems. We can write it in the form of a function like below.

Recursive structure example

$$opt[i, j] = \max\{opt[i-1, j], opt[i, j-1]\}$$

Implementation of DP algorithms

- Because DP algorithm breaks down a problem in a non-trivial manner, in the implementation of DP algorithms, we need to consider two things:
    1. The extra information of how we break down the problem.
    2. The solutions to each of the subproblem associated with the extra information.
- Usually, an array or a table is required to store the partial solutions and extra information.

0/1 Knapsack problem

0/1 Knapsack problem

Input: an integer total weight $W > 0$
a set of n items, each with an integer weight $w_i > 0$
a value $v_i > 0$ for each item i

Output: a subset S of items that
maximizes $\sum_{i \in S} v_i$ s.t. $\sum_{i \in S} w_i \le W$

- 0/1 Knapsack problem is different from Fractional Knapsack problem. In 0/1 Knapsack problem, if we take an item, we have to take the whole item.

Does Greedy work?

- Total weight: 5

Item	Weight (units)	Value	Value-to-Weight ratio
A	1	60	60
B	2	100	50
C	3	120	40

- Greedy Algorithm total value: 160
  1. Item A (Remaining capacity: 4)
  2. Item B (Remaining capacity: 2)
- Optimal choice total value: 220
  1. Item B (Remaining capacity: 3)
  2. Item C (Remaining capacity: 0)
- Greedy algorithm does not always produce the optimal solution for the 0/1 Knapsack problem.

Is it easy?

- There is always the trivial algorithm that records the value of every combination of choices. The total running time is up to 2^n.
- Just by removing the ability of taking fractional item, it suddenly makes the problem extremely hard. There seems to be no algorithms to solve it in polynomial time.
- However we can apply DP and solve it in time $O(nW)$, W is the total weight given in the input.

Recursive structure for 0/1 Knapsack Problem

- Define the optimum solution for the subproblem:
  $opt[i, W']$ when budget is W' and items are $\{1, 2, 3, ..., i\}$.
  The solution for the original problem is therefore $opt[n, W]$.
- We consider items one by one.
  1. If the ith item's weight is greater than W', we know that at this W' we cannot take item i, so optimal solution would be the same as $opt[i-1, W']$.
  2. If the ith item's weight is less than or equal W', we know that it can be possible that we take item i. We check whether taking or not taking it would give us higher value.
     If we don't take it, we know the optimal solution is the same as $opt[i-1, W']$.
     If we take it, we know that among the total W' there is at least w_i that is taken by item i, we want to find the optimal solution for the remaining weight at $opt[i-1, W'-w_i]$ and add v_i to it. Then we take the greater one to be the optimal solution of $opt[i, W']$.

Recursive structure for 0/1 Knapsack Problem

- We have the following recursive structure:

$$\mathrm{opt}\left[i, W'\right] = \begin{cases} 0 & i = 0 \\ \mathrm{opt}\left[i-1, W'\right] & i > 0, w_i > W' \\ \max\left\{ \begin{array}{c} \mathrm{opt}\left[i-1, W'\right] \\ \mathrm{opt}\left[i-1, W' - w_i\right] + v_i \end{array} \right\} & i > 0, w_i \leq W' \end{cases}$$

The DP table for storing the recursive structure

- Total weight: 5

Item	Weight (units)	Value	Value-to-Weight ratio
C	3	120	40
A	1	60	60
B	2	100	50

- Observe that $opt[i, W']$ is 2-dimensional. We can allocate a table to store the recursive structure. We have 3 items and $W = 5$. The size of the table is $(3+1) \times (5+1) = 24$

i \ W'	0	1	2	3	4	5
0						
1						
2						
3						

The DP table for storing the recursive structure

- Total weight: 5

Item	Weight (units)	Value	Value-to-Weight ratio
C	3	120	40
A	1	60	60
B	2	100	50

- Observe that $opt[i, W']$ is 2-dimensional. We can allocate a table to store the recursive structure. We have 3 items and $W = 5$. The size of the table is $(3+1) \times (5+1) = 24$

i \ W'	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						

The DP table for storing the recursive structure

- Total weight: 5

Item	Weight (units)	Value	Value-to-Weight ratio
C	3	120	40
A	1	60	60
B	2	100	50

- Observe that $opt[i, W']$ is 2-dimensional. We can allocate a table to store the recursive structure. We have 3 items and $W = 5$. The size of the table is $(3+1) \times (5+1) = 24$

i \ W'	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	120	120	120
2						
3						

The DP table for storing the recursive structure

- Total weight: 5

Item	Weight (units)	Value	Value-to-Weight ratio
C	3	120	40
A	1	60	60
B	2	100	50

- Observe that $opt[i, W']$ is 2-dimensional. We can allocate a table to store the recursive structure. We have 3 items and $W = 5$. The size of the table is $(3+1) \times (5+1) = 24$

i \ W'	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	120	120	120
2	0	60	60	120	180	180
3						

The DP table for storing the recursive structure

- Total weight: 5

Item	Weight (units)	Value	Value-to-Weight ratio
C	3	120	40
A	1	60	60
B	2	100	50

- Observe that $opt[i, W']$ is 2-dimensional. We can allocate a table to store the recursive structure. We have 3 items and $W = 5$. The size of the table is $(3+1) \times (5+1) = 24$

i \ W'	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	120	120	120
2	0	60	60	120	180	180
3	0	0	100	160	180	220

Using the extra information to recover the choice

- The reason why we used colored text in the DP table is that we want to record every decision we have taken during the process of filling the table.
- We can use this extra information to recover the “witness” of this optimum solution. In particular, the set of chosen items is the witness for this problem.

i \ W'	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	120	120	120
2	0	60	60	120	180	180
3	0	0	100	160	180	220

Using the extra information to recover the choice

i \ W'	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	120	120	120
2	0	60	60	120	180	180
3	0	0	100	160	180	220

- We start from $opt[n, W]$, the red color means we picked this item 3. We then subtract the weight of item 3 and look for the weight at $5 - w_3 = 3$ of the last row.
- The $opt[2, 3]$ gave us a pink color. This means we did not pick item 2. Then we do not subtract the weight and still look for the weight 3 of the last row.
- The $opt[1, 3]$ gave us a red color. This means we picked item 1. So we picked item 1 and 3 which are B and C from the input.

Running time is pseudo polynomial

- In our approach, every entry in the DP table takes $O(1)$ to fill. There are $O(nW)$ entries in this table so the running time is $O(nW)$.
- This is polynomial only when $W \in O(n^c)$ for some constant c. But W can be unbounded (i.e. exponentially big in terms of n). In this case the DP table will be huge.

Intuition

- How do we know what the problem recursive structure look like?
  Try to parameterize the problem. Some solutions to the parameterized subproblems are built upon other parameterized subproblems.

Weighed Activity Selection

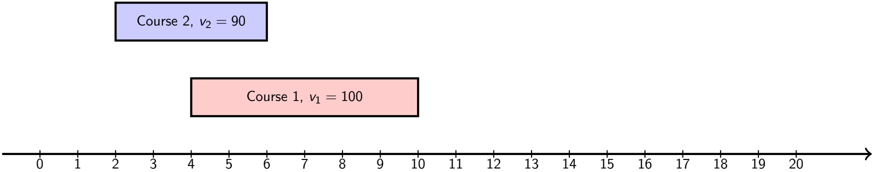
Weighted Activity Selection

Input: n jobs, job i with start time s_i and finish time f_i
each job has a value $v_i > 0$
i and j are compatible if $[s_i, f_i)$ and $[s_i, f_i)$ are disjoint

Output: a subset of compatible jobs with maximum total value.

- Recall that we have can solve the non-weighted activity selection problem by greedy algorithm.
- Does greedy algorithm still work for the weighted version?

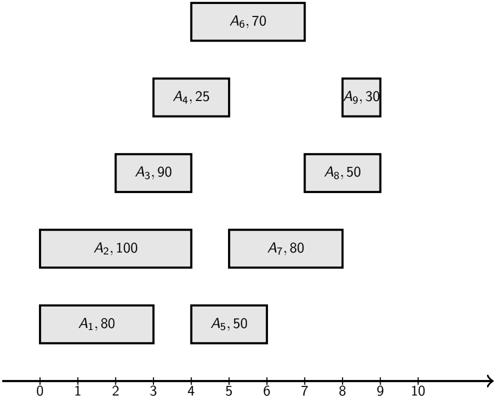
Try greedy strategy



- Optimum: 100, greedy: 90.
- We can instantly find a counter example such that taking the earliest finish time does not produce the optimum solution for the weighted version.
- It appears that we cannot make our decision solely based on duration/weight/weight to duration ratio.

Try Dynamic Programming

- Each item can either be in the solution or not in the solution. We still consider the items one by one based on the order of finish time. The $opt[i]$ stands for the solution for activities from 1 to i of finish time order.



Recursive structure

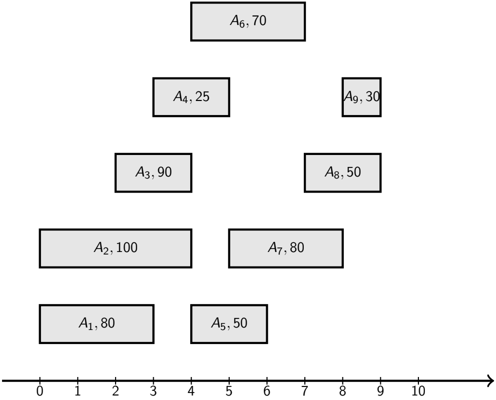
- If the activity i is not in the solution, then we know the optimal solution is the same as $opt[i-1]$.
- If the activity i is in the solution, suppose this activity starts at s_i, then we know that the optimal solution from 0 to s_i plus this activity i will be the optimal solution from 0 to s_i. The subproblem includes the subset of all compatible activities whose finish times are before s_i. Since we have sorted the activities by the finish time, it is easy to find the last activity p_i whose finish time is right before s_i. We know that the optimal solution is build based on that $opt[p_i]$ plus the value of activity i.

Recursive structure

$$\text{opt}[i] = \max\left\{\text{opt}[i-1], v_i + \text{opt}\left[p_i\right]\right\}$$

Running the DP algorithm on the example

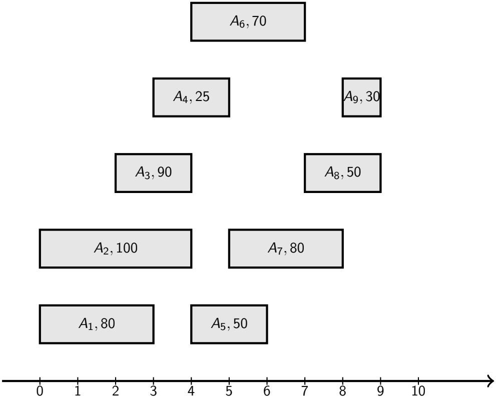
- $\text{opt}[i] = \max\{\text{opt}[i-1], v_i + \text{opt}[p_i]\}, \text{opt}[0] = 0$



i	opt[i]
1	80
2	100
3	100
4	80+25=105
5	100+50=150
6	100+70=170
7	105+80=185
8	170+50=220
9	220

Recovering our choices using the extra information pre_i

- $\text{opt}[i] = \max\left\{\text{opt}[i-1], v_i + \text{opt}\left[p_i\right]\right\}, \text{opt}[0] = 0$



i	opt[i]	pre_i
1	80	0
2	100	0
3	100	-
4	80+25=105	1
5	100+50=150	3
6	100+70=170	3
7	105+80=185	4
8	170+50=220	6
9	220	-

- The optimum solution is 220 with activity 2,6,8.

The algorithm

DP algorithm for weighted activity selection

Sort the activities by finish times
Compute $p_1, p_2, ..., p_n$
$opt[0] = 0$
for $i = 1$ to N **do**
    if $opt[i-1] \geq opt[p_i] + v_i$ **then**
        $opt[i] = opt[i-1]$
        $pre_i = null$
    else
        $opt[i] = opt[p_i] + v_i$
        $pre_i = p_i$
return $opt[n]$

Running time

- Sorting takes $O(n \log n)$.
- Computing $p_1, p_2, \ldots p_n$ takes $O(n \log n)$.
- Filling in DP array takes $O(n)$.
- So the total running time is $O(n \log n)$.

Longest Common Subsequence

- What is a subsequence of a string?
- $T = abdb$, $S = abcdbddab$, T is a subsequence of S.
- What is a common subsequence of a pair of strings?
- $T = abdb$, $S = abcdbddab$, $R = acccbcbdb$, T is one of the common subsequences of R and S.
- Question: Is T the longest common subsequence?

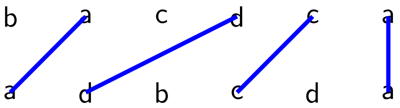
Longest Common Subsequence

LCS - Longest Common Subsequence

Input: String $A[1..n]$ and $B[1..m]$
Output: The longest common subsequence C of A and B

- Example: $A = bacdca$, $B = adbcda$, $LCS(A, B) = adca$
- Exercise: Implement a trivial algorithm

Dynamic Programming



- We want to catch every moment when some matched pair of characters appears.
- Define the subproblem $opt[i,j] = LCM(A[1..i], B[1..j])$, we know that setting different i,j will guarantee that we don't miss any of that moment. We then just need to solve every one of them, for every i,j.

Recursive structure

- There can be three cases:
  1. $A[i]$ matches $B[j]$. This case we know $opt[i, j]$ increased by 1.
  2. $A[i]$ does not match $B[j]$, We want to ask if $A[i]$ matches with any previous $B[j']$ or $B[j]$ matches with any previous $A[i']$. We consider both cases so not a single case can be missed. Either we keep i and reduce j by 1 or keep j and i by 1.
- $$opt[i,j] = \begin{cases} \mathrm{opt}[i-1, j-1] + 1 & \text{if } A[i] = B[j] \\ \max \begin{cases} opt[i-1, j] \\ \mathrm{opt}[i, j-1] \end{cases} & \text{if } A[i] \neq B[j] \end{cases}$$

LCS DP algorithm

LCS

for $j \leftarrow 0$ to m **do**
  $\text{opt}[0, j] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  $\text{opt}[i, 0] \leftarrow 0$
  for $j \leftarrow 1$ to m **do**
    if $A[i] = B[j]$ **then**
      $\text{opt}[i, j] \leftarrow \text{opt}[i-1, j-1] + 1$; Store i, j yellow;
    else if $\text{opt}[i, j-1] \geq \text{opt}[i-1, j]$ **then**
      $\text{opt}[i, j] \leftarrow \text{opt}[i, j-1]$; Store i, j pink;
    else
      $\text{opt}[i, j] \leftarrow \text{opt}[i-1, j]$; Store i, j red;

DP table

	1	2	3	4	5	6
A	b	a	c	d	c	a
B	a	d	b	c	d	a

	0	1	2	3	4	5	6
0	0 $\perp$	0 $\perp$	0 $\perp$	0 $\perp$	0 $\perp$	0 $\perp$	0 $\perp$
1	0 $\perp$	0 $\leftarrow$	0 $\leftarrow$	1 $\nwarrow$	1 $\leftarrow$	1 $\leftarrow$	1 $\leftarrow$
2	0 $\perp$	1 $\nwarrow$	1 $\leftarrow$	1 $\leftarrow$	1 $\leftarrow$	1 $\leftarrow$	2 $\nwarrow$
3	0 $\perp$	1 $\uparrow$	1 $\leftarrow$	1 $\leftarrow$	2 $\nwarrow$	2 $\leftarrow$	2 $\leftarrow$
4	0 $\perp$	1 $\uparrow$	2 $\nwarrow$	2 $\leftarrow$	2 $\leftarrow$	3 $\nwarrow$	3 $\leftarrow$
5	0 $\perp$	1 $\uparrow$	2 $\uparrow$	2 $\leftarrow$	3 $\nwarrow$	3 $\leftarrow$	3 $\leftarrow$
6	0 $\perp$	1 $\nwarrow$	2 $\uparrow$	2 $\leftarrow$	3 $\uparrow$	3 $\leftarrow$	4 $\nwarrow$

The answer is *adca*. Characters matched are marked yellow.

Running time

- The running time is $O(m * n)$ since every entry takes constant to compute.

Can we save some space?

- The DP table requires $O(m * n)$ to store.
- It appears that the ith row of the DP table depends only on the $(i-1)$th row. If we don't want to recover the subsequence, can we remove the redundant space?
- Exercise: Rewrite the program to reduce the space usage to $O(n)$