CS4349 SPRING 2020. DP Notes.

Dynamic programming is a design technique where a solution to a problem is obtained from solutions of subproblems. Thus, solution to a problem is written in terms of the solutions of subproblems.

1) Maximum Subarray Sum.

Measure employed is $C[i]$ that denotes the maximum sum obtainable from a subarray terminating at position i of the given array $A[1..n]$

<u>Recursive definition</u>.

$$C[i] = \max(A[i],\ A[i] + C[i-1]).$$

The max. sum obtained by a subarray terminating at position 'i' either includes a prefix or consists solely of $A[i]$.
If it includes a prefix then such a prefix must be a subarray that terminates at position $i-1$, otherwise we will not have a valid subarray terminating at 'i'.
Thus in this case optimum value corr. to $A[i]+C[i-1]$.

2. Longest Common Subsequence.

Measure: $C[i,j]$ denotes the length of longest <u>CS</u> of X_i, Y_j.

<u>Recursive defn.</u>

$$C[i,j] = \begin{cases} \text{If } (X[i] = Y[j]) : C[i-1, j-1]+1 \\ \text{Else } \max\begin{cases} C[i-1, j], \\ C[i, j-1] \end{cases} \end{cases}$$

<u>Computational rule</u>: Recursive defn. and the associated realization of the measure. E.G. In 1) we apply the defn. of $C[i]$ from left to right $(2..n)$ where $C[1] = A[1]$.

In 2) We init $\forall_j\ C[0,j]=0$ & $\forall_i\ C[i,0]=0$. Then we fill the table row-wise.

Both $C[i]$ ~~of~~ 1) and $C[i,j]$ of 2 are filled in $O(1)$ time.

This yields a T.C. of $O(n)$ for 1) and $O(mn)$ for 2).

3) L.C. Substring:

$C[i,j]$: Length of the longest common substring of suffix of X_i and suffix of Y_j.

R. Defn: $C[i,j] = \begin{cases} 0 & \text{if } (X[i] \neq Y[j]) \\ 1 + C[i-1, j-1] & \text{otherwise.} \end{cases}$

Computation: Row-wise from top-row. $\forall_i\, C[i,0]=0$, $\forall_j\, C[0,j]=0$. T.C. $= O(mn)$.

4) 0-1 Knapsack problem.

A straight forward method might require enumeration of all subsets of the given set of objects.

Measure $T[i,j]$: optimum value that can be obtained by choosing from $\{O_1, O_2, O_3, \ldots O_i\}$ with a weight limit of j.

Recursive defn: $T[i,j] = \text{Max}\{T[i-1, j],\ T[i-1, j-w_i] + v_i\}$

Valid only when $j \geq w_i$

If O_i is not chosen then we must spend weight 'j' optimally on the first $i-1$ objects.

Otherwise, one spends w_i units of weight on O_i. Thus, remaining $j-w_i$ must be optimally used among $\{O_1, O_2, \ldots O_{i-1}\}$

Computation: Rowwise from top. $O(nW)$. $n \rightarrow$ # of objects
$W \rightarrow$ Weight limit.

Rod cutting. A particular length can be employed at most once.

This reduces to 0-1 Knapsack where the length of the given rod (Original) L Corr. W.

Each piece with length $= l_i$ and price $= p_i$

Corr to an object with weight $= w_i$ and value $= v_i$.

6) <u>Coin Change problem.</u> There are n coins specified as $(c_1, c_2, c_3, \dots c_n)$ where c_i denotes the value of i^{th} coin. Given a target amount V, you are supposed to issue change for V in terms of $c_1 \dots c_n$ s.t. the number of coins is minimized. A coin can be used any number of times.

Recursive defn. $T[i,j]$ denotes the min. number of coins from $c_1, c_2 \dots c_i$ that are required to provide change for amount j.

$$T[i,j] = \min\{\underbrace{T[i-1, j]}_{c_i \text{ is not used}}, \underbrace{1 + T[i, j - c_i]}_{c_i \text{ is used and can be reused}}\}$$

Note that the first index is unaltered.
1 → The coin c_i is to be counted

7) <u>Matrix Chain Multiplication:</u> Input $A_1, A_2, \dots A_n$ matrices where $A_1 \times A_2 \times A_3 \times A_4 \dots A_n$ must be computed. $\forall_i\ A_i \times A_{i+1}$ is well defined.

$C_{m \times n} \times D_{n \times p} = E_{m \times p}$ n is the common dimension, <u>cost = m.n.p.</u>

Say $A_1 = 10 \times 100$, $A_2 = 100 \times 5$ and $A_3 = 5 \times 50$. This input is specified as $(10, 100, 5, 50)$ <u>i.e</u> 10×100, <u>100×5, 5×50</u>. <u>i.e.</u> $A_i : P_{i-1} \times P_i$

$((A_1 A_2) A_3)$: Cost $= 5000 + 2500 = \underline{7500}$.

Cost of $((A_1)(A_2 A_3)) = 100 \times 5 \times 50 + 10 \times 100 \times 50 = \underline{75,000}$.

<u>Optimal substructure</u> Optimal solution for $A_i A_{i+1} \dots A_j$ requires optimal solutions for $A_i \dots A_k$, $A_{k+1} \dots A_j$ $\forall_k$.

<u>Measure</u> $C[i,j]$: Optimum number of scalar multiplications required for $A_i \times A_{i+1} \times \dots A_j$.

$$C[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{if } i < j \end{cases}$$

Computation. $\forall_i \; C[i,i] = 0$ $\underbrace{\forall_{i>j} \; C[i,j]}_{\text{Invalid and will not be used.}}$

" $\forall_{i<n} \; C[i,i+1] = p_{i-1} \, p_i \, p_{i+1}$

We fill the remaining $\frac{n(n-1)}{2} - n - 1$ entries in the increasing order of $\underline{j-i}$.

Each cell takes $O(n)$ time & there are $\frac{(n-1)(n-2)}{2}$ cells.

$\therefore$ Time complexity is $O(n^3)$.

Space complexity is $\Theta(n^2)$

Scanned with CamScanner

Greedy notes CS4349.002.
Fall 2019, Dept. of CS, UTD.

① Fractional knapsack problem: Given n objects specified as $\forall_i\ (w_i, v_i)$ where w_i and v_i are weight and value of the i^{th} object, you are to identify the maximum cumulative value one can obtain for a specified weight limit of W. The term "fractional" refers to the fact that an item can be divided and $f \cdot w_i$ yields a value of $f \cdot v_i$ where $0 \le f \le 1$. $\forall_i (v_i > 0, w_i > 0)$ and $W > 0$.

That is we must maximize $\sum f_i v_i$ subject to $\sum f_i w_i \le W$.
We assume that $W < \sum w_i$ & $\forall_i\ v_i/w_i$ are distinct.

Algo A: 1. Sort objects by decreasing order of v_i/w_i. From now on $\tilde{O}_1$ refers to the first object in the sorted order, $\tilde{O}_2$ second etc.. Note that a particular $\tilde{O}_i$ can be traced back to a particular O_j

2. a) Remaining weight $= W$. Total value, $V = 0$. $\forall_i\ C[i] = 0$

b) While (remaining weight, $W_r > 0$)

{ Choose the maximum possible fraction of the leftmost available object $\tilde{O}_i$ such that $f_i w_i \le W_r$.

$V \leftarrow V + f_i v_i$; $W_r \leftarrow W_r - f_i w_i$; $C[i] \leftarrow f_i$;

}

3. Print V, $\forall_i\ C[i]$.

Obs1. We do not run out of objects because (a) $W < \sum w_i$.
Note that if $W \geq \sum w_i$ we can simply select all objects and $V = \sum v_i$.

Obs2 The solution is of the form $\left(1^* f^? 0^*\right)$, where f denotes a fraction in $(0,1)$. (C[i] read from left to right) The first time we are unable to select an entire object we run out of residual weight, thereafter, no object can be selected. Thus, the last selected object has a fraction of 1 or 1<. If a fraction of 1< is selected corr. $\tilde{O}_i$ then it is the last selected object.

Scanned with CamScanner

The solution vector consisting of chosen fractions is a vector in $\mathbb{R}^n$ where each dimension is in $[0, 1]$. The candidate solutions set here has larger cardinality than that of 0-1 knapsack.

Claim. The solution of A is optimum.

Proof: Let the solution of A be $\underline{S = (f_1, f_2, f_3, \ldots f_n)}$ where $f_i = x[i]$.
We assume the contrary, i.e. let $S^* = (f_1^*, f_2^*, \ldots f_n^*)$ be the optimum where $S \neq S^*$.
Let i be the leftmost index where $f_i \neq f_i^*$. By construction, (i.e. design of A)
$f_i > f_i^*$. Let $\delta = f_i - f_i^*$.
From S^* we construct another solution as follows (new soln is called S^{**})
From items $i+1 \ldots n$ of S^* reduce a total weight of $\delta * (w_i)$ and add the same weight to O_i then
we obtain $S^{**} = (f_1, f_2, f_3 \ldots f_i, f_{i+1}^{**}, f_{i+2}^{**}, \ldots f_n^{**})$
The additional value that S^{**} has w.r.t. S^* at index i
is $\underline{\delta \cdot v_i}$. Likewise, the max. amount by which S^* can exceed S^{**} in indices $i+1 \ldots n$ is $\delta . w_i . \frac{v_{i+1}}{w_{i+1}}$

because $\forall_i \quad \frac{v_i}{w_i} > \frac{v_{i+1}}{w_i}$.

However $\quad \delta . w_i . \frac{v_i}{w_i} > \delta . w_i . \frac{v_{i+1}}{w_{i+1}}$.

Therefore the value corr. to sol $S^{**} >$ that of S^* ($\leftrightarrow$).

It follows that $S^* \neq S$ cannot be an optimum solution.

② Activity selection problem.

Input: A set of activities $A = \{a_1, a_2, a_3, \dots a_n\}$ where a_i is specified by a half open interval $[s_i, s_f)$ where $s_i \to$ start time and $s_j \to$ finish time. Two activities a_i, a_j can be scheduled together if $[s_i, f_i) \cap [s_j, f_j) = \phi$.

Otherwise, only one of them can be scheduled.

Output: Determine the maximum number of activities that can be scheduled.

Algorithm A: 1. Sort activities in the increasing order of finish time. Let us call this new sequence $A^* = (a_1^*, a_2^*, \dots a_n^*)$.

Def: If 2 activities can be scheduled together, say a_i, a_j, then they are compatible otherwise they are incompatible.

Note: a_1^* is always feasible because there is no prior selected activity.

2. a. $S[1] \leftarrow (f_1^*, 1)$. S: Solution array.

Last_selected $= 1$; selected_index $= 1$;

b. Select the leftmost activity with index > selected_index that is compatible with S[last_selected].

If such activity exists, say a_j^* then

{ last_selected++; S[last_selected] ← (f_j, j);

Go to 2b. }

Otherwise { Terminate and print last_selected, S; }

Claim: S is an optimal solution.

Proof: Assume the contrary, let $S^* (\neq S)$ be an optimum solution where $|S^*| > |S|$.

Let both solutions be arranged in the increasing order of finish times.

$S = (i_1, i_2, i_3, \dots i_p)$ and $S^* = (j_1, j_2, j_3, \dots j_k)$ where i_x, j_y denote the corresponding activities and $k > p$. (I)

Let x be the leftmost activity where $i_x \neq j_x$, i.e. S & S^* differ.

Obs. F.T. of $i_1 \le$ F.T. of j_1 by the design of Algorithm A. -- (A)

Claim F.T of $i_x \le$ F.T. of j_x for all x.

Proof: By induction: (A) shows that basis holds.

Assume that $F.T.(i_{q-1}) \le F.T.(j_{q-1})$ then we will show that $F.T(i_q) \le F.T(j_q)$ proving the inductive step.

After selecting i_{q-1} Algorithm A picks the activity that is compatible with i_{q-1} and has the least F.T. If j_q is such an activity then i_q can be j_q otherwise i_q is an activity whose F.T $\le$ F.T. of j_q.

(Because any activity that starts after j_{q-1} finishes is compatible with i_{q-1} because $F.T.(j_{q-1}) \geq F.T(i_{q-1})$.)

$\therefore$ It follows that $F.T.(i_q) \le F.T(j_q)$. The claim follows.

From the above claim it follows that $F.T.(i_x) \leq F.T.(j_x)$.

Further, because $F.T.(i_{x-1}) = F.T.(j_{x-1})$, i_x and j_{x-1}, and i_{x-1} and j_x are compatible pairs. Moreover, i_x is also compatible with j_{x+1} because $F.T.(i_x) \le F.T.(j_x)$.

ent (Thus, in S^* j_x can be replaced by i_x to obtain S^{**} that is also a feasible solution with a cardinality of k.

The above argument can be repeated for any subsequent index.)

Thus, j_{p+1} --- j_k can be appended to S as i_p and j_{p+1} are compatible. Thus, Algorithm A would have appended these activities or corresponding activities with finish times at most F.T.s of activities in S^*.

Thus, $k \ne p$ or S is also an optimal solution. ∎

: By Argument A, one can replace each activity of S^* with corr. activity of S.