

COL 351 : ANALYSIS & DESIGN OF ALGORITHMS

LECTURE 21

DYNAMIC PROGRAMMING II: KNAPSACK

SEPT 22, 2026

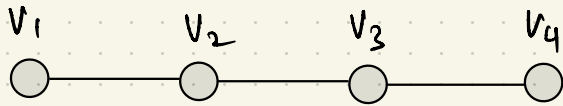
|

ROHIT VAISH

MAX WEIGHT INDEPENDENT SET ON PATHS

MAX WEIGHT INDEPENDENT SET ON PATHS

input: a path graph $G = (V, E)$ with nonnegative weights on vertices $\{w_v\}_{v \in V}$

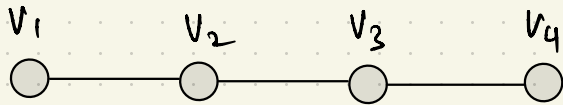


MAX WEIGHT INDEPENDENT SET ON PATHS

input: a path graph $G = (V, E)$ with nonnegative weights on vertices $\{w_v\}_{v \in V}$

output: an independent set $S \subseteq V$ of G with maximum $\sum_{v \in S} w_v$

subset of non-adjacent vertices

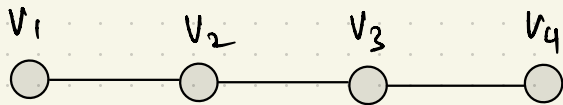


MAX WEIGHT INDEPENDENT SET ON PATHS

input: a path graph $G = (V, E)$ with nonnegative weights on vertices $\{w_v\}_{v \in V}$

output: an independent set $S \subseteq V$ of G with maximum $\sum_{v \in S} w_v$

subset of non-adjacent vertices



FAIL

Brute force

Greedy

Divide and conquer

OPTIMAL SUBSTRUCTURE

Optimal solution of overall problem built up from optimal solution of subproblems in a prescribed way.

OPTIMAL SUBSTRUCTURE

$$G = v_1 \text{ --- } v_2 \text{ --- } \dots \text{ --- } v_{n-2} \text{ --- } v_{n-1} \text{ --- } v_n$$

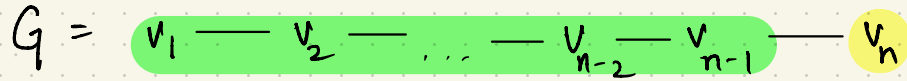
S : max wt independent set of graph G

OPTIMAL SUBSTRUCTURE

$$G = v_1 - v_2 - \dots - v_{n-2} - v_{n-1} - v_n$$

S : max wt independent set of graph G

OPTIMAL SUBSTRUCTURE



S : max wt independent set of graph G

if $v_n \notin S$

$\Rightarrow S$ must be a max wt ind. set of $G' := G \setminus \{v_n\}$.

OPTIMAL SUBSTRUCTURE

$$G = v_1 \text{ --- } v_2 \text{ --- } \dots \text{ --- } v_{n-2} \text{ --- } v_{n-1} \text{ --- } v_n$$

S : max wt independent set of graph G

if $v_n \notin S$

$\Rightarrow S$ must be a max wt ind. set of $G' := G \setminus \{v_n\}$.

if $v_n \in S$

$\Rightarrow S \setminus \{v_n\}$ must be a max wt ind. set of $G'' := G \setminus \{v_{n-1}, v_n\}$.

A RECURSIVE ALGORITHM

A RECURSIVE ALGORITHM

input: a path graph G with vertices $v_1, v_2, \dots, v_n$, nonnegative weights $\{w_i\}$

output: a maximum-weight independent set of G

if $n = 1$

return $\{v_1\}$

A RECURSIVE ALGORITHM

input: a path graph G with vertices $v_1, v_2, \dots, v_n$, nonnegative weights $\{w_i\}$

output: a maximum-weight independent set of G

if $n = 1$

return $\{v_1\}$

// recursion when $n \geq 2$

$S_1 :=$ recursively compute an MWIS of $G \setminus \{v_n\}$

$S_2 :=$ recursively compute an MWIS of $G \setminus \{v_{n-1}, v_n\}$

return S_1 or $S_2 \cup \{v_n\}$, whichever has higher weight

A RECURSIVE ALGORITHM

Claim: The recursive algorithm for MWIS is correct.

Proof: Exercise, by induction on number of vertices n .

Claim: The recursive algorithm for MWIS takes exponential time.

Proof: $T(n) = T(n-1) + T(n-2) + O(1)$

branching factor = 2 ... but very little progress

$T(n) \geq \text{Fibonacci}(n) \sim \text{const.}^n$ No better than brute force



Among the exponentially-many recursive calls,
how many **distinct** subproblems are considered?

$$\Theta(n)$$

The prefixes of graph G $v_1 - v_2 - v_3 \dots v_i - v_{i+1} \dots v_n$

ELIMINATING REDUNDANCY

The first time we solve a subproblem, cache its solution in a global table for $O(1)$ time lookup later on.

(i.e., **memoization**)

BLENDING CACHE INTO PSEUDOCODE

BLENDING CACHE INTO PSEUDOCODE

let $G_i :=$ first i vertices of graph G and corresponding edges

$$v_1 - v_2 - \dots - v_{i-1} - v_i$$

Plan: populate array A left to right with $A[i] =$ value of MWIS of G_i .

BLENDING CACHE INTO PSEUDOCODE

let $G_i :=$ first i vertices of graph G and corresponding edges

$$v_1 - v_2 - \dots - v_{i-1} - v_i$$

Plan: populate array A left to right with $A[i] =$ value of MWIS of G_i .

// initialization

$$A[0] = 0$$

$$A[1] = w_1$$

// main loop

for $i = 2, 3, \dots, n$:

$$A[i] := \max \{ A[i-1], A[i-2] + w_i \}.$$

BLENDING CACHE INTO PSEUDOCODE

Let $G_i :=$ first i vertices of graph G and corresponding edges

$$v_1 - v_2 - \dots - v_{i-1} - v_i$$

Plan: populate array A left to right with $A[i] =$ value of MWIS of G_i .

// initialization

$$A[0] = 0$$

$$A[1] = w_1$$

Running time: $O(n)$

Correctness: same as recursive algorithm

// main loop

for $i = 2, 3, \dots, n$:

$$A[i] := \max \{ A[i-1], A[i-2] + w_i \}.$$

BLENDING CACHE INTO PSEUDOCODE

Let $G_i :=$ first i vertices of graph G and corresponding edges

$$v_1 - v_2 - \dots - v_{i-1} - v_i$$

Plan: populate array A left to right with $A[i] =$ value of MWIS of G_i .

// initialization

$$A[0] = 0$$

$$A[1] = w_1$$



Computes the **weight** of Ind Set,
but **not the set itself**

// main loop

for $i = 2, 3, \dots, n$:

$$A[i] := \max \{ A[i-1], A[i-2] + w_i \}.$$

RECONSTRUCTION ALGORITHM

Idea: Trace back through array A to reconstruct optimal solution.

Key point:

vertex v_i belongs to
MWIS of G_i $\iff w_i +$
MWIS of $G_{i-2} \geq$ MWIS of G_{i-1}

$S := \emptyset$

While $i \geq 1$

if $A[i-1] \geq A[i-2] + w_i$
decrease i by 1

else

add v_i to S , decrease i by 2

return S

// scan from right to left

// $v_i \notin$ MWIS

// $v_i \in$ MWIS

A DAG VIEW OF DYNAMIC PROGRAMMING

1.

2.

3.

A DAG VIEW OF DYNAMIC PROGRAMMING

Subproblem 1

Subproblem 2

Subproblem 3

Subproblem 4

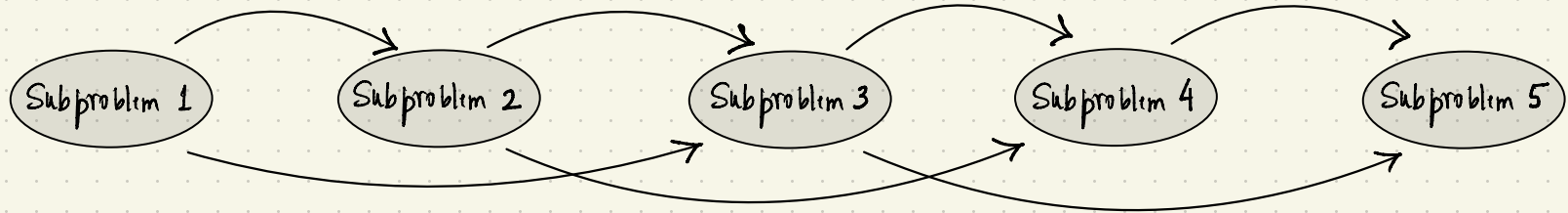
Subproblem 5

1. Identify a small number of subproblems

2.

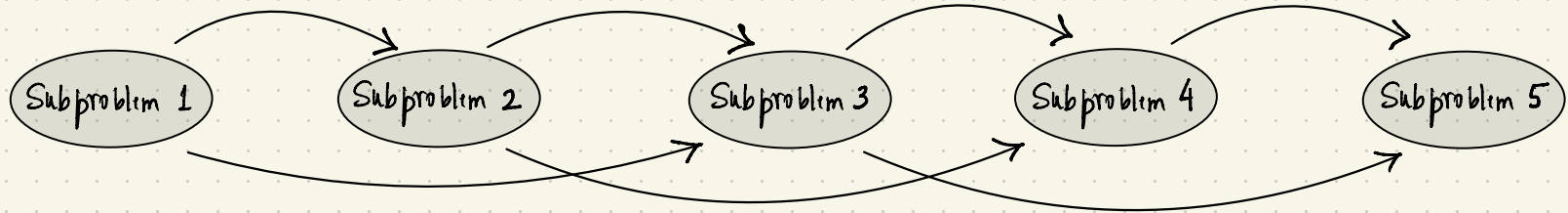
3.

A DAG VIEW OF DYNAMIC PROGRAMMING



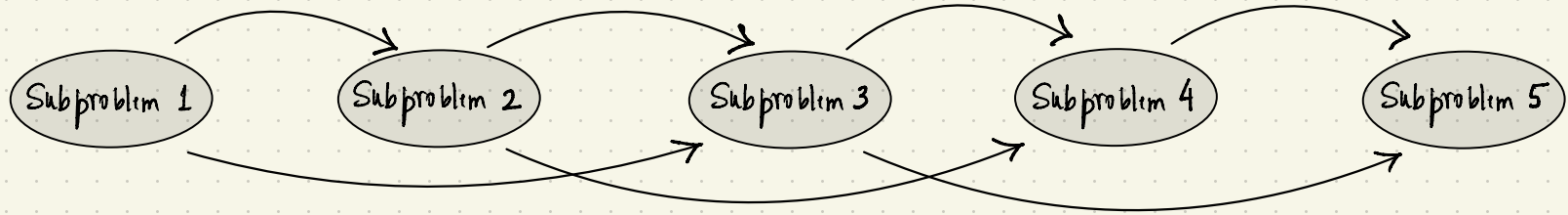
1. Identify a small number of subproblems
2. Show how to quickly and correctly solve "large" subproblems given solution of "smaller" ones.
- 3.

A DAG VIEW OF DYNAMIC PROGRAMMING



1. Identify a small number of subproblems
2. Show how to quickly and correctly solve "large" subproblems given solution of "smaller" ones.
3. " " " " " " **final** problem given solutions of all other subproblems.

A DAG VIEW OF DYNAMIC PROGRAMMING



1. Identify a small number of subproblems

e.g., max wt independent set of Q_i for $i \in \{0, 1, 2, \dots, n\}$

2. Show how to quickly and correctly solve "large" subproblems given solution of "smaller" ones.

e.g., recurrence $A[i] = \max \{ A[i-1], A[i-2] + w_i \}$.

3. " " " " " **final** problem

given solutions of all other subproblems.

e.g., return $A[n]$

I spent the Fall quarter (of 1950) at [RAND](#). My first task was to find a name for multistage decision processes. An interesting question is, "Where did the name, dynamic programming, come from?" The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named [Wilson](#). He was [Secretary of Defense](#), and he actually had a pathological fear and hatred of the word "research". I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

— Richard Bellman, *Eye of the Hurricane: An Autobiography* (1984, page 159)

I spent the Fall quarter (of 1950) at [RAND](#). My first task was to find a name for multistage decision processes. An interesting question is, "Where did the name, dynamic programming, come from?" The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named [Wilson](#). He was [Secretary of Defense](#), and he actually had a pathological fear and hatred of the word "research". I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

— Richard Bellman, *Eye of the Hurricane: An Autobiography* (1984, page 159)



KNAPSACK

KNAPSACK

input : ①

②

KNAPSACK

input: ① n items, each having a

- value v_i (non negative)

- size s_i (non negative and integral)

②

KNAPSACK

input : ① n items , each having a

- value v_i (non negative)

- size s_i (non negative and integral)

② capacity C (non negative and integral)

KNAPSACK

input : ① n items , each having a

- value v_i (non negative)
- size s_i (non negative and integral)

② capacity C (non negative and integral)

output : a subset $S \subseteq \{1, 2, \dots, n\}$

that maximizes $\sum_{i \in S} v_i$

KNAPSACK

- input :
- ① n items , each having a
 - value v_i (non negative)
 - size s_i (non negative and integral)
 - ② capacity C (non negative and integral)

output : a subset $S \subseteq \{1, 2, \dots, n\}$

that maximizes $\sum_{i \in S} v_i$

subject to $\sum_{i \in S} s_i \leq C$

EXAMPLE

item	value	size
1	3	4
2	2	3
3	4	2
4	4	3

capacity = 6

EXAMPLE

item	value	size
1	3	4
2	2	3
3	4	2
4	4	3

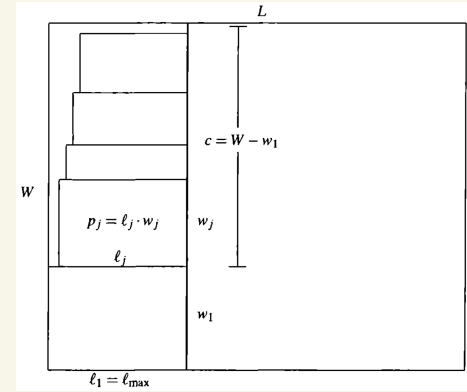
capacity = 6

APPLICATIONS

APPLICATIONS



Scheduling advertisements

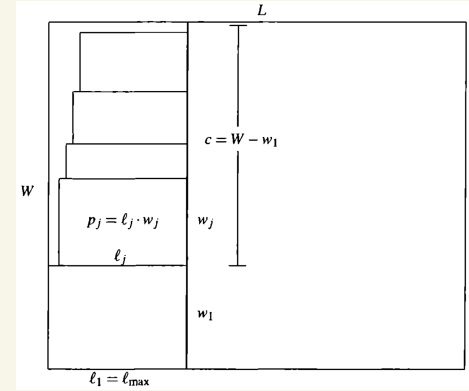


Profit - maximizing cutting

APPLICATIONS



Scheduling advertisements



Profit - maximizing cutting

Other applications : Credit assignment, Cryptosystems, ...

OPTIMAL SUBSTRUCTURE

OPTIMAL SUBSTRUCTURE

Let $S :=$ a max value solution of given knapsack instance

OPTIMAL SUBSTRUCTURE

Let $S :=$ a max value solution of given knapsack instance

Case I : item $n \notin S$

Case II : item $n \in S$

OPTIMAL SUBSTRUCTURE

Let $S :=$ a max value solution of given knapsack instance

Case I : item $n \notin S$

$\Rightarrow S$ must be optimal for the first $(n-1)$ items

Case II : item $n \in S$

OPTIMAL SUBSTRUCTURE

Let $S :=$ a max value solution of given knapsack instance

Case I : item $n \notin S$

$\Rightarrow S$ must be optimal for the first $(n-1)$ items
and residual capacity C .

Case II : item $n \in S$

OPTIMAL SUBSTRUCTURE

Let $S :=$ a max value solution of given knapsack instance

Case I : item $n \notin S$

$\Rightarrow S$ must be optimal for the first $(n-1)$ items
and residual capacity C .

Case II : item $n \in S$

$\Rightarrow S \setminus \{n\}$ must be optimal for the first $(n-1)$ items
and residual capacity $C - s_n$.

OPTIMAL SUBSTRUCTURE

Let $S :=$ a max value solution of given knapsack instance

Case I : item $n \notin S$

$\Rightarrow S$ must be optimal for the first $(n-1)$ items
and residual capacity C .

Case II : item $n \in S$

$\Rightarrow S \setminus \{n\}$ must be optimal for the first $(n-1)$ items
and residual capacity $C - s_n$.

buffer for item n 

THE SUB PROBLEMS

THE SUBPROBLEMS

Subproblems along **two** dimensions

THE SUBPROBLEMS

Subproblems along **two** dimensions

- * all possible prefixes of items $\{1, 2, \dots, i\}$
- * all possible (integral) residue capacities $x \in \{0, 1, \dots, C\}$

THE SUBPROBLEMS

Subproblems along **two** dimensions

* all possible prefixes of items $\{1, 2, \dots, i\}$

* all possible (integral) residue capacities $x \in \{0, 1, \dots, C\}$

recall C and all
sizes s_i are integral

THE SUBPROBLEMS

Subproblems along **two** dimensions

* all possible prefixes of items $\{1, 2, \dots, i\}$

* all possible (integral) residue capacities $x \in \{0, 1, \dots, C\}$

recall C and all
sizes s_i are integral

Design a recurrence to systematically solve all subproblems.

KNAPSACK RECURRENCE

KNAPSACK RECURRENCE

Define $V_{i,n} :=$ value of best solution that

(a) uses only the first i items, and

(b) has total size $\leq n$

KNAPSACK RECURRENCE

Define $V_{i,n} :=$ value of best solution that

(a) uses only the first i items, and

(b) has total size $\leq n$

For every $i \in \{1, 2, \dots, n\}$ and every $n \in \{0, 1, 2, \dots, C\}$

KNAPSACK RECURRENCE

Define $V_{i,n} :=$ value of best solution that

(a) uses only the first i items, and

(b) has total size $\leq n$

For every $i \in \{1, 2, \dots, n\}$ and every $n \in \{0, 1, 2, \dots, C\}$

$$V_{i,n} = \max \left\{ V_{(i-1), n} , V_{(i-1), n-s_i} + v_i \right\}$$

excluding item i

including item i

KNAPSACK RECURRENCE

Define $V_{i,n} :=$ value of best solution that

(a) uses only the first i items, and

(b) has total size $\leq n$

For every $i \in \{1, 2, \dots, n\}$ and every $n \in \{0, 1, 2, \dots, C\}$

$$V_{i,n} = \max \left\{ V_{(i-1), n}, V_{(i-1), n-s_i} + v_i \right\} \text{ if } s_i \leq n$$

KNAPSACK RECURRENCE

Define $V_{i,n} :=$ value of best solution that

(a) uses only the first i items, and

(b) has total size $\leq n$

For every $i \in \{1, 2, \dots, n\}$ and every $n \in \{0, 1, 2, \dots, C\}$

$$\begin{aligned} V_{i,n} &= \max \left\{ V_{(i-1), n} , V_{(i-1), n-s_i} + v_i \right\} \text{ if } s_i \leq n \\ &= ? \text{ if } s_i > n. \end{aligned}$$

KNAPSACK RECURRENCE

Define $V_{i,n} :=$ value of best solution that

(a) uses only the first i items, and

(b) has total size $\leq n$

For every $i \in \{1, 2, \dots, n\}$ and every $n \in \{0, 1, 2, \dots, C\}$

$$\begin{aligned} V_{i,n} &= \max \left\{ V_{(i-1), n} , V_{(i-1), n-s_i} + v_i \right\} \text{ if } s_i \leq n \\ &= V_{(i-1), n} \text{ if } s_i > n. \end{aligned}$$