

# Intro to Analysis of Algorithms

## Dynamic Programming

### Chapter 4

Michael Soltys

CSU Channel Islands

[Ed: 4th, last updated: October 2, 2025]

# Longest Monotone Subsequence

Input:  $d, a_1, a_2, \dots, a_d \in \mathbb{N}$ .

Output:  $L =$  length of the longest monotone non-decreasing subsequence.

Note that a subsequence need not be consecutive, that is  $a_{i_1}, a_{i_2}, \dots, a_{i_k}$  is a monotone subsequence provided that

$$1 \leq i_1 < i_2 < \dots < i_k \leq d,$$

$$a_{i_1} \leq a_{i_2} \leq \dots \leq a_{i_k}.$$

# Dynamic Prog approach

1. Define an array of sub-problems
2. Find the recurrence
3. Write the algorithm

We first define an array of subproblems:  $R(j)$  = length of the longest monotone subsequence which ends in  $a_j$ . The answer can be extracted from array  $R$  by computing  $L = \max_{1 \leq j \leq n} R(j)$ .

The next step is to find a recurrence. Let  $R(1) = 1$ , and for  $j > 1$ ,

$$R(j) = \begin{cases} 1 & \text{if } a_i > a_j \text{ for all } 1 \leq i < j \\ 1 + \max_{1 \leq i < j} \{R(i) \mid a_i \leq a_j\} & \text{otherwise} \end{cases}.$$

## Algorithm 18

```
1:  $R(1) \leftarrow 1$ 
2: for  $j : 2..d$  do
3:      $\max \leftarrow 0$ 
4:     for  $i : 1..j - 1$  do
5:         if  $R(i) > \max$  and  $a_i \leq a_j$  then
6:              $\max \leftarrow R(i)$ 
7:         end if
8:     end for
9:      $R(j) \leftarrow \max + 1$ 
10: end for
```

## Question

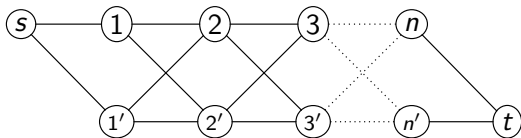
Once  $R$ , and  $L$ , have been computed how do we build the *actual* monotone subsequence?

# All pairs shortest path

Input: Directed graph  $G = (V, E)$ ,  $V = \{1, 2, \dots, n\}$ , and a cost function  $C(i, j) \in \mathbb{N}^+ \cup \{\infty\}$ ,  $1 \leq i, j \leq n$ ,  $C(i, j) = \infty$  if  $(i, j)$  is not an edge.

Output: An array  $D$ , where  $D(i, j)$  the length of the shortest directed path from  $i$  to  $j$ .

## Exponentially many paths Problem: 4.5





Define an array of subproblems: let  $A(k, i, j)$  be the length of the shortest path from  $i$  to  $j$  such that all *intermediate* nodes on the path are in  $\{1, 2, \dots, k\}$ . Then  $A(n, i, j) = D(i, j)$  will be the solution. The convention is that if  $k = 0$  then  $\{1, 2, \dots, k\} = \emptyset$ .

Define a recurrence: we first initialize the array for  $k = 0$  as follows:  $A(0, i, j) = C(i, j)$ .

Now we want to compute  $A(k, i, j)$  for  $k > 0$ .

To design the recurrence, notice that the shortest path between  $i$  and  $j$  either includes  $k$  or does not.

Assume we know  $A(k - 1, r, s)$  for all  $r, s$ .

Suppose node  $k$  is not included. Then, obviously,  $A(k, i, j) = A(k - 1, i, j)$ .

If, on the other hand, node  $k$  occurs on a shortest path, then it occurs exactly once, so  $A(k, i, j) = A(k - 1, i, k) + A(k - 1, k, j)$ .

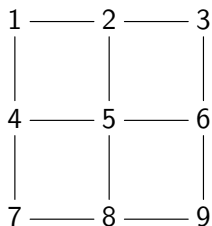
Therefore, the shortest path length is obtained by taking the minimum of these two cases:

$$A(k, i, j) = \min\{A(k - 1, i, j), A(k - 1, i, k) + A(k - 1, k, j)\}.$$

## Algorithm 19

```
1: for  $i : 1..n$  do
2:     for  $j : 1..n$  do
3:          $B(i,j) \leftarrow C(i,j)$ 
4:     end for
5: end for
6: for  $k : 1..n$  do
7:     for  $i : 1..n$  do
8:         for  $j : 1..n$  do
9:              $B(i,j) \leftarrow \min\{B(i,j), B(i,k) + B(k,j)\}$ 
10:        end for
11:    end for
12: end for
13: return  $D \leftarrow B$ 
```

## Example



$k = 0$  can be read directly from the graph  
(assume all edges worth 1).

$k = 1$								
	1	$\infty$	1	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
		1	2	1	$\infty$	$\infty$	$\infty$	$\infty$
			$\infty$	$\infty$	1	$\infty$	$\infty$	$\infty$
				1	$\infty$	1	$\infty$	$\infty$
					1	$\infty$	1	$\infty$
						$\infty$	$\infty$	1
							1	$\infty$
								1

$k = 2$								
	1	2	1	2	$\infty$	$\infty$	$\infty$	$\infty$
		1	2	1	$\infty$	$\infty$	$\infty$	$\infty$
			3	2	1	$\infty$	$\infty$	$\infty$
				1	$\infty$	1	$\infty$	$\infty$
					1	$\infty$	1	$\infty$
						$\infty$	$\infty$	1
							1	$\infty$
								1

# The “overwriting” trick

“Overwriting” not a problem on line 9 of algorithm.



## Bellman-Ford algorithm: §4.2.1

$$\text{OPT}(i, v) = \min\{\text{OPT}(i-1, v), \min_{w \in V}\{c(v, w) + \text{OPT}(i-1, w)\}\}$$

where  $\text{OPT}(i, v)$  is the shortest  $i$ -path from  $v$  to  $t$  (we want the shortest path from  $s$  to  $t$ ).

# Knapsack Problem

Input:  $w_1, w_2, \dots, w_d, C \in \mathbb{N}$ , where  $C$  is the knapsack's capacity.

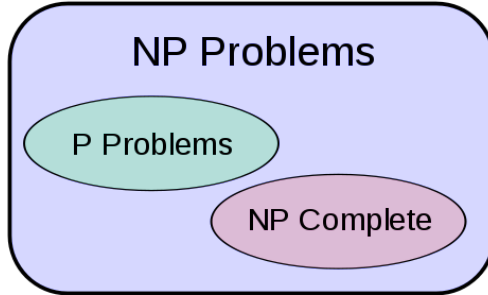
Output:  $\max_S \{K(S) \mid K(S) \leq C\}$ , where  $S \subseteq [d]$  and

$$K(S) = \sum_{i \in S} w_i.$$

First example of an NP-hard problem.

CHOTCHKIES RESTAURANT	
~ APPETIZERS ~	
MIXED FRUIT	2.15
FRENCH FRIES	2.75
SIDE SALAD	3.35
HOT WINGS	3.55
MOZZARELLA STICKS	4.20
SAMPLER PLATE	5.80
~ SANDWICHES ~	
BARBECUE	6.55







Define an array of subproblems: we consider the first  $i$  weights (i.e.,  $[i]$ ) summing up to an *intermediate* weight limit  $j$ .

We define a Boolean array  $R$  as follows:

$$R(i, j) = \begin{cases} \text{T} & \text{if } \exists S \subseteq [i] \text{ such that } K(S) = j \\ \text{F} & \text{otherwise} \end{cases},$$

for  $0 \leq i \leq d$  and  $0 \leq j \leq C$ .

Once we have computed all the values of  $R$  we can obtain the solution  $M$  as follows:  $M = \max_{j \leq C} \{j \mid R(d, j) = \text{T}\}$ .

Define a recurrence: we initialize  $R(0, j) = F$  for  $j = 1, 2, \dots, C$ , and  $R(i, 0) = T$  for  $i = 0, 1, \dots, d$ .

We now define the recurrence for computing  $R$ , for  $i, j > 0$ , in a way that hinges on whether we include object  $i$  in the knapsack.

Suppose that we do *not* include object  $i$ . Then, obviously,  $R(i, j) = T$  iff  $R(i - 1, j) = T$ .



Suppose, on the other hand, that object  $i$  is included. Then it must be the case that  $R(i, j) = \text{T}$  iff  $R(i - 1, j - w_i) = \text{T}$  and  $j - w_i \geq 0$ , i.e., there is a subset  $S \subseteq [i - 1]$  such that  $K(S)$  is exactly  $j - w_i$  (in which case  $j \geq w_i$ ).

$R$	0	$\dots$	$j-w_i$	$\dots$	$j$	$\dots$	$C$
0	T	F...F	F	F...F	F	F...F	F
	T						
	$\vdots$						
	T						
$i-1$	T		<b>c</b>		<b>b</b>		
$i$	T				<b>a</b>		
	T						
	$\vdots$						
	T						
$d$	T						

Putting it all together we obtain the following recurrence for  $i, j > 0$ :

$$R(i, j) = \text{T} \iff R(i-1, j) = \text{T} \vee (j \geq w_i \wedge R(i-1, j-w_i) = \text{T}).$$

```

1:  $S(0) \leftarrow T$ 
2: for  $j : 1..C$  do
3:      $S(j) \leftarrow F$ 
4: end for
5: for  $i : 1..d$  do
6:     for decreasing  $j : C..1$  do
7:         if  $(j \geq w_i \text{ and } S(j - w_i) = T)$  then
8:              $S(j) \leftarrow T$ 
9:         end if
10:    end for
11: end for

```

# General Knapsack Problem

Input:  $w_1, w_2, \dots, w_d, v_1, \dots, v_d, C \in \mathbb{N}$

Output:  $\max_{S \subseteq [d]} \{V(S) | K(S) \leq C\}$ ,  $K(S) = \sum_{i \in S} w_i$ ,  
 $V(S) = \sum_{i \in S} v_i$ .

$$V(i, j) = \max\{V(S) | S \subseteq [i] \text{ and } K(S) = j\},$$

for  $0 \leq i \leq d$  and  $0 \leq j \leq C$ .

Problem: what is the recurrence for this problem?

# Approximating SKS

Greedy “solution” to SKS:

order the weights from heaviest to lightest, keep adding for as long as possible.

Let  $M$  be the optimal solution, and let  $\bar{M}$  be the solution obtained from the greedy approach.

Performance:  $1/2$ .

Let  $S_0$  be the set of weights we got from greedy, so  $K(S_0) = \bar{M}$ .

If  $S_0 = \emptyset$ , then  $\bar{M} = M$ .

If  $S_0 = S$  (all weights in), then  $\bar{M} = M$ .

OTHERWISE:

Assume we throw out weights greater than  $C$  (they won't be added anyway). Let  $w_j$  be the first weight that has been rejected, after some weights have been added . . . .

# Activity Selection

Input: A list of activities  $(s_1, f_1, p_1), \dots, (s_n, f_n, p_n)$ , where  $p_i > 0$ ,  $s_i < f_i$  and  $s_i, f_i, p_i$  are non-negative real numbers.

Output: A set  $S \subseteq [n]$  of selected activities such that no two selected activities overlap, and the profit  $P(S) = \sum_{i \in S} p_i$  is as large as possible.

An *activity*  $i$  has a fixed start time  $s_i$ , finish time  $f_i$  and profit  $p_i$ . Given a set of activities, we want to select a subset of non-overlapping activities with maximum total profit.

Define an array of subproblems: sort the activities by their finish times,  $f_1 \leq f_2 \leq \dots \leq f_n$ .

As it is possible that activities finish at the same time, we select the *distinct* finish times, and denote them  $u_1 < u_2 < \dots < u_k$ , where, clearly,  $k \leq n$ .

For instance, if we have activities finishing at times 1.24, 4, 3.77, 1.24, 5 and 3.77, then we partition them into four groups: activities finishing at times  $u_1 = 1.24$ ,  $u_2 = 3.77$ ,  $u_3 = 4$ ,  $u_4 = 5$ .



Let  $u_0$  be  $\min_{1 \leq i \leq n} s_i$ , i.e., the earliest start time. Thus,

$$u_0 < u_1 < u_2 < \dots < u_k,$$

as it is understood that  $s_i < f_i$ . Define an array  $A(0..k)$  as follows:

$$A(j) = \max_{S \subseteq [n]} \{P(S) \mid S \text{ is feasible and } f_i \leq u_j \text{ for each } i \in S\},$$

where  $S$  is *feasible* if no two activities in  $S$  overlap. Note that  $A(k)$  is the maximum possible profit for all feasible schedules  $S$ .

Define a recurrence for  $A(0..k)$ .

In order to give such a recurrence we first define an auxiliary array  $H(1..n)$  such that  $H(i)$  is the index of the largest distinct finish time no greater than the start time of activity  $i$ .

Formally,  $H(i) = \ell$  if  $\ell$  is the largest number such that  $u_\ell \leq s_i$ . To compute  $H(i)$ , we need to search the list of distinct finish times.

To do it efficiently, for each  $i$ , apply the binary search procedure that runs in logarithmic time in the length of the list of distinct finish times (try  $\ell = \lfloor \frac{k}{2} \rfloor$  first).

Since the length  $k$  of the list of distinct finish times is at most  $n$ , and we need to apply binary search for each element of the array  $H(1..n)$ , the time required to compute all entries of the array is  $O(n \log n)$ .

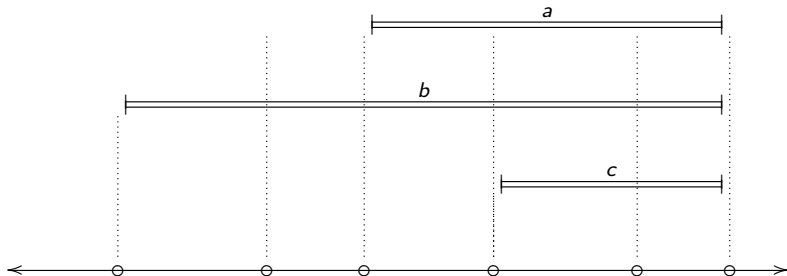
We initialize  $A(0) = 0$ , and we want to compute  $A(j)$  given that we already have  $A(0), \dots, A(j-1)$ .

Consider  $u_0 < u_1 < u_2 < \dots < u_{j-1} < u_j$ .

Can we beat profit  $A(j-1)$  by scheduling some activity that finishes at time  $u_j$ ? Try all activities that finish at this time and compute maximum profit in each case. We obtain the following recurrence:

$$A(j) = \max\{A(j-1), \max_{1 \leq i \leq n} \{p_i + A(H(i)) \mid f_i = u_j\}\},$$

where  $H(i)$  is the greatest  $\ell$  such that  $u_\ell \leq s_i$ .



$$s_b = u_H(b)$$

$$u_H(a)$$

$$s_b$$

$$s_c = u_H(c)$$

$$u_{j-1}$$

$$u_j$$

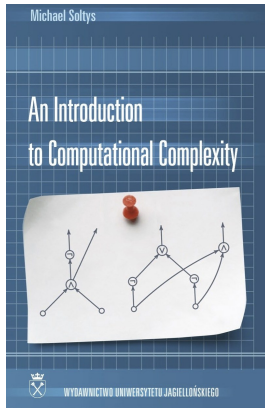
```

 $A(0) \leftarrow 0$ 
for  $j : 1..k$  do
     $\max \leftarrow 0$ 
    for  $i = 1..n$  do
        if  $f_i = u_j$  then
            if  $p_i + A(H(i)) > \max$  then
                 $\max \leftarrow p_i + A(H(i))$ 
            end if
        end if
    end for
    if  $A(j-1) > \max$  then
         $\max \leftarrow A(j-1)$ 
    end if
     $A(j) \leftarrow \max$ 
end for

```

# Introduction to Complexity

This material is not in the IAA textbook but here:



A TM  $M$  is of *time complexity*  $T(n)$  if whenever  $M$  is given an input  $w$ ,  $|w| = n$ , then  $M$  *halts* after making at most  $T(n)$  many moves.

$L \in \text{TIME}(f(n))$  if there exists a deterministic TM  $M$  of time complexity  $O(f(n))$  that decides  $L$ .

$L \in \text{NTIME}(f(n))$  if there exists a nondeterministic TM  $M$  of time complexity  $O(f(n))$  that decides  $L$ .

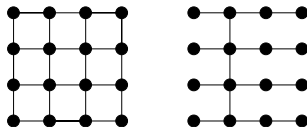
$L$  is in the class  $P$  if  $L \in \text{TIME}(n^k)$  for some fixed  $k$ .

$L$  is in the class  $NP$  if  $L \in \text{NTIME}(n^k)$  for some fixed  $k$ .

Observation:  $P \subseteq NP$ ; Question:  $NP \subseteq P$  ?

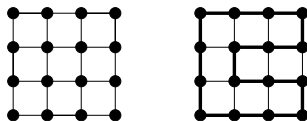
Ex. of a language in P:

$\{\langle G, k \rangle \mid G \text{ has a spanning tree of weight } \leq k\}$ . ( $k = 15$ )



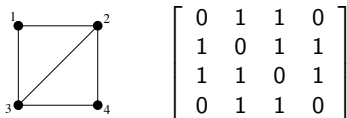
Ex. of a language in NP believed not to be in P:

$\{\langle G, k \rangle \mid G \text{ has a complete cycle of weight } \leq k\}$ . ( $k = 16$ )





A graph  $G$  can be encoded as an adjacency matrix. For example, the graph given below would have the adjacency matrix given by:



If  $P$  is a *decision problem*, the related language  $L_P$  consists of the encodings (under some fixed convention) of all the “yes” instances of  $P$ .

## Feasibility Thesis:

Polynomial time algorithm  $\equiv$  polynomial time TM.

A problem  $P_1$  is *reducible in polynomial time* to a problem  $P_2$  if there exists a polynomial time function  $f$  such that:

$$\langle I \rangle \in L_{P_1} \iff \langle f(I) \rangle \in L_{P_2}$$

$L$  is *NP-complete* if:

1.  $L \in \text{NP}$
2. Every language  $L' \in \text{NP}$  is polynomial time reducible to  $L$ .

Ex. Traveling Salesman Problem

$L$  is NP-complete is *evidence* of  $L$  not being in P

(see *Computers and Intractability* by Michael Garey and David Johnson.)

**Theorem:** If  $P_1$  is NP-complete,  $P_2$  is in NP, and there is a polynomial time reduction of  $P_1$  to  $P_2$ , then  $P_2$  is also NP-complete.

**Proof:** Every language  $L$  in NP is reducible to  $L_{P_1}$ , by completeness, and  $P_1$  is reducible to  $P_2$ . Enough to show transitivity of reductions.

**Theorem:** If some NP-complete problem  $P$  is in P, then  $P=NP$ .

**Proof:** Follows from the fact that all languages in NP are polynomial time reducible to  $P$ .

## Satisfiability

*Boolean Expressions* are built from: Boolean variables  $x, y, z, \dots$ , Boolean values 0, 1, and Boolean connectives:  $\vee, \wedge, \neg$ , and parenthesis.

Ex.  $\neg x \vee (y \wedge z)$

If  $\phi$  is a Boolean expression, then a *truth assignment*  $T$  is an assignment of truth values to the variables of  $\phi$ .

Ex.  $T(x) = 0, T(y) = 1, T(z) = 1$ , then  
 $T(\neg x \vee (y \wedge z)) = \neg 0 \vee (1 \wedge 1) = 1 \vee 1 = 1$ .

$T$  *satisfies*  $\phi$  if  $T(\phi) = 1$ , and  $\phi$  is *satisfiable* if  $\exists T$  s.t.  $T(\phi) = 1$ .

The *satisfiability problem* is: given a Boolean expression, is it satisfiable?

$\text{SAT} = \{\langle \phi \rangle \mid \phi \text{ is satisfiable}\}$

(i.e., SAT is the language corresponding to the satisfiability problem).

**Cook's Theorem:** SAT is NP-complete.

**PROOF:** SAT is in NP.

Let  $L$  be any language in NP.

We show there exists a polynomial time function  $f$  s.t.:

$$w \in L \iff f(w) = \phi \in \text{SAT}$$

$\exists$  non-det TM  $M$  s.t.  $L = L(M)$  and  $M$  always halts within  $n^k$  many steps on inputs  $w$ ,  $|w| = n$ , for fixed  $k$ .

Given  $w$ ,  $f$  outputs a Boolean formula  $\phi$  which encodes a computation of  $M$  on  $w$  and is satisfiable  $\iff M$  accepts  $w$ .