Chapter 3

Dynamic Programming

Dynamic programming is another important algorithmic technique. The main idea behind it is to solve a big problem by combining the solutions to smaller subproblems.

We usually want to maximize profit or minimize cost. A dynamic programming algorithm works by creating an array of related but simpler subproblems, and then, it computes the solution to the big complicated problem by using the solutions to the easier subproblems which are stored in this array.

A general way of finding a Dynamic Programming solution to a problem:

1. Define a class of subproblems.
2. Give a recurrence based on solving each subproblem in terms of simpler subproblems.
3. Give an algorithm for computing the recurrence.

3.1 Longest Monotone Subsequence Problem

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

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

Note: the subsequence need not be consecutive. Thus $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ is a monotone subsequence provided $1 \leq i_1 < i_2 < \ldots < i_k \leq d$ and $a_{i_1} \leq a_{i_2} \leq \ldots \leq a_{i_k}$.

Example 3.1.1 The length of the longest monotone subsequence of $\{4, 6, 5, 9, 1\}$ is 3.

We want to give a dynamic programming solution to the LMS Problem:

1. Define an array of subproblems: $R(j) =$ length of the longest monotone subsequence which ends in a_j. Then the answer would be $L = \max_{1 \leq j \leq n} R(j)$.

2. Find a recurrence: let $R(1) = 1$, and for $j > 1$,

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

3. Write an algorithm that computes R:

```
R(1) ← 1
for j : 2..d
    max ← 0
    for i : 1..j − 1
        if R(i) > max and a_i ≤ a_j then
            max ← R(i)
        end if
    end for
    R(j) ← max +1
end for
```

Exercise 3.1.1 Once we have computed all the values of the array R, how could we build an actual monotone non-decreasing subsequence of length L?

Exercise 3.1.2 The *pre-condition* of a piece of code (as 3. above) is an assertion about the input; the *post-condition* is an assertion about the output. The *correctness* of the code is the following statement: *if the pre-condition holds, then so does the post-condition.* Showing correctness usually involves designing a loop invariant, proving that it is indeed a loop invariant, and concluding correctness. What would be the appropriate pre/post-conditions of the above algorithms? Prove correctness with an appropriate loop invariant.

Exercise 3.1.3 Consider the following variation of the Longest Monotone Subsequence problem:

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

Output: What is the length of the longest subsequence of $a_1, a_2, \ldots, a_d$, where any two consecutive members of the subsequence differ by at most 1?

For example, the longest such subsequence of $\{7, 6, 1, 4, 7, 8, 20\}$ is $\{7, 6, 7, 8\}$, so in this case the answer would be 4. Give a dynamic programming solution to this problem.

3.2 All Pairs Shortest Path Problem

Input: Directed graph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$. Cost $C(i, j) \in \mathbb{R}^+ \cup \{\infty\}$ (think of the costs as distances), $1 \leq i, j \leq n$, $C(i, j) = \infty$ if (i, j) is not an edge.

Problem: Find $D(i, j)$, the length of the shortest directed path from i to j.

Note that for a general graph on n nodes there are exponentially many (in n) paths, so exhaustive search is not feasible.

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

2. Define a recurrence: Initialization: for $k = 0$, $A(0, i, j) = C(i, j)$. Now for $k > 0$. To come up with a 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 node k occurs on a shortest path, then it occurs exactly once, so $A(k, i, j) = A(k-1, i, k) + A(k-1, k, j)$. The shortest path length is obtained by taking the minimum of these two cases:

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

3. Write an algorithm:

   It turns out that we only need space for a two-dimensional array $B(i, j) = A(k, i, j)$. To compute $A(k, *, *)$ from $A(k-1, *, *)$, we can overwrite $A(k-1, *, *)$. The following algorithm is known as *Floyd's algorithm*:

```
for i : 1..n do
    for j : 1..n do
        B(i, j) ⟵ C(i, j)          (Initialize array B, i.e. copy C there)
    end for
end for

for k : 1..n do
(at this point B(i, j) = A(k − 1, i, j) for all i, j)
    for i : 1..n do
        for j : 1..n do
            B(i, j) ⟵ min{B(i, j), B(i, k) + B(k, j)}
        end for
    end for
end for
D ⟵ B
return D
```

The running time of this algorithm is $O(n^3)$.

Exercise 3.2.1 Why does the overwriting method in the above pseudo-code work? The worry is that $B(i, k)$ or $B(k, j)$ may have already been updated (if $k < j$ or $k < i$). However, the overwriting *does* work; explain why. We could have avoided a 3-dimensional array by keeping two 2-dimensional arrays instead, and then overwriting would not be an issue at all; how would that work?

Exercise 3.2.2 What are good pre/post-conditions? What is an appropriate loop invariant?

3.3 Simple Knapsack Problem (SKS)

Input: $d, w_1, \ldots, w_d, C \in \mathbb{N}$ (where C is the capacity)

Output: $\max_S\{K(S)|K(S) \leq C\}$, where $S \subseteq \{1, \ldots, d\}$, $K(S) = \sum_{i \in S} w_i$.

This is an NP-hard problem, which means that we cannot expect to find a polynomial time algorithm that works in general. We'll give a Dynamic Programming solution that works for relatively small C. It is important that the inputs $w_1, \ldots, w_d, C$ are integers (and in fact positive integers).

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

   $$R(i,j) = \begin{cases} True & \text{if } \exists S \subseteq \{1,\ldots,i\} \text{ s.t. } K(S) = j, \text{ i.e., the sum of weights is exactly } j. \\ False & \text{otherwise} \end{cases}$$

   $0 \leq i \leq d$, $0 \leq j \leq C$. We can compute the solution from R as follows:

   $$M = \max_{j \leq C}\{j|R(d,j) = True\}.$$

2. Define a recurrence: Initialize: $R(0,j) = False$, $j > 0$; $R(i,0) = True$, $i = 0, 1, \ldots, d$. Before writing a recurrence for $i, j \geq 1$, consider the following: Suppose we do not include object i. Then, obviously, $R(i,j) = True$ iff $R(i-1,j) = True$. Suppose object i is included. Then it must be the case that $R(i,j) = True$ iff $R(i-1, j-w_i) = True$, $j - w_i \geq 0$ (i.e., there is a subset S of objects $\{1, \ldots, i-1\}$ s.t. the sum of weights $K(S)$ is exactly $j - w_i$, $j \geq w_i$).

   We obtain the following recurrence for $i, j \geq 1$: $R(i,j) = True$ iff $R(i-1,j) = True$ or ($j \geq w_i$ and $R(i-1, j-w_i) = True$).

3. Write an algorithm: We will use the same space saving trick as before by using one-dimensional array $S(j)$ for $R(i,j)$.

```
S(0) ⟵ True (Initialize array for i = j = 0)
for j : 1..C do
    S(j) ⟵ False (Initialize array for i = 0 and j = 1..C)
end for
for i : 1..d
    for decreasing j : C downto 1
        if (j ≥ w_i and S(j − w_i) = True) then
            S(j) ⟵ True
        end if
    end for
    (at this point S(j) = R(i, j) for all j)
end for
```

Exercise 3.3.1 We are using a one dimensional array to keep track of a two dimensional array, and again the overwriting is not a problem; explain why.

Exercise 3.3.2 The assertion $S(j) = R(i, j)$ can be proved by induction on the number of times the inner loop body is executed. This assertion implies that upon termination of the algorithm, $S(j) = R(d, j)$ for all j. Prove this formally, by giving pre/post-conditions, a loop invariant, and a standard proof of correctness.

Exercise 3.3.3 Find an input for which the program would make an error if the inner loop "for decreasing $j : C..1$" were changed to "for $j : 1..C$".

Exercise 3.3.4 Show how to construct the actual optimal set of weights once R has been computed.

Exercise 3.3.5 Define a "natural" greedy algorithm for solving SKS; let $\overline{M}$ be the output of this algorithm, and let M be the output of the dynamic programming solution given in this section. Show that either $\overline{M} = M$ or $\overline{M} > \frac{1}{2}C$.

3.3.1 Dispersed Knapsack

Define the problem *Dispersed Knapsack* as follows:
Input: $w_1, \ldots, w_d, C$, all positive integers, such that the w_i's satisfy the following condition:

$$w_i \geq \sum_{j=i+1}^{d} w_j, \quad \text{for } i = 1, \ldots, d-1$$

Output: $S_{\max} \subseteq \{1, \ldots, d\}$ such that $K(S_{\max}) = \max_{S \subseteq \{1,\ldots,d\}} \{K(S) | K(S) \leq C\}$.

Exercise 3.3.6 Give a greedy algorithm which solves Dispersed Knapsack by filling in the following:

$S \longleftarrow \emptyset$
for $i : 1..d$

  if _____ then

    _____

  end if
end for

Exercise 3.3.7 Give a definition of what it means for an intermediate solution S in the above algorithm to be "promising". Show that the loop invariant "S is promising" implies that the greedy algorithm gives the optimal solution.

Exercise 3.3.8 Show that "S is promising" (as defined in exercise 3.3.7) is a loop invariant of the greedy algorithm in exercise 3.3.6.

3.4 The (General) Knapsack Problem

We generalize the Simple Knapsack Problem (SKS) to the Knapsack Problem by assigning a positive integer value v_i to each weight w_i.

Input: $d, w_1, \ldots, w_d, v_1, \ldots, v_d, C \in \mathbb{N}$ (where C is the capacity)

Output: Find $\max_S\{V(S)|K(S) \leq C\}$, where $S \subseteq \{1, \ldots, d\}$, $K(S) = \sum_{i \in S} w_i$, and $V(S) = \sum_{i \in S} v_i$.

Thus $V(S)$ is the total value of the set S of weights. The goal is to maximize $V(S)$, subject to the contraint that $K(S)$ (the sum of the weights in S) is at most C.

The Simple Knapsack Problem is the special case in which $v_i = w_i, 1 \leq i \leq d$.

To solve the general Knapsack problem, we start by computing the same Boolean array $R(i, j)$ that was used to solve SKS. Thus $R(i, j)$ ignores the values v_i, and only depends on the weights w_i. Next we define another array $V(i, j)$ that depends on the values v_i as follows: For $0 \leq i \leq d$, $0 \leq j \leq C$

$$V(i, j) = \max\{V(S)|S \subseteq \{1, \ldots, i\} \text{ and } K(S) = j\}$$

Exercise 3.4.1 Give a recurrence for computing the array $V(i, j)$, using the Boolean array $R(i, j)$. (You may assume that the array $R(i, j)$ has already been computed.) Also, give an algorithm for computing $V(i, j)$.

Exercise 3.4.2 It turns out that if the above definition of $V(i, j)$ is changed so that we only require $K(S) \leq j$ instead of $K(S) = j$, then the Boolean array $R(i, j)$ is not needed in the recurrence. Show how to do this.

3.5 The Activity Selection Problem

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. (An example is a set of lectures with fixed start and finish times that need to be scheduled in a single class room.

Input: A list of activities (s_1, f_1, p_1),..., (s_n, f_n, p_n). Assume $p_i > 0$, $s_i < f_i$, and $s_i, f_i, p_i \in \mathbb{R}$ where $1 \leq i \leq n$.

Output: Find a set $S \subseteq \{1, \ldots, n\}$ 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.

We describe a Dynamic Programming solution.

1. Define an array (of subproblems):

   Sort the activities by their finish time: $f_1 \leq f_2 \leq \ldots \leq f_n$. It is possible that two or more activities have the same finish time. Partition the activities according to their finish times. (Remark: This partitioning is not strictly necessary, although it may lead to a more efficient algorithm in some cases.) Denote these *distinct* finish times by $u_1 < u_2 < \ldots < u_k$.

Example 3.5.1 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 1.24, 3.77, 4 and 5 respectively. In this case $u_1 = 1.24$, $u_2 = 3.77$, $u_3 = 4$, $u_4 = 5$.

Let u_0 be $\min_{1 \le i \le n} s_i$, i.e., the earliest start time. Thus, $u_0 < u_1 < u_2 < \ldots < u_k$. Define an array $A(0..k)$ as follows.

$$A(j) = \max_{S \subseteq \{1,\ldots,n\}} \{P(S) | S \text{ is feasible and } f_i \le u_j \text{ for each } i \in S\}.$$

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.

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

   In order to give a recurrence for computing $A(0 \ldots k)$ we define an 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, $H(i) = l$ if l is the largest number such that $u_l \le s_i$. 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 $l = \lfloor \frac{k}{2} \rfloor$ 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 $O(n \log n)$.

   Initialization: $A(0) = 0$.

   Want $A(j)$ given $A(0), \ldots, A(j-1)$. Consider $u_0 < u_1 < u_2 < \ldots < u_{j-1} < u_j$. 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:

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

   where $H(i)$ is the last finish time before the start of activity i.

Exercise 3.5.1 Write the program.

Example 3.5.2 Consider the following example:

Activity i:	1	2	3	4
Start s_i:	0	2	3	2
Finish f_i:	3	6	6	10
Profit p_i:	20	30	20	30
$H(i)$:	0	0	1	0

j:	0	1	2	3
u_j:	0	3	6	10
$A(j)$:	0	20	40	40

$A(2) = \max\{20, 30 + A(0), 20 + A(1)\} = 40$ and $A(3) = \max\{40, 30 + A(0)\} = 40$. The answer is $\max P = A(k) = 40$.

Exercise 3.5.2 Given that A has been computed, how do you find a set of activities S such that $P(S) = A(k)$? **Hint:** If $A(k) = A(k-1)$, then we know that no selected activity finishes at time u_k, so we go on to consider $A(k-1)$. If $A(k) > A(k-1)$, then some selected activity finishes at time u_k. How do we find this activity? We must find an i_0 such that $p_{i_0} + A(H(i_0)) = A(k)$ and $f_{i_0} = u_k$, output i_0, and go on to consider $A(H(i_0))$.

3.6 Job Scheduling with Profits

A job differs from an activity in that a job can be scheduled any time as long as it finishes by its deadline; an activity has a fixed start time and finish time. Because of the flexibility in scheduling jobs, it is "harder" to find an optimal schedule for jobs than to select an optimal subset for activities, and we need a 2 dimensional array, rather than a 1 dimensional one.

Before we considered job scheduling problems for the case in which each job takes unit time. We now generalize this to the case in which each of n jobs i has duration d_i, deadline t_i, and profit p_i. We assume that d_i and t_i are positive integers, but the profit p_i can be a positive real number.

The goal is to find a feasible schedule $C(1 \ldots n)$ for the n jobs for which the profit $P(C)$ is maximized.

Our convention is that job i is scheduled to begin at time $C(i) \geq 0$, except that $C(i) = -1$ if job i is not scheduled by C.

The profit is:

$$P(C) = \sum_{C(i) \geq 0} p_i$$

We say that the schedule $C(1 \ldots n)$ is *feasible* if the following two conditions hold:

1. If $C(i) \geq 0$, then $C(i) + d_i \leq t_i$ (each scheduled job finishes by its deadline) and,
2. If $i \neq j$ and $C(i) \geq 0$ and $C(j) \geq 0$ then either $C(i) + d_i \leq C(j)$ or $C(j) + d_j \leq C(i)$ (no two scheduled jobs overlap).

Now we can define the problem Job Scheduling with Profits:

Input: A list of jobs (d_1, t_1, p_1),..., (d_n, t_n, p_n)

Output: A feasible schedule $C(1), \ldots, C(n)$ such that the profit $P(C)$ is the maximum possible among feasible schedules.

Notice that this problem is at "least as hard as SKS." In fact an SKS instance $w_1, \ldots, w_n, C$ can be viewed as a job scheduling problem in which each duration $d_i = w_i$, each deadline $t_i = C$, and each profit $p_i = w_i$. Then the maximum profit of any schedule is the same as the maximum weight that can be put into the knapsack.

To give a Dynamic Programming solution to the job scheduling problem, we start by sorting the jobs according to deadline. Thus we assume that

$$t_1 \leq t_2 \leq \ldots \leq t_n$$

It turns out that to define a suitable array A for solving the problem, we must consider all possible integer times t, $0 \leq t \leq t_n$ as a deadline for the first i jobs. It is not enough to only consider the specified deadline t_i given in the problem input.

Thus define the array $A(i, t)$ for $0 \leq i \leq n, 0 \leq t \leq t_n$ by:

$$A(i,t) = \max\{P(C) | \begin{cases} C \text{ is a feasible schedule,} \\ \text{only jobs in } \{1, \ldots, i\} \text{ are scheduled,} \\ \text{all scheduled jobs finish by time } t. \end{cases} \}$$

Exercise 3.6.1 Find a recurrence for $A(i,t)$. (**Hint:** Consider the two cases that either job i occurs or does not occur in the optimal schedule (and note that job i will not occur in the optimal schedule if $d_i > \min\{t_i, t\}$). If job i does not occur, we already know the optimal profit. If job i occurs in an optimal schedule, then we may as well assume that it is the last job (among jobs $\{1,\ldots,i\}$) to be scheduled, because it has the lastest deadline. Hence we assume that job i is scheduled as late as possible, so that it finishes either at time t, or at time t_i, which ever is smaller, i.e. it finishes at time $t_{\min} = \min\{t_i, t\}$.)

3.7 Context-Free Grammars

An *alphabet* is a finite set of symbols (when taling about context-free grammars, symbols are oftern called *terminals*), for example $\Sigma = \{0,1\}$ or $\Sigma = \{a,b,c,\ldots,z\}$. Given an alphabet Σ, we denote the set of all finite strings over Σ as Σ^*. For example, if $\Sigma = \{0,1\}$ then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$, the set of all binary strings, where ε is the empty string. If w is a string, $|w|$ is the length of w, so $|\varepsilon| = 0$ and $|101001| = 6$.

A *context-free grammar* is a tuple $G = (V, T, R, S)$, where V is a (finite) set of *variables*, T is a (finite) set of terminals, R is a (finite) set of rules, and S denotes the *starting variable*. All rules are of the form $X \longrightarrow \alpha$ where $X \in V$ and $\alpha \in (V \cup T)^*$, i.e., α is a (finite) string of variables and terminals.

A *derivation* of a string w is denoted as follows: $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_k$, where $\alpha_k = w$ and each α_{i+1} is obtained from α_i by one of the rules (meaning that a variable X in α_i is replaced by α to obtain α_{i+1} if $X \longrightarrow \alpha$ was one of the rules of the grammar).

The set $L(G)$, called *the language of the grammar* G, is the set of all those strings in T^* which have G-derivations.

Sometimes, for the sake of succinctness, if there are several rules for the same variable X, for example $X \longrightarrow \gamma_1, \cdots, X \longrightarrow \gamma_k$, we write $X \longrightarrow \gamma_1 | \cdots | \gamma_k$ as if it were one rule, but the symbol "|" denotes that this is a family of rules for X.

Example 3.7.1 Consider the following grammar $G_{\text{pal}} = (\{S\}, \{0,1\}, \{S \longrightarrow \varepsilon|0|1|0S0|1S1\}, S)$. In this grammar there is only one variable, S, which is also (necessarily) the starting variable, and there are five rules, represented in the succinct form by $S \longrightarrow \varepsilon|0|1|0S0|1S1$, and there are two terminals $0, 1$. For example, $S \Rightarrow 0S0 \Rightarrow 01S10 \Rightarrow 01010$ shows that $01010 \in L(G_{\text{pal}})$. In fact, G_{pal} generates precisely the set of all *palindromes* over 0,1, i.e., the set of all binary strings that read the same backwards as forwards.

Exercise* 3.7.1 Show that $L_{\text{pal}} = L(G_{\text{pal}})$, in other words, show that G_{pal} is the grammar that generates precisely the set of all palindromes over $\{0,1\}$. To show "$\subseteq$" use induction on the length of strings to show that each palindrome has a derivation; for "$\supseteq$" use induction on the length of derivation.

A context-free grammar is in *Chomsky Normal Form (CNF)* if all the rules are of the form: $A \longrightarrow BC$, $A \longrightarrow a$, and $S \longrightarrow \varepsilon$, where A, B, C are variables (not necessarily distinct), a is a terminal, and S is the starting variable (so note that ε can only be generated from the starting variable).

Example 3.7.2 The grammar G_{pal} can be translated into CNF as follows: first $S \longrightarrow \varepsilon|0|1$ is allowed, so we have to deal with $S \longrightarrow 0S0$ and $S \longrightarrow 1S1$. We introduce two new variables U_0, U_1 and the following two rules, $U_0 \longrightarrow 0$ and $U_1 \longrightarrow 1$, and replace $S \longrightarrow 0S0|1S1$ with $S \longrightarrow U_0SU_0|U_1SU_1$. Now the only problem is that there are too many variables (we are only allowed 2 per body of rule in CNF), so we introduce two more new variabels V_0, V_1 and transform $S \longrightarrow U_0SU_0|U_1SU_1$ as follows: $S \longrightarrow U_0V_0$, $V_0 \longrightarrow SU_0$, and $S \longrightarrow U_1V_1$, $V_1 \longrightarrow SU_1$. Putting it all together, we have a new grammar G'_{pal} given by $(\{S, U_0, U_1, V_0, V_1\}, \{0, 1\}, R, S)$ and R is given by the following rules:

$$\{S \longrightarrow \varepsilon|0|1,\ U_0 \longrightarrow 0,\ U_1 \longrightarrow 1,\ S \longrightarrow U_0V_0,\ V_0 \longrightarrow SU_0,\ S \longrightarrow U_1V_1,\ V_1 \longrightarrow SU_1\}$$

and note that G'_{pal} is also a grammar for the language of palindromes, and furthermore it is in CNF.

Exercise* **3.7.2** Every context-free grammar can be transformed to an equivalent grammar (i.e., generating the same set of strings of terminals) in CNF. (**Hint:** the main ideas are given in example 3.7.2; you will have to deal with the case $X \longrightarrow Y$ (the so called *unit rules*), and $X \longrightarrow \varepsilon$ where X is not the starting variable (the so called *ε-rules*).)

3.7.1 The CYK algorithm

Given a grammar G in CNF, and a string $w = a_1a_2 \dots a_n$, we can test whether $w \in L(G)$ using the CYK dynamic algorithm. The algorithm, on input $\langle G, w = a_1a_2 \dots a_n\rangle$, builds an $n \times n$ table T, where each entry contains a subset of V. At the end, $w \in L(G)$ iff the start variable S is contained in position $(1, n)$ of T. The main idea is to put variable X_1 in position (i, j) if X_2 is in position (i, k) and X_3 is in position $(k + 1, j)$ and $X_1 \longrightarrow X_2X_3$ is a rule. The reasoning is that X_1 is in position (i, k) iff $X_1 \stackrel{*}{\Rightarrow} a_i \dots a_k$, that is, the substring $a_i \dots a_k$ of the input string can be generated from X_1.

Let $V = \{X_1, X_2, \dots, X_m\}$.

for $i = 1..n$
  for $j = 1..m$
    Place variable X_j in (i, i) iff $X_j \longrightarrow a_i$ is a rule of G
for $i < j$
  for $k = i..(j - 1)$
    if $(\exists X_p \in (i, k) \wedge \exists X_q \in (k + 1, j)$ & $\exists$ rule $X_r \longrightarrow X_pX_q)$
      Put X_r in (i, j)

Example 3.7.3 In the following example, we show which entries in the table we need to use to comptue the contents of $(2, 5)$.

x	(2,2)	(2,3)	(2,4)	**(2,5)**
x	x			(3,5)
x	x	x		(4,5)
x	x	x	x	(5,5)

3.8 Computing the determinant of a matrix

The determinant has been a subject of study for over 200 years[1]. Given its importance in linear algebra and geometry, it is not surprising that it has been studied by famous mathematicians such as Leibnitz, Gauss, Jacobi, and many others.

Definition 3.8.1 Let S_n be the set of permutations on n numbers, i.e., S_n is the set of bijections $\sigma : [n] \longrightarrow [n]$. The *identity* permutation is denoted id, and $\mathrm{id}(i) = i$ for all i. A *transposition* is a permutation which exchanges only two numbers, that is, σ is a transpostion if $\sigma(i) = j$ and $\sigma(j) = i$, for some $i \neq j$, and for all $k \notin \{i, j\}$, $\sigma(k) = k$. Let $\sigma^0(i) = i$, $\sigma^1(i) = \sigma(i)$, and $\sigma^{n+1}(i) = \sigma^n(\sigma(i))$. A *cycle* is denoted by $(a_1 a_2 \dots a_k)$, and it represents a permutation of the form $\sigma(a_i) = a_{i+1}$ for $i < k$, and $\sigma(a_k) = a_1$. The product of two permutations σ_1, σ_2 is their composition, i.e., $\sigma_1 \cdot \sigma_2 := \sigma_1 \circ \sigma_2$. In other words, $(\sigma_1 \cdot \sigma_2)(i) = \sigma_1(\sigma_2(i))$. (Note that this means that the product of permutations is evaluated from right to left.)

Exercise 3.8.1 Show that $|S_n| = n!$.

Exercise 3.8.2 Show that any permutation σ can be represented as a product of disjoint cycles.

Exercise 3.8.3 Show that any permutation σ can be represented as a product of transpostions (use exercise 3.8.2). Furthermore, show that the parity of the number of transpostions in any such representation is unique. Using this fact, we define $\mathrm{sgn}(\sigma)$ as the parity of the number of transpositions in (any) representation of σ, so $\mathrm{sgn} : S_n \longrightarrow \{0, 1\}$.

We restrict ourselves to square matrices; the determinant of an $n \times n$ matrix A is given by

$$\det(A) = \sum_{\sigma \in S_n} \mathrm{sgn}(\sigma) \prod_{i=1}^{n} a_{i\sigma(i)}.$$

Example 3.8.1 If A is a 1×1 matrix, then $A = (a)$, so $\det(A) = a$. If A is a 2×2 matrix or a 3×3 matrix, then it is of the form:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \qquad \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix},$$

respectively, and the determinants are

$$a_{11}a_{22} - a_{21}a_{12}, \qquad a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31},$$

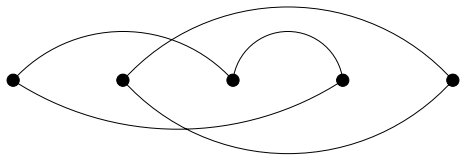
respectively.

For n large enough, $n!$ is too big to compute $\det(A)$ directly from the above definition. We are going to present a fast sequential dynamic programming solution for computing $\det(A)$. Furthermore, our solution will have the advantage of not requiring division (which, for example, would be necessary if we used standard Gaussian Elimination to compute the determinant).

[1]The material for this section comes from *Determinant: Combinatorics, Algorithms, and Complexity*, by M. Mahajan and V. Vinay, Chicago Journal of Theoretical Computer Science, 1997.

Let G_A be the graph *induced* by A; G_A has n vertices, and it is a weighted directed graph where the weight of edge (i, j) is a_{ij}. A permutation in S_n therefore corresponds to a *cycle cover*: the cycle decomposition of the permutation, when interpreted as a graph, induces a partition on the vertex set into disjoint cycles.

Example 3.8.2 The permutation $\sigma \in S_5$ given by $\sigma(1) = 3, \sigma(2) = 5, \sigma(3) = 4, \sigma(4) = 1, \sigma(5) = 2$ gives rise to the following cycle cover:



where the nodes are number $1, \ldots, 5$ from left to right.

As was mentioned above, counting cycle covers is not feasible. So we cannot implement this idea directly. We will get around the problem of counting cycle covers by counting *clow sequences* instead. There are more clow sequences than cycle covers, but there are efficient ways of computing their total weight, and *bad* clow sequences (those which do not correspond to cycle covers) cancel out!

A *clow* (this word is derived from *closed walk*) is a walk $(w_1, \ldots, w_l)$ starting from vertex w_1 and ending at the same vertex, where any (w_i, w_{i+1}) is an edge in the graph. Vertex w_1 is the least-numbered vertex in the clow, and it is called the *head* of the clow. We also require that the head occur only once in the clow. This means that there is exactly one incoming edge (w_l, w_1) and one outgoing edge (w_1, w_2) at w_1.

A *clow sequence* is a sequence of clows $(C_1, \ldots, C_k)$ with two properties: (i) the sequence is ordered by heads:

$$\text{head}(C_1) < \ldots < \text{head}(C_k)$$

and (ii) the total number of edges, counted with multiplicity, adds to n.

Exercise 3.8.4 Show that a cycle cover is a clow sequence. Give an example of a clow sequence which is not a cycle cover.

We will now show how to associate a sign with a clow sequence that is consistent with the definition of the sign of a cycle cover.

Exercise 3.8.5 Show that the sign of a cycle cover is $(-1)^{n+k}$, where n is the number of vertices in the graph, and k is the number of components in the cycle cover.

Thus, by analogy with cycle covers, we define the sign of a clow sequence to be $(-1)^{n+k}$ where n is the number of vertices in the graph, and k is the number of clows in the sequence.

We will associate a weight with a clow sequence that is consistent with the contribution of a cycle cover. The *weight* of a clow C, $w(C)$, is the product of the weights of the edges in the walk while accounting for multiplicity.

Example 3.8.3 $w((1,2,3,2,3)) = a_{12}a_{23}^2a_{32}a_{31}$

The weight of a clow sequence $(C_1, \ldots, C_k)$ is

$$\prod_{i=1}^{n} w(C_i)$$

Theorem 3.8.1

$$\det(A) = \sum_{\pi \text{ is a clow sequence}} \text{sign}(\pi) w(\pi)$$

PROOF: The idea is to show that clow sequences which are not cycle covers cancel out. To do that, we are going to consider the set of all clow sequences, and define a mapping on it, φ, which is a bijection with the property that φ^2 is the identity map. Furthermore, φ will be the identity on cycle covers, and for clow sequences which are *not* cycle covers, it will map them to another clow sequence with the same edge set but opposite parity. The existence of such a φ will prove the theorem.

We now show that φ exists by defining it as follows: given a clow sequence $(C_1, \ldots, C_k)$ choose the smallest i such that $(C_{i+1}, \ldots, C_k)$ is a set of disjoint (simple) cycles. If $i = 0$, then φ maps $(C_1, \ldots, C_k)$ to itself (since it is obviously a cycle cover).

Otherwise, traverse C_i starting from its head until one of two things happen:

1. we hit a vertex that touches one of $(C_{i+1}, \ldots, C_k)$, or
2. we hit a vertex that completes a simple cycle within C_i.

We call this vertex v. Given the way that we chose i, such a v must exist.

Exercise 3.8.6 Show that vertex v cannot satisfy both of the above two conditions.

We consider the two cases separately.

Case 1. Suppose that v touches C_j in $(C_{i+1}, \ldots, C_k)$. Then, φ maps $(C_1, \ldots, C_k)$ to the following clow sequence:

$$(C_1, \ldots, C_{i-1}, C_i', C_{i+1}, \ldots, C_{j-1}, C_{j+1}, \ldots, C_k) \tag{3.1}$$

where C_i' is obtained by merging C_i and C_j as follows: insert the cycle C_j into C_i at the first occurence (from the head of C_i) of v.

Exercise 3.8.7 Suppose that $C_i = (8, 11, 10, 14)$ and $C_j = (9, 10, 12)$. Compose C_i'.

The head of C_i' is clearly the head of C_i.

Exercise 3.8.8 Argue that that the new clow sequence has the same (multi)set of edges, and hence the same weight as the original, and opposite parity.

Case 2. Suppose that v completes a simple cycle C in C_i. By exercise 3.8.6 we know that C does not intersect $(C_{i+1}, \ldots, C_k)$. We modify the sequence by deleting C from C_i and introducing C as a new clow in a an appropriate position, depending on the minimum labeled vertex in C, which me make the head of C.

Exercise 3.8.9 Show what to do if $C = (8, 11, 10, 12, 9, 10, 14)$.

Exercise 3.8.10 Show that the new sequence is indeed a clow sequence, and show that it is of opposite parity but same weight as the original.

Exercise 3.8.11 What would happen if $i = k$?

Exercise 3.8.12 Show that if we applied φ to the clow sequence given by (3.1) we would get back $(C_1, \ldots, C_k)$.

This completes the proof. □

Exercise 3.8.13 Show that

$$\det(A) = \sum_{\mathbf{C} \text{ is a c.s. with head of 1st clow 1}} \operatorname{sgn}(\mathbf{C}) w(\mathbf{C})$$

We are now ready to start desiging our dynamic program for computing the determinant. For a given $n \times n$ matrix A, we define a layered, directed acyclic graph H_A with three special vertices, s, t_+, t_-, having the following property:

$$\det(A) = \sum_{\rho: s \rightsquigarrow t_+} w(\rho) - \sum_{\rho: s \rightsquigarrow t_-} w(\rho) \tag{3.2}$$

Here the weight of the path ρ is simply the product of the weights of the edges appearing on it. The idea is that $s \rightsquigarrow t_+$ (or $s \rightsquigarrow t_-$) paths will be in one-to-one correspondence with clow sequences of positive (negative) sign.

The vertex set of H_A is $\{s, t_+, t_-\}$ together with all the vertices described by $[p, h, u, i]$ where $p \in \{0, 1\}$ is the parity, h, u are vertices in $[n]$, and i is a number in $\{0, 1, \ldots, n-1\}$.

Exercise 3.8.14 What is $|V_{H_A}|$, that is, how many vertices are in H_A?

Here is the meaning of the vertices: if a path ρ (starting from s) reaches a vertex $[p, h, u, i]$, this indicates that in the clow sequence being constructed along this path, p is the parity of the quantity "n+(the number of components already constructed)," h is the head of the clow currently being constructed, u is the vertex that the current clow has reached, and i is the number of edges traversed so far (in this and preceding clows). Finally, a path from s to t_+ (t_-) corresponds to a clow sequence of positive (negative) parity.

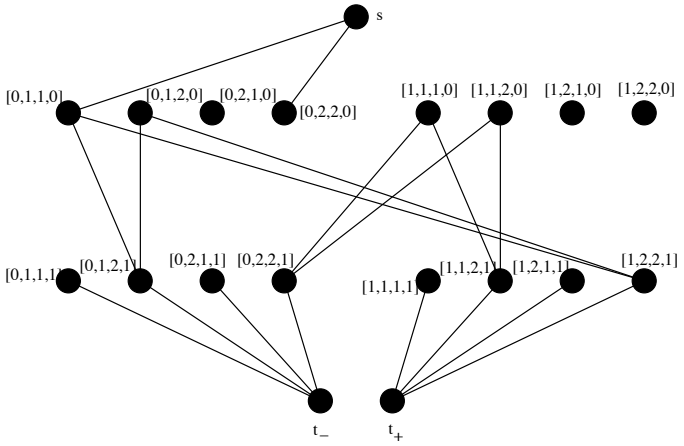
Now we describe the edges of H_A:

- $(s, [b, h, h, 0])$ for $h \in [n]$ and b the parity of n; the weight of this edge is 1.
- $([p, h, u, i], [p, h, v, (i+1)])$ if $v > h$ and $(i+1) < n$; the weight of this edge is a_{uv}.

- $([p, h, u, i]\,, [\bar{p}, h', h', (i+1)])$ if $h' > h$ and $(i+1) < n$; the weight of this edge is a_{uh}.
- $([p, h, u, (n-1)]\,, t_+)$ if $p = 1$; the weight of this edge is a_{uh}.
- $([p, h, u, (n-1)]\,, t_-)$ if $p = 0$; the weight of this edge is a_{uh}.

Exercise 3.8.15 Give a big-Oh estimate of $|E_{H_A}|$, i.e., an estimate of the number of edges in H_A.

Example 3.8.4 Consider a 2×2 matrix A, where in fact clow sequences and cycle covers coincide. In this case H_A is given by:



Note that the edge from s to $[0, 1, 1, 0]$ is labeled with 1, and the two edges from $[0, 1, 1, 0]$ to $[0, 1, 2, 1]$ and $[1, 2, 2, 1]$ are labeled with a_{12} and a_{11}, respectively. The edges from $[0, 1, 2, 1]$ to t_- and from $[1, 2, 2, 1]$ to t_+ are labeled with a_{21} and a_{22} respectively. There is just one path from s to t_+, of total weight $a_{11}a_{22}$, and the one path from s to t_- has total weight $a_{12}a_{21}$, and their difference gives us the determinant, as it should.

Theorem 3.8.2 (3.2) holds for H_A defined as above.

Exercise 3.8.16 Prove the above theorem; show that paths in H_A from s to t_+ (t_-) correspond in a bijective way to clow sequences in G_A. As a warm up, consider first the clow sequence given by $((1, 2, 5, 2, 4, 7), (3, 8, 10), (6, 9, 11))$; what is the corresponding path in H_A?

We are now ready to present the dynamic algorithm. We say that a vertex $[p, h, u, i]$ is at layer i in H_A. Vertices t_+ and t_- are at layer n, and s is at layer 0. The algorithm proceeds by computing, in stages, the sum of the weighted paths from s to any vertex at layer i in H_A. After n stages, it has the values at t_+ and t_-, and therefore it also has $\det(A)$.

(*Initialize values to 0*)
For $u, v, i \in [n]$ and $p \in \{0, 1\}$
    $V[p, u, v, (i-1)] \leftarrow 0$
$V[t_+] \leftarrow 0$

$V[t_-] \leftarrow 0$
$b \leftarrow$ parity of n
(*Set selected values at layer 0 to 1*)
For $u \in [n]$
    $V[b, u, u, 0] \leftarrow 1$
(*Process outgoing edges from each layer*)
For $i = 0$ to $(n-2)$
    For $u, v \in [n]$ such that $u \leq v$ and $p \in \{0, 1\}$
        For $w \in \{u+1, \ldots, n\}$
            $V[p, u, w, (i+1)] \leftarrow V[p, u, w, (i+1)] + V[p, u, v, i] \cdot a_{vw}$
            $V[\bar{p}, w, w, (i+1)] \leftarrow V[\bar{p}, w, w, (i+1)] + V[p, u, v, i] \cdot a_{vu}$
For $u, v \in [n]$ such that $u \leq v$ and $p \in \{0, 1\}$
    $V[t_+] \leftarrow V[t_+] + V[1, u, v, (n-1)] \cdot a_{vu}$
    $V[t_-] \leftarrow V[t_-] + V[0, u, v, (n-1)] \cdot a_{vu}$
Return $V[t_+] - V[t_-]$

3.9 Further examples and exercises

Consider the *Consecutive Subsequence Sum Problem*:
Input: Real numbers $r_1, \ldots, r_n$ (for some $n \in \mathbb{N}$).
Output: For each consecutive subsequence of the form $r_i, r_{i+1}, \ldots, r_j$ let

$$S_{ij} = r_i + r_{i+1} + \cdots + r_j$$

where $S_{ii} = r_i$. Find $M = \max_{1 \leq i \leq j \leq n} S_{ij}$.

For example, if $n = 7$ and the input sequence is $1, -5, 3, -1, 2, -8, 3$, then $M = S_{35} = 3 + (-1) + 2 = 4$.

This problem can be solved in time $O(n^2)$ by systematically computing all of the sums S_{ij} and finding the maximum. However, there is a more efficient dynamic programming solution which runs in time $O(n)$.

Define the array $M(1..n)$ by:

$$M(j) = \max\{S_{1j}, S_{2j}, \ldots, S_{jj}\}$$

In the above example:

j	=	1	2	3	4	5	6	7
r_j	=	1	-5	3	-1	2	-8	3
$M(j)$	=	1	-4	3	2	4	-4	3

Exercise 3.9.1 Explain how to find the solution M from the array $M(1..n)$.

Exercise 3.9.2 Complete the four lines indicated in the program below for computing the values of the array $M(1..n)$, given $r_1, r_2, \ldots, r_n$.

```
M(1) ⟵ (1)
for j:2..n

    if (2)
    then
         M(j) ⟵ (3)
    else
         M(j) ⟵ (4)
    end if
end for
```