Dynamic Programming (DP)

- It is another design technique, not a specific algorithm.
- It had been developed back in the day when programming meant tabular method (like linear programming). Therefore, "Programming" in this context refers to a tabular method, not to writing computer code.
- It is similar to the Divide-and-Conquer(D&C) method that means it solves problems by combining the solutions to subproblems.

D&C	DP
D&C algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.	In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share sub-subproblems
A D&C algorithm does **more work** than necessary, repeatedly solving the common sub-subproblems.	A DP algorithm solves every sub-subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the **sub-subproblem** is encountered

- Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions.
- Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution ***an optimal solution*** to the problem, as opposed to ***the optimal solution***, since there may be several solutions that achieve the optimal value.
- The development of a dynamic-programming algorithm can be broken into a sequence of four steps:

  (1) Characterize the structure of an optimal solution.
  (2) Recursively define the value of an optimal solution.
  (3) Compute the value of an optimal solution in a bottom-up fashion.
  (4) Construct an optimal solution from computed information.

Note:

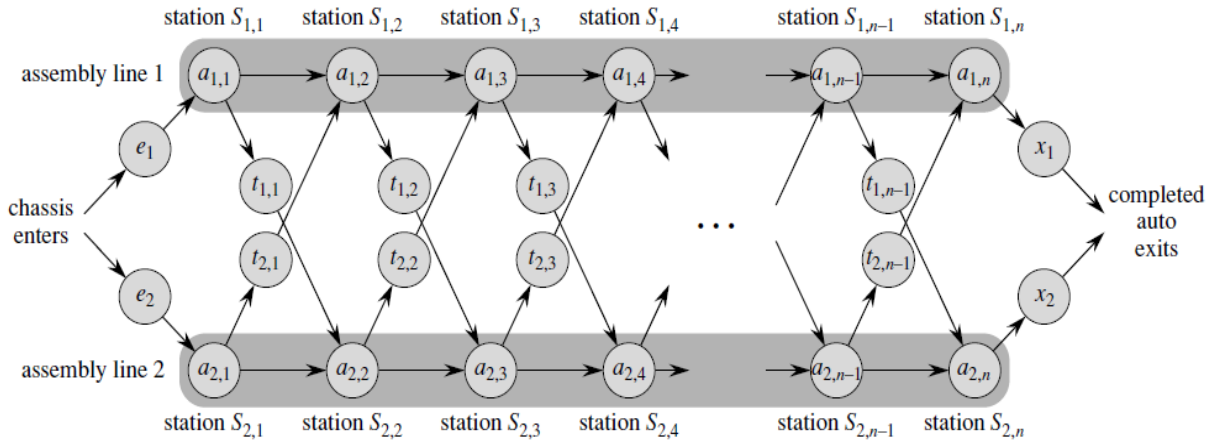
1) Steps 1–3 form the basis of a dynamic-programming solution to a problem.
2) Step 4 can be omitted if only the value of an optimal solution is required.
3) When we do perform step 4, we sometimes maintain additional information during the computation in step 3 to ease the construction of an optimal solution.

Assembly-line scheduling

A **simple dynamic-programming example** (can be solved by a graph algorithm), but a good warm-up example for dynamic programming.

Problem Description:

XYZ company produces automobiles in a factory that has two assembly lines, as shown below:



// An automobile chassis enters each assembly line, has parts added to it at a number of stations, and a finished auto exits at the end of the line.

- Each assembly line has n stations, numbered j = 1, 2, . . ., n. We denote the j^{th} station on line i (where i is 1 or 2) by $S_{i,j}$.
- The j^{th} station on line 1 ($S_{1,j}$) performs the same function as the j^{th} station on line 2 ($S_{2,j}$).
- The time required at each station varies (even between stations at the same position on the two different lines) (Because the stations were built at different times and with different technologies).
- We denote the assembly time required at station $S_{i,j}$ by $a_{i,j}$.
  // a chassis enters station 1 of one of the assembly lines, and it progresses from each station to the next.
- There is also an entry time e_i for the chassis to enter assembly line i and an exit time x_i for the completed auto to exit assembly line i. Therefore, Entry times e_1 & e_2 and Exit times x_1 and x_2.)
  // Normally, once a chassis enters an assembly line, it passes through that line only.
- The time to go from one station to the next within the same assembly line is negligible.

  // However, It is allowed to switch the partially-completed auto from one assembly line to the other after any station, but the chassis still passes through the n stations in order.

- The time to transfer a chassis away from assembly line i after having gone through station $S_{i,j}$ is $t_{i,j}$, where $i = 1,2$ and $j = 1, 2, \ldots, n-1$ (since after the nth station, assembly is complete).

Problem Statement:

The problem is to determine which stations to choose from line 1 and which to choose from line 2 in order to **minimize the total time** through the factory for one auto.

Brute force method:

- If we are given a list of which stations to use in line 1 and which to use in line 2, it is easy to compute in $\Theta(n)$ time how long it takes a chassis to pass through the factory.
- Unfortunately, there are 2^n possible ways to choose stations.
- Thus, determining the fastest way through the factory by enumerating all possible ways and computing how long each takes would require $\Omega(2^n)$ time, which is infeasible when n is large.

Step 1: The structure of the fastest way through the factory

The first step of the dynamic-programming paradigm is to **characterize the structure of an optimal solution.**

Let us consider the fastest possible way for a chassis to get from the starting point through station $S_{1,j}$.

If $j = 1$, there is only one way that the chassis could have gone, and so it is easy to determine how long it takes to get through station $S_{1,j}$.

For $j = 2, 3, \ldots, n$, however, there are two choices:

Choice 1: the chassis could have come from station $S_{1,j-1}$ and then directly to station $S_{1,j}$ (the time for going from station $j-1$ to station j on the same line being negligible).

Choice 2: the chassis could have come from station $S_{2,j-1}$ and then been transferred to station $S_{1,j}$, the transfer time being $t_{2,j-1}$

First, let us suppose that the fastest way through station $S_{1,j}$ is through station $S_{1,j-1}$. The key observation is that the chassis must have taken a fastest way from the starting point through station $S_{1,j-1}$. Why? If there were a faster way to get through station $S_{1,j-1}$, we could substitute this faster way to yield a faster way through station $S_{1,j}$: a contradiction.

Similarly, let us now suppose that the fastest way through station $S_{1,j}$ is through station $S_{2,j-1}$. Now we observe that the chassis must have taken a fastest way from the starting point through station $S_{2,j-1}$. The reasoning is the same: if there were a faster way to get through station $S_{2,j-1}$, we could substitute this faster way to yield a faster way through station $S_{1,j}$, which would be a contradiction.

In general, we can say that for assembly-line scheduling, an optimal solution to a problem (finding the fastest way through station $S_{1,j}$) contains within it an optimal solution to subproblems (finding the fastest way through either $S_{1,j-1}$ or $S_{2,j-1}$).

We refer to this property as optimal substructure, and it is one of the hallmarks of the applicability of dynamic programming. We use optimal substructure to show that we can construct an optimal solution to a problem from optimal solutions to subproblems.

For assembly-line scheduling, we reason as follows. If we look at a fastest way through station $S_{1,j}$, it must go through station $j-1$ on either line 1 or line 2. Thus, the fastest way through station $S_{1,j}$ is either

- the fastest way through station $S_{1,j-1}$ and then directly through station $S_{1,j}$, or
- the fastest way through station $S_{2,j-1}$, a transfer from line 2 to line 1, and then through station $S_{1,j}$.

Using symmetric reasoning, the fastest way through station $S_{2,j}$ is either

- the fastest way through station $S_{2,j-1}$ and then directly through station $S_{2,j}$, or
- the fastest way through station $S_{1,j-1}$, a transfer from line 1 to line 2, and then through station $S_{2,j}$.

Therefore, in order to solve the problem of finding the fastest way through station j of either line, we solve the subproblems of finding the fastest ways through station $j-1$ on both lines.

Step 2: A recursive solution

The second step of the dynamic-programming paradigm is to define the value of an optimal solution recursively in terms of the optimal solutions to subproblems.

Let $f_i[j]$ denote the fastest possible time to get a chassis from the starting point through station Si, j.

Our ultimate goal is to determine the fastest time to get a chassis all the way through the factory, which we denote by f*. The chassis has to get all the way through station n on either line 1 or line 2 and then to the factory exit.

$$f^* = \min(f_1[n] + x_1,\ f_2[n] + x_2)$$

(Eq. (1))

It is also easy to reason about $f_1[1]$ and $f_2[1]$, To get through station 1 on either line, a chassis just goes directly to that station. Thus,

$$\begin{aligned} f_1[1] &= e_1 + a_{1,1} \\ f_2[1] &= e_2 + a_{2,1} \end{aligned}$$

Now let us consider how to compute $f_i[j]$ for $j = 2, 3, \ldots, n$ (and $i = 1, 2$).

Also, the fastest way through station $S_{1,j}$ is either the fastest way through station $S_{1,j-1}$ and then directly through station $S_{1,j}$. ⇒ In this case $f_1[j] = f_1[j-1] + a_{1,j}$

the fastest way through station $S_{2,j-1}$, a transfer from line 2 to line 1, and then through station $S_{1,j}$. ⇒ In this case $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$. Therefore,

$$f_1[j] = \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j})$$

for $j = 2, 3, \ldots, n$. Symmetrically, we have

$$f_2[j] = \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})$$

Combining the above equations, we obtain the recursive equations:

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

(Eq. (2) & (3))

To help us keep track of how to construct an optimal solution, let us define $l_i[j]$ to be the line number, 1 or 2, whose station $j - 1$ is used in a fastest way through station $S_{i,j}$. Here, $i = 1,2$ and $j = 2, 3, \ldots, n$.

We also define l^* to be the line whose station n is used in a fastest way through the entire factory.

Using the values of l^* and $l_i[j]$, we would trace a fastest way through the factory as follows:

Starting with $l^* = 1$, we use station $S_{1,6}$.

Now we look at $l_1[6]$, which is 2, and so we use station $S_{2,5}$.

Continuing, we look at $l_2[5] = 2$ (use station $S_{2,4}$),

$l_2[4] = 1$ (station $S_{1,3}$),

$l_1[3] = 2$ (station $S_{2,2}$), and

$l_2[2] = 1$ (station $S_{1,1}$).

Step 3: Computing the fastest times

At this point, it would be a simple matter to write a recursive algorithm based on equation (1) and the recurrences (2) and (3) to compute the fastest way through the factory. There is a problem with such a recursive algorithm: its running time is exponential in n.

Proof:

let $r_i(j)$ be the number of references made to $f_i[j]$ in a recursive algorithm.

From Eq. (1), $r_1(n) = r_2(n) = 1$

From the recurrences (2) and (3), we have

$r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1)$

Claim

$r_i(j) = 2^{n-j}$.

Proof Induction on j, down from n.

Basis: $j = n$. $2^{n-j} = 2^0 = 1 = r_i(n)$.

Inductive step: Assume $r_i(j+1) = 2^{n-(j+1)}$.

$$\begin{aligned} \text{Then } r_i(j) &= r_i(j+1) + r_2(j+1) \\ &= 2^{n-(j+1)} + 2^{n-(j+1)} \\ &= 2^{n-(j+1)+1} \\ &= 2^{n-j} . \end{aligned}$$

Therefore, $f_1[1]$ alone is referenced 2^{n-1} times! So top down isn't a good way to compute $f_i[j]$.

We can do much better if we compute the $f_i[j]$ values in a different order from the recursive way.

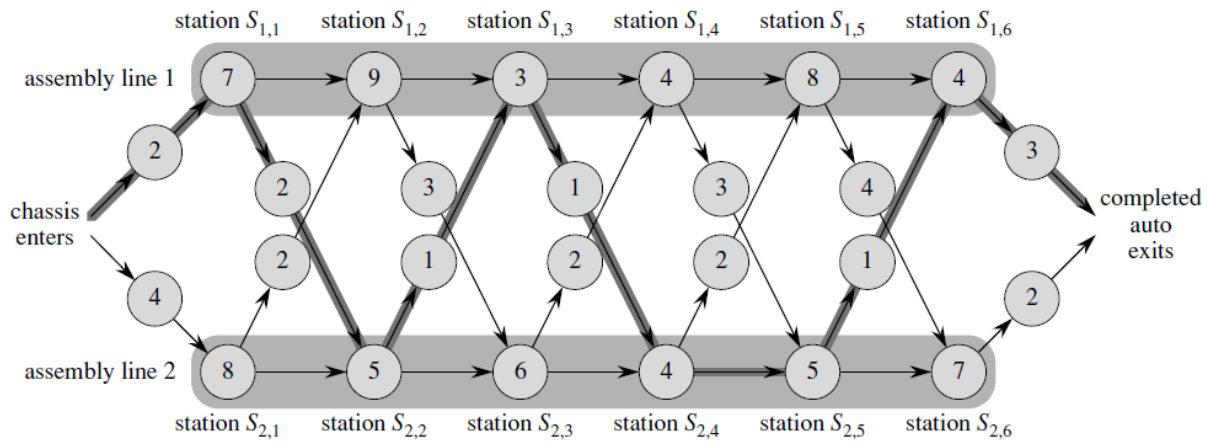
Observation: $f_i[j]$ depends only on $f_1[j-1]$ and $f_2[j-1]$ (for $j \geq 2$).

So compute in order of *increasing* j.

Therefore, by computing the $f_i[j]$ values in order of increasing station numbers j—left to right, we can compute the fastest way through the factory, and the time it takes, in $\Theta(n)$ time.

```
FASTEST-WAY(a, t, e, x, n)
 1  f_1[1] ← e_1 + a_{1,1}
 2  f_2[1] ← e_2 + a_{2,1}
 3  for j ← 2 to n
 4       do if f_1[j − 1] + a_{1,j} ≤ f_2[j − 1] + t_{2,j−1} + a_{1,j}
 5            then f_1[j] ← f_1[j − 1] + a_{1,j}
 6                 l_1[j] ← 1
 7            else f_1[j] ← f_2[j − 1] + t_{2,j−1} + a_{1,j}
 8                 l_1[j] ← 2
 9          if f_2[j − 1] + a_{2,j} ≤ f_1[j − 1] + t_{1,j−1} + a_{2,j}
10            then f_2[j] ← f_2[j − 1] + a_{2,j}
11                 l_2[j] ← 2
12            else f_2[j] ← f_1[j − 1] + t_{1,j−1} + a_{2,j}
13                 l_2[j] ← 1
14  if f_1[n] + x_1 ≤ f_2[n] + x_2
15     then f* = f_1[n] + x_1
16          l* = 1
17     else f* = f_2[n] + x_2
18          l* = 2
```

Example:



(a)

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

(b)

One way to view the process of computing the values of $f_i[j]$ and $l_i[j]$ is that we are filling in table entries.

we fill tables containing values $f_i[j]$ and $l_i[j]$ from left to right (and top to bottom within each column).

To fill in an entry $f_i[j]$ we need the values of $f_1[j-1]$ and $f_2[j-1]$ and, knowing that we have already computed and stored them, we determine these values simply by looking them up in the table.