CS4311
Design and Analysis of Algorithms

Lecture 9: Dynamic Programming I
About this lecture

• **Divide-and-conquer** strategy allows us to solve a big problem by handling only smaller sub-problems.

• Some problems may be solved using a stronger strategy: **dynamic programming**.

• We will see some examples today.
Assembly Line Scheduling

• You are the boss of a company which assembles Gundam models to customers
Assembly Line Scheduling

• Normally, to assemble a Gundam model, there are $n$ sequential steps

Step 0: getting model  →  Step 1: assembling body  →  Step 2: assembling legs

...  →  Step $n-1$: polishing  →  Step $n$: packaging
Assembly Line Scheduling

• To improve efficiency, there are two separate assembly lines:

Line 1

Line 2
Assembly Line Scheduling

- Since different lines hire different people, processing speed is not the same:

  Line 1: 5 → 1 → 3 → ... → 2 → 3

  Line 2: 2 → 6 → 1 → ... → 3 → 1

  E.g., Line 1 may need 34 mins, and Line 2 may need 38 mins
Assembly Line Scheduling

• With some transportation cost, after a step in a line, we can process the model in the other line during the next step.
Assembly Line Scheduling

• When there is an urgent request, we may finish faster if we can make use of both lines + transportation in between.

E.g., Process Step 0 at Line 2, then process Step 1 at Line 1, => better than process both steps in Line 1
Assembly Line Scheduling

Question: How to compute the fastest assembly time?

Let $p_{1,k} =$ Step $k$'s processing time in Line 1

$p_{2,k} =$ Step $k$'s processing time in Line 2

$t_{1,k} =$ transportation cost from Step $k$ in Line 1 (to Step $k+1$ in Line 2)

$t_{2,k} =$ transportation cost from Step $k$ in Line 2 (to Step $k+1$ in Line 1)
Assembly Line Scheduling

Let $f_{1,j}$ = fastest time to finish Steps 0 to $j$, ending at Line 1

$f_{2,j}$ = fastest time to finish Steps 0 to $j$, ending at Line 2

So, we have:

$f_{1,0} = p_{1,0}$, $f_{2,0} = p_{2,0}$

Fastest time = $\min \{ f_{1,n}, f_{2,n} \}$
Assembly Line Scheduling

How can we get $f_{1,j}$?

Intuition:

Let $(1,j) = j^{th}$ step of Line 1

The fastest way to get to $(1,j)$ must be:

- First get to the $(j-1)^{th}$ step of each line using the fastest way, and choose whichever one that goes to $(1,j)$ faster

Is our intuition correct?
Assembly Line Scheduling

Lemma: For any $j > 0$,

$$f_{1,j} = \min \left\{ f_{1,j-1} + p_{1,j}, \ f_{2,j-1} + t_{2,j-1} + p_{1,j} \right\}$$

$$f_{2,j} = \min \left\{ f_{2,j-1} + p_{2,j}, \ f_{1,j-1} + t_{1,j-1} + p_{2,j} \right\}$$

Proof: By induction + contradiction

Here, optimal solution to a problem (e.g., $f_{1,j}$) is based on optimal solution to subproblems (e.g., $f_{1,j-1}$ and $f_{2,j-1}$)

$\Rightarrow$ optimal substructure property
Define a function `Compute_F(i, j)` as follows:

\[ \text{Compute}_F(i, j) \quad \text{/* Finding } f_{i,j} \text{ */} \]

1. if \((j == 0)\) return \(p_{i,0}\);
2. \(g = \text{Compute}_F(i, j-1) + p_{i,j}\);
3. \(h = \text{Compute}_F(3-i, j-1) + t_{3-i, j-1} + p_{i,j}\);
4. return \(\min \{ g, h \} \);

Calling \(\text{Compute}_F(1,n)\) and \(\text{Compute}_F(2,n)\) gives the fastest assembly time.
Assembly Line Scheduling

Question: What is the running time of Compute_F(i,n)?

Let T(n) denote its running time.

So, \( T(n) = 2T(n-1) + \Theta(1) \)

\( \Rightarrow \) By Recursion-Tree Method,

\[ T(n) = \Theta(2^n) \]
Assembly Line Scheduling

To improve the running time, observe that:

To Compute_F(1,j) and Compute_F(2,j), both require the SAME subproblems:
- Compute_F(1,j-1) and Compute_F(2,j-1)

So, in our recursive algorithm, there are many repeating subproblems which create redundant computations!

Question: Can we avoid it?
Bottom-Up Approach (Method I)

• We notice that
  \[ f_{i,j} \] depends only on \( f_{1,k} \) or \( f_{2,k} \) with \( k < j \)

• Let us create a 2D table \( F \) to store all \( f_{i,j} \) values once they are computed

• Then, let us compute \( f_{i,j} \) from \( j = 0 \) to \( n \)
Bottom-Up Approach (Method I)

BottomUp_F() /* Finding fastest time */

1. \( F[1,0] = p_{i,0} \), \( F[2,0] = p_{2,0} \);

2. for \( (j = 1,2,\ldots, n) \) {
   Compute \( F[1,j] \) and \( F[2,j] \);
   // Based on \( F[1,j-1] \) and \( F[2,j-1] \)
}

3. return \( \min \{ F[1,n], F[2,n] \} \);

Running Time = \( \Theta(n) \)
Memoization (Method II)

• Similar to Bottom-Up Approach, we create a table $F$ to store all $f_{i,j}$ once computed.

• However, we modify the recursive algorithm a bit, so that we still solve compute the fastest time in a Top-Down.

• Assume: entries of $F$ are initialized empty.

Memoization comes from the word “memo”
Original Recursive Algorithm

Compute_F(i, j) /* Finding f_{i,j} */

1. if (j == 0) return p_{i,0};
2. g = Compute_F(i, j-1) + p_{i,j};
3. h = Compute_F(3-i, j-1) + t_{3-i, j-1} + p_{i,j};
4. return min \{ g, h \};
Memoized Version

Memo_F(i, j) /* Finding f_{i,j} */

1. if (j == 0) return p_{i,0};
2. if (F[i,j-1] is empty)
   \[
   F[i,j-1] = \text{Memo}_F(i,j-1);
   \]
3. if (F[3-i,j-1] is empty)
   \[
   F[3-i,j-1] = \text{Memo}_F(3-i,j-1);
   \]
4. \[g = F[i,j-1] + p_{i,j} ;\]
5. \[h = F[3-i,j-1] + t_{3-i,j-1} + p_{i,j} ;\]
6. return min \{ g, h \};
Memoized Version (Running Time)

To find Memo_F(1, n):

1. Memo_F(i, j) is only called when F[i, j] is empty (it becomes nonempty afterwards)
   \[\Theta(n)\] calls

2. Each Memo_F(i, j) call only needs \(\Theta(1)\) time apart from recursive calls

Running Time = \(\Theta(n)\)
Dynamic Programming

The previous strategy that applies “tables” is called **dynamic programming (DP)**

[ Here, programming means: a good way to plan things / to optimize the steps ]

• A problem that can be solved efficiently by DP often has the following properties:
  1. **Optimal Substructure** (allows recursion)
  2. **Overlapping Subproblems** (allows speed up)
Assembly Line Scheduling

Challenge: We now know how to compute the fastest assembly time. How to get the exact sequence of steps to achieve this time?

Answer: When we compute $f_{i,j}$, we remember whether its value is based on $f_{1,j-1}$ or $f_{2,j-1}$. 

⇒ easy to modify code to get the sequence
Sharing Gold Coins

Five lucky pirates has discovered a treasure chest with 1000 gold coins ...
Sharing Gold Coins

There are rankings among the pirates:

1  2  3  4  5

... and they decide to share the gold coins in the following way:
Sharing Gold Coins

First, Rank-1 pirate proposes how to share the coins...

• If at least half of them agree, go with the proposal
• Else, Rank-1 pirate is out of the game

Hehe, I am going to make the first proposal ... but there is a danger that I cannot share any coins
Sharing Gold Coins

If Rank-1 pirate is out, then Rank-2 pirate proposes how to share the coins...

• If at least half of the remaining agree, go with the proposal
• Else, Rank-2 pirate is out of the game

Hehe, I get a chance to propose if Rank-1 pirate is out of the game
Sharing Gold Coins

In general, if Rank-1, Rank-2, ..., Rank-k pirates are out, then Rank-\((k+1)\) pirate proposes how to share the coins...

- If at least half of the remaining agree, go with the proposal
- Else, Rank-\((k+1)\) pirate is out of the game

Question: If all the pirates are smart, who will get the most coin? Why?