Lecture 12 – Dynamic Programming
Prof. Manolis Kellis

Today: Dynamic Programming

• Refresher: Lecture 1 – Interval Scheduling
  – **Definitions:** Guess, recurse, combine
    – **Guess:** find where to break up current step
    – **Recurse:** solve subproblems recursively
    – **Combine:** construct curr. soln from sub solns
• Games: Alternating Coin Game / minimax
• Dictionary: Optimal Binary Search Trees
• Language: Parsing Context-Free-Grammars

Remember lecture 1: Interval scheduling

• **DP** = recursive formula + overlapping subproblems
• **Algorithm:**
  – Sort all \( r_i \) by the **finish** time
  – **Choice:** consider \( r_1 \):
    - If \( r_1 \) **not in** \( \text{OPT} (R) \), then the optimal cost is
      \[
      \text{score}(\text{OPT}(R)) = \text{score}(\text{OPT}(R-\text{r}_1))
      \]
    - If \( r_1 \) **in** \( \text{OPT} (R) \), then the optimal cost is
      \[
      \text{score}(\text{OPT}(R)) = w_1 + \text{score}(\text{OPT}(R-\text{r}_1-\text{INC}(r_1))),
      \text{where INC}(r_i) \text{ are the intervals incompatible with } r_i
      \]
  – **Recurrence:**
    \[
    \text{score}(\text{OPT}(R)) = \max \left[ w_1 + \text{score}(\text{OPT}(R-\text{r}_1-\text{INC}(r_1))), \text{score}(\text{OPT}(R-\text{r}_i)) \right]
    \]

Reveal repeated sub-problems

• Key insight: In the recursion, we keep repeating the same operations, over and over again

Order computation bottom-up

• If we make additions from the bottom up, the sub-problems become identical

Top-down decision tree

• Find best-scoring combination going down
• Exponential \( O(2^n) \) paths given binary choices
• **DP** explores them all, in linear space/time!
Fill table of intermediate results
- We can thus simply ‘cache’ (remember, ‘store’) the best score obtained at each decision point
- the choice (Y|N) that led to it

Table lookup instead of computation
- We can now look-up the solution instead of recomputing it, saving a lot of work!

Traceback: constructing optimal solution
- Once we have the max score, we can trace back our choices, to find path of the optimal solution

Dynamic programming solution runtime
1. Sort all (stat and) end times \( \rightarrow O(n \log n) \)
2. Construct predecessor variable \( \rightarrow O(n) \)
3. Compute: \( M[i]=\max\{M[i-1], w[i]+M[P(i)]\} \) for \( i=1..n \) \( \rightarrow O(n) \)
4. Traceback: follow \( P(i) \) pointers, based on \( ch[i] \) choices \( \rightarrow O(n) \)

Constructing the predecessor matrix
- \( O(n^2) \) solution
  - Simply walk backwards until compatible interval found
- \( O(n \log n) \) solution
  - Do a binary search since the end times are already sorted!
- \( O(n) \) (once start + end are sorted)
1. Jointly sort all endpoints, remember start or end
   - sort: \( a_1, a_2, a_4, b_1, a_3, b_2, b_3, a_6, a_5, b_4, b_5, b_6 \)
   - \( \pi = 0, 0, 0, 1, 1, 2, 3, 3, 3, 4, 4, 5, 6 \) Single traversal
2. For each \( b_i \) encounter, set variable \( \pi \) to \( i \)
3. For each \( a_i \) encounter, set variable \( p(i) \) to \( \pi \)

Today: Dynamic Programming
- Refresher: Lecture 1 – Interval Scheduling
- Definitions: Guess, recurse, combine
  - Guess: find where to break up current step
  - Recurse: solve subproblems recursively
  - Combine: construct curr. soln from sub solns
- Games: Alternating Coin Game / minimax
- Dictionary: Optimal Binary Search Trees
- Language: Parsing Context-Free-Grammars
Today: Dynamic Programming

- **Refresher:** Lecture 1 – Interval Scheduling
- **Definitions:** Guess, recurse, combine
  - Guess: find where to break up current step
  - Recurse: solve subproblems recursively
  - Combine: construct curr. soln from sub solns
- **Games:** Alternating Coin Game / minimax
- **Dictionary:** Optimal Binary Search Trees
- **Language:** Parsing Context-Free-Grammars

---

**Example 1: Alternating Coin Game**

- Row of \( n \) coins of values \( v_1, v_2, ..., v_n \), where \( n \) even
- Players play in turns
- In each turn, a player removes either the first or the last coin, receives it permanently and receives its value.

\[
\begin{array}{c}
\text{player 1: 6}
\end{array} \quad \begin{array}{c}
\text{player 2: 14}
\end{array}
\]

**Question:** Given \( v_1, v_2, ..., v_n \) can player 1 win no matter how well player 2 plays?

\[
v(i,j) : \text{value player 1 can guarantee to himself if remaining coins are } v_i, v_{i+1}, ..., v_j
\]

**Q:** Who plays if remaining coins are \( v_i, ..., v_j \)?

**A:** If \( j-i \) even, player 2
- \( j-i \) odd, player 1

**Q:** how relate \( v(i,j) \) to subproblems?

**A:** two cases:
- \( j-i \) is even (player 2 plays)
  - guess whether he picks \( v_i \) or \( v_j \)
  - choose the one to minimize \( v(i,j) \)
  \[
v(i,j) = \min \left( v(i+1,j), v(i,j-1) \right)
\]
- \( j-i \) is odd (player 1 plays)
  - guess whether to pick coin \( v_i \) or \( v_j \)
  - choose the one to maximize \( v(i,j) \)
  \[
v(i,j) = \max \left( v_0 + v(i+1,j), v_j + v(i,j-1) \right)
\]
Filling in the DP matrix

Our choice
Adversary
Our choice
Adversary
Our choice

Filling in the DP matrix

Our choice
Adversary
Our choice
Adversary
Our choice
Today: Dynamic Programming

- **Refresher**: Lecture 1 – Interval Scheduling
- **Definitions**: **Guess, recurse, combine**
  - **Guess**: find where to break up current step
  - **Recurse**: solve subproblems recursively
  - **Combine**: construct curr. soln from sub solns
- **Games**: Alternating Coin Game / minimax
- **Dictionary**: Optimal Binary Search Trees
- **Language**: Parsing Context-Free-Grammars

---

**Example 2**: Optimal Binary Search Trees

**Input**: keys $k_1, k_2, \ldots, k_n$, where $k_1 < k_2 < \cdots < k_n$

Search probabilities $p_1, \ldots, p_n$

**Goal**: store keys in BST to minimize expected search cost, namely

$$\min_{i=1}^{n} p_i \cdot \left( \text{depth}(k_i) + 1 \right)$$

**Real-world Application**: English-French dictionary

- Want more frequent words to have faster look-up time
- Key $k_i$

**Work to done**: searching for word

---

**Algorithm 1**: Enumerate all possible BSTs on $k_1, k_2, \ldots, k_n$

- **n=2**
- $k_2$
- $k_1$
- $p_1 + 2p_2$
- $k_3$
- $k_2$
- $2p_1 + p_2$

- **n=3**
- $k_3$
- $k_2$
- $k_1$
- $p_1 + 2p_2 + 3p_3$
- $p_3 + 2p_1 + 3p_2$
- $p_3 + 2p_1 + 3p_2$
- $p_1 + 2p_2 + 3p_3$

**Cost** = $10 + 2 \times 1 + 2 \times 2 + 3 \times 8 = 54$

---

**Algorithm 2**: place key $k_i$ with highest search probability $p_i$ at root of tree

- Then recursively solve problem for left keys $k_1, \ldots, k_{i-1}$ and right keys $k_{i+1}, \ldots, k_n$

- **Correct?**

- $P_5 = 10$

- $P_5 = 9$

- $P_3 = 8$

- **Cost** = $9 + 2 \times 10 + 2 \times 8 + 3 \times 1 = 48$

---

**Today: Dynamic Programming**

- **Refresher**: Lecture 1 – Interval Scheduling
- **Definitions**: **Guess, recurse, combine**
  - **Guess**: find where to break up current step
  - **Recurse**: solve subproblems recursively
  - **Combine**: construct curr. soln from sub solns
- **Games**: Alternating Coin Game / minimax
- **Dictionary**: Optimal Binary Search Trees
- **Language**: Parsing Context-Free-Grammars
**Algorithm 3: Dynamic Programming**

Q: What guess would allow me to split into subproblems?

A: Which key to put in the root?

$e(i,j)$ = cost of optimal BST containing keys $k_i, k_{i+1}, \ldots, k_j$.

$$e(i,j) = \begin{cases} P_0 & \text{if } i = j \\ \min_{r \in S_i} \left( e(i,r-1) + e(r+1,j) + \frac{1}{2} \sum_{k_j \in S_i} \right) & \text{otherwise} \end{cases}$$

accounts for $P_r$ of the root node, as well as the increase in depth by $1$ of all keys in subtrees of $k_r$.

$O(n^2) \times O(n)$ subproblems, $O(n^2)$ work per subproblem.

**Example 3: Parsing Context Free Grammars**

Non-terminal symbols $\Sigma = \{ +, -, 0, 1, \ldots, 9, \star, (, ), \}$

Non-terminal symbols $N = \{ INTEGER, SIGN, DIGITS, DIGIT, EXPR, EXPRS, TUPLE \}$

Special starting non-terminal symbol: $EXPR$

Rules $R$: see above.

Problem: given string $s \in \Sigma^*$, decide whether it can be generated by the CFG starting w/ the starting non-terminal symbol.

**Example:**

```
(5 + 3, (2*(42)))
```

1. DP subproblem
   $\Pi(i,j, X)$: 0/1 depending on whether string $S[i:j]$ can be generated starting from $X$.
2. Guessing: what rule $X \rightarrow X_1 \ldots X_m$
   apply and at which index $k_i$ each $X_k$ starts (note $k_1 = 1$ is forced).
3. Combining:
   $\Pi(i,j, X) = 1$ iff for some guess $(X \rightarrow X_1 \ldots X_m)$:
   $$\Pi(k_1,k_2) = 1 \quad \forall i \in 1, m, k_{i+1} = k_i + 1$$
Base cases:

\[ \pi(i,j, \text{ terminal}) = \begin{cases} \emptyset, & \text{if } j = 0 \text{ or } s[i] = \text{ terminal symbol} \\ \{ 1 \}, & \text{if } j = i \\ 0, & \text{otherwise} \end{cases} \]

\[ \pi(i,j, \varepsilon) = \begin{cases} \emptyset, & \text{if } j = i \\ 1, & \text{otherwise} \end{cases} \]

\[ \pi(i,j, \text{ anything}) = 0 \]

Interested in \( \pi(0, |S|, \text{ non-terminal}) \)

Runtime:

\[ \# \text{ subproblems: } \leq |S|^2 \times |N| \]

\[ \# \text{ guesses for subproblem } \pi(i,j,x): \leq (|\text{rules}|) \times |S|^{m-1} \]

\[ \text{work per guess: } O(m) \]

\[ \Rightarrow \text{ runtime } \leq (|\text{rules}|) \times |S|^{m-1} \times O(m) \]

\[ = O(|S|^2 \times |N| \times |\text{CFG}| \times |S|^{m-1} \times m) \]

\[ = O(|S|^m \times |N| \times |\text{FG}| \times m) \]

- Fine if RHS's are bounded (small m)
- If not, convert to Chomsky Normal Form
  - All RHS's are of the form nonterm nonterm or terminal

Challenge: DP that runs in \( O(|S|^m \times |N| \times |\text{FG}|) \) without resorting to Chomsky Normal Form?