Chapter 5

Dynamic Programming

Dynamic Programming Algorithms

- This is a method of solving complex problems by breaking them down into simpler subproblems
- ‘Programming’ refers to t
- abular method not writing of computer code
- Unlike Divide and Conquer where subproblems are independent, DP is applicable when the subproblems are not independent; they share subsubproblems
- DP algorithms solves every subsubproblem once and saves the answer in a table
- It is applied for optimization problems; where there can be many possible solutions. The goal is usually to find a an optimal (minimum or maximum) solution

Solving a dynamic DP problems

- Solving a dynamic problem involves four steps

1. Define subproblems
2. Write down the recurrence that relates to subproblems
3. Recognize and solve the base cases
4. Construct an optimal solution combined.

- The final step is not a must it be executed.

Elements of a DP

A. Optimal Substructure
- If an optimal solution to the problem contains within it optimal solutions to subproblems optimal solution is built from the solutions to subproblems
- The range of subproblems considered include those used in an optimal solution
- Example: Assembly-Line Scheduling
  - The fastest way through station j of either line contained within it fastest way through station j-1

Elements of a DP

- In identifying Optimal Substructure, the following steps are followed:

1. Show that a solution to the problem consists of making a choice; making this choice leaves one or more subproblems to be solved
2. You suppose that for a given problem, you are given the choice that leads to an optimal solution.
3. Given this choice, you determine which subproblems ensue and how best to characterize the resulting space of subproblems
4. Show that the solutions to the subproblems used within the optimal solution to the problem must themselves be using a “cut-and-paste” technique

Elements of a DP

- Optimal substructure varies across problem domains in two ways:
  1. How many subproblems are used in a an optimal solution to the original problem
  2. How many choices we have in determining which subproblem(s) to use in an optimal solution
- Example:
  - Assembly line scheduling uses one subproblem and two choices
- The running time of DP therefore is computed:
  - Number of subproblems X Number of choices
  - Assembly line: $\Theta(n) \times 2 = \Theta(n)$
  - Matrix-chain multiplication: $\Theta(n^2) \times n-1 = \Theta(n^3)$

Elements of a DP

B. Overlapping subproblems
   - The space of subproblems must be small; a recursive algorithm for the problem solves the same subproblem over and over rather than generating new ones

C. Reconstructing an optimal solution
   - Store each choice made in each subproblem in a table so that to avoid reconstruction of the information

DP Algorithm - Examples

- Coin Changing
- Assembly line scheduling
- Matrix-Chain Multiplication
- Longest Common Subsequence
- Optimal Binary Search Trees
- Finding nth Fibonacci Number

Coin Changing

- Goal: Given a set of discrete coin denomination values, make change for amount A using minimum number of coins
  - For Kenya Shillings coin denominations, a greedy approach can be shown to be optimal. Assume denom is array such that
    - denom[1] = 10, denom[2] = 5, denom[3] = 1

```
Coin-Changer-Greedy(A)
1. i = 1
2. while (A > 0)
3. c = A/denom[i]
4. print c coins of denom[i]
5. A = A - c * denom[i]
6. i++
```

Coin Changing

- Goal: Given discrete coin denomination values, make change for amount A using minimum number of coins (bottom-up approach)
- If different (lets say KShs. after ‘inflation’) denominations are used, the greedy approach is no longer optimal. Assume denom is array such that denom[1] = 40, denom[2] = 15, denom[3] = 6, denom[4] = 1
- Try using the algorithm on the last slide to make change for 91 cents with the fewest number of coins. Greedy approach clearly is not optimal in this simple example.
- Idea of DP solution: Keep a running count, for all denominations, of how many coins of each type it takes to make change for a given amount, along with a way to trace exactly which coins go into the optimal solution.

Assembly-line Scheduling

- Manufacturing problem to find the fastest way through a factory using 2 assembly lines.
- Considers assembly lines with n stations each.
- Parameters:
  - $S_{i,j}$: The jth station on assembly line i
  - $a_{i,j}$: Assembly time at the jth station on line i
  - e_i : Time to enter assembly line i
  - x_i : Time to exit from assembly line i
  - $t_{i,j}$: Time to transfer from station i on line 1 to line j
- Problem: determine which stations to choose from line 1 and which to choose from line 2 in order to minimize assembly time.

Assembly-line Scheduling

- First, consider the optimal way for a product to get from the starting point through station $S_{1,j}$.
  - If j = 1, there is one choice --> go through station 1,1
  - For j = 2,3,4,..., n, there are 2 choices
    - the fastest way through $S_{1,j-1}$ and then directly through $S_{1,j}$, or
    - the fastest way through $S_{2,j-1}$, a transfer from line 2 to line 1, and then through station $S_{1,j}$
- Problem has optimal substructure, because it can be expressed recursively. To find the fastest way through station j, we solve the sub-problems of finding the fastest way through station j - 1.

Matrix-Chain Product

- Matrix-Chain Multiplication Problem:
  - Given matrices A1, A2, A3, ..., An, where the dimension of A_i is
  - $d_{i-1} \times d_i$, determine the minimum number of multiplications needed to compute the product $A_1 \cdot A_2 \cdot ...A_n$. This involves finding the optimal way to parenthesize the matrices.

Optimal Binary Search Trees

Introduction

OPTIMAL BINARY SEARCH TREE

Def:

- optimal binary search tree is a binary search tree which focuses on reducing the cost of a binary search tree.
- It may not have the lowest height !
- This optimal binary search tree records the probability, the cost and the root of the tree
- This type of tree has two type of keys:
    - data keys i.e. internal keys
    - dummy keys(leaves) i.e. external keys

- Data keys may contain at least one dummy key
- To get the search there are two probabilities
- The probability of success i.e (i=1~n) Σp_i+ (i=0~n) $\Sigma q_i=1$.
- It has n keys (representation $k_1,k_2,\ldots,k_n$) in sorted order (so that $k_1<k_2<\ldots<k_n$), and we wish to build a binary search tree from these keys. For each k_i ,we have a probability p_i that a search will be for k_i.
- Because we have probabilities of searches for each key and each dummy key, we can determine the expected cost of a search in a given binary search tree T.

- Let us assume that the actual cost of a search is the number of nodes examined, i.e., the depth of the node found by the search in T,plus1.
- Then the expected cost of a search in T is : (The second statement)
- E[search cost in T]

$$= (i=1\sim n) \sum p_i \cdot (\text{depth}_T(k_i)+1) + (i=0\sim n) \sum q_i \cdot (\text{depth}_T(d_i)+1)$$

$$=1 + (i=1\sim n) \sum p_i \cdot \text{depth}_T(k_i) + (i=0\sim n) \sum q_i \cdot \text{depth}_T(d_i)$$

Where depth_T denotes a node’s depth in the tree T.

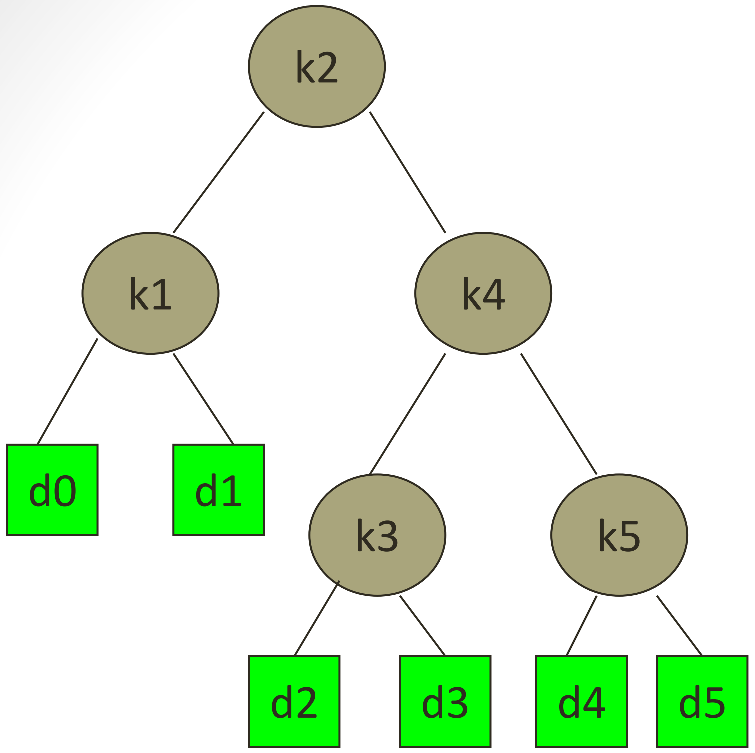

Figure (a)

i	0	1	2	3	4	5
pi		0.15	0.10	0.05	0.10	0.20
qi	0.05	0.10	0.05	0.05	0.05	0.10

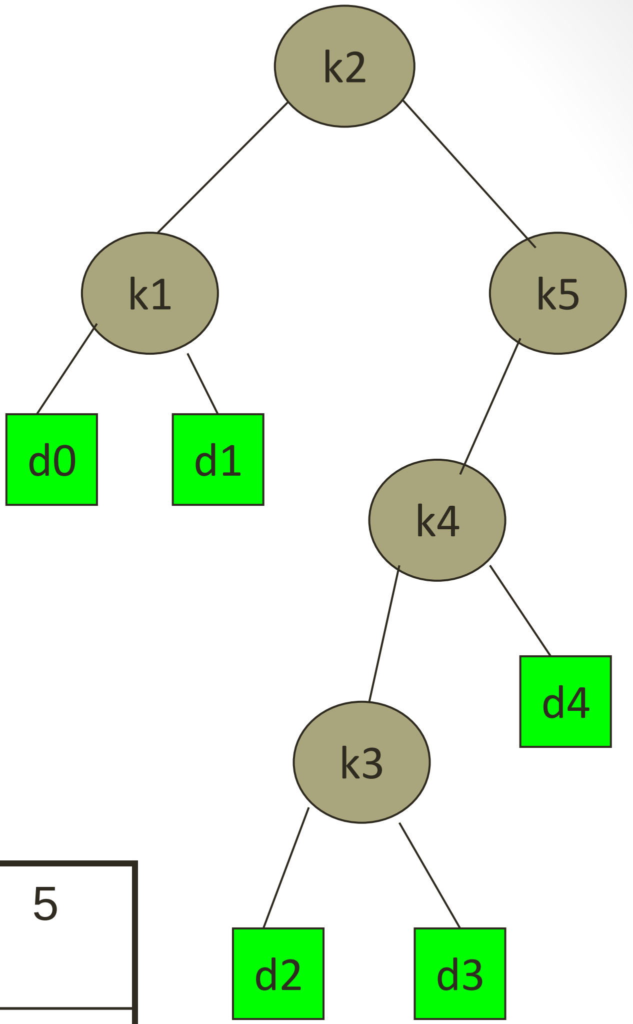

Figure (b)

Optimal binary search tree

- And the total cost = (0.30 + 0.10 + 0.15 + 0.20 + 0.60 + 0.15 + 0.30 + 0.20 + 0.20 + 0.20 + 0.40) = 2.80
- So Figure (a) costs 2.80 ,on another, the Figure (b) costs 2.75, and that tree is really optimal.
- We can see the height of (b) is more than (a) , and the key k5 has the greatest search probability of any key, yet the root of the OBST shown is k2.

Characteristics of OBST

- All the data keys must contain at least one dummy key except the root
- Every sub tree must have a contagious range
- The key at any internal node is greater than all the keys in the lest sub trees and less than all the keys in the right sub tree

```
• For i      1 to n+1
        do e[i,i-1]      $q_{i-1}$
        do w[i,i-1]      $q_{i-1}$
  For l    1 to n
        do for i    1 to n-l +1
                do j    i+l-1
                e[i,j]    ∞
                w[i,j]    w[i,j-1]+$p_{j+}q_j$
                For r     i to j
                    do t    e[i,r-1]+e[r+1,j]+w[i,j]
                        if t<e[i,j]
                          then e[i,j]     t
                          root [i,j]     r
Return e and root
```

Analysis

- The elements considered in the analysis are:
- k_n = key (data key)
- p_n = probability of searching k_n
- d_n = dummy key
- q_n = probability of searching d_n

APPLICATIONS

- Dictionary problems
- Used in *many* search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries

- Used in solving salesman problems.