Dynamic Programming

Agenda

- Dynamic Programming
- Coin row problem
- Find the max subarray - Kadane’s algorithm

Dynamic Programming

- D*ynamic Programming* (DP) is a general algorithm design technique for solving problems with *overlapping* subproblems

- Main idea of the classical DP:
  - Given a recurrence, solve for increasing inputs
  - Store the solutions for *all* subproblems in a table

Example 1: Fibonacci Sequence

Recall the definition of the Fibonacci sequence:

$F(0) = 0$

$F(1) = 1$

$F(n) = F(n\text{-}1) + F(n\text{-}2)$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55…

Recursive solution: Return the nth number in the Fibonacci sequence

```
F(n)
    if n=0
        return 0
    if n=1
        return 1
    return F(n-1) + F(n-2)
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55…

Fibonacci – Linear Complexity with DP

- Computing the n^{th} Fibonacci using Dynamic Programming (DP)
  - Solve for increasing inputs and record results in an array F

F[0] = 0
F[1] = 1
F[2] = 1+0 = 1
.
.
.
F[i] = F[i-1] + F[i-2]

0	**1**	**1**	**2**	**3**	**5**	**8**	**13**	**21**	**34**	**55**	**...**
0	1	2	3	4	5	6	7	8	9	10	

Fibonacci – Linear Complexity with DP

- Computing the n^{th} Fibonacci using Dynamic Programming (DP)
    - Solve for increasing inputs and record results in an array F

```
DP-ALGORITHM Fib(n)

F[0] = 0; F[1] = 1
for i ← 2 to n do
        F[i] = F[i-1] + F[i-2]

return F[n]
```

0	1	1	2	3	5	8	13	21	34	55	...

- *Time Efficiency*: O(n)

Coin-Row Problem

There is a row of *n* coins whose values are some positive integers

$$c_1, c_2, \ldots, c_n$$

not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two adjacent coins may be picked up.

E.g. 5, 1, 2, 10, 6, 2. What is the best selection?

Coin-Row Example

Index i	0	1	2	3	4	5	6
Coin C	--	5	1	2	10	6	2
Solution S	0	5					

Base cases:
S(0)=0
S(1)=c(1)

If we have 0 coins the max is 0
If we have one coin the max is that one coin

Coin-Row Example

- DP Algorithm:

```
S[0] ← 0;  S[1] ← C[1]
for i ← 2 to n do
        S[i] ← max(C[i] + S[i-2], S[i-1])
```

- Trace DP Algorithm on coin row 5, 1, 2, 10, 6, 2:

Index i	0	1	2	3	4	5	6
Coin C	--	5	1	2	10	6	2
Solution S	0	5	5	7	15	15	17

- Optimal solution: 17, found at S(n)

Coin-Row Dynamic Programming Algorithm

```
ALGORITHM CoinRow(C[1, ..., n])
// Input : an array C of positive integers indicating coin values
// Output: maximum amount of money that can be picked up
// Idea: Solve for increasing inputs and record results in an array S

S[0] ← 0;  S[1] ← C[1]
for i ← 2 to n do
        S[i] ← max(C[i] + S[i-2], S[i-1])
return S[n]
```

- Time efficiency: O(n)

Kadane’s Algorithm

Problem Statement:

Given an array of integers (which may include negative numbers), find the contiguous subarray with the maximum sum.

Example Input:
arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]

✅ **Maximum subarray sum is 6**, from subarray [4, -1, 2, 1]

Kadane’s Algorithm

Example Input:

arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]

Step-by-Step Execution:

✅ **Maximum subarray sum is 6**, from subarray [4, -1, 2, 1]

Index i	0	1	2	3	4	5	6	7	8
Value	-2	1	-3	4	-1	2	1	-5	4
Solution S	-2	1	-2	4	3	5	6	1	5

At each step take the max(value(i), value(i)+solution(i-1))

Kadane’s Algorithm

Kadane’s Algorithm:

1. Dynamic programming table:
   - Initialize a table, T to hold the current max sum at index place i
   - T[i] = holds the max sum up to i.
   - T[0]=array[0]

2.Iterate through the array 1..n-1:
   - Check if adding the current value to the running sum is greater than the current value.
   - If yes, add the current value to the sum and store in T[i].
   - Otherwise start over with a new subarray sum. T[i]=array[i]
   - The result is the max value stored in the table.

Time Complexity:

- O(n) (single pass through the array, and single pass through the table to extract solution)