Chapter 5- Dynamic Programming

2021

Prepared by: Beimnet G.

Dynamic Programming

- Divide and conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
- But sometimes, these subproblems overlap and we find that divide and conquer does more work than necessary, repeatedly solving the subproblems.

Dynamic Programming

- This is where dynamic programming comes in.
- A dynamic-programming algorithm solves each subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subproblem.

Dynamic Programming: History

Coined by Richard Bellman.

Programming means optimization in the mathematical world.

And **Dynamic** because…. it sounded cool.

A more descriptive term would be a lookup table.

Dynamic Programming: Paradigm

In a problem, when trying to solve it, I end up asking the same question over and over again trying to find the solution. So rather than solving these questions over and over, let me remember the answer to a question after solving for it once and next time this question comes up, I'll just look up the solution to it.

Dynamic Programming

Basic Idea: split into subproblems, solve, store solutions, reuse solutions

DP = recursion + reuse (memoization)

Dynamic Programming: Fibonacci Series

Series: 1, 1, 2, 3, 5, 8 .....

$$F_n = F_{n-1} + F_{n-2}$$

Goal: $F_n = ?$

Dynamic Programming: Fibonacci Series

Naive Approach:

```
Fib (n)
{
    if n <= 2{
        f=1
    }
    else{
        f= Fib (n-1) +Fib (n-2)
    }
    return f;
}
```

Runtime ?

$T(n)= T(n-1) + T(n-2) + \Theta(1)$

Exponential time => bad

Can we make this better?

Dynamic Programming: Fibonacci Series

How can we make it better?

Memoization!

Memoization: remember solutions to already solved subproblems and use those solutions instead of solving them again.

T(n)= # of subproblems + time it takes to solve a subproblem

Dynamic Programming: Fibonacci Series

```
memo=[    ]
Fib (n){
    if n in memo
        return memo[n]
    if n <= 2
        f=1
    else
        f= Fib (n-1) +Fib (n-2)
    memo[n]=f
    return f;
}
```

Memoized DP Algorithm

$T(n)= \Theta (n)$

Cost only comes from the non memoized recursion calls.

=>Top Down Approach

Dynamic Programming: Fibonacci Series

```
r=[]
Fib (n){
    for k=1 to n
        if k<= 2
            f=1
        else
            f=r[k-1]+r[k-2]
        r[k]=f
    return r[n]
}
```

Bottom Up Approach

$T(n)= \Theta (n)$

Does a topological sort of the subproblems.

Dynamic Programming

When developing a dynamic-programming algorithm, we follow 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, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Dynamic Programming: Rod Cutting

Given a rod of length n and a table of prices p_i for i= 1, 2, 3, … n for a rods of length i, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

Note: that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

Dynamic Programming: Rod Cutting

In how many different ways can you cut a rod of length n?

If an optimal solution cuts a rod into k pieces for some k, $1 \leq k \leq n$ then,

$n = i_1 + i_2 + i_3 + \ldots i_k$ i.e it's be cut into k pieces of lengths $i_1, i_2, i_3 \ldots i_k$

This provides a maximum revenue $r_n = p_{i1} + p_{i2} + p_{i3} + \ldots + p_{ik}$

Dynamic Programming: Rod Cutting

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Dynamic Programming: Rod Cutting

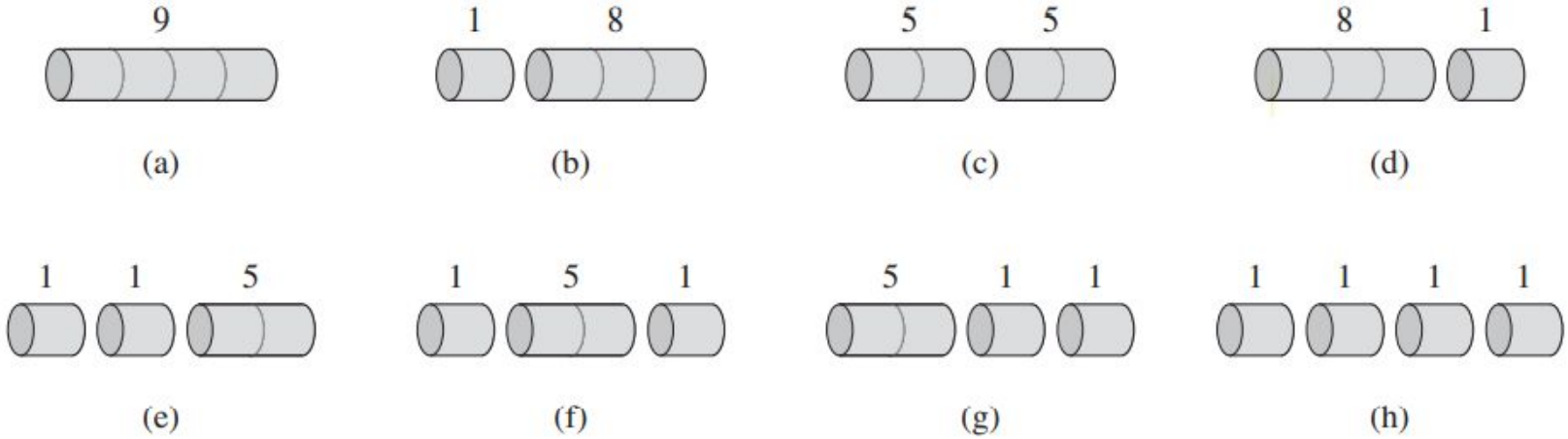


Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

Dynamic Programming: Rod Cutting

```
given p[] of length n
CUT-ROD(n){
    if n==0
        return 0
    q=-∞
    for i=1 to n
        q= max (q, p[i] + CUT-ROD(n-i))
    return q
}
```

Naive Approach

$T(n)= \Theta(2^n)$

Dynamic Programming: Rod Cutting

```
given p[] of length n
memo=[ ]  //an empty array
CUT-ROD(n){
    if memo[n] ≠ null
        return memo[n]
    if n==0
        q= 0
    else
        q=-∞
        for i=1 to n
            q= max (q, p[i] + CUT-ROD(n-i))
    memo[n]=q
    return q;
}
```

Memoized Algorithm

$T(n)= \Theta(n^2)$

Dynamic Programming: Rod Cutting

```
give p[] of length n
CUT-ROD(n){
    r=[]
    r[0]=0
    for j=1 to n
        q=-∞
        for i=1 to j
            q= max(q, p[i]+ r[j-1])
        r[j]=q
    return r[n]
}
```

Bottom Up Approach

$T(n)= \Theta(n^2)$

Dynamic Programming: 1/0 KnapSack

Given weights and values of n items, put these items in a knapsack of capacity C to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively and an integer C which represents knapsack capacity, **find out the maximum value** (subset of val[]) **such that sum of the weights of this subset is smaller than or equal to C**. You cannot break an item, either pick the complete item or don’t pick it (0-1 property).

Dynamic Programming: 1/0 KnapSack

Brute force approach: consider all subsets of items and calculate the total weight and value of all subsets. From all such subsets, pick the maximum value subset.

Dynamic Programming: 1/0 KnapSack

Optimal Substructure: There can only be two cases for every item.

Case 1: The item is included in the optimal subset.

Case 2: The item is not included in the optimal set.

Therefore, the decision made when considering the n^{th} item is based on the max of the following two values.

1. Maximum value obtained by n-1 items and W weight (excluding nth item).
2. Value of nth item plus maximum value obtained by n-1 items and C minus the weight of the nth item (including nth item).

If the weight of ‘nth’ item is greater than ‘C’, then the nth item cannot be included and Case 1 is the only possibility.

Dynamic Programming: 1/0 KnapSack

```
wt=[wt1,wt2,wt3...]
val=[v1,v2,v3...]
KnapSack(C, n ){ // C- max capacity of KnapSack, n- no of items
    if(n==0 || C==0){
        return 0;
    }
    if (wt[n]> C){
        return KnapSack(C, n-1)
    }
    else{
        vin= val[n] + KnapSack(C- wt[n], n-1)
        vex= KnapSack(C, n-1)
        return max(vin,vex)
    }
}
```

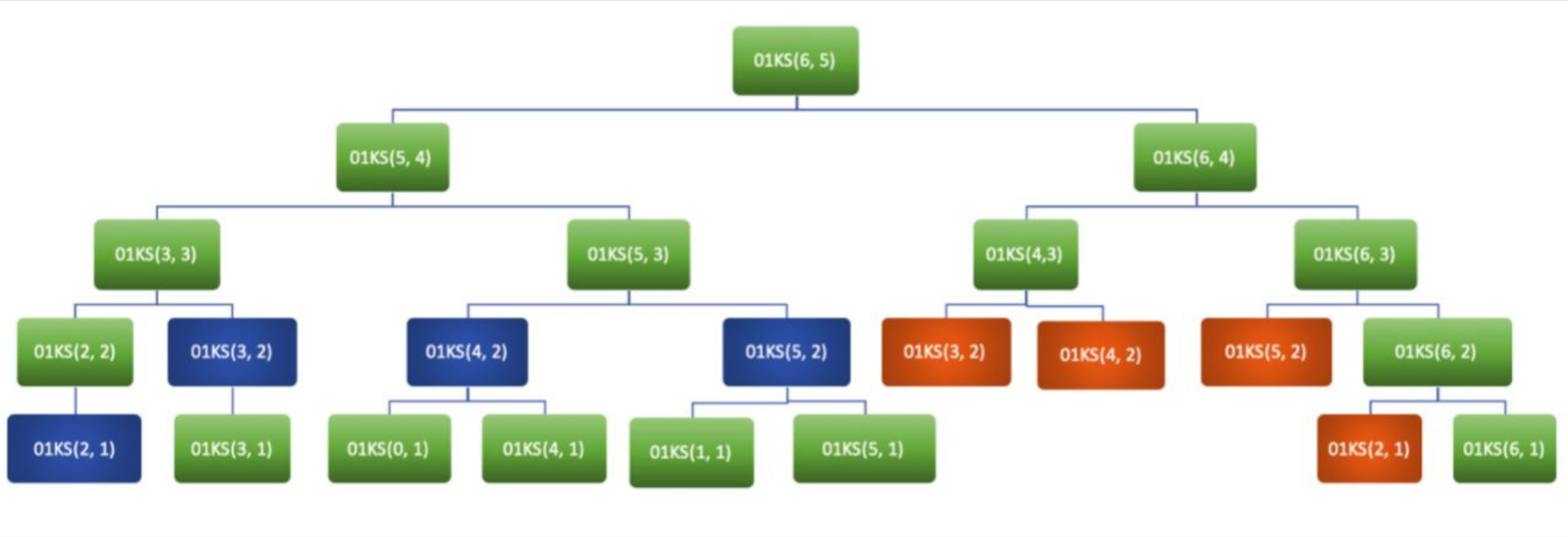
Brute Force Approach

T(n) -> exponential

What are the overlapping problems?

How can we optimize run time?

Dynamic Programming: 1/0 KnapSack



Dynamic Programming: 1/0 KnapSack

```
wt=[wt1,wt2,wt3...]
val=[v1,v2,v3...]

memo[x][y]=[-1]
KnapSack(C, n ){ // C- max capacity of KnapSack
    if(n==0 || C==0){
        return 0;
    }
    if(memo[n][C]>-1){
        return memo[n][C]
    }
    if (wt[n]> C){
        //return KnapSack(C, n-1)
        memo[n][C]=KnapSack(C, n-1)
    }
    else{
        vin= val[n] + KnapSack(C- wt[n], n-1)
        vex= KnapSack(C, n-1)
        //return max(vin,vex)
        memo[n][C]=max(vin,vex)
    }
}
```

Memoized Top Down recursion

T(n) -> O(n*C)

Is this polynomial?

More Dynamic Problems

1. Ugly numbers
2. Coin change problem
3. Longest common substring
4. Shortest Paths