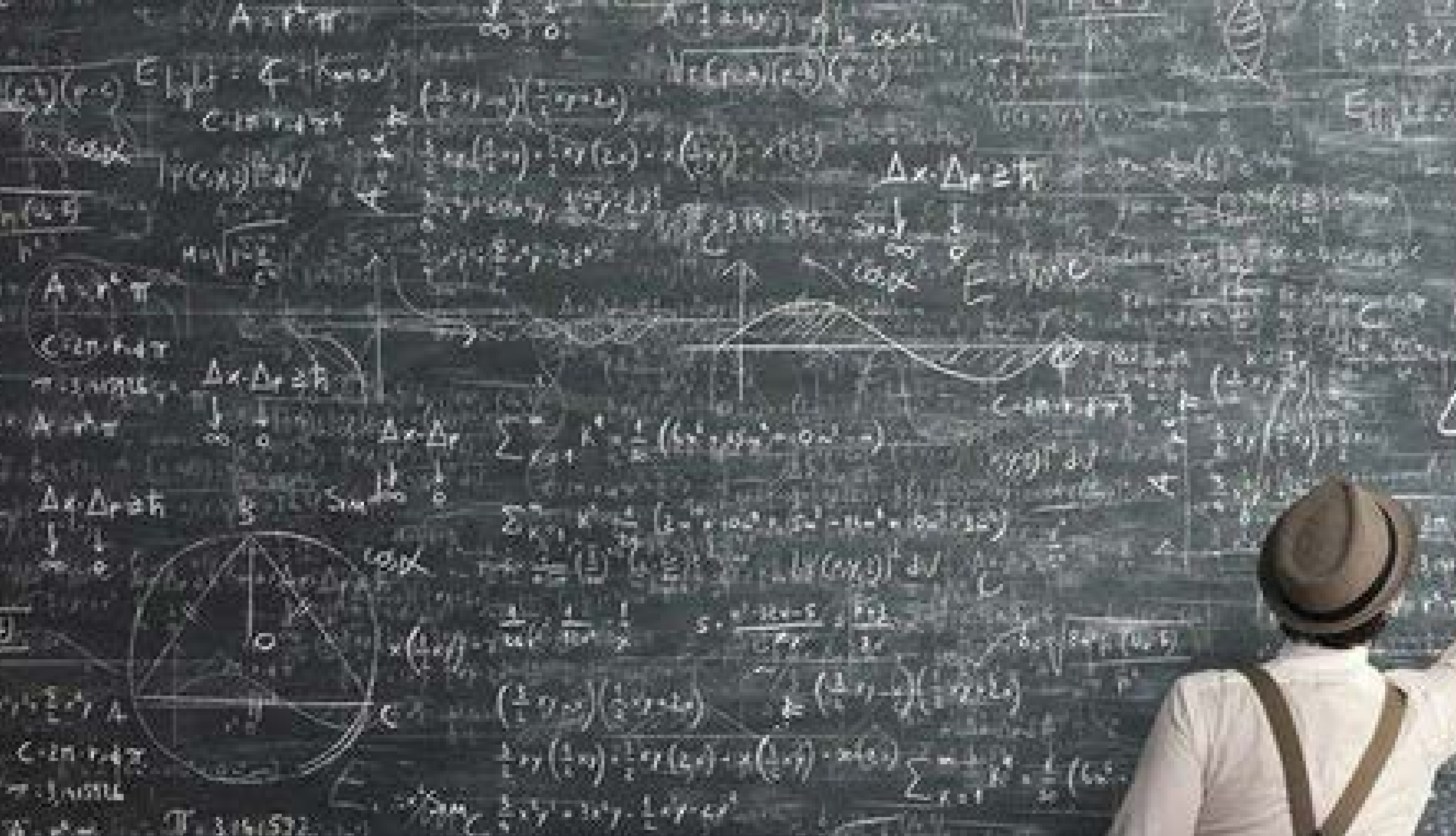


Dynamic programming

Ja-Hee Kim

Agenda

Introduction

Techniques

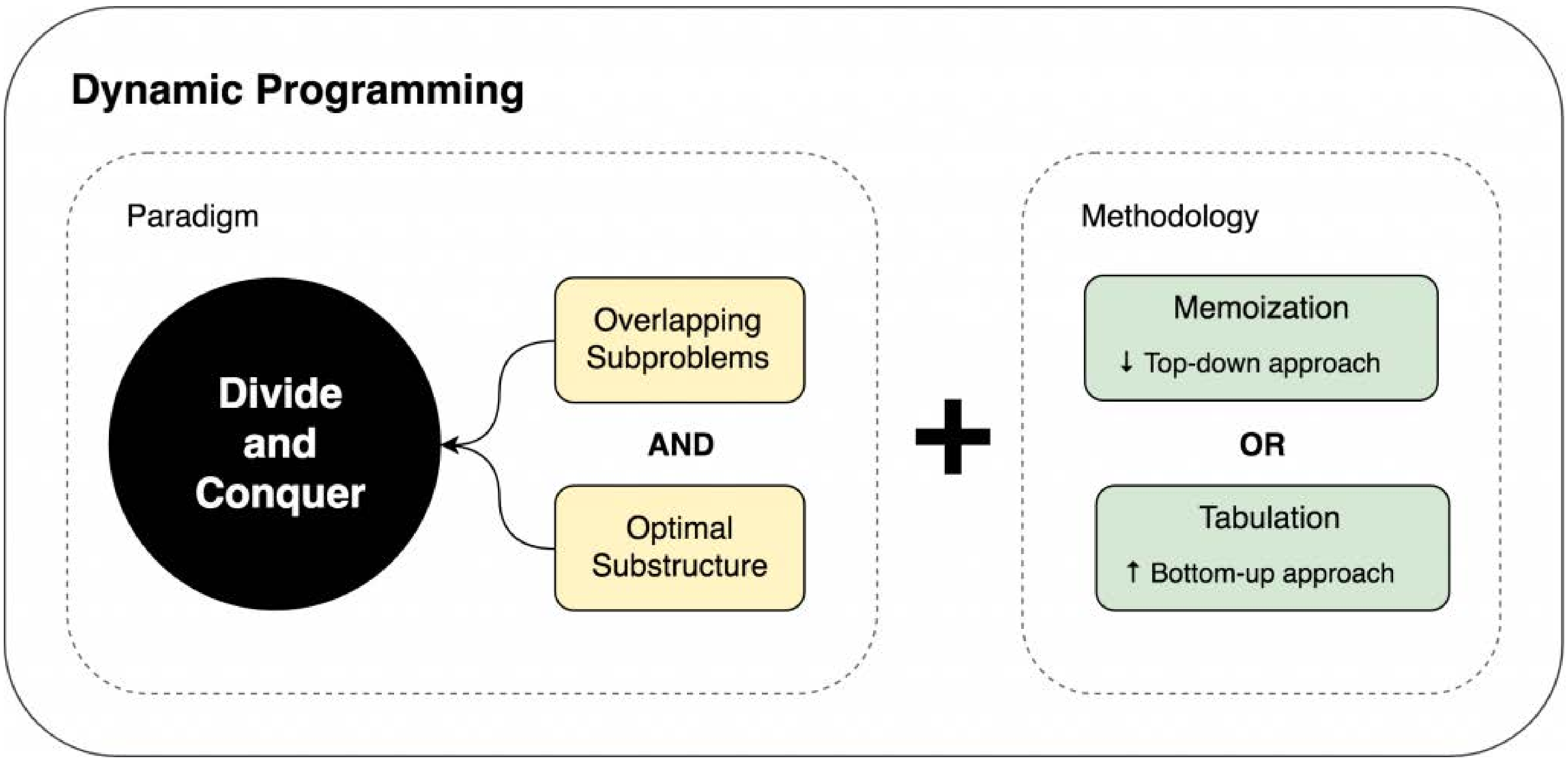
Examples

Introduction

Dynamic programming

- A mathematical optimization method
- A computer programming method.
- solving a complex problem by **breaking it down** into a collection of simpler sub-problems
- solving each of those sub-problems just once, and **storing their solutions** using a memory-based data structure
- Dynamic programming paradigm is similar to divide and conquer paradigm but it avoids recursion.

D&Q vs DP


Dynamic Programming
Paradigm
Divide and Conquer
Overlapping Subproblems
AND
Optimal Substructure
+
Methodology
Memoization
↓ Top-down approach
OR
Tabulation
↑ Bottom-up approach

Prerequisites

- In order that the dynamic programming paradigm can be applicable, a divide and conquer problem should have both of the following attributes:
  - **Overlapping sub-problems**
    - Found solutions of sub-problems involves solving the same sub-problem multiple times.
    - Binary search vs Fibonacci numbers
  - **Optimal substructures**
    - its overall optimal solution can be constructed from the optimal solutions of its sub-problems.
    - https://youtu.be/JWTqsNvtwP4

Techniques

Tabulation vs Memoization

- Two patterns
  - Tabulation
    - Bottom Up
    - Base case → n
  - Memoization
    - Top Down
    - speed up computer programs by storing the results of expensive function calls and returning the cached result

Bottom up approach

```
public static long bottomUp(int n) {
	for (int i = lastFibIndex+1; i <= n; i++)
		fib[i] = fib[i-1] + fib[i-2];
	if (n > lastFibIndex) lastFibIndex = n;
	return fib[n];
}
```

Top down approach

```
public static long topDown(int n) {
	if (n < lastFibIndex) return fib[n];
	fib[n]=topDown(n-2)+topDown(n-1);
	lastFibIndex =n;
	return fib[n];
}
```

Tabulation vs Memoization

	Tabulation	Memoization
State	State Transition relation is difficult to think	State transition relation is easy to think
Code	Code gets complicated when lot of conditions are required	Code is easy and less complicated
Speed	Fast, as we directly access previous states from the table	Slow due to lot of recursive calls and return statements
Subproblem solving	If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor	If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required
Table Entries	In Tabulated version, starting from the first entry, all entries are filled one by one	Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. The table is filled on demand.

Steps to solve a DP

1. Identify if it is a DP problem
2. Decide a state expression with least parameters
3. Formulate state relationship
4. Do tabulation (or add memoization)

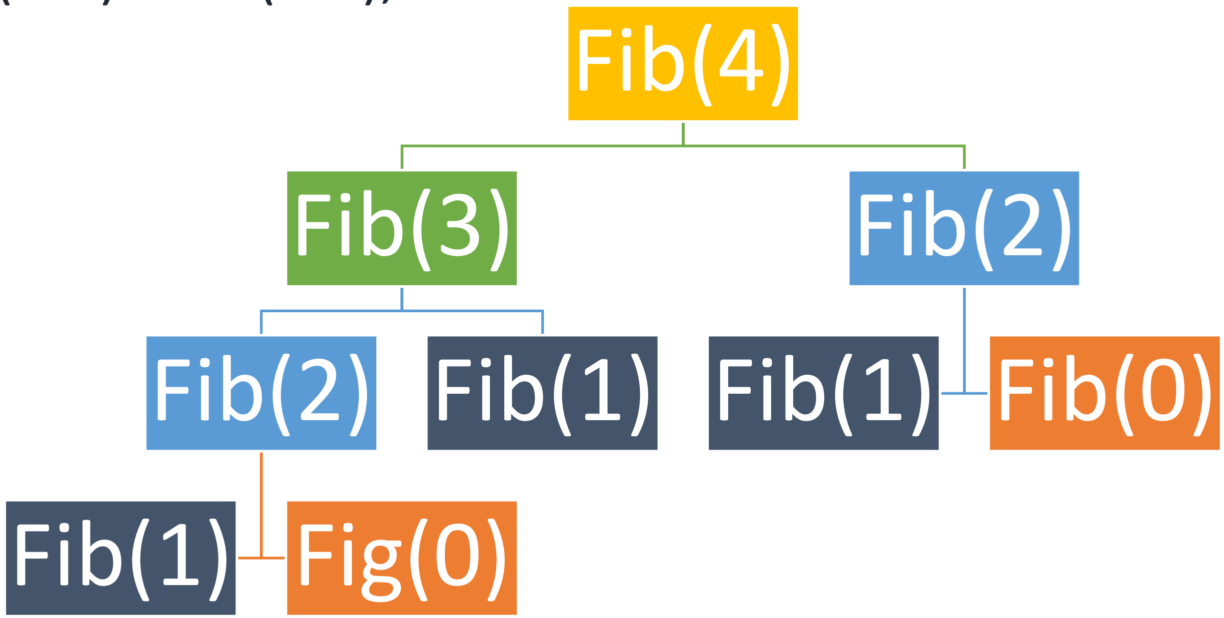
Example: Fibonacci number

$$Fib(n) = Fib(n-1) + Fib(n-2), \text{ for } n > 1$$

Step1: classify a problem

- Optimization
  - Minimize or maximize certain quantity
- Counting problem
  - count the arrangements under certain condition
- the overlapping sub-problems property

Fib(n) = Fib(n-1) + Fib(n-2), for n > 1



Step2: decide the state

- Decide states and their transitions
  - State:
    - the set parameters identified uniquely
    - As small as possible
  - Transition
    - It causes state changes
    - It usually means your choice.

- Fib(n) = Fib(n-1) + Fib(n-2), for n > 1
  - State: n
  - Transition:
    - n-> n-1 and n-2

Step3: Formulating a relation among the states

- Hardest part
- For example, formulating mathematics induction

- Fib(n) = Fib(n-1) + Fib(n-2), for n > 1
  - the expression itself

Step4: bottom up or top down

- Declare an array for tabulation or memoization
- Another way is to add tabulation and make solution iterative.

```
public static long topDown(int n) {
    if (n < lastFibIndex) return fib[n];
    fib[n]=topDown(n-2)+topDown(n-1);
    lastFibIndex =n;
    return fib[n];
}
```

```
public static long bottomUp(int n) {
    for (int i = lastFibIndex+1; i <= n; i++)
        fib[n]= fib[n-1] + fib[n-2];
    if (n > lastFibIndex) lastFibIndex = n;
    return fib[n];
}
```

```
public static long iteration(int n) {
        if (n<2) return n;
        long f0=0, f1=1, f2=1;
        for (int i=2; i<n; i++)
                f0 = f1;    f1 = f2;    f2 = f1 + f0;
        }
        return f2;
}
```

Examples

Fibonacci numbers
Shortest path: Floyd Warshall algorithm

Subset sum problem
0-1 knapsack problem

Subset sum

- Given a set of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to given sum.

- Step1: Counting problem(count the arrangements under certain condition)
- Step2:
  - State: a subset of something.
  - Transition: an element is included in addition

Subset sum

- Step3:
  - Base case:
    - sum = 0: return true
    - n = 0 but sum!= 0: return false
  - Set[n-1] > sum:
    - Ignore
    - return isSubsetSum(set, n-1, sum);
  - isSubsetSum(set, n, sum) =

```
isSubsetSum(set, n-1, sum) ||          // element is not included
isSubsetSum(set, n-1, sum-set[n-1])//element is included
```

Subset sum

- Step4:tabulation

sum

elements	0	1	2	3	4	5	6	7	8	9
0										
4										
12										
5										
2										

Subset sum

- set[] = {4, 12, 5, 2}, sum = 9
  - Base case:
    - sum = 0: return true
    - n = 0 but sum!= 0: return false
  - Set[n-1] > sum:
    - Ignore
    - return isSubsetSum(set, n-1, sum);
  - isSubsetSum(set, n, sum) =

    isSubsetSum(set, n-1, sum) || // element is not included

    isSubsetSum(set, n-1, sum-set[n-1])//element is included

	0	1	2	3	4	5	6	7	8	9
0	true	false	false	false	false	false	false	false	false	false
4	true	false	false	false	true	false	false	false	false	false
12	true	false	false	false	true	false	false	false	false	false
5	true	false	false	false	true	true	false	false	false	true
2	true	false	true	false	true	true	true	true	false	true

Knapsack problem

- Given a knapsack weight W and a set of n items with certain value v_i and weight w_i, we need to pack items whose sum of weights is less than W and whose sum of values is maximum. We allow to use unlimited number of instances of an item.’
- Step1: Optimization(Minimize or maximize certain quantity)
- Step2:
  - State: the value of selected items
  - Transition: change the number of items

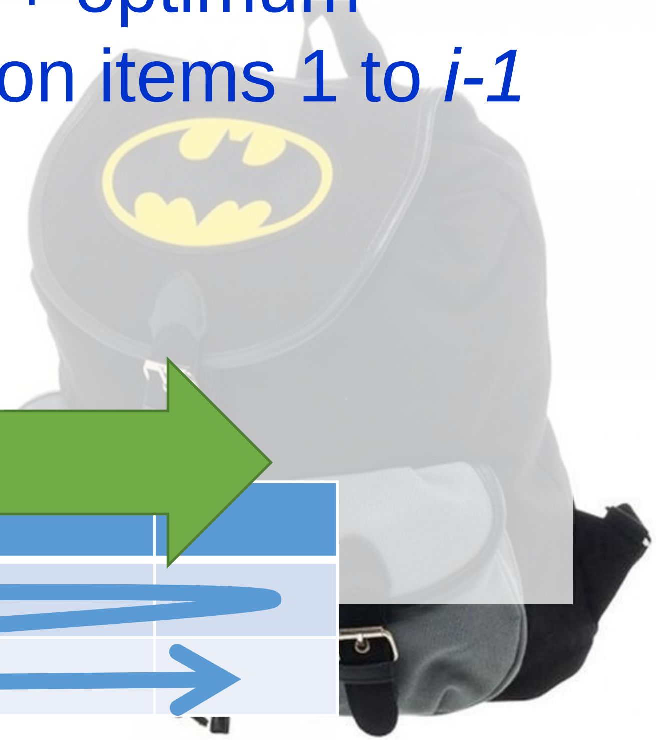
Knapsack problem

- Step3: Formula
  - **f(i,w)=Max[vi + f(i,w-wi) ; f(i-1,w)]**

ONE Item *i* + optimum combination of weight *w-wi*

NO Item *i* + optimum combination items 1 to *i-1*

- Step4: Tabulation



Knapsack problem

- Optimum output of a combination of items 1 to *i* with a cumulated weight of *w* or less.

- knapsack – 10kg capacity
- Item 1: $5 (3kg)
- Item 2: $7 (4kg)
- Item 3: $8 (5kg)

Knapsack Problem

	1	2	3	4	5	6	7	8	9	10
1										
2										
3										

w

i

f(i,w)

Knapsack Problem

Table

	1	2	3	4	5	6	7	8	9	10
1										
2										
3										

Using only item 1

W

i

Knapsack Problem

Table

	1	2	3	4	5	6	7	8	9	10
1										
2										
3										

Using only item 1&2

W

i

Knapsack Problem

Table

	1	2	3	4	5	6	7	8	9	10
1										
2										
3	**Using only item 1,2 & 3**									

W

i

Knapsack Problem

2 items n°1

2 w1 = 6

Table

	1	2	3	4	5	6	7	8	9	10
1	0	0	5	5	5	10	10	10	15	
2										
3										

0 items n°1

1 items n°1

w1 = 3

- Item 1: v1=$5 ; w1=3kg
- Item 2: v2=$7 ; w2=4kg
- Item 3: v3=$8 ; w3=5kg

Knapsack Problem

- Item 1: v1=$5 ; w1=3kg
- Item 2: v2=$7 ; w2=4kg
- Item 3: v3=$8 ; w3=5kg

Table

	1	2	3	4	5	6	7	8	9	10
1	0	0	5	5	5	10	10	10	15	15
2	0	0	5	7	7					
3										

+ x2 (= 7)

f(i,w)=Max[xi + f(i,w-wi) ; f(i-1,w)]

Knapsack Problem

- Item 1: v1=$5 ; w1=3kg
- Item 2: v2=$7 ; w2=4kg
- Item 3: v3=$8 ; w3=5kg

Table

	1	2	3	4	5	6	7	8	9	10
1	0	0	5	5	5	10	10	10	15	15
2	0	0	5	7	7	10				
3										

7

14

$$f(i,w)=Max[\ xi + \cancel{f(i-1,w-wi)}\ ;\ f(i-1,w)\ ]$$

Knapsack Problem

COMPLETED TABLE

	1	2	3	4	5	6	7	8	9	10
1	0	0	5	5	5	10	10	10	15	15
2	0	0	5	7	7	10	12	14	15	17
3	0	0	5	7	8	10	12	14	15	17

Thanks