Dynamic Programming (1)

By: Aminul Islam

Based on Chapter 3 of Foundations of Algorithms

Objectives

Objectives

- Describe the Dynamic Programming Technique
- Contrast the Divide and Conquer and Dynamic Programming approaches to solving problems
- Identify when dynamic programming should be used to solve a problem

Dynamic Programming

Dynamic Programming

- Dynamic Programming is an algorithm design technique to solve the recursive problems in more efficient manner

Dynamic Programming

- Dynamic Programming is an algorithm design technique to solve the recursive problems in more efficient manner
- Like divide and conquer, DP solves problems by combining solutions to subproblems

Dynamic Programming

- Dynamic Programming is an algorithm design technique to solve the recursive problems in more efficient manner
- Like divide and conquer, DP solves problems by combining solutions to subproblems
- Unlike divide and conquer, subproblems are not independent (overlapping subproblems)

Dynamic Programming

- Dynamic Programming is an algorithm design technique to solve the recursive problems in more efficient manner
- Like divide and conquer, DP solves problems by combining solutions to subproblems
- Unlike divide and conquer, subproblems are not independent (overlapping subproblems)
- Let's start with a simple example

Fibonacci Numbers

Fibonacci Numbers

- Recall what it was . . .

Fibonacci Numbers

- Recall what it was . . .
  - Fib(0)=0
  - Fib(1)=1

Fibonacci Numbers

- Recall what it was . . .
  - Fib(0)=0
  - Fib(1)=1
  - Fib(n)=Fib($n-1$)+ Fib($n-2$)

Fibonacci Numbers

- Recall what it was ...
  - Fib(0)=0
  - Fib(1)=1
  - Fib(n)=Fib($n-1$)+ Fib($n-2$)
- How to obtain Fib(n)?

Fibonacci Numbers

- Recall what it was ...
  - Fib(0)=0
  - Fib(1)=1
  - Fib(n)=Fib($n-1$)+ Fib($n-2$)
- How to obtain Fib(n)?
  - Recall the recursive algorithm

Fibonacci (Recursive)

Fibonacci (Recursive)

```
int Fib (int n) {
        if (n==0)
            return 0;
        if (n==1)
            return 1;
        return Fib(n-1)+Fib(n-2);
}
```

Fibonacci (Recursive)

```
int Fib (int n) {
        if (n==0)
            return 0;
        if (n==1)
            return 1;
        return Fib(n-1)+Fib(n-2);
}
```

- This recursive version is an example of

Fibonacci (Recursive)

```
int Fib (int n) {
        if (n==0)
            return 0;
        if (n==1)
            return 1;
        return Fib(n-1)+Fib(n-2);
}
```

- This recursive version is an example of D&C approach

Fibonacci (Recursive)

```
int Fib (int n) {
        if (n==0)
            return 0;
        if (n==1)
            return 1;
        return Fib(n-1)+Fib(n-2);
}
```

- This recursive version is an example of D&C approach
- Why D&C is not a good approach for solving Fibonacci?

Fibonacci (Recursive)

```
int Fib (int n) {
        if (n==0)
            return 0;
        if (n==1)
            return 1;
        return Fib(n-1)+Fib(n-2);
}
```

- This recursive version is an example of D&C approach
- Why D&C is not a good approach for solving Fibonacci?
- Let's see how is the dynamic version

Fibonacci (Using DP)

Fibonacci (Using DP)

```
int fibDP(int n) {
        int memo[] = new int[n+1];
        memo[0]=0;
        memo[1]=1;
        for (int i=2; i<=n; i++) {
                memo[i]=memo[i-1]+memo[i-2];
        }
        return memo[n];
}
```

Fibonacci (Using DP)

```
int fibDP(int n) {
        int memo[] = new int[n+1];
        memo[0]=0;
        memo[1]=1;
        for (int i=2; i<=n; i++) {
                memo[i]=memo[i-1]+memo[i-2];
        }
        return memo[n];
}
```

- Order of complexity (when there is no pre-computed result in memory)

Fibonacci (Using DP)

```
int fibDP(int n) {
        int memo[] = new int[n+1];
        memo[0]=0;
        memo[1]=1;
        for (int i=2; i<=n; i++) {
                memo[i]=memo[i-1]+memo[i-2];
        }
        return memo[n];
}
```

- Order of complexity (when there is no pre-computed result in memory) $\Theta(n)$

Fibonacci (Using DP)

```
int fibDP(int n) {
        int memo[] = new int[n+1];
        memo[0]=0;
        memo[1]=1;
        for (int i=2; i<=n; i++) {
                memo[i]=memo[i-1]+memo[i-2];
        }
        return memo[n];
}
```

- Order of complexity (when there is no pre-computed result in memory) $\Theta(n)$
- Order of complexity (when there is pre-computed result in memory)

Fibonacci (Using DP)

```
int fibDP(int n) {
        int memo[] = new int[n+1];
        memo[0]=0;
        memo[1]=1;
        for (int i=2; i<=n; i++) {
                memo[i]=memo[i-1]+memo[i-2];
        }
        return memo[n];
}
```

- Order of complexity (when there is no pre-computed result in memory) $\Theta(n)$
- Order of complexity (when there is pre-computed result in memory) $\Theta(1)$

Lessons Learnt about DP

Lessons Learnt about DP

- The key is to remember what has been computed so far

Lessons Learnt about DP

- The key is to remember what has been computed so far
  - This is called “memoization”

Lessons Learnt about DP

- The key is to remember what has been computed so far
  - This is called "memoization"
  - This is in fact "reusing" computation

Lessons Learnt about DP

- The key is to remember what has been computed so far
  - This is called “memoization”
  - This is in fact “reusing” computation
- Identify what are the subproblems

Lessons Learnt about DP

- The key is to remember what has been computed so far
  - This is called "memoization"
  - This is in fact "reusing" computation
- Identify what are the subproblems
  - They help to solve the actual problem instance

Lessons Learnt about DP

- The key is to remember what has been computed so far
  - This is called "memoization"
  - This is in fact "reusing" computation
- Identify what are the subproblems
  - They help to solve the actual problem instance
  - Save the result obtained from subproblems (memoize)

DP reduces computation by

DP reduces computation by

- Solving subproblems in a bottom-up fashion.

DP reduces computation by

- Solving subproblems in a bottom-up fashion.
- Storing solution to a subproblem the first time it is solved

DP reduces computation by

- Solving subproblems in a bottom-up fashion.
- Storing solution to a subproblem the first time it is solved
- Looking up the solution when subproblem is encountered again

Dynamic Programming

Dynamic Programming

- We need to find a recursive formula for the solution

Dynamic Programming

- We need to find a recursive formula for the solution
- We can recursively solve subproblems, starting from the trivial case, and save their solutions in memory

Dynamic Programming

- We need to find a recursive formula for the solution
- We can recursively solve subproblems, starting from the trivial case, and save their solutions in memory
- In the end we'll get the solution of the whole problem

Bionomial Coefficient Problem

Bionomial Coefficient Problem

$\binom{n}{k} = \frac{n!}{k!(n-k)!}$ where $0 \leq k \leq n$

Bionomial Coefficient Problem

$\binom{n}{k} = \frac{n!}{k!(n-k)!}$ where $0 \leq k \leq n$

- This is difficult to calculate for large values of n

Bionomial Coefficient Problem

$\binom{n}{k} = \frac{n!}{k!(n-k)!}$ where $0 \leq k \leq n$

- This is difficult to calculate for large values of n
- We can eliminate the use of $n!$ using the following formula:

Bionomial Coefficient Problem

$\binom{n}{k} = \frac{n!}{k!(n-k)!}$ where $0 \leq k \leq n$

- This is difficult to calculate for large values of n
- We can eliminate the use of $n!$ using the following formula:

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ 1 & \text{if } k = 0 \text{ or } k = n \end{cases}$$

Bionomial Coefficient using D&C

Bionomial Coefficient using D&C

```
Problem:  Compute the binomial coefficient
Inputs:  nonnegative integers n and k, where k ≤ n
Outpts:  bin, the binomial coefficient (n choose k)
int bin (int n, int k)
{
    if (k == 0 || n == k)
     return 1;
    else
     return bin(n − 1, k − 1) + bin(n − 1, k);
}
```

Bionomial Coefficient using D&C

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ 1 & \text{if } k = 0 \text{ or } k = n \end{cases}$$

```
Problem:  Compute the binomial coefficient
Inputs:  nonnegative integers n and k, where k ≤ n
Outpts:  bin, the binomial coefficient (n k)
int bin (int n, int k)
{
   if (k == 0||n == k)
    return 1;
   else
    return bin(n − 1, k − 1) + bin(n − 1, k);
}
```

Bionomial Coefficient using D&C

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ 1 & \text{if } k = 0 \text{ or } k = n \end{cases}$$

```
Problem:  Compute the binomial coefficient
Inputs:  nonnegative integers n and k, where k ≤ n
Outpts:  bin, the binomial coefficient (n k)
int bin (int n, int k)
{
    if (k == 0 || n == k)
     return 1;
    else
     return bin(n − 1, k − 1) + bin(n − 1, k);
}
```

- What's the time complexity and order of complexity?

Bionomial Coefficient using D&C

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ 1 & \text{if } k = 0 \text{ or } k = n \end{cases}$$

Problem: Compute the binomial coefficient
Inputs: nonnegative integers n and k, where $k \leq n$
Outpts: bin, the binomial coefficient $\binom{n}{k}$

```
int bin (int n, int k)
{
    if (k == 0||n == k)
     return 1;
    else
     return bin(n − 1, k − 1) + bin(n − 1, k);
}
```

- What's the time complexity and order of complexity? $T(n) \approx 2T(n-1)$ and $T(n) = \Theta(2^n)$

Bionomial Coefficient using D&C

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ 1 & \text{if } k = 0 \text{ or } k = n \end{cases}$$

Problem: Compute the binomial coefficient

Inputs: nonnegative integers n and k, where $k \leq n$

Outpts: bin, the binomial coefficient $\binom{n}{k}$

```
int bin (int n, int k)
{
    if (k == 0 || n == k)
     return 1;
    else
     return bin(n - 1, k - 1) + bin(n - 1, k);
}
```

- What's the time complexity and order of complexity? $T(n) \approx 2T(n-1)$ and $T(n) = \Theta(2^n)$
- D&C approach is not efficient. Why?

Bionomial Coefficient using D&C

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ 1 & \text{if } k = 0 \text{ or } k = n \end{cases}$$

```
Problem:  Compute the binomial coefficient
Inputs:  nonnegative integers n and k, where k ≤ n
Outpts:  bin, the binomial coefficient (n k)
int bin (int n, int k)
{
   if (k == 0||n == k)
    return 1;
   else
    return bin(n − 1, k − 1) + bin(n − 1, k);
}
```

- What's the time complexity and order of complexity? $T(n) \approx 2T(n-1)$ and $T(n) = \Theta(2^n)$
- D&C approach is not efficient. Why?
- What's the time complexity and order of complexity for $T(n) = 7T(\frac{n}{2})$, where $T(1) = 1$?

Bionomial Coefficient using D&C

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ 1 & \text{if } k = 0 \text{ or } k = n \end{cases}$$

Problem: Compute the binomial coefficient
Inputs: nonnegative integers n and k, where $k \leq n$
Outpts: bin, the binomial coefficient $\binom{n}{k}$

```
int bin (int n, int k)
{
    if (k == 0||n == k)
     return 1;
    else
     return bin(n − 1, k − 1) + bin(n − 1, k);
}
```

- What's the time complexity and order of complexity? $T(n) \approx 2T(n-1)$ and $T(n) = \Theta(2^n)$
- D&C approach is not efficient. Why?
- What's the time complexity and order of complexity for $T(n) = 7T(\frac{n}{2})$, where $T(1) = 1$? $T(n) = n^{\log_2 7} = n^{2.8}$