ساختمان داده و الگوریتم ها

مبحث هجدهم: برنامه نویسی پویا

سجاد شیرعلی شهرضا

پاییز 1402

شنبه، 25 آذر 1402

Based on Ohio State University CSE680 Slides from Autumn 2011 by Roger Crawfis

اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 15

مقدمه ای بر برنامه نویسی پویا

یک روش طراحی الگوریتم

DYNAMIC PROGRAMMING

Dynamic programming (DP) is an algorithm design paradigm.
It’s often used to solve optimization problems (e.g. *shortest* path).

Fibonacci Number

- $F(n) = F(n-1) + F(n-2)$
- $F(0) = 0$
- $F(1) = 1$
- Other values:
- $F(2) = 1$
- $F(3) = 2$
- $F(4) = 3$
- $F(5) = 5$
- $F(6) = 8$

DYNAMIC PROGRAMMING

Dynamic programming (DP) is an algorithm design paradigm.
It's often used to solve optimization problems (e.g. *shortest* path).

We'll see two examples of DP today:
Fibonacci and Matrix Multiplication.
We will go over some DP practice problems in depth next time.

But first, an overview of DP!

DYNAMIC PROGRAMMING

Elements of dynamic programming:

DYNAMIC PROGRAMMING

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.

DYNAMIC PROGRAMMING

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.

e.g. $\mathbf{f_n = f_{n-1} + f_{n-2}}$

DYNAMIC PROGRAMMING

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.

e.g. $\mathbf{f_n = f_{n-1} + f_{n-2}}$

Overlapping sub-problems: the subproblems overlap a lot!
This means we can save time by solving a sub-problem once & cache the answer.
(this is sometimes called “memoization”)

DYNAMIC PROGRAMMING

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.

e.g. $f_n = f_{n-1} + f_{n-2}$

Overlapping sub-problems: the subproblems overlap a lot!
This means we can save time by solving a sub-problem once & cache the answer.
(this is sometimes called "memoization")

e.g. **Use value of f_{n-k} multiple times while calculating f_n**

DYNAMIC PROGRAMMING

Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

DYNAMIC PROGRAMMING

Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

Bottom-up: iterates through problems by size and solves the small problems first (kind of like taking care of base cases first & building up).

DYNAMIC PROGRAMMING

Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

Bottom-up: iterates through problems by size and solves the small problems first (kind of like taking care of base cases first & building up).

e.g. compute $\mathbf{f_2 = f_1 + f_0}$, then $\mathbf{f_3 = f_2 + f_1}$, then $\mathbf{f_4 = f_3 + f_2}$, …, and finally $\mathbf{f_n = f_{n-1} + f_{n-2}}$

DYNAMIC PROGRAMMING

Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

Bottom-up: iterates through problems by size and solves the small problems first (kind of like taking care of base cases first & building up).
e.g. compute $\mathbf{f_2 = f_1 + f_0}$, then $\mathbf{f_3 = f_2 + f_1}$, then $\mathbf{f_4 = f_3 + f_2}$, …, and finally $\mathbf{f_n = f_{n-1} + f_{n-2}}$

Top-down: instead uses recursive calls to solve smaller problems, while using memoization/caching to keep track of small problems that you've already computed answers for (simply fetch the answer instead of re-solving that problem and waste computational effort)

DYNAMIC PROGRAMMING

Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

Bottom-up: iterates through problems by size and solves the small problems first (kind of like taking care of base cases first & building up).

e.g. compute $\mathbf{f_2 = f_1 + f_0}$, then $\mathbf{f_3 = f_2 + f_1}$, then $\mathbf{f_4 = f_3 + f_2}$, …, and finally $\mathbf{f_n = f_{n-1} + f_{n-2}}$

Top-down: instead uses recursive calls to solve smaller problems, while using memoization/caching to keep track of small problems that you've already computed answers for (simply fetch the answer instead of re-solving that problem and waste computational effort)

e.g. **Try to computer f_n by trying to fetch f_{n-1} and f_{n-2}, and compute them if needed**

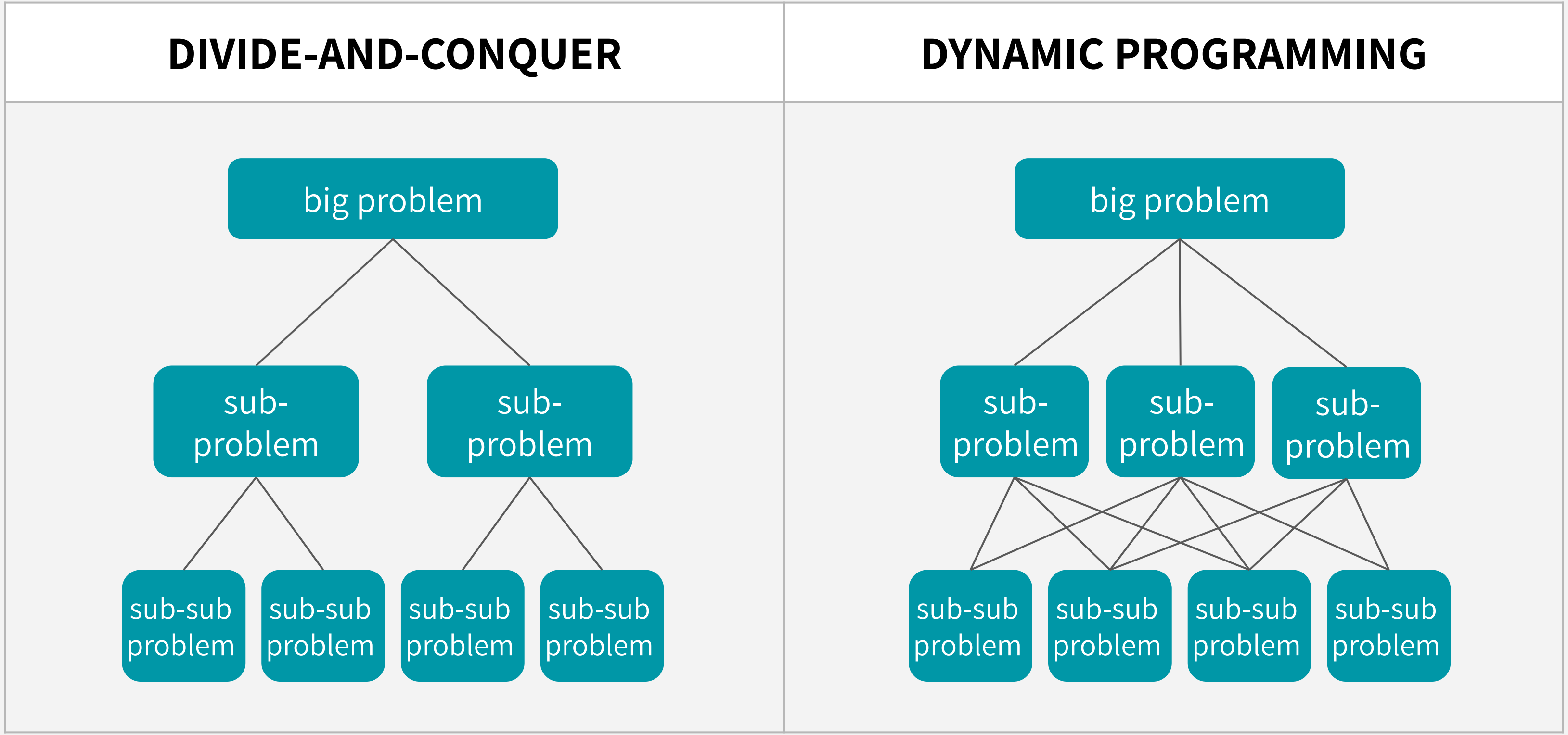
DYNAMIC PROGRAMMING

Why “dynamic programming”?

Richard Bellman invented the term in the 1950’s. He was working for the RAND corporation at the time, which was employed by the Air Force, and government projects needed flashy non-mathematical non-researchy names to get funded and approved.

“It’s impossible to use the word dynamic in a pejorative sense…
I thought dynamic programming was a good name.
It was something not even a Congressman could object to.”

DIVIDE & CONQUER vs DP



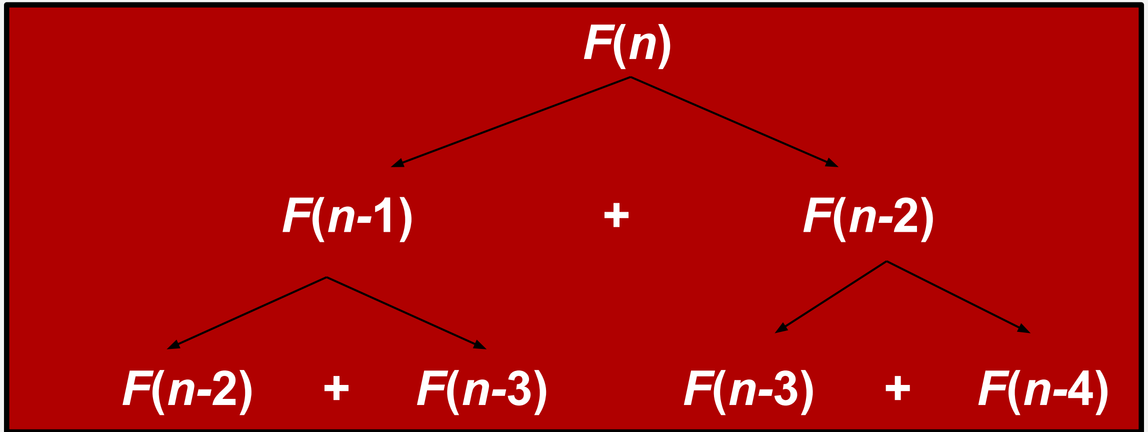
سوال؟

محاسبه عدد فیبوناچی

Recursive Calculation

- Also known as **Top-Down** approach

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



Runtime

- ?

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

Runtime

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

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

Runtime

- $T(n) = T(n-1) + T(n-2) + 1$
- Upper bound: $\mathbf{O(2^n)}$

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

Runtime

- $T(n) = T(n-1) + T(n-2) + 1$
- Upper bound: $\mathbf{O(2^n)}$
- Lower bound: $\mathbf{\Omega(2^{n/2})}$

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

Runtime

- $T(n) = T(n-1) + T(n-2) + 1$
- Upper bound: $\mathbf{O(2^n)}$
- Lower bound: $\mathbf{\Omega(2^{n/2})}$
- $T(n)$ grows very similarly to $F(n)$
  - Actually, we have: $T(n) = \Theta(F(n))$

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

Efficient Calculation

- Use a bottom-up approach
- F(0) = 0
- F(1) = 1
- F(2) = 1+0 = 1
- …
- F(n-2) = F(n-3) + F(n-4)
- F(n-1) = F(n-2) + F(n-3)
- F(n) = F(n-1) + F(n-2)

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

Runtime and Memory

- Time: ?

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

Runtime and Memory

- Time: **O(n)**

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

Runtime and Memory

- Time: **O(n)**
- Space (memory): **?**

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

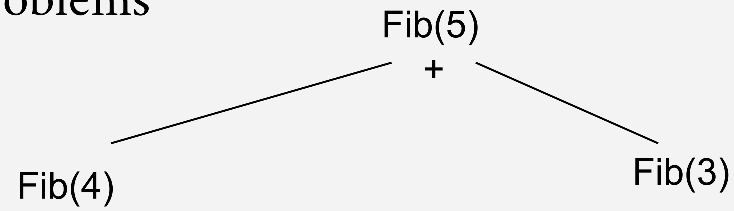
Runtime and Memory

- Time: **O(n)**
- Space (memory): **O(n)**

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

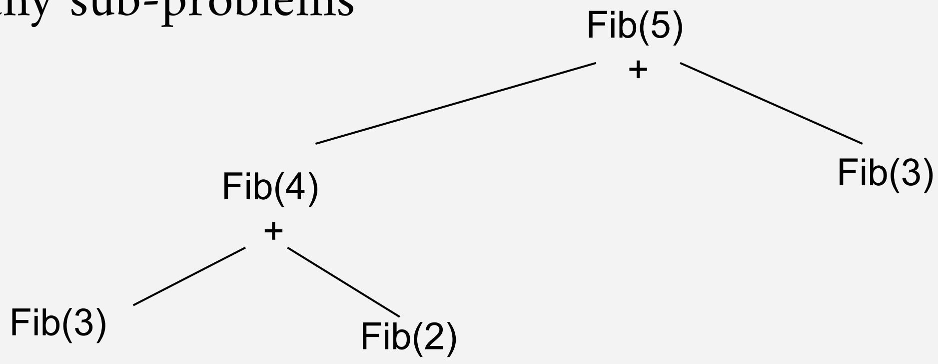
Why Top-Down Approach is sooooo Bad?

- Recomputes many sub-problems



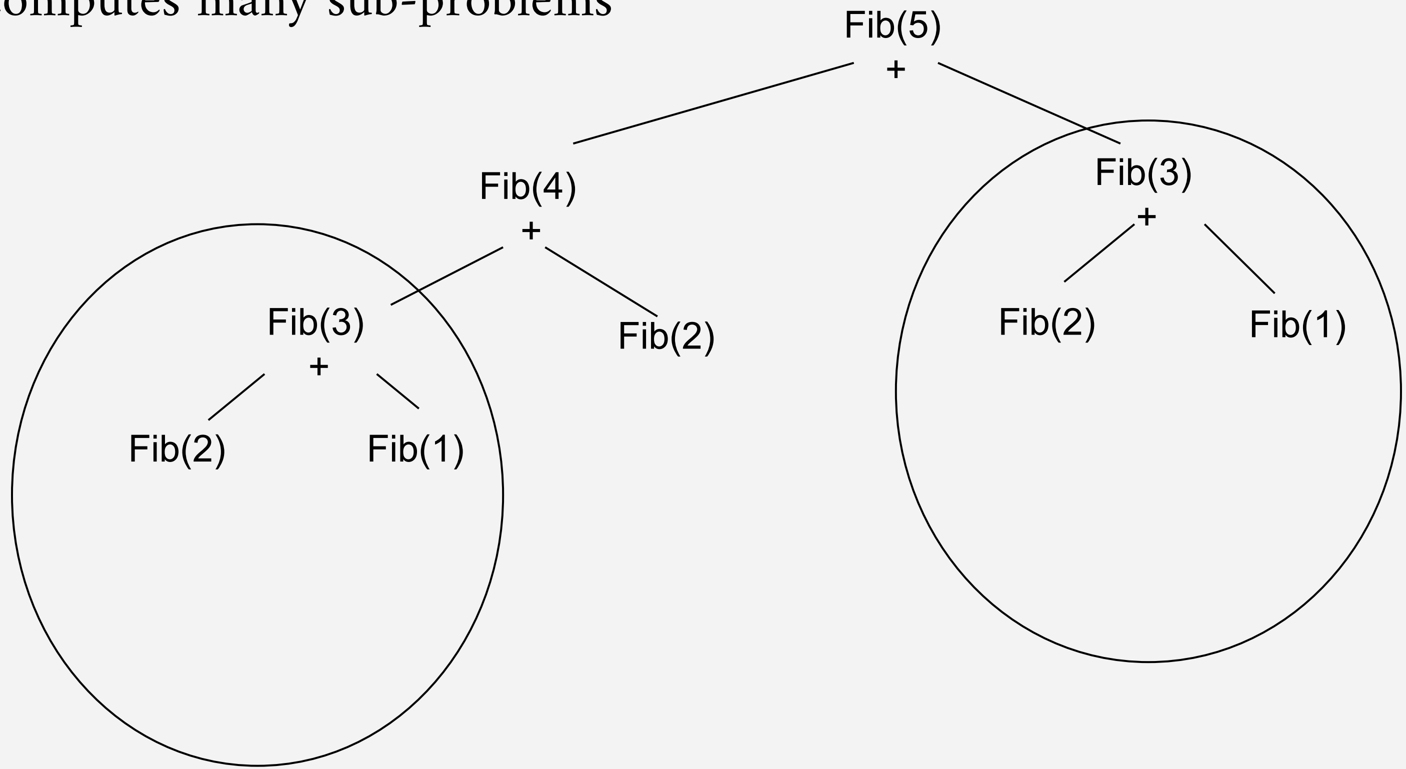
Why Top-Down Approach is sooooo Bad?

- Recomputes many sub-problems



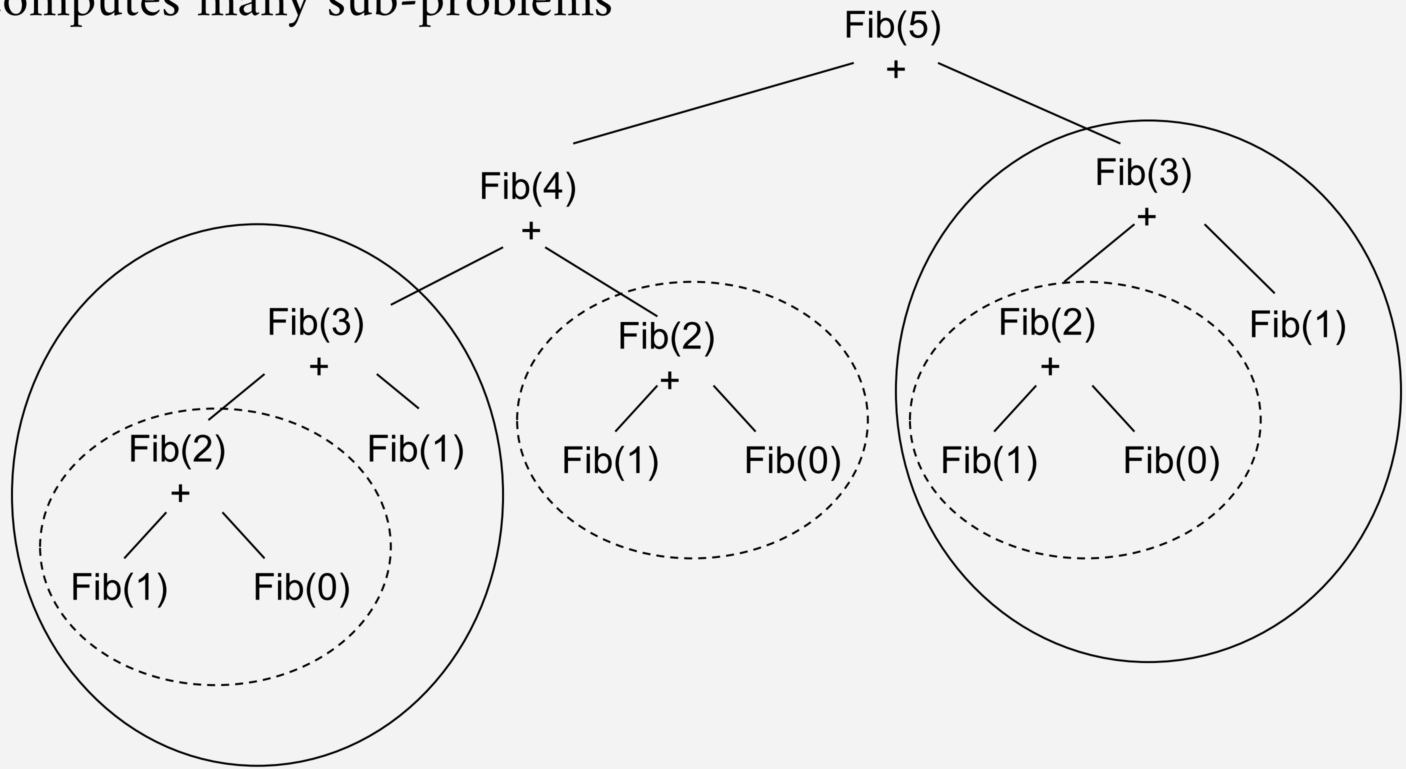
Why Top-Down Approach is sooooo Bad?

- Recomputes many sub-problems



Why Top-Down Approach is sooooo Bad?

- Recomputes many sub-problems



Dynamic Programming Idea

- Divide problem into subproblems and combine their answers
  - Similar to **Divide and Conquer,**
- Subproblems are not independent
  - Unlike divide and conquer
- Subproblems may share subsubproblems

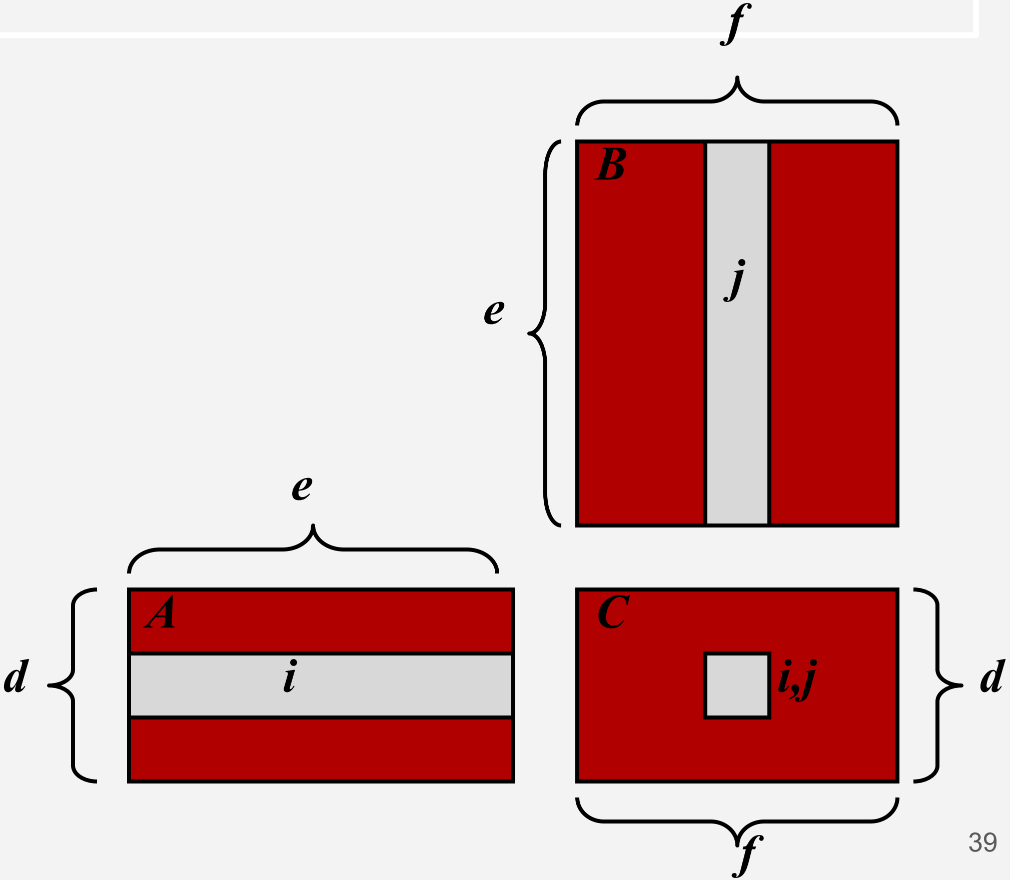
سوال؟

ضرب ماتریس ها

Matrix Multiplication

- C = A*B
- A is d × e and B is e × f

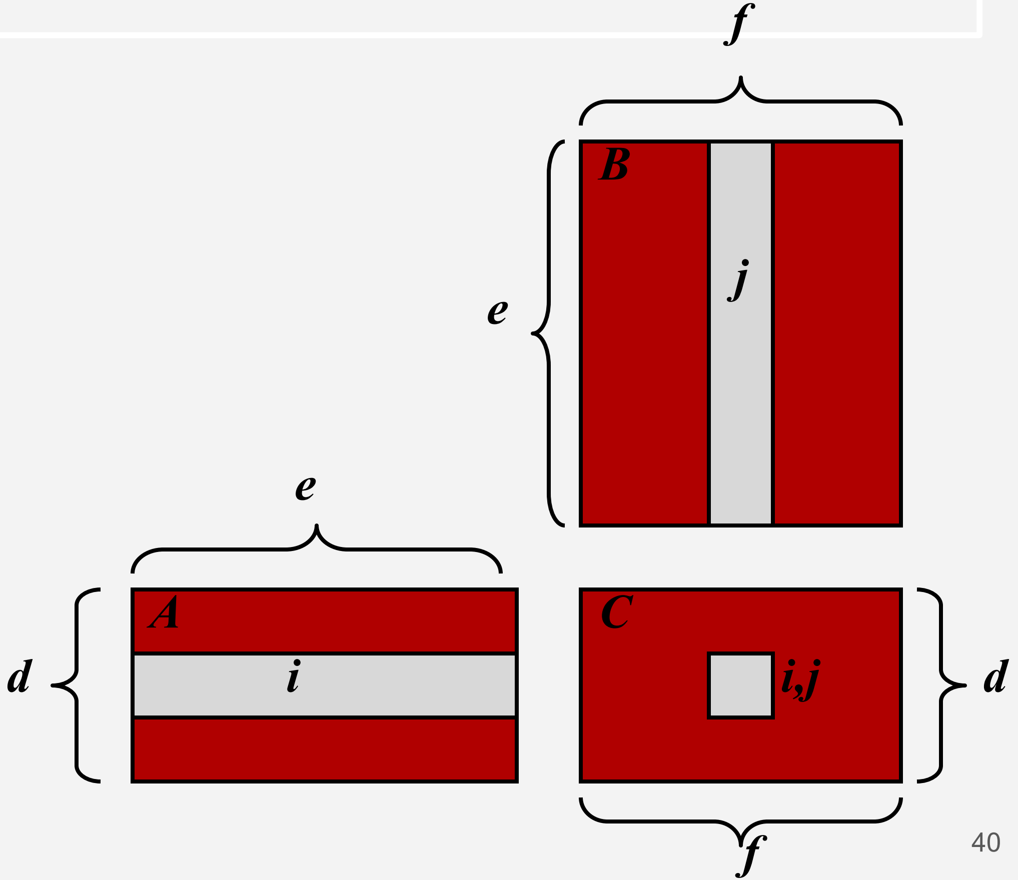
$$C[i,j] = \sum_{k=0}^{e-1} A[i,k] * B[k,j]$$



Matrix Multiplication

- C = A*B
- A is d × e and B is e × f
- Time: **?**

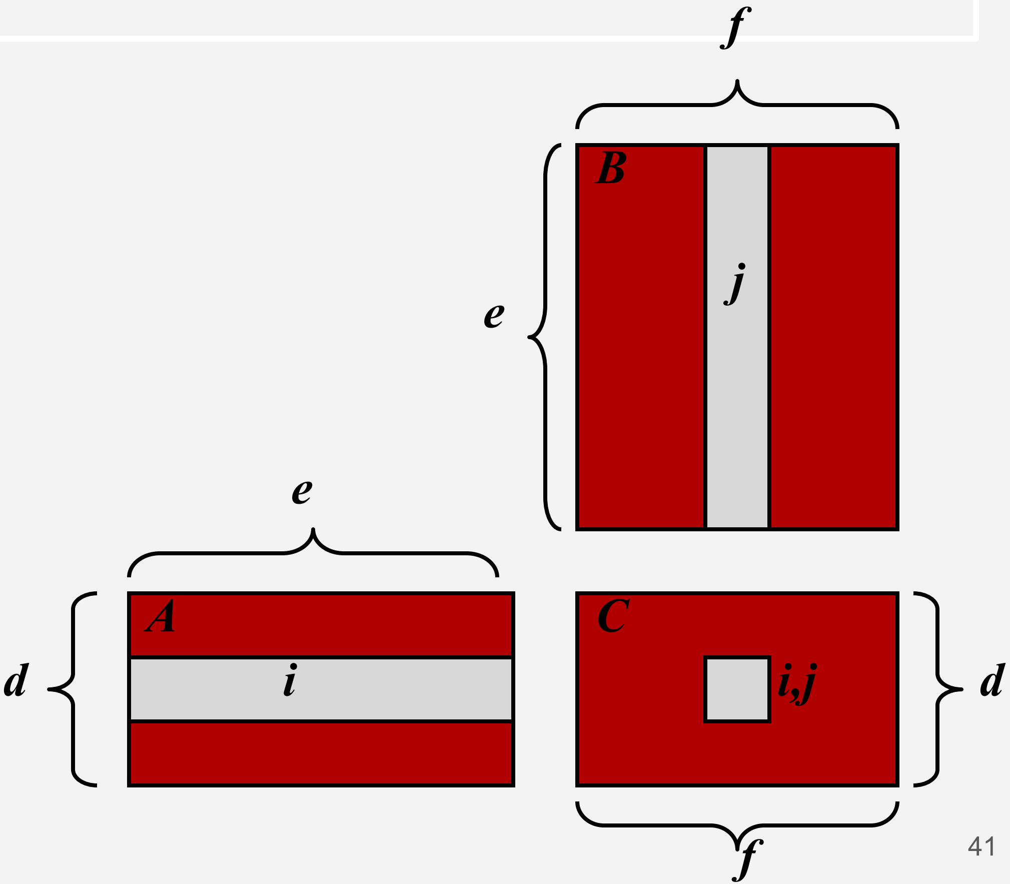
$$C[i,j] = \sum_{k=0}^{e-1} A[i,k] * B[k,j]$$



Matrix Multiplication

- C = A*B
- A is d × e and B is e × f
- Time: $O(d \cdot e \cdot f)$

$$C[i,j] = \sum_{k=0}^{e-1} A[i,k] * B[k,j]$$



Matrix Chain-Products

- Compute $A = A_0 * A_1 * \ldots * A_{n-1}$
- A_i is $d_i \times d_{i+1}$
- Problem: How to parenthesize?

Matrix Chain-Products

- Compute $A=A_0*A_1*\ldots*A_{n-1}$
- A_i is $d_i \times d_{i+1}$
- Problem: How to parenthesize?
- Example:
  - B is 3×100
  - C is 100×5
  - D is 5×5

Matrix Chain-Products

- Compute $A=A_0*A_1*\ldots*A_{n-1}$
- A_i is $d_i \times d_{i+1}$
- Problem: How to parenthesize?
- Example:
  - B is 3×100
  - C is 100×5
  - D is 5×5
  - (B*C)*D takes $1500 + 75 = 1575$ ops

Matrix Chain-Products

- Compute $A=A_0*A_1*\ldots*A_{n-1}$
- A_i is $d_i \times d_{i+1}$
- Problem: How to parenthesize?
- Example:
  - B is 3×100
  - C is 100×5
  - D is 5×5
  - (B*C)*D takes 1500 + 75 = 1575 ops
  - B*(C*D) takes 1500 + 2500 = 4000 ops

Matrix Chain-Products

- Compute $A=A_0*A_1*\ldots*A_{n-1}$
- A_i is $d_i \times d_{i+1}$
- Problem: How to parenthesize?
- Example:
  - B is 3 × 100
  - C is 100 × 5
  - D is 5 × 5
  - (B*C)*D takes 1500 + 75 = 1575 ops
  - B*(C*D) takes 1500 + 2500 = 4000 ops
    - Worst option (slower)

Solution 1: Try different options

- Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
- Calculate number of ops for each one
- Pick the one that is best

Solution 1: Try different options

- Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
- Calculate number of ops for each one
- Pick the one that is best

- Runtime: **?**

Solution 1: Try different options

- Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
- Calculate number of ops for each one
- Pick the one that is best

- Runtime:
  - The number of parenthesizations = the number of binary trees with n nodes

Solution 1: Try different options

- Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
- Calculate number of ops for each one
- Pick the one that is best

- Runtime:
  - The number of parenthesizations = the number of binary trees with n nodes
    - Exponential!
    - Known as **the Catalan number**
      - Almost 4^n

Solution 2: Find first multiplication

- Select the product that uses the fewest operations
    - A greedy choice (will learn more about in a few weeks!)

Solution 2: Find first multiplication

- Select the product that uses the fewest operations
  - A greedy choice (will learn more about in a few weeks!)
- Counter-example:
  - A is 101×11
  - B is 11×9
  - C is 9×100
  - D is 100×99

Solution 2: Find first multiplication

- Select the product that uses the fewest operations
  - A greedy choice (will learn more about in a few weeks!)
- Counter-example:
  - A is 101×11
  - B is 11×9
  - C is 9×100
  - D is 100×99
  - This solution answer: A*((B*C)*D))
    - 109989+9900+108900=228,789 ops

Solution 2: Find first multiplication

- Select the product that uses the fewest operations
  - A greedy choice (will learn more about in a few weeks!)
- Counter-example:
  - A is 101×11
  - B is 11×9
  - C is 9×100
  - D is 100×99
  - This solution answer: A*((B*C)*D))
    - 109989+9900+108900=228,789 ops
  - Optimal solution: (A*B)*(C*D)
    - 9999+89991+89100=189,090 ops

سوال؟

Suboptimal Structure

- Consider the final multiplication in optimal solution
  - Assume it is at index ***k***
  - $(A_0 * \ldots * A_k) * (A_{k+1} * \ldots * A_{n-1})$

Suboptimal Structure

- Consider the final multiplication in optimal solution
  - Assume it is at index ***k***
  - $(A_0*\ldots*A_k)*(A_{k+1}*\ldots*A_{n-1})$
- Cost of this solution:

$$N_{0,n-1} = \min_{0 \le k < n-1} \{N_{0,k} + N_{k+1,n-1} + d_0 d_{k+1} d_n\}$$

  - $\boldsymbol{N_{0,k}}$: Cost of calculating $(A_0*\ldots*A_k)$
  - $\boldsymbol{N_{k+1,n}}$: Cost of calculating $(A_{k+1}*\ldots*A_{n-1})$
  - $\boldsymbol{d_0 d_{k+1} d_n}$: Cost of the final multiplication
    - A_i is a $d_i \times d_{i+1}$ dimensional matrix

Define Subproblem

- Problem: minimum cost of calculating $A_i * A_{i+1} * \ldots * A_j$
  - Use $N_{i,j}$ to denote the minimum cost
- $N_{k,k} = 0$ for all k
- Answer to the main question: $N_{0,n-1}$
- Recursive formula for $N_{i,j}$:

$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

Dependent Subproblems

- Sub-problems are not independent
    - Sub-problems of size m, are independent.
- Example: $N_{2,6}$ and $N_{3,7}$ both need solutions to $N_{3,6}$, $N_{4,6}$, $N_{5,6}$, and $N_{6,6}$.
- Example of high sub-problem overlap
    - Pre-computing common subproblems significantly speed up the algorithm

Naive Recursive Solution

Algorithm ***RecursiveMatrixChain***(S, i, j):

Input: sequence S of n matrices to be multiplied

Output: number of operations in an optimal parenthesization of S

if i=j

then return 0

for $k \leftarrow i$ **to** j **do**

$N_{i,j} \leftarrow \min\{N_{i,j},$ ***RecursiveMatrixChain***(S, i, k) $+$ ***RecursiveMatrixChain***$(S, k+1, j)$ $+ d_i d_{k+1} d_{j+1}\}$

return $N_{i,j}$

Naive Recursive Solution

- Too expensive ($O(2^n)$) similar to Fibonacci case)

Algorithm ***RecursiveMatrixChain***(S, i, j):

  Input: sequence S of n matrices to be multiplied

  Output: number of operations in an optimal parenthesization of S

 if i=j

  then return 0

 for $k \leftarrow i$ **to** j **do**

  $N_{i,j} \leftarrow \min\{N_{i,j},$ ***RecursiveMatrixChain***(S, i, k) $+$ ***RecursiveMatrixChain***$(S, k+1, j)$ $+ d_i d_{k+1} d_{j+1}\}$

 return $N_{i,j}$

Bottom-up Approach

- Easy to calculate $N_{i,i}$ (equals to 0!)
  - Start with them!
- Then do problems of length 2,3,…, n

Algorithm ***matrixChain***(***S***):

Input: sequence ***S*** of ***n*** matrices to be multiplied

Output: number of operations in an optimal parenthesization of ***S***

for $i \leftarrow 1$ **to** $n - 1$ **do**

$N_{i,i} \leftarrow \mathbf{0}$

for $b \leftarrow 1$ **to** $n - 1$ **do**

{ $b = j - i$ is the length of the problem }

for $i \leftarrow 0$ **to** $n -$ b - 1 **do**

$j \leftarrow i + b$

$N_{i,j} \leftarrow +\infty$

for $k \leftarrow i$ **to** $j - 1$ **do**

$N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

return $N_{0,n-1}$

Bottom-up Approach

- Easy to calculate $N_{i,i}$ (equals to 0!)
  - Start with them!
- Then do problems of length 2,3,…, n
- Running time: ?

Algorithm ***matrixChain***(***S***):

Input: sequence ***S*** of ***n*** matrices to be multiplied

Output: number of operations in an optimal parenthesization of ***S***

for $i \leftarrow 1$ **to** $n - 1$ **do**

$N_{i,i} \leftarrow \mathbf{0}$

for $b \leftarrow 1$ **to** $n - 1$ **do**

{ $b = j - i$ is the length of the problem }

for $i \leftarrow 0$ **to** $n - b - 1$ **do**

$j \leftarrow i + b$

$N_{i,j} \leftarrow +\infty$

for $k \leftarrow i$ **to** $j - 1$ **do**

$N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

return $N_{0,n-1}$

Bottom-up Approach

- Easy to calculate $N_{i,i}$ (equals to 0!)
    - Start with them!
- Then do problems of length 2,3,…, n
- Running time: $O(n^3)$

Algorithm *matrixChain*(*S*):

    Input: sequence S of n matrices to be multiplied

    Output: number of operations in an optimal parenthesization of S

for $i \leftarrow 1$ **to** $n - 1$ **do**

  $N_{i,i} \leftarrow \mathbf{0}$

for $b \leftarrow 1$ **to** $n - 1$ **do**

  { $b = j - i$ is the length of the problem }

  for $i \leftarrow 0$ **to** $n - b - 1$ **do**

      $j \leftarrow i + b$

      $N_{i,j} \leftarrow +\infty$

      for $k \leftarrow i$ **to** $j - 1$ **do**

          $N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

return $N_{0,n-1}$

Example Step

A_0: 30 X 35; A_1: 35 X15; A_2: 15X5;
A_3: 5X10; A_4: 10X20; A_5: 20 X 25

0	1	2	3	4	5	
0	15,750	7,875	9,375	11,875	15,125	0
	0	2,625	4,375	7,125	10,500	1
		0	750	2,500	5,375	2
			0	1,000	3,500	3
				0	5,000	4
					0	5

$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

$$
\begin{aligned}
&N_{1,4} = \min\{ \\
&N_{1,1} + N_{2,4} + d_1 d_2 d_5 = 0 + 2500 + 35 * 15 * 20 = 13000, \\
&N_{1,2} + N_{3,4} + d_1 d_3 d_5 = 2625 + 1000 + 35 * 5 * 20 = 7125, \\
&N_{1,3} + N_{4,4} + d_1 d_4 d_5 = 4375 + 0 + 35 * 10 * 20 = 11375 \\
&\} \\
&= 7125
\end{aligned}
$$

DP vs Naive Recursive Solution

- Naive Recursive: $\boldsymbol{O(2^n)}$
  - Solves $O(2^n)$ sub-problems
- DP solution (bottom-up): $\boldsymbol{\Theta(n^3)}$
  - Only $\Theta(n^2)$ distinct sub-problems
  - Implies a high overlap of sub-problems

سوال؟