Dynamic Programming (I)

Charlie Li

Reminder

- Assuming all variables are initialized to zero in all the codes appearing in this powerpoint

Perquisites – things learned before

- Recursion
- Divide and Conquer
- Time complexity analysis (using big O notation)

Fibonacci sequence - recall

- Define:
  - $f(1) = f(2) = 1$
  - $f(n) = f(n - 1) + f(n - 2)$ for $n > 2$
- How can we compute $f(n)$?

Fibonacci sequence - recursion

- We can use recursion and the definition of f(n).
- Code:

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

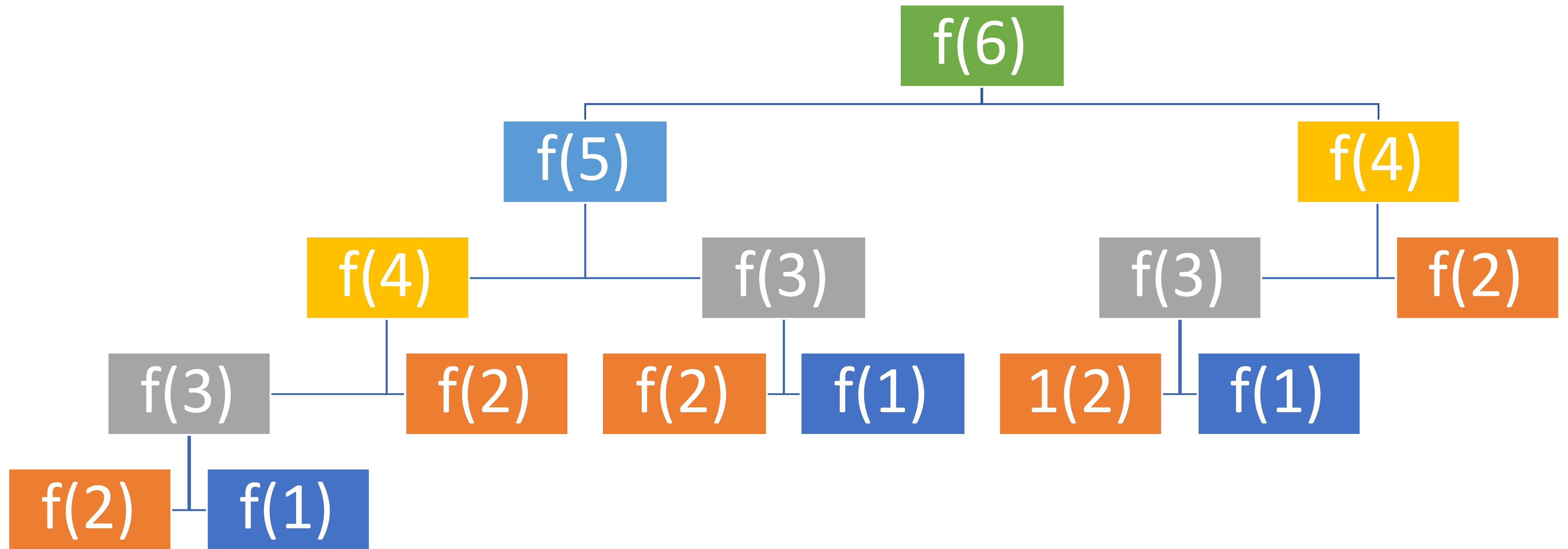
Fibonacci sequence - recursion

- Time complexity?
- $T(n) = T(n - 1) + T(n - 2) + 1 = O(f(n))$ (or simply $O(2^n)$)
- And we know f(n) grows pretty quickly, so this algorithm runs slow
- But why? Can we speed up the computation?

Fibonacci sequence - recursion

- Lets see how do we compute f(6)

Fibonacci sequence - recursion



Fibonacci sequence - recursion

- As we can see, most of the functions are evaluated more than once
- We say these computing power is wasted
- How can we improve the algorithm?

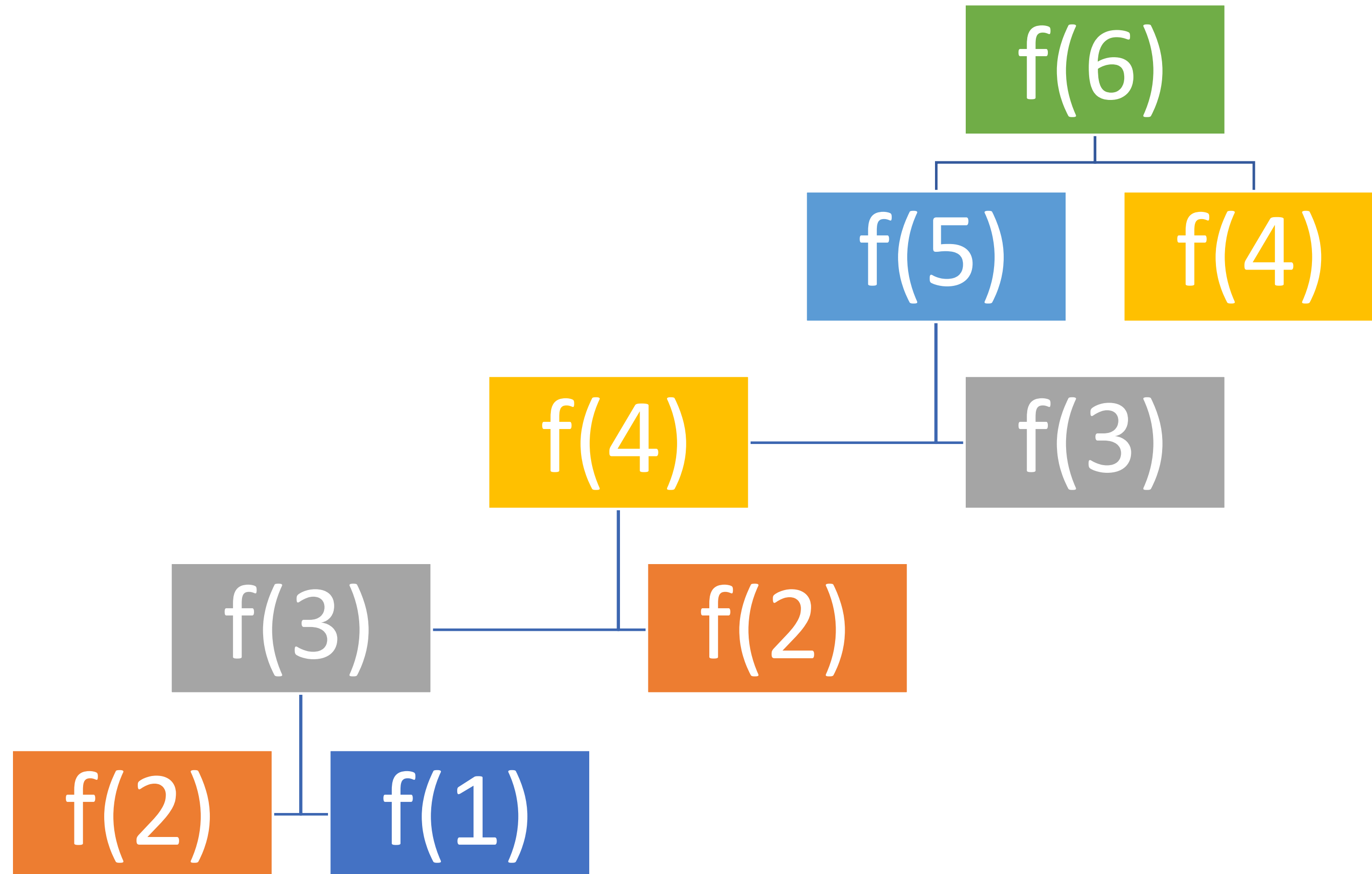
Fibonacci sequence - memorization

- One way to optimize is to store the evaluated value of the function so that we do not need to compute it again
- Code:

```
int f (int n) {
    if (dp[n]) return dp[n];
    if (n == 1 || n == 2) return 1;
    else return dp[n] = f(n - 1) + f(n - 2);
}
```

- Lets see how this algorithm looks like

Fibonacci sequence - memorization



Fibonacci sequence - memorization

- As we can see, we now reduced our computation a lot.
- What is the time complexity now?
- $T(n) = T(n - 1) + 1 = O(n)$
- Wow, now we can compute the Fibonacci sequence in very short time
- But remember, $f(n)$ grows pretty fast, so if n is large, you may need to use long long or HPA for the calculations

Fibonacci sequence - memorization

- Here, we use a top down approach to solve the problem
- ie. We try to find the solution to our problem (f(n)) directly
  we find the solution to its sub-problem (f(n - 1) and f(n - 2)) whenever we need and record it down for future usage
- But why does it work?
- It’s because that the sub-problem (f(n - 1) and f(n - 2)) of our problem (f(n)) can be calculated in the same manner so that we can use a recursion to get the work done

Dynamic programming

- No we come to learn a power technique, dynamic programming
- A problem can be solved using dynamic programming should fulfill:
  - It have optimal substructure
    - An optimal solution can be constructed efficiently form optimal solutions of its sub-problems
  - It have overlapping sub-problems
    - Can be broken down into sub-problems which are reused several times; Or
    - A recursive algorithm for the problem solves the same sub-problem
- Also we will define a ‘state’ for every sub-problems and we need to make sure that the solution to a certain sub-problem only depends on the its state

Fibonacci sequence – bottom up DP

- Let define dp[n] to be f(n)
- We then have the base cases dp[1] = dp[2] = 1
- And the relationship of the problems (aka. transition formula) is
  - dp[n] = dp[n - 1] + dp[n - 2], n > 2
- So we can use a for loop to do this
- Code:

```
dp[1] = dp[2] = 1;
for (int i = 3; i <= n; i++) dp[n] = dp[n - 1] + dp[n – 2];
```

- When you apply the transition formula, make sure that the sub-problems is already solved when using the bottom up approach

Maximum subarray sum - problem

- Let a_1, a_2, a_3, …, a_n be a sequence with length n
- Define subarray of the sequence to be the list a_i, a_{i+1}, …, a_{j-1}, a_j for some $i < j$
- The problem is to find the maximum sum of the subarray if we choose some i, j smartly
- If all a_i are negative, the answer is zero

Maximum subarray sum – brute force

- We can use brute force to find the answer
- Code:

```
for (int i = 1; i <= n; i++) for (int j = i; j <= n; j++) {
        sum = 0;
        for (int k = i; k <= j; k++) sum += a[k];
        ans = max(ans, sum);
}
```

- Time complexity?
- $O(n^3)$

Maximum subarray sum – brute force

- We can use partial sum (aka. delta array) to optimize
- Code:

```
for (int i = 1; i <= n; i++) sum[i] = sum[i - 1] + a[i];
for (int i = 1; i <= n; i++) for (int j = i; j <= n; j++)
        ans = max(ans, sum[j] – sum[i - 1]);
```

- Time complexity?
- $O(n^2)$
- Can it be faster?
- Yes, we can use DP, how?

Maximum subarray sum - DP

- When doing DP problems, we often define the state to be something like “The optimal solution from 1 to i” or “The optimal solution containing i”
- Here, we use the latter one
- We define dp[i] to be the maximum subarray sum where a_i is the last element of the subarray
- ie. dp[i] = $\max\limits_{1 \le j \le i} \sum_{k=j}^{i} a_k$
- And the answer to the problem is $\max\limits_{1 \le i \le n} dp[i]$
- Here, the optimal solution is either concatenate to the previous one or being independent
- So, our transition formula is
- dp[i] = max(dp[i - 1] + a[i], a[i])

Maximum subarray sum - DP

- Code:

```
for (int i = 1; i <= n; i++) dp[i] = max(dp[i - 1] + a[i], a[i]);
for (int i = 1; i <= n; i++) ans = max(ans, dp[i]);
```

- Time complexity?
- O(n)

Maximum subarray sum - DP

“I have learned that we can use divide and conquer to solve the same question with the same time complexity, why should I learn DP?”

- The code is shorter which bring a lot of benefits
  - Easier to debug
  - Quicker to code
- DP is guaranteed to give us the optimal solution if we define the state and transition formula correctly
- However, the divide and conquer solution is needed if we are finding the online maximum subarray sum in which case DP is not the fastest solution

Jumper - problem

- In a city, there is N buildings, counting from the left, the ith building has the height h_i
- There is a man want to jump from building 1 to building N
- When he jump from building i to building j, the energy consumed is $|h_i – h_j|$
- Although he can jump very high, but he is so scared that he can only jump through at most K gaps between the buildings
- Also, he can only jump to the right
- What is the minimum energy consumption?

Jumper - DP

- Define dp[i] be the minimum energy consumption for the man to jump to building i
- The transition formula is
- $dp[i] = \min_{i-k \le j < i} (|h[i] - h[j]| + dp[j])$
- Since the man can only jump from a building between i – k and i, so we just need to find the minimum one

Jumper - DP

- Code:

```
for (int i = 2; i <= N; i++) {
    dp[i] = INF;
    for (int j = max(1, i- k); j < i; j++)
        dp[i] = min(dp[i], dp[j] + abs(h[i] – h[j]));
}
```

- Time complexity?
- O(NK)

Longest common subsequence - problem

- Given two string S and T, find the length of longest common subsequence
- Here, subsequence of a string S is any string that can be obtained by removing some(possibly none) characters of S

- EG.
- S = “ABCDEFG”
- T = “XVBDGH”
- Longest common subsequence = “BDG” (length 3)

Longest common subsequence – brute force

- It is not an easy job to think of a brute force solution for the problem
- Perhaps we should go ahead to think of the DP solution
- Remember what I have talked before
- When doing DP problems, we often define the state to be something like “The optimal solution from 1 to i” or “The optimal solution containing i”
- This time, we may choose the former one
- But since now we have 2 strings, so we are make it a 2D array

Longest common subsequence - DP

- Define dp[i][j] be the length of longest common subsequence of string S[0], S[1], …, S[i] and string T[0], T[1], …, T[j]
- Transition formula:
- dp[i][j] = max(dp[i - 1][j], dp[i][j -1]) if S[i] != T[j]
- dp[i][j] = max(dp[i - 1][j], dp[i][j – 1], dp[i - 1][j - 1] + 1) if S[i] == T[j]

- If S[i] != T[j], then the answer will be just the same as we either concatenate S[i] to S[0], S[1], … S[i -1] or concatenate T[j] to T[0], T[1], …, T[j-1] without changing the second string
- But if S[i] == T[j], there may be one more option, that is concatenating both at the same time, and increase the answer by 1

Longest common subsequence - DP

- Code:

```
dp[0][0] = S[0] == T[0] ? 1 : 0;
    for (int i = 0; S[i]; i++) for (int j = 0; T[j]; j++) {
        dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        if (S[i] == T[j]) dp[i][j] = max(dp[i][j], dp[i - 1][j - 1] + 1);
    }
```

- Time complexity?
- O(|S||T|)

A short break

Knapsack problem - problem

- Given N items, each with a weight w_i and value v_i
- Our bag have a capacity of K which means for any set of items, we can put them in iff their total weight is at most K
- We want to maximize the total value of items that we put into our bag
- Noted that the order of putting the items does not matter
- for each item, we either put it or not put it into our bag

Knapsack problem – brute force

- We can use brute force to find the answer
- Code:

```
for (int i = 1; i <= (1 << N); i++) {
    sum_w = 0;
    sum_v = 0;
    for (int j = 0; j < N; j++) {
        if (i & (1 << j)) {
            sum_w += w[j];
            sum_v += v[j];
        }
    }
    if (sum_w <= K) ans = max(ans, sum_v);
}
```

Knapsack problem – brute force

- Time Complexity?
- $O(2^N)$

- Seems very slow…
- How can we do faster?

Knapsack problem – wrong attempt

- Perhaps we may define the efficiency of each item e_i to be v_i/w_i
- And perhaps we may put the most efficient item into our bag, then the second and so on
- However, this is wrong because we cannot cut our item into pieces
- Can you think of a counter example?

Knapsack problem – DP

- Define dp[i][j] to be the largest total value of items using item 1, item 2, up to item i with total weight at most j.
- The answer is dp[N][K]
- The transition formula is
- dp[i][j] = max(dp[i - 1][j], dp[i - 1][j – w[i]] + v[i])

- This means we are either putting or not putting the item into our bag

Knapsack problem – DP

- Code:

```
for (int i = 1; i <= N; i++) for (int j = w[i]; j <= K; j++)
        dp[i][j] = max(dp[i][j], dp[i - 1][j – w[i]] + v[i]);
```

- Time complexity?
- O(NK)

Knapsack problem - Variations

- What if:
  - Additionally, the bag has a capacity of volume M and each item has volume u_i?
  - We can pick as many item i as we want?
  - We can pick at most p_i of the item i
  - We can pick item i only if we have picked item j for some i?

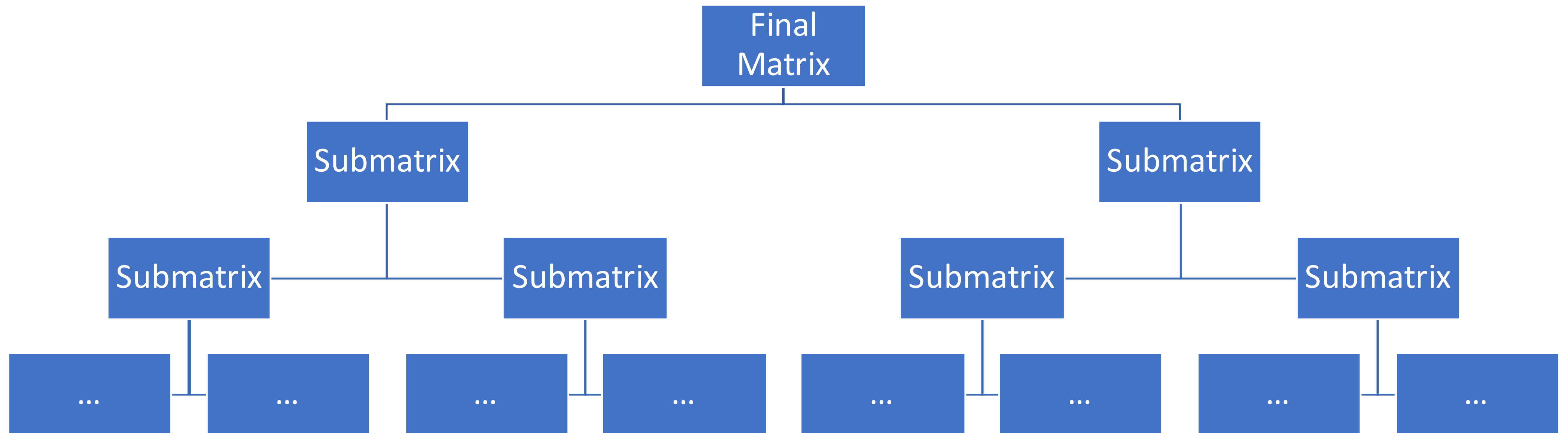
Matrix multiplication - problem

- A $m \times n$ matrix is like a 2D array of numbers with m rows and n columns
- Let A_1 and A_2 be matrices, A_1A_2 exist iff no. of columns in A_1 equals to no. of rows in A_2
- Let say, A_1 is an $m \times p$ matrix and A_2 is an $p \times n$ matrix, then A_1A_2, the multiple of two matrices, will be a $m \times n$ matrix, the cost of such operation is $m \times p \times n$
- Matrix multiplication is associative, ie. $(A_1A_2)A_3 = A_1(A_2A_3)$
- But matrix multiplication is not commutative, ie. A_1A_2 != A_2A_1
- Given N + 1 numbers, where the ith one is p_i where A_i is a $p_i \times p_{i+1}$ matrix, find the minimum cost to multiply all the matrix together

Matrix multiplication - problem

- Eg.
- A_1 has dimension 5×10
- A_2 has dimension 10×15
- A_3 has dimension 15×5
- If we calculate $(A_1 A_2) A_3$, the total cost is $5 \times 10 \times 15 + 5 \times 15 \times 5 = 1125$
- If we calculate $A_1 (A_2 A_3)$, the total cost is $10 \times 15 \times 5 + 5 \times 10 \times 5 = 1000$
- So the minimum cost is 1000

Matrix multiplication



Matrix multiplication - DP

- How can we define the states?

Matrix multiplication - DP

- How can we define the states?
- We can define dp[i][j] to be the minimum cost to multiply $A_iA_{i+1}...A_j$
- And the answer is dp[1][N]
- Then what is the transition formula?

Matrix multiplication - DP

- How can we define the states?
- We can define dp[i][j] to be the minimum cost to multiply $A_i A_{i+1} \ldots A_j$
- And the answer is dp[1][N]
- Then what is the transition formula?
- $dp[i][j] = \min_{i<k\leq j}(dp[i][k-1] + dp[k][j] + p[i] \times p[k] \times p[j+1])$
- For every i, j, we are finding the optimal way to cut them into two submatrices
- We now have both the states and the transition formula but how can we actually calculate all dp[i][j]?

Matrix multiplication – Wrong DP

- Code:

```
for (int i = 1; i <= N; i++) for (int j = i + 1; j <= N; j++) {
    dp[i][j] = INF;
    for (int k = i + 1; k <= j; k++)
        dp[i][j] = min(dp[i][j], dp[i][k - 1] + dp[k][j] + p[i] * p[k] * p[j + 1])
}
```

Matrix multiplication – Wrong DP

- Code:

```
for (int i = 1; i <= N; i++) for (int j = i + 1; j <= N; j++) {
    dp[i][j] = INF;
    for (int k = i + 1; k <= j; k++)
        dp[i][j] = min(dp[i][j], dp[i][k - 1] + dp[k][j] + p[i] * p[k] * p[j + 1])
}
```

- Remember what I have said before:
- When you apply the transition formula, make sure that the sub-problems is already solved when using the bottom up approach

Matrix multiplication - DP

- Due to the nature of this question, it is a little bit hard to design the order of dp[i][j] that we need to calculate, it will be better if we use a top down approach

Matrix multiplication – top down DP

- Code:

```
long long solve(int i, int j) {
    if (i == j) return 0;
    if (v[i][j]) return dp[i][j];
    v[i][j] = true;
    dp[i][j] = INF;
    for (int k = i + 1; k <= j; k++)
        dp[i][j] = min(dp[i][j], solve(i, k - 1) + solve(k, j) + p[i] * p[k] * p[j + 1]);
    return dp[i][j];
}
```

Matrix multiplication – top down DP

- Time complexity?
- No. of states: $O(n^2)$
- State transition: $O(n)$
- Total: $O(n^3)$

Practice problems

Let’s think of the solutions

Palindrome subsequence - problem

- Given a string S with length N
- You are going to find the length of the longest palindrome subsequence
- A subsequence of a string S is any string that can be obtained by removing some(possibly none) characters in S
- A palindrome is a string that looks the same after it is reversed

- How can we define the state and the transition formula?
- What is the time complexity of the algorithm?

Palindrome subsequence - solution

- It is easier to use top down approach
- States:
  - Define dp[i][j] to be the length of maximum length of palindrome subsequence for the substring S[i], S[i+1], …, S[j]
- Transition formulars:
  - dp[i][j] = dp[i+1][j-1] + 2 if S[i] == S[j] (this means we cut the first and the last character)
  - dp[i][j] = max(dp[i+1][j], dp[i][j - 1]) if S[i] != S[j] (this means we cut either the first or the last character)
- Base cases:
  - dp[i][j] = 0 if i > j
  - dp[i][j] = 1 if i == j
- Time complexity: $O(n^2)$

Balanced parentheses - problem

- Given the length of the string is 2N
- Find the no. of different strings such that it contain exactly N open brackets ‘(’ and N close brackets ‘)’ and it is balance
- We say the parentheses are balanced if the no. of open brackets is not less than the no. of close brackets in any prefix.
- Since the number is large, you are only required to output the answer mod $10^9 + 7$

- How can we define the state and the transition formula?
- What is the time complexity of the algorithm?

Balanced parentheses - solution

- State:
  - Define dp[i][j] to be the no. of strings (mod $10^9 + 7$) with length i, containing exactly j open brackets (ie. containing exactly i – j close brackets) andfor any of its prefix, no. of open brackets is no less than no. of close brakcets
- Transition formula:
  - dp[i][j] = dp[i - 1][j - 1] if (j - 1) == (i - 1) - (j - 1) (we can only put an open bracket)
  - dp[i][j] = dp[i – 1][j] + dp[i - 1][j – 1] if (j - 1) > (i - 1) - (j – 1) (we can either put an open or a close bracket)
- Base case:
  - dp[1][1] = 1
- Time complexity: $O(n^2)$

More problems:

- Maximum subarray sum
  - 01010 Diamond Chain
  - 01016 Diamond Ring
  - M0822 Diamond Chain II
- Knapsack problem
  - 05011 Coin
  - T043 Need for speed
- Palindrome
  - I0011 Palindrome
  - CF 607B Zuma

More problems:

- Parentheses
  - CF 629C Famil Door and Brackets
- Combinatorics
  - CF 553A Kyoya and Colored Balls
- Probabilities
  - CF 540D Bad Luck Island
- Extra
  - M1724 Guess the Number
    - Easier to solve some kind of ‘reverse’ problem