AI2615 算法设计与分析 2020-2021 春季学期

Lecture 10 – Dynamic Programming 3

2021 年 4 月 30 日

Lecturer: 张驰豪 *Scribe:* 陶表帅

We will continue studying dynamic programming in this lecture. In particular, we will see some advanced techniques for reducing the number of sub-problems/states and for designing sub-problems such that computing recurrence relations can be done faster.

1 Non-attacking Kings on Chessboard

In a single round of a chess game, a king can move in one of the eight directions (two horizontal directions, two vertical directions and four diagonal directions) by one step. The labels "X" in Fig. 1 shows a king's movement in one round. Given a $m \times n$ chessboard and multiple kings placed on it, we say that the kings are *non-attacking* if no king can move to the position of any other king in one round.

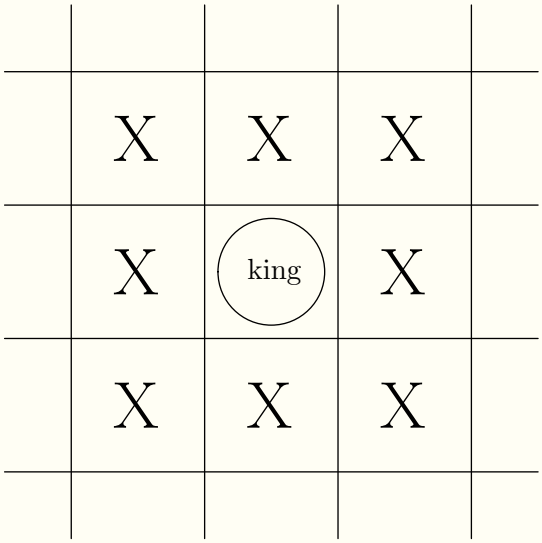

Figure 1: A king's feasible movement in one round.

Problem 1. Given a $m \times n$ chessboard with $n \leq m$, how many different ways are there to place non-attacking kings?

This problem can be solved by a brute-force search: just check for all the possible 2^{mn} kings placements, and see how many of them are non-attacking. This requires time complexity $O(2^{mn})$. We will present a dynamic programming algorithm that solves this problem with time complexity $O(4^n \cdot m)$.

A natural idea is to define $F(i)$ for each $i = 1, \ldots, m$ such that $F(i)$ is the number of valid placements if restricted to the first i rows of the board. However, it is hard to find a recurrence relation if we define the sub-problems in this way. This is because the placement of the kings on the $(i-1)$-th row affects the validity of the placement on the i-th row. For example, if a king is placed at $(i-1, j)$ on the $(i-1)$-th row, we know that the three positions $(i, j-1), (i, j), (i, j+1)$ on the i-th row are not allowed to place any kings. Therefore, to figure out how $F(i)$ is related to $F(i-1), \ldots, F(1)$, we need the extra information on how kings on the $(i-1)$-th row are placed. This motivates us to define the sub-problem in the following way.

We denote the set $\{1, \ldots, n\}$ by $[n]$. For each $i = 1, \ldots, m$ and each $S \subseteq [n]$, let $F(i, S)$ be the number of valid placements in the first i rows, given that the placement on the i-th row is according to S (i.e., a king is placed at (i, j) for each $j \in S$). Given $S \subseteq [n]$, we say that S is *non-contiguous* if there is no pair of two adjacent numbers in S. Obviously, if S describes the placement of the kings on a row, S is required to be non-contiguous. If S and S' describe the placements of the kings in two adjacent rows, we require that $S \cap S' = \emptyset$ (no pair of two kings can be vertically adjacent) and that $S \cup S'$ is non-contiguous (no pair of two kings can be horizontally or diagonally adjacent). We then have the following natural recurrence relation:

$$F(i, S) = \sum_{S' : S' \subseteq [n], S' \cap S = \emptyset, S' \cup S \text{ is non-contiguous}} F(i-1, S').$$

For initial condition, we have

$$F(1, S) = \begin{cases} 1 & \text{if } S \text{ is non-contiguous} \\ 0 & \text{otherwise} \end{cases}.$$

The output we need is

$$\sum_{S : S \subseteq [n], S \text{ is non-contiguous}} F(n, S).$$

This finishes all the essential details for the dynamic programming algorithm.

There are $2^n \times m$ sub-problems/states of this dynamic programming algorithm. The recurrence relation can be computed in $O(2^n)$ time, as there are less than 2^n possible subsets for S'. The overall time complexity is therefore $O(4^n \cdot m)$.

Exercise 2. The one-round movements for a "knight" and a "queen" are shown in Fig. 2. Can we use the similar dynamic programming technique to solve Problem 1 if we are placing knights? What about queens?

2 Largest Number in Every k Consecutive Numbers

In this section, we consider the problem of finding the largest number in every k consecutive numbers in an array.

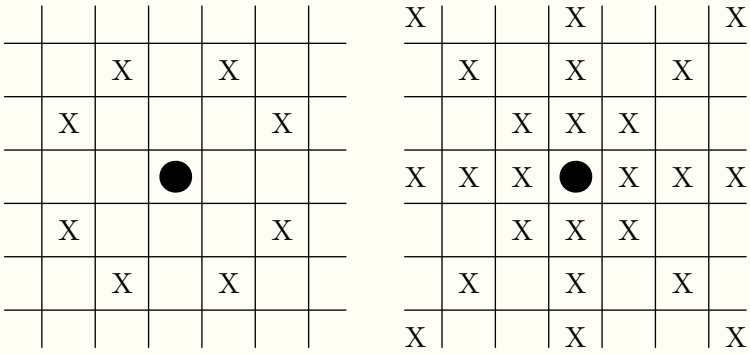

Figure 2: The one-round feasible movements for a knight (left-hand side) and a queen (right-hand side).

Problem 3. Let $a[1 \cdots n]$ be an array of numbers and k be an integer less than n. Let $b[1 \cdots n]$ be the array defined by $b[i] = \max_{j:\max\{i-k+1,1\} \le j \le i} \{a_j\}$. Compute b.

The naïve way of computing b require $O(nk)$ time. We will see a clever dynamic programming algorithm that runs in $O(n)$ time. Below, an *interval* refers to a sub-array consisting of k consecutive numbers, or a sub-array consisting of less than k consecutive numbers that starts at $a[1]$. By saying the i-th interval, we mean the interval ended at $a[i]$, i.e., the interval from $a[\max\{i-k+1,1\}]$ to $a[i]$. Problem 3 then lets us find the largest number in every interval.

The technical heart of the algorithm is to maintain an array C which stores the indices of the numbers each of who has a chance to become the largest number in an interval. To be more specific, the algorithm will iteratively calculate $b[1], b[2], \ldots, b[n]$; after calculating $b[i]$, C stores the indices of the numbers in the i-th interval such that each of those numbers can potentially be the largest number in the i'-th interval for some $i' > i$.

Example 4. Let $a = \{3,5,7,12,3,5,4,5,6,7,5,10,8\}$ and $k = 4$. Suppose we have just calculated $b[7]$. That is, we have identified that the largest number in the 7-th interval $\{12,3,5,4\}$ is 12. We have $C = \{6,7\}$ at current stage. Notice that the fourth number 12, as well as any number before the fourth number, can no longer be the largest number in any later intervals, as any later interval will not even contain any of them. Therefore, we only need to consider $a[5] = 3$, $a[6] = 5$, and $a[7] = 4$. Next, $a[5] = 3$ cannot be the largest number in any later intervals, because any later intervals containing $a[5]$ will also contain $a[6]$, and we have $a[6] > a[5]$, which implies $a[5] = 3$ will never be the largest number. Finally, $a[6] = 5$ and $a[7] = 4$ both have a potential to become the largest number. In particular, if $a[8]$ and $a[9]$ are both less than $a[6]$, then $a[6]$ is the largest number in the 9-th interval. If $a[8]$, $a[9]$ and $a[10]$ are all less than $a[7]$, then $a[7]$ is the largest number in the 10-th interval. Therefore, C stores the indices 6 and 7. We remark that we do not see $a[i+1], \ldots, a[n]$ by the time we are updating C at the i-th iteration of the algorithm. By saying a number "has a potential to become the largest number in a later interval", we mean that there exist certain $a[i+1], \ldots, a[n]$ such that this number is the largest number in one (or more) of the later intervals.

We have the following observations.

Proposition 5. *By the end of the i-th iteration, if there exist two indices j and j' such that $a[j] < a[j']$ and $j < j' \le i$, we must have $j \notin C$. On the other hand, we must have $j \in C$ if $j \ge i - k + 2$ and $a[j] \ge a[j']$ for any j' with $j < j' \le i$.*

Proof. Suppose we are at the end of the i-th iteration, and we have $a[j] < a[j']$ and $j < j'$. At any later interval $i' > i$, if i'-th interval does not contain $a[j]$, $a[j]$ naturally will not be the largest number in this interval. If i'-th interval contains $a[j]$, since $j < j' \le i < i'$, $a[j']$ is also contained in this interval. In this case, $a[j]$ will not be the largest number due to the presence of $a[j']$. As a result, $j \notin C$.
To show the second part, it suffices to notice that $a[j]$ will be the largest number in $(j + k - 1)$-th interval if all of $a[i + 1], a[i + 2], \ldots, a[j + k - 1]$ are less than $a[j]$. □

Proposition 6. *Suppose C stores the indices by the ascending order. After each iteration of the algorithm, the sequence $a[C[i]]$ is decreasing in i.*

Proof. This follows straightforwardly from the first part of Proposition 5. □

With the help from C, it is then easy to find $b[i]$. Since C stores the indices of all the first $i - 1$ numbers who can potentially be the largest, by Proposition 6, we directly have $b[i] = \max\{a[C[1]], a[i]\}$. It remains to figure out how to update C by the end of the i-th iteration.
Firstly, we need to check if the first index of C is "expired" after i-th iteration. That is, we check $C[1]$, and we remove $C[1]$ from C if $C[1] < i - k + 2$. In particular, notice that the $(i+1)$-th interval starts at $a[i - k + 2]$. It is also easy to see that, by the end of each iteration, at most one index from C can be expired (why?).
Secondly, after seeing the value of $a[i]$, we need to remove those $j \in C$ such that $a[j]$ no longer has a chance to be the largest number in a later interval. By proposition 5, we need to remove exactly those $j \in C$ such that $a[j] < a[i]$.
We can use a queue to store C, as we only need to remove elements from both ends of C and add elements from one end. The algorithm is presented in Algorithm 1.
The time complexity for Algorithm 1 is $O(n)$, as it can be easily checked that each number is included in C exactly once and is removed from C at most once.

3 Longest Increasing Subsequence (Revisited)

We have seen a dynamic programming algorithm for the longest increasing subsequence problem with time complexity $O(n^2)$. In this lecture, we will see a more clever dynamic programming algorithm that runs in $O(n \log n)$ time.

Problem 7. Given a sequence of n numbers $a[1 \cdots n]$, find the length of the longest (strictly) increasing subsequence.

Algorithm 1 Find largest number in every k consecutive numbers.

Input: $a[1 \cdots n]$, $k \in \mathbb{Z}^+$

Output: $b[1 \cdots n]$, where $b[i]$ is the largest number in the i-th interval

```
C ← {1}
b[1] = a[1]
for i = 2, ..., n:
    b[i] ← max{a[i], a[C[1]]}
    if C[1] < i − k + 2: remove the first element from C
    while a[i] is greater than the last element of C:
        remove the last element from C
    endwhile
    append a[i] to the end of C
endfor
return b
```

Let $F(i, j)$ be defined as follows: consider the set of all increasing subsequences with length j in $a[1 \cdots i]$; in this set of subsequences, consider the subsequence with the smallest ending number; $F(i, j)$ is then defined to be the ending number of this subsequence; if $a[1 \cdots i]$ does not contain a subsequence with length j, set $F(i, j) = \infty$. Let $\texttt{len}_i$ be the maximum j such that $F(i, j) < \infty$. It is clear that $\texttt{len}_i$ is the length of the longest increasing subsequence in $a[1 \cdots i]$.

Example 8. Consider the input $a[1 \cdots 8] = \{1, 8, 3, 6, 5, 4, 7, 2\}$. Let us find $F(5, j)$ for each j. In $a[1 \ldots 5] = \{1, 8, 3, 6, 5\}$.

- The set of all increasing subsequences with length 1 is $\{1\}, \{8\}, \{3\}, \{6\}$ and $\{5\}$. Among then, $\{1\}$ has the smallest ending number, which is 1. Therefore, $F(5, 1) = 1$.
- The set of all increasing subsequences with length 2 is $\{1, 8\}, \{1, 3\}, \{1, 6\}, \{1, 5\}, \{3, 6\}$ and $\{3, 5\}$. Among then, $\{1, 3\}$ has the smallest ending number, which is 3. Therefore, $F(5, 2) = 3$.
- The set of all increasing subsequences with length 3 is $\{1, 3, 6\}$ and $\{1, 3, 5\}$. Among then, $\{1, 3, 5\}$ has the smallest ending number, which is 5. Therefore, $F(5, 2) = 5$.
- $a[1 \ldots 5] = \{1, 8, 3, 6, 5\}$ does not contain any subsequence with length 4, or more than 4. Therefore, $F(5, 4) = F(5, 5) = \cdots = \infty$.

In this example, we have $\texttt{len}_5 = 3$.

For the initial condition, we obviously have $F(1, 1) = a[1]$, $F(1, 2) = F(1, 3) = \cdots = \infty$, and $\texttt{len}_1 = 1$. The output of the algorithm is simply $\texttt{len}_n$. It remains to find the recurrence relation.

Proposition 9. *For each $i = 1, \ldots, n$, $\{F(i, 1), F(i, 2), \ldots, F(i, \texttt{len}_i)\}$ is a strictly increasing sequence.*

Proof. Consider any i and any j_1, j_2 such that $j_1 < j_2 \le \texttt{len}_i$. We aim to show that $F(i, j_1) < F(i, j_2)$. Let $S[1 \cdots j_2]$ be an increasing subsequence of $a[1 \cdots i]$ such that $F(i, j_2) = S[j_2]$. Consider the subsequence $S[1 \cdots j_1]$. Since S is increasing, we have $S[j_1] < S[j_2]$. We have identified an increasing subsequence of $a[1 \cdots i]$ with length j_1 that ends at $S[j_1]$, and we have $S[j_1] < S[j_2] = F(i, j_2)$. This implies $F(i, j_1) \le S[j_1] < F(i, j_2)$. □

Suppose we already know $F(i-1,1), \ldots, F(i-1, \texttt{len}_{i-1})$. We need to find $\texttt{len}_i$ and $F(i,1), \ldots, F(i, \texttt{len}_i)$ in order to build a recurrence relation.

Proposition 10. $\texttt{len}_{i-1} \le \texttt{len}_i \le \texttt{len}_{i-1} + 1$. *In addition, if* $\texttt{len}_i = \texttt{len}_{i-1} + 1$, *then every increasing subsequence of* $a[1 \cdots i]$ *with length* $\texttt{len}_i$ *must end at* $a[i]$.

Proof. $\texttt{len}_i \ge \texttt{len}_{i-1}$ is trivial, as every increasing subsequence of $a[1 \cdots i-1]$ is also an increasing subsequence of $a[1 \cdots i]$. It is also easy to see that $\texttt{len}_i \le \texttt{len}_{i-1} + 1$: if there is an increasing subsequence of $a[1 \cdots i]$ with length at least $\texttt{len}_{i-1} + 2$, there must exist an increasing subsequence of $a[1 \cdots i-1]$ with length at least $\texttt{len}_{i-1} + 1$, which contradicts to that the longest increasing subsequence of $a[1 \ldots i-1]$ has length $\texttt{len}_{i-1}$. For the second part of the proposition, if there exists an increasing subsequence with length $\texttt{len}_{i-1} + 1$ in $a[1 \cdots i]$ that does not end at $a[i]$, then this is also a subsequence of $a[1 \cdots i-1]$, which contradicts to that the longest increasing subsequence of $a[1 \ldots i-1]$ has length $\texttt{len}_{i-1}$. □

We then discuss two cases: 1) $a[i] > F(i-1, \texttt{len}_{i-1})$ and 2) $a[i] \le F(i-1, \texttt{len}_{i-1})$.

Case 1. In this case, there exists an increasing subsequence of $a[1 \cdots i-1]$ with length $\texttt{len}_{i-1}$ that ends at a value less than $a[i]$. Then, appending $a[i]$ to this subsequence yields a subsequence of $a[1 \cdots i]$ with length $\texttt{len}_{i-1} + 1$. By Proposition 10, this implies $\texttt{len}_i = \texttt{len}_{i-1} + 1$, and all increasing subsequences of $a[1 \cdots i]$ with length $\texttt{len}_i$ must end at $a[i]$. Therefore, $F(i, \texttt{len}_i) = a[i]$.
In addition, for each $j = 1, \ldots, \texttt{len}_{i-1}$, we have $F(i, j) = F(i-1, j)$. To see this, Proposition 9 implies that $F(i-1, j) \le F(i-1, \texttt{len}_{i-1}) < a[i]$. For each $j = 1, \ldots, \texttt{len}_{i-1}$, there exists an increasing subsequence of $a[1 \cdots i-1]$ with length j that ends at a value of $F[i-1, j]$. Since this is also an increasing subsequence of $a[1 \cdots i]$, we have $F[i, j] \le F[i-1, j]$. Suppose for the sake of contradiction that $F(i, j) < F(i-1, j)$. There exists an increasing subsequence of $a[1 \ldots i]$ with length j that ends at a value less than $F(i-1, j)$. Since we have seen $a[i] > F(i-1, \texttt{len}_{i-1}) \ge F(i-1, j)$, this increasing subsequence must not end at $a[i]$. As a result, this subsequence is also a subsequence of $a[1 \cdots i-1]$, contradicting to that the minimum ending value of a subsequence of $a[1 \cdots i-1]$ with length j is $F(i-1, j)$.
In conclusion, for Case 1, we have $\texttt{len}_i = \texttt{len}_{i-1} + 1$, $F(i, \texttt{len}_i) = a[i]$ and $F(i, j) = F(i-1, j)$ for each $j = 1, \ldots, \texttt{len}_{i-1}$.

Case 2. Firstly, we show that $\texttt{len}_i = \texttt{len}_{i-1}$. Suppose $\texttt{len}_i > \texttt{len}_{i-1}$. Proposition 10 implies $\texttt{len}_i = \texttt{len}_{i-1} + 1$ and all increasing subsequences of $a[1 \cdots i]$ with length $\texttt{len}_i$ must end at $a[i]$. If we truncate $a[i]$ from each of those increasing subsequences, each of those subsequences will be a subsequence of $a[1 \cdots i-1]$ with length $\texttt{len}_{i-1}$ and will end at a value strictly less than $a[i]$. Since $a[i] \le F(i-1, \texttt{len}_{i-1})$, each of those subsequences with length $\texttt{len}_{i-1}$ ends at a value strictly less than $F(i-1, \texttt{len}_{i-1})$, which contradicts to our definition of $F(\cdot, \cdot)$.

Secondly, let j^* be the minimum index such that $F(i-1, j^*) \ge a[i]$. Proposition 9 implies that $F(i-1, j) > a[i]$ for all $j > j^*$ and that we can find j^* by a binary search. We will show that

$$F(i,j) = \begin{cases} F(i-1,j) & \text{if } j < j^* \quad (a) \\ a[i] & \text{if } j = j^* \quad (b) \\ F(i-1,j) & \text{if } j > j^* \quad (c) \end{cases}$$

Proof of (a). By our definition of j^*, we have $F(i-1, j) < a[i]$. We first have $F(i, j) \le F(i-1, j)$, as every increasing subsequence of $a[1 \cdots i-1]$ with length j is also an increasing subsequence of $a[1 \cdots i]$. Consider an increasing subsequence of $a[1 \cdots i]$ with length j that has the minimum ending value $F(i, j)$. If this subsequence ends at $a[i]$, we have $F(i, j) = a[i] > F(i-1, j)$, which contradicts to $F(i, j) \le F(i-1, j)$. Thus, this subsequence must not end at $a[i]$. Then, this is also an increasing subsequence of $a[1 \cdots i-1]$ which ends at $F(i-1, j)$, and we have $F(i, j) = F(i-1, j)$.

Proof of (b). Let S be an increasing subsequence of $a[1 \cdots i-1]$ with length j^* that ends at $F(i-1, j^*)$. Since $F(i-1, j^*-1) < a[i]$ and $F(i-1, j^*) \ge a[i]$, we have $S[j^*-1] < a[i]$ and $S[j^*] \ge a[i]$. Consider the subsequence S' obtained by replacing the last number of S by $a[i]$. This is an increasing subsequence of $a[1 \cdots i]$ with length j^* that ends at $a[i]$. Thus, $F(i, j) \le a[i]$. On the other hand, for an arbitrary increasing subsequence of $a[1 \cdots i]$ with length j^*, if it does not end at $a[i]$, then ending value is $F(i-1, j^*)$, which is at least $a[i]$. Therefore, we also have $F(i, j) \ge a[i]$.

Proof of (c). It suffices to show that any increasing subsequence of $a[1 \cdots i]$ with length j must not end at $a[i]$. First of all, since $j > j^*$, we have $F(i-1, j-1) \ge F(i-1, j^*) \ge a[i]$. If there exists an increasing subsequence of $a[1 \cdots i]$ with length j that ends at $a[i]$, then the $(j-1)$-th entry of this subsequence should be strictly less than $a[i]$. We have found an increasing subsequence of $a[1 \cdots i-1]$ with length $j-1$ that ends at a value strictly less than $a[i]$, contradicting to that $F(i-1, j-1) \ge a[i]$.

When implementing this algorithm, we can omit the first argument of $F(\cdot, \cdot)$. The algorithm is presented in Algorithm 2.

The time complexity for Algorithm 2 is $O(n \log n)$: for each of the n iterations, Case 1 can be handled in a constant time, and Case 2 can be handled in $O(\log n)$ time since a binary search requires $O(\log n)$ time.

Algorithm 2 A better dynamic programming algorithm for longest increasing subsequence

Input: $a[1 \cdots n]$

Output: the length of the longest increasing subsequence of a

initialize F such that $F(1) = a[1]$ and $F(2) = \cdots = F(n) = \infty$
`len` $\leftarrow 1$
for $i = 2, \ldots, n$:
    if $a[i] > F($`len`$)$: // Case 1
        `len` $\leftarrow$ `len` $+1$
        $F($`len`$) \leftarrow a[i]$
    else // Case 2
        find $j^* \leftarrow \operatorname{argmin}_j \{F(j) \geq a[i]\}$ by a binary search
        $F(j^*) \leftarrow a[i]$
    endif
endfor
return `len`

4 Test Solidity of Eggs

Problem 11. We have m identical eggs and a building with n levels. The *solidity* of those eggs is defined by the maximum number $s \in \{0, 1, \ldots, n\}$ such that the egg remains intact when being thrown at the s-th level of the building. In particular, if the egg is broken when being thrown at the first level, its solidity is 0. We would like to decide the solidity of those identical eggs by performing a sequence of experiments. In each experiment, an egg is thrown at a chosen particular level of the building. If this egg is intact after the experiment, we can reuse it in the next experiment; otherwise, this egg can no longer be used. As a requirement, the sequence of experiments must be able to decide the solidity in the worst case. In this problem, we are to decide the number of experiments needed to decide the solidity of the eggs in the worst case.

Let us first look at some examples to understand the problem better.
Suppose $m = 1$. The number of experiments needed is n. In fact, the only strategy is sequentially throwing the egg from levels $1, 2, \ldots, n$. If the egg is broken after being thrown at the i-th level, we know the solidity is $i - 1$. We can prove by induction that this is the only strategy that can guarantee us to know the solidity of the egg. For the base step, our first experiment must be throwing the egg at the first level. If we throw the egg at level $i > 1$ instead, there is a possibility that the egg breaks. In this case, we have no more eggs for experiments, and the solidity can be any of $0, 1, \ldots, i - 1$. Thus, there is a possibility we fail to decide the solidity of the egg. For the inductive step, suppose the first i experiments must be throwing the eggs at level $1, \ldots, i$ respectively. If the next egg is not thrown from level $i + 1$, it must be thrown at level $i' > i + 1$

(there is no point to redo any experiments for throwing the eggs from level $1,\dots,i$). However, there is a possibility that the egg breaks, and the solidity can be any of $i+1, i+2,\dots,i'$, which is undecided. Thus, the $(i+1)$-th experiment must be throwing the egg from the $(i+1)$-th level, which proves the inductive step. After showing that sequentially throwing the egg from levels $1,2,\dots,n$ is the only strategy, it is easy to see that the number of experiments needed to decide the solidity is n in the worst case: there is a possibility that the solidity is exactly n, in which case we need to perform all those n experiments.

Suppose $m = 2$ and $\sqrt{n}$ is an integer. We can test the solidity by only $O(\sqrt{n})$ experiments. Firstly, we sequentially test the first egg at levels $\sqrt{n}, 2\sqrt{n},\dots,n$, and we can find the number i^* such that the egg is intact at level $i^*\sqrt{n}$ and broken at level $(i^*+1)\sqrt{n}$. After that, we use the second egg to test for level $i^*\sqrt{n}+1, i^*\sqrt{n}+2,\dots,(i^*+1)\sqrt{n}-1$. This can safely decide the solidity of the eggs.

Suppose $m \geq \lceil \log_2(n+1) \rceil$. The number of the eggs is sufficient for us to perform a binary search. Therefore, we can decide the solidity of the eggs by $\lceil \log_2(n+1) \rceil$ experiments. It is also easy to see that this is the best we can do.

4.1 A Dynamic Programming Algorithm

Let $F(i,j)$ be the number of experiments needed to decide the solidity if we have a total of i eggs and j levels. We need to find $F(m,n)$. We have seen in the first example that $F(1,j) = j$ for each $j = 1,\dots,n$. We set $F(i,0) = 0$. It now remains to find a recurrence relation for $F(i,j)$.

Suppose in the first experiment we throw an egg at level k. If the egg is broken, we know the solidity is one of $0,1,\dots,k-1$. We have $i-1$ eggs now, and we can assume without loss of generality that the building has only $k-1$ levels. Therefore, the total number of experiments needed (after this experiment) is $F(i-1,k-1)$.

If the egg is intact, we know the solidity is one of $k, k+1,\dots,j$, and we still have i eggs. We can assume we now have a new building with $j-k$ levels. Then, throwing an egg at the k'-th level in this new building is equivalent as throwing it at the $(k'+k)$-th level in the original building. In particular, level k in the original building corresponds to level 0 in the new building. The the total number of experiments needed (after the first experiment) is $F(i,j-k)$.

Since we are considering the worst case, we need to be able to handle both scenarios (broken and intact). Thus, we need to take the *maximum* of $F(i-1,k-1)$ and $F(i,j-k)$. Therefore, the recurrence relation is

$$F(i,j) = \min_{k=1,\dots,j} \left\{ \max\{F(i-1,k-1), F(i,j-k)\} + 1 \right\}.$$

This gives us a dynamic programming algorithm with time complexity $O(mn^2)$: the total number of states is mn, and each state requires $O(n)$ to process the recurrence relation.

This algorithm can be optimized to $O(n^2 \log n)$. We have seen that the number of experiments needed is $\lceil \log_2(n+1) \rceil$ when $m \geq \lceil \log_2(n+1) \rceil$. Therefore, we can let the algorithm directly output $\lceil \log_2(n+1) \rceil$ when

$m \geq \lceil \log_2(n+1) \rceil$. We only need dynamic programming for $m < \lceil \log_2(n+1) \rceil$. By this optimization, we have only $O(n \log n)$ states, and the overall time complexity is $O(n^2 \log n)$.
This algorithm can be further optimized. We prove the following proposition first.

Proposition 12. $F(i, j) \geq F(i, j-1)$.

Proof. We have a sequence of $F(i, j)$ experiments to decide the solidity when the building has a total of j levels. This same sequence of experiments can be used to decide the solidity if the building only have a total of $j-1$ levels. In addition, if there is an experiment that throws an egg at level j in this sequence, we can just remove this experiment from the sequence. Therefore, the number of experiments needed for the case with $j-1$ levels is at most $F(i, j)$. □

With this proposition, we know that $F(i-1, k-1)$ is weakly increasing in k, and $F(i, j-k)$ is weakly decreasing in k. From Fig. 3, it is easy to see that $\max\{F(i-1, k-1), F(i, j-k)\}$ is minimized when $F(i-1, k-1)$ and $F(i, j-k)$ are approximately equal (notice that k only takes integer values, so there may not exist a k such that the two functions are equal). More precisely, if there exists a k^* such that $F(i-1, k^*-1) = F(i, j-k^*)$, $\max\{F(i-1, k-1), F(i, j-k)\}$ is minimized at k^*; otherwise, $\max\{F(i-1, k-1), F(i, j-k)\}$ is minimized at either $k^\dagger$ or $k^\dagger + 1$, where $k^\dagger$ is defined such that $F(i-1, k^\dagger - 1) < F(i, j-k^\dagger)$ and $F(i-1, (k^\dagger + 1) - 1) > F(i, j-(k^\dagger + 1))$. Since $F(i-1, k-1) - F(i, j-k)$ is weakly increasing in k, we can use a binary search to find k^* or $k^\dagger$. Thus, it only requires $O(\log n)$ time to process the recurrence relation. The overall time complexity is then $O(n(\log n)^2)$.

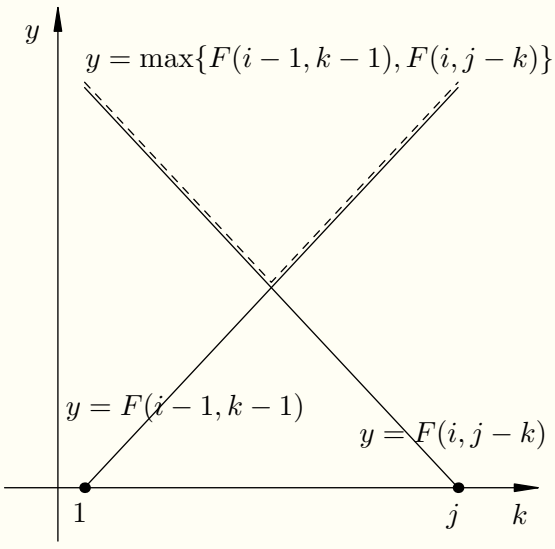

Figure 3: $\max\{F(i-1, k-1), F(i, j-k)\}$ is minimized when $F(i-1, k-1)$ and $F(i, j-k)$ are approximately equal.

4.2 A Better Dynamic Programming Algorithm

Although we have optimized the dynamic programming algorithm in the previous section, there exists a better dynamic programming algorithm that is designed in a different way.

Let $G(i, j)$ be the *maximum total number of levels* of the building such that we can find the solidity of the eggs with j eggs and i experiments. To solve Problem 11, we need to find $k \in \mathbb{Z}^+$ such that $G(k-1, m) < n$ and $G(k, m) \geq n$. For initial conditions, we have $G(0, j) = 0$ for all j (if we do not need to test at all, then the number of levels must be 0), and $G(i, 0) = 0$ for all i (if we do not have any egg, the number of levels must be 0).

To find a recurrence relation, suppose we have j eggs and we only need to perform i experiments to decide the solidity. Suppose we have a total of n levels, and the first experiment in an optimal strategy is to throw the egg at the k-th level. If the egg breaks, we know that the solidity is one of $0, 1, \ldots, k-1$; otherwise, we know the solidity is one of $k, k+1, \ldots, n$. We have $i-1$ experiments left in both cases. We have $j-1$ eggs left in the first case, and we have j eggs left in the second case. To find $G(i, j)$, we need to maximize the values for both $k-1$ and $n-k$.

For $k-1$, we need to make sure that the solidity can still be decided with $j-1$ eggs and $i-1$ experiments if the total number of the levels is $k-1$. The maximum possible value for $k-1$ is $G(i-1, j-1)$.

For $n-k$, we need to make sure that the solidity can still be decided with j eggs and $i-1$ experiments if the total number of levels is $n-k$. The maximum possible value for $n-k$ is $G(i-1, j)$.

Putting together, the maximum possible value of n is

$$n = (n-k) + (k-1) + 1 = G(i-1, j) + G(i-1, j-1) + 1.$$

Thus, we have the recurrence relation

$$G(i, j) = G(i-1, j) + G(i-1, j-1) + 1.$$

To analyze the time complexity for this algorithm, first notice that we only need to worry about those $G(i, j)$ such that $j \leq m < \lceil \log_2(n+1) \rceil$. The total number of states in this dynamic programming algorithm is then $O(\log n)$ times the minimum number of k such that $G(k, m) \geq n$, and each state requires a constant time to compute by the recurrence relation. Now we find an upper bound on the minimum number of k satisfying $G(k, m) \geq n$.

Proposition 13. $G(i, j) \geq \binom{i}{\min\{i, j\}}$.

Proof. We prove this by induction. For the base step, we can compute by the recurrence relation that $G(1, j) = 1$ for all j. We also have $\binom{1}{\min\{1, j\}} = 1$. The inequality holds.

For the inductive step, suppose the inequality holds for each of $i = 1, \ldots, i'$ and all j. We aim to show that the inequality holds for $i = i' + 1$ and all j. If $j \geq i' + 1$, we have $G(i' + 1, j) > 1$ (clearly, with $i' + 1$

experiments and at least as many eggs as this, we can decide the solidity for more than 1 level of the building), and we have $\binom{i'+1}{\min\{i'+1,j\}} = 1$. The inequality holds. If $j < i'+1$, we have $\min\{i', j\} = j$, and by the recurrence relation and induction hypothesis,

$$G(i'+1, j) = G(i', j) + G(i', j-1) + 1 > \binom{i'}{j} + \binom{i'}{j-1} = \binom{i}{j},$$

where the last equality is by Pascal's identity. □

With this proposition, if we let $\phi(n)$ be the minimum integer k such that $\binom{k}{m} \geq n$, the time complexity of the algorithm is $O(\phi(n) \log n)$. Notice that $\phi(n)$ grows slowly in n. For m as small as 2, we have $\phi(n) = \sqrt{n}$. It grows even slower for larger m. The only case when $\phi(n)$ grows linearly is when $m = 1$. However, as we have seen at the beginning, the number of experiments needed for only 1 egg is exactly n, and we do not need to run the dynamic programming algorithm to figure this out.
Combining all these, the algorithm is presented in Algorithm 3.

Algorithm 3 The clever dynamic programming algorithm for Problem 11

Input: $m, n \in \mathbb{Z}^+$

Output: the number of experiments needed

```
if m ≥ ⌈log_2(n+1)⌉, return ⌈log_2(n+1)⌉
if m = 1, return n
set G(0, j) ← 0 for each of j = 0, 1, ..., m
for i = 1, 2, 3, ..., ∞:
    G(i, 0) = 0
    for j = 1, ..., m:
        G(i, j) ← G(i−1, j) + G(i−1, j−1) + 1
    endfor
    if G(i, m) ≥ n, break
endfor
return i
```

5 Product of Sets, Quadrangle Inequality

Definition 14. Given two sets L_1 and L_2, the *product* of L_1 and L_2 is defined as $L_1 \times L_2 = \{(a_1, a_2) \mid a_1 \in L_1, a_2 \in L_2\}$.

In this section, we assume computing the product $L_1 \times L_2$ requires $|L_1| \cdot |L_2|$ operations. Suppose we want to compute the product of $L_1 \times L_2 \times L_3$, and let $|L_1| = n_1$, $|L_2| = n_2$ and $|L_3| = n_3$. If we compute the product in

the left-to-right order, the number of operations required is $n_1 n_2 + n_1 n_2 n_3$. If we compute $L_2 \times L_3$ first and then compute $L_1 \times (L_2 \times L_3)$, the number of operations required is $n_2 n_3 + n_1 n_2 n_3$. We can see that different orders of computing a chain of set products require different number of operations. This is similar to the case of chain matrix multiplication we have seen in Lecture 8.

Problem 15. Given $L_1, \ldots, L_n$, find the minimum number of operations to compute $L_1 \times \cdots \times L_n$.

Let $|L_i| = n_i$ for each $i = 1, \ldots, n$. Let $c(i, j)$ be the minimum number of operations to compute $L_i \times \cdots \times L_j$. We have seen in Lecture 8 the following recurrence relation holds:

$$c(i, j) = w(i, j) + \min_{k:i<k\leq j} \{c(i, k-1) + c(k, j)\},$$

where $w(i, j) = n_i n_{i+1} \cdots n_j$. We have n^2 states and each state requires $O(n)$ time to compute the recurrence relation. Therefore, the overall time complexity is $O(n^3)$. In this lecture, we will see how to improve this to $O(n^2)$. This result is due to Yao [1980].

The two key properties of $w(\cdot, \cdot)$ that enable this improvement are

- **Monotonicity**: $w(i, j) \leq w(i', j')$ if $[i, j] \subseteq [i', j']$, and
- **Quadrangle Inequality (QI)**: $w(i, j) + w(i', j') \leq w(i', j) + w(i, j')$ for $i \leq i' \leq j \leq j'$.

A geometric interpretation of QI is available in Fig. 4. Perhaps "anti-quadrangle inequality" is a more intuitive name for this inequality: the inequality is of the opposite direction to its geometric counterpart.

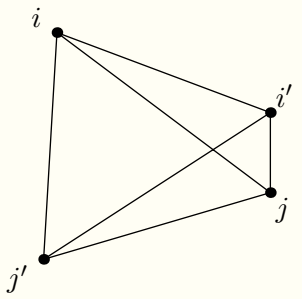

Figure 4: A geometric interpretation of the quadrangle inequality.

Theorem 16. *If $w(\cdot, \cdot)$ satisfies monotonicity and QI, then $c(1, n)$ can be computed in time $O(n^2)$.*

Before proving this theorem, we first verify that monotonicity and QI hold for w. The monotonicity is obvious. For QI, let $a = n_i \cdots n_{i'-1}$, $b = n_{i'} \cdots n_j$, and $c = n_{j+1} \cdots n_{j'}$. Then QI becomes

$$ab + bc \leq b + abc.$$

This is true since

$$0 \leq b(a-1)(c-1).$$

Theorem 16 follows from the following two lemmas. In the remaining part of this section, we use $c_k(i, j)$ to denote $w(i, j) + c(i, k-1) + c(k, j)$.

Lemma 17. *if $w(\cdot,\cdot)$ satisfies QI and monotonicity, then $c(\cdot,\cdot)$ also satisfies QI.*

Proof. The proof is by induction on the length $\ell = j' - i$ of the "long side" of the quadrangle inequality

$$c(i,j) + c(i',j') \le c(i',j) + c(i,j') \qquad \text{for } i \le i' \le j \le j'. \tag{1}$$

First not that (1) is trivial when $i = i'$ or $j = j'$. Therefore, (1) is true when $\ell \le 1$. Inductively, consider two cases: A) $i < i' = j < j'$, and B) $i < i' < j < j'$. (See Fig. 5.)

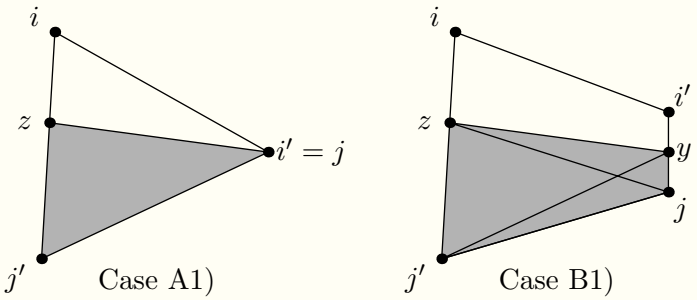

Figure 5: The proof of Lemma 17

Case A). $i < i' = j < j'$.
In this case, (1) becomes the (inverse) trangle inequality:

$$c(i,j) + c(j,j') \le c(i,j') \qquad \text{for } i < j < j'. \tag{2}$$

Suppose $c(i,j')$ is minimized at $k = z$; that is, $c(i,j') = c_z(i,j')$ where we recall that we use $c_k(i,j)$ to denote $w(i,j) + c(i,k-1) + c(k,j)$. There are two symmetric subcases.
Case A1). $z \le j$.
We have $c(i,j) \le c_z(i,j) = w(i,j) + c(i,z-1) + c(z,j)$. Therefore,

$$\begin{aligned} c(i,j) + c(j,j') &\le w(i,j) + c(i,z-1) + c(z,j) + c(j,j') \\ &\le w(i,j') + c(i,z-1) + c(z,j') \\ &= c(i,j'), \end{aligned}$$

where we used the monotonicity of w and the induction hypothesis (2) at $z \le j \le j'$.
Case A2). $z \ge j$. This is symmetric with A1), with all the intervals reversed.
Case B). $i < i' < j < j'$.
Assume the two terms on the right-hand side of (1) achieve their values at $k = y$ and $k = z$ respectively. That is,

$$c(i',j) = c_y(i',j), \qquad \text{and} \qquad c(i,j') = c_z(i,j').$$

We again look at two symmetric subcases.

Case B1). $z \le y$.

We have

$$c(i', j') \le c_y(i', j'), \qquad \text{and} \qquad c(i, j) \le c_z(i, j).$$

Adding them up, we obtain

$$\begin{aligned} & c(i, j) + c(i', j') \\ \le & c_z(i, j) + c_y(i', j') \\ = & w(i, j) + w(i', j') + c(i, z-1) + c(z, j) + c(i', y-1) + c(y, j'). \end{aligned}$$

Applying QI of w and the induction hypothesis (1) at the points $z \le y < j < j'$, we have

$$\begin{aligned} & c(i, j) + c(i', j') \\ \le & w(i', j) + w(i, j') + c(i, z-1) + c(i', y-1) + c(y, j) + c(z, j') \\ \le & c_y(i', j) + c_z(i, j') \\ = & c(i', j) + c(i, j'). \end{aligned}$$

Case B2). $z \ge y$. This again reduced to B1) when all intervals are reversed. □

Let us use $K_c(i, j)$ to denote $\max\{k \mid c_k(i, j) = c(i, j)\}$; so $K_c(i, j)$ is the largest index k where the minimum is achieved in (1). (We define $K_c(i, i) = i$.)

Lemma 18. *If the function $c(\cdot, \cdot)$ satisfies QI, then we have*

$$K_c(i, j) \le K_c(i, j+1) \le K_c(i+1, j+1) \qquad \text{for } i \le j. \tag{3}$$

Proof. It is trivially true when $i = j$. Therefore, we assume $i < j$ from now on. To prove the first inequality $K_c(i, j) \le K_c(i, j+1)$, we show that for $i < k \le k' \le j$,

$$c_{k'}(i, j) \le c_k(i, j) \qquad \Longrightarrow \qquad c_{k'}(i, j+1) \le c_k(i, j+1). \tag{4}$$

Taking the QI of c at $k \le k' \le j < j+1$, we have

$$c(k, j) + c(k', j+1) \le c(k', j) + c(k, j+1).$$

Adding $w(i, j) + w(i, j+1) + c(i, k-1) + c(i, k'-1)$ to both sides, we get

$$c_k(i, j) + c_{k'}(i, j+1) \le c_{k'}(i, j) + c_k(i, j+1),$$

from which (4) follows. Similarly, the second inequality $K_c(i, j+1) \le K_c(i+1, j+1)$ follows from the QI of c at $i < i+1 \le k \le k'$. □

Lemma 18 says that the matrix $K_c(i,j)$ is non-decreasing along each row and column. Let $\delta = j - i$. We use the following topological order for the dynamic programming: at the outer for-loop, we enumerate $\delta = 0, 1, \ldots, n-1$; at the inner for-loop, we enumerate $i = 1, 2, \ldots, n-\delta$. To compute $c(i,j) = c(i, i+\delta)$ by the recurrence relation $c(i,j) = w(i,j) + \min_{k:i<k\le j}\{c(i,k-1) + c(k,j)\}$, Lemma 18 suggests that we only need to search for k from $K_c(i, i+(\delta-1))$ to $K_c(i+1, i+1+(\delta-1))$ (instead of from i to $j = i+\delta$). Notice that the values for $K_c(i, i+(\delta-1))$ and $K_c(i+1, i+1+(\delta-1))$ have already been computed at the previous iteration of the outer for-loop.

To analyze the time complexity, we compute the number of the values for k that have been considered in each iteration of the outer for-loop. We have see that, for each i, we need to search for k from $K_c(i, i+(\delta-1))$ to $K_c(i+1, i+1+(\delta-1))$, which means that we need to search for $K_c(i+1, i+1+(\delta-1)) - K_c(i, i+(\delta-1))$ different values. Therefore, the total number of values searched is

$$\sum_{i=1}^{n-\delta} (K_c(i+1, i+1+(\delta-1)) - K_c(i, i+(\delta-1))) = K_c(n-\delta+1, n) - K_c(1, \delta) \le n.$$

For the first equality above, notice that most of the K_c terms are cancelled in the summation. Since it takes $O(n)$ time to process each iteration of the outer for-loop, the overall time complexity is $O(n^2)$.

参考文献

F Frances Yao. Efficient dynamic programming using quadrangle inequalities. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 429–435, 1980. 13