Mathematical Modeling Skill

The computer is a tool to solve problems in our daily life and work, and the duty of programmers is to abstract mathematical models from real problems and solve them with programming languages. Therefore, the mathematic modeling skill is important for software engineers.

Candidates should select the appropriate data structure to model a problem. The problems in our life and work are various, but there are only a few common data structures. Candidates could make choices according to properties of the problem as well as performance and difficulties of development. For instance, candidates may model a circle with a set of numbers as a looped linked list in order to solve the problem "Last Number in a Circle" (Question 96).

Candidates should also analyze the hidden rules or patterns inside the problem and implement them with appropriate algorithms. For example, the interview problem of "Probabilities of Dice Points" (Question 95) is essentially equivalent to calculate a sequence of $f(n)=f(n-1)+f(n-2)+f(n-3)+f(n-4)+f(n-5)+f(n-6)$. Additionally, in order to find the minimum number of moves to sort a set of cards, we have to find the longest increasing subsequence at first. If candidates are familiar with algorithms, they should be able to find the longest increasing subsequence with dynamic programming.

Probabilities of Dice Points

Question 95 Given a number n, which stands for n dice, please print probabilities of all possible sums of dice points.

For example, if there are two dice, there are two ways to get three points: (1) one die has one point, and the other has two points; and (2) one die has two points, and the other has one. However, there is only one way to get twelve points, with six points on each die. Therefore, the possibility to get three points is higher than to get twelve points with two dice.

A die is often a rounded cube, with each of its faces showing a different number from 1 to 6. The minimum sum of n dice is n, and the maximum is $6n$. Additionally, points shown by n dice have 6^n permutations.

A function $f(i)$ is defined for the number of situations for dice to show i points. When we get $f(i)(0<n\leq 6n)$, the probability to get i points with n dice is $f(i)/6^n$.

When there is only one die, $f(1), f(2), \ldots, f(6)$ are initialized to 1. A die is added at each step. If there are $f(i)$ situations to get i points at a certain step, there are $f(i)=f(i-1)+f(i-2)+f(i-3)+f(i-4)+f(i-5)+f(i-6)$ with one more die at the next step. This is a typical recursive equation. However, the complexity grows exponentially if it is solved recursively because there are overlapped calculations. A better choice is based on iteration. More discussions about the efficiency difference between recursion and iteration are available in the section *Fibonacci Sequence*.

The iterative solution can be implemented as shown in Listing 8-18.

Listing 8-18. *C++ Code to Get Probabilities of Dice Points*

```
void PrintProbability(int number) {

if(number < 1)

    return;


const int maxValue = 6;

int* pProbabilities[2];

pProbabilities[0] = new int[maxValue * number + 1];

pProbabilities[1] = new int[maxValue * number + 1];

for(int i = 0; i < maxValue * number + 1; ++i) {

    pProbabilities[0][i] = 0;

    pProbabilities[1][i] = 0;

}


int flag = 0;

for (int i = 1; i <= maxValue; ++i)

    pProbabilities[flag][i] = 1;


for (int k = 2; k <= number; ++k) {

    for(int i = 0; i < k; ++i)

        pProbabilities[1 - flag][i] = 0;


    for (int i = k; i <= maxValue * k; ++i) {

        pProbabilities[1 - flag][i] = 0;

        for(int j = 1; j <= i && j <= maxValue; ++j)

            pProbabilities[1 - flag][i] += pProbabilities[flag][i - j];

    }


    flag = 1 - flag;

}


double total = pow((double)maxValue, number);

for(int i = number; i <= maxValue * number; ++i) {

    double ratio = (double)pProbabilities[flag][i] / total;

    printf("%d: %e\n", i, ratio);

}


delete[] pProbabilities[0];

delete[] pProbabilities[1];

}
```

Two arrays, `pProbabilities[0]` and `pProbabilities[1]`, are defined in the code above. The i^{th} element in these two arrays stands for the number of situations for dice to show i points. These two arrays are swapped (modifying the variable `flag`) for the next step to add one more die.

Source Code:

```
095_DicesProbability.cpp
```

Test Cases:

- Normal Test Cases (Probabilities of points for 1, 2, 3, 4 dice)
- Boundary Test Cases (Input 0)
- Performance Test Cases (Somewhat big numbers for the count of dice, such as 11)

Last Number in a Circle

Question 96 A circle is composed of n numbers, 0, 1, …, n-1. The m^{th} number is removed every time, counting from the number 0 for the first removal. What is the last number left?

For example, a circle is composed of five numbers 0, 1, 2, 3, and 4 (Figure 8-3). If the third number is removed from the circle repeatedly, four numbers are deleted in the sequence of 2, 0, 4, and 1, and the last number remaining in the circle is 3.

This is the classic Josephus problem, and there are two solutions for it. The first solution is to simulate a circle with a looped list, and the other one is to analyze the pattern of deleted numbers at each step.

Simulating a Circle with a Looped List

Since this problem is about a circle, an intuitive solution is to simulate the circle with a looped list. A loop with n nodes is created first, and then the m^{th} node is deleted from it at each step.

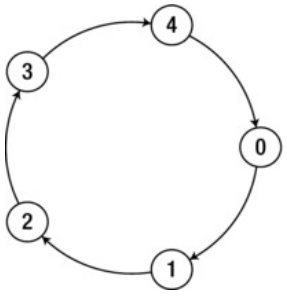

Figure 8-3. *A circle with five numbers*

The code in Listing 8-19 is based on the type `list` in the C++ standard template library. An instance of `std::list` is not looped. When the iterator reaches the tail node, it returns back to the head node, so the list is traversed in a circled sequence. This solution can be implemented as the code in C++ as shown in Listing 8-19.

Listing 8-19. *C++ Code for the Last Number Remaining in a Circle (Version 1)*

```
int LastRemaining_Solution1(unsigned int n, unsigned int m) {

if(n < 1 || m < 1)

    return -1;


list<int> numbers;

for(unsigned int i = 0; i < n; ++ i)

    numbers.push_back(i);


list<int>::iterator current = numbers.begin();

while(numbers.size() > 1) {

    for(int i = 1; i < m; ++ i) {

        current ++;

        if(current == numbers.end())

            current = numbers.begin();

    }


    list<int>::iterator next = ++ current;

    if(next == numbers.end())

        next = numbers.begin();


    -- current;

    numbers.erase(current);

    current = next;

}


return *(current);

}
```

This solution has to traverse the circled list many times. It costs O(m) time to delete a node, so its overall complexity is O(mn) on a circle with n nodes originally. Additionally, it consumes O(n) space to create a looped list. Let's explore a more effective solution.

Analyzing the Pattern of Deleted Numbers

A function $f(n,m)$ is defined for the last remaining number in a circle with n numbers from 0 to n-1 when the m^{th} number is deleted at every step.

Among the n numbers in the circle, the first deleted number is (m-1)%n. (m-1)%n is denoted as k for simplicity. When k is deleted, there are n-1 numbers (0, 1, …, k-1, k+1, …, n-1) left, and it counts from k+1 to delete the next m^{th} number. Therefore, numbers can be reordered as k+1, …, n-1, 0, 1, …, k-1 in order to place k+1 as the first number. The pattern of the reordered sequence is different from the original sequence since it does not start from 0. Another function $f'(n\text{-}1,m)$ is defined for the last remaining number in these n-1 numbers, and $f(n,m)=f'(n\text{-}1,m)$.

The sequence with n-1 numbers k+1, …, n-1, 0, 1, …, k-1 can be projected to a new sequence from 0 to n-2:

k+1 → 0
k+2 → 1
…
n-1 → n-k-2
0 → n-k-1
1 → n-k
…
k-1 → n-2

If the projection is defined as p, $p(x)=(x\text{-}k\text{-}1)\%n$. The reversed projection is $p^{-1}(x)=(x+k+1)\%n$.

Since the projected sequence starts from 0 and has a pattern similar to the original sequence from 0 to n-1, the last number left in the projected sequence with n-1 numbers also follows the function f, and it is $f(n\text{-}1,m)$.

Therefore, the last number remaining in the sequence before the projection is $f'(n\text{-}1,m)=p^{-1}[f(n\text{-}1,m)]=[f(n\text{-}1,m)+k+1]\%n$. Because $k=(m\text{-}1)\%n$, $f(n,m)=f'(n\text{-}1,m)=[f(n\text{-}1),m)+m]\%n$.

We have found a recursive equation to get the last remaining number from n numbers when the m^{th} number is deleted at each step. If there is only one number in the original circle, it is obvious that the last remaining number is 0. Therefore, $f(n,m)$ is defined as follows:

$$f(n,m)=\begin{cases}0 & n=1\\ (f(n-1,m)+m)\%n & n>1\end{cases}$$

The equation can be implemented easily with both recursion and iteration. The code in Listing 8-20 is based on iteration.

Listing 8-20. *C++ Code for the Last Number Remaining in a Circle (Version 2)*

```
int LastRemaining_Solution2(unsigned int n, unsigned int m) {

if(n < 1 || m < 1)

    return -1;


int last = 0;

for (int i = 2; i <= n; i ++)

    last = (last + m) % i;


return last;

}
```

Even though this analysis is very complicated, the implementation code looks quite concise. More importantly, it is more efficient because its time complexity is O(n) and space complexity is O(1).

Source Code:

```
096_LastNumberInCircle.cpp
```

Test Cases:

- Normal Test Cases (The input m is less than n, such as to delete the 3rd number at each step from a circle with 5 numbers originally; the input m is greater than or equal to n, such as to delete the 6th or

7^{th} number at each step from a circle with 6 numbers originally)

- Boundary Test Cases (The circle is empty)
- Performance Test Cases (The inputs *m* and *n* are somewhat big, such as to delete the 997^{th} number at each step from a circle with 4000 numbers originally)

Minimum Number of Moves to Sort Cards

Question 97 You are required to sort *n* cards numbered from 1 to *n*. You can choose one card and move it to any place you want (insert to any place, not swap). Given a sequence, please implement a function to return the minimum move count to sort these cards.
For example, given a sequence {1, 2, 5, 3, 4, 7, 6}, you can move 5 and insert it between 4 and 7, and the sequence becomes {1, 2, 3, 4, 5, 7, 6}. This is one move. If 7 is moved behind 6, the whole sequence gets sorted. Therefore, it needs two steps at least to sort cards in the sequence of {1, 2, 5, 3, 4, 7, 6}.

In order to sort a sequence with minimum number of moves, we first find the longest increasing subsequence, and then it is only necessary to insert numbers out of the longest increasing subsequence into the appropriate places. For example, one of the longest increasing subsequences of {1, 2, 5, 3, 4, 7, 6} is {1, 2, 3, 4, 6}. If the two numbers 5 and 7 are inserted into right places of {1, 2, 3, 4, 6}, the whole sequence is sorted.

Therefore, if we get the length of the longest increasing subsequence (which is not necessarily to be continuous), it is easy to know the minimum number of moves, as shown in Listing 8-21.

Listing 8-21. *Java Code for the Minimum Number of Moves to Sort a Sequence*

```
int minMoveCount(int[] seq) {
return seq.length - longestIncreasingLength(seq);
}
```

It is a classic problem to get the length of the longest increasing subsequence, and there are two solutions available for it.

Based on Dynamic Programming Costing O(n^2) Time

A function $f(i)$ is defined to indicate the maximum length of all subsequences ending with the i^{th} number in the input array (denoted as $A[i]$). We define another function $g(i,j)$, which stands for the maximum length of incremental subsequences ending with $A[i]$, whose preceding number in the incremental subsequence is $A[j]$.

The required output is $max\{f(i)\}$ where $0 \le i < n$ and n is the length of array. We can get $f(i)$ with the following equation:

$$f(i) = max(g(i,j)) \text{ where } g(i,j) = \begin{cases} 1 & A[i] > A[j] \\ f(j)+1 & A[i] < A[j] \end{cases}, 0 \le j <$$

Take the sample sequence {1, 2, 5, 3, 4, 7, 6} as an example. The longest increasing subsequence ending with the third element is the subsequence {1, 2, 5} itself, and the length is 3.

Let's move on to analyze the longest increasing subsequence ending with the fourth element. The increasing subsequence may come from the first element and the length of {1, 3} is 2.

Similarly, the increasing subsequence may come from the second element, and the length of {1, 2, 3} is 3. However, the increasing subsequence cannot come from the third element because {1, 2, 5, 3} is not incremental. Therefore, the maximum length of the increasing subsequence ending with the fourth element is 3.

The maximum lengths of increasing subsequences ending with the last three elements are 4, 5, and 5 respectively. Therefore, the maximum length of all increasing subsequences is 5.

Even though this problem is analyzed recursively, it is more efficient to implement a solution iteratively with an array, as demonstrated in Listing 8-22.

Listing 8-22. *Java Code for the Length of Longest Increasing Subsequence (Version 1)*

```
int longestIncreasingLength(int[] seq){
int len = seq.length;
int[] lookup = new int[len];

for(int i = 0; i < len; ++i) {
    int longestSoFar = 0;
    for(int j= 0; j < i; ++j) {
        if(seq[i] > seq[j] && lookup[j] > longestSoFar)
            longestSoFar = lookup[j];
    }
    lookup[i] = longestSoFar + 1;
}

int longestLength = 0;
for(int i = 0; i < len; ++i) {
    if(lookup[i] > longestLength)
        longestLength = lookup[i];
}

return longestLength;
}
```

The i^{th} number in the auxiliary array `lookup` is for $f(i)$ in the equation above.

It has to compare each number with O(n) numbers ahead of it in an array with n numbers, so it costs O(n^2) to get maximum length of all increasing subsequences.

Based on Binary Search Costing O($n$log$n$) Time

An auxiliary array is utilized to store ending elements of the increasing subsequences while we scan the input sequence and analyze its elements one by one. When the first number 1 of the sequence {1, 2, 5, 3, 4, 7, 6} is scanned, the longest increasing subsequence so far only has one element, and we insert the number 1 at the end into the auxiliary array, and the array becomes {1}.

Similarly, when the second and third numbers are scanned, the longest increasing subsequence so far has three elements and the auxiliary array contains {1, 2, 5}.

The next number to be scanned is 3. The longest increasing subsequence so far has three elements, which are {1, 2, 5} or {1, 2, 3}. At this time, the new end number 3 is not added into the auxiliary array directly, but it replaces the last number 5 because it is less than 5 and so the array becomes {1, 2, 3}.

The reason the number 5 was replaced by 3 is that for the step to insert, the number 4 has no effect as the number 5 is the last number in the subsequence {1, 2, 5}, and the next number 4 cannot be appended to the subsequence because 4 is less than 5. However, when the number is replaced with 3, the array still keeps increasing after the number 4 is inserted, and a longer increasing subsequence {1, 2, 3, 4} with four numbers is found.

When a number in the input sequence is scanned, it is compared with the greatest number in the auxiliary array. If the newly scanned number is greater, it is inserted into the array directly. Otherwise, it replaces the least number of those greater than it in the array in order to allow a wider range of values to be added after it. In order to find the greatest number in the array as well as to find the first number greater than a target value efficiently, the array keeps sorted.

When the next number 7 is scanned, it is inserted into the auxiliary array and the array becomes {1, 2, 3, 4, 7}. And then the array becomes {1, 2, 3, 4, 6} after the last number 6 is scanned. The previous end number 7 is replaced because it is greater than 6.

The maximum length of all increasing subsequences of {1, 2, 5, 3, 4, 7, 6} is five because there are five elements in the auxiliary array {1, 2, 3, 4, 6} finally.

It should be noticed that the auxiliary array is not guaranteed to be the longest increasing subsequence. For example, the auxiliary array is {1, 3, 5, 6} after all numbers in the sequence {2, 4, 1, 5, 6, 3} are handled. The auxiliary array is not the longest increasing subsequence, but its length is the same as the longest increasing subsequence, which is {2, 4, 5, 6}.

Because the auxiliary array is sorted, the binary search algorithm is applied to find the first number greater than a target. This solution can be implemented as shown in Listing 8-23.

Listing 8-23. *Java Code for the Length of Longest Increasing Subsequence (Version 2)*

```
int longestIncreasingLength(int[] seq) {
int len = seq.length;
int[] lookup = new int[len];

lookup[0] = seq[0];
int longestLength = 1;
for(int i = 1; i < len; ++i) {
    if(seq[i] > lookup[longestLength - 1]) {
        longestLength++;
        lookup[longestLength - 1] = seq[i];
    }
    else {
        int low = 0;
        int high = longestLength - 1;
```

```
            while(low != high) {
                int mid= (low + high) / 2;
                if(lookup[mid] < seq[i]) {
                    low = mid + 1;
                }
                else {
                    high = mid;
                }
            }

            lookup[low] = seq[i];
        }
    }

    return longestLength;
}
```

As we know, it takes O(log$n$) time to search in a sorted array with n elements, so the overall time complexity is ($n$log$n$).

Source Code:

097_MinimalMoves.java

Test Cases:

- Normal Test Cases (An arbitrary array with some numbers)
- Boundary Test Cases (There is only one number in the array; the array is increasingly/decreasingly sorted)

Most Profit from Stock

Question 98 Stock prices are stored in an array in the order of date. How do you get the most profit from a sequence of stock prices?
For example, the most profit to be gained from the sequence of ordered stock prices {9, 11, 5, 7, 16, 1, 4, 2} is 11, bought when the price was 5 and sold when the price was 16.

The stock profit is the difference in prices in buying and selling stock. Of course, we can sell stock only after we buy it. A pair is composed of a buying price and a selling price. Therefore, the most profit is the maximum difference of all pairs in a sequence of stock prices. This problem is essentially to get the maximum difference for all pairs in an array.

The naive brute-force solution is quite straightforward. We can get the result for each number minus every number on its left side, and then get the maximum difference after comparisons. Since O(n) minus operations are required for each number in an array with n numbers, the overall time complexity is O(n^2). A brute-force solution usually is not the best one. Let's try to reduce the times for these minus operations.

Based Divide and Conquer

An array is divided into two sub-arrays of the same size. The maximum difference of all pairs occurs in one of the three following situations: (1) two numbers in a pair are both in the first sub-array; (2) two numbers in a pair are both in the second sub-array; or (3) the minuend is the maximum number in the second sub-array, and the subtrahend is the minimum number in the first sub-array.

It is not difficult to get the minimum number in the first sub-array and the maximum number in the first sub-array. How about getting the maximum difference of all pairs inside two sub-arrays? They are actually subproblems of the original problem and we can solve them via recursion. Listing 8-24 contains the sample code of this solution.

Listing 8-24. *C++ Code to Get Pair with Maximum Difference (Version 1)*

```
int MaxDiff_Solution1(int numbers[], unsigned length) {
    if(numbers == NULL && length < 2)
        return 0;

    int max, min;
    return MaxDiffCore(numbers, numbers + length - 1, &max, &min);
}

int MaxDiffCore(int* start, int* end, int* max, int* min) {
    if(end == start) {
        *max = *min = *start;
        return 0x80000000;
    }

    int* middle = start + (end - start) / 2;

    int maxLeft, minLeft;
    int leftDiff = MaxDiffCore(start, middle, &maxLeft, &minLeft);

    int maxRight, minRight;
    int rightDiff = MaxDiffCore(middle + 1, end, &maxRight, &minRight);

    int crossDiff = maxRight - minLeft;

    *max = (maxLeft > maxRight) ? maxLeft : maxRight;
    *min = (minLeft < minRight) ? minLeft : minRight;

    int maxDiff = (leftDiff > rightDiff) ? leftDiff : rightDiff;
    maxDiff = (maxDiff > crossDiff) ? maxDiff : crossDiff;
    return maxDiff;
}
```

With the function `MaxDiffCore`, we get the maximum difference of pairs in the first sub-array (`leftDiff`), and then get the maximum difference of pairs in the second sub-array (`rightDiff`). We continue to calculate the difference between the maximum in the second sub-array and the minimum number in the first sub-array (`crossDiff`). The greatest value of the three differences is the maximum difference of the whole array.

We can get the minimum and maximum numbers, as well as their difference in O(1) time, based on the comparison of minimum and maximum numbers of sub-arrays, so the time complexity of the recursive solution is $T(n)=2T(n/2)+O(1)$. We can demonstrate that its time complexity is O(n) with the Master Theory.

Storing the Minimum Numbers while Scanning

Let's define $diff[i]$ for the difference of a pair whose minuend is the i^{th} number in an array, and the subtrahend corresponding to the maximum $diff[i]$ should be the minimum of all numbers on the left side of the i^{th} number in an array. We can get the minimum numbers on the left side of each i^{th} number in an array while scanning the array once. The code in Listing 8-25 is based on this solution.

Listing 8-25. *C++ Code to Get Pair with Maximum Difference (Version 2)*

```
int MaxDiff_Solution2(int numbers[], unsigned length) {
    if(numbers == NULL && length < 2)
        return 0;

    int min = numbers[0];
    int maxDiff =  numbers[1] - min;

    for(int i = 2; i < length; ++i) {
```

```
        if(numbers[i - 1] < min)
            min = numbers[i - 1];

        int currentDiff = numbers[i] - min;
        if(currentDiff > maxDiff)
            maxDiff = currentDiff;
    }

    return maxDiff;
}
```

It is obvious that its time complexity is O(*n*) since it is only necessary to scan an array with length *n* once. It is more efficient than the first solution on memory consumption, which requires O(log*n*) memory for call stack due to recursion.
Source Code:

```
098_MaximalProfitBuyingSellingStock.cpp
```

Test Cases:

- Normal Test Cases (An arbitrary array with some numbers for stock prices)
- Boundary Test Cases (There is only one number in the array; the array is increasingly/decreasingly sorted)
- Robustness Test Cases (The pointer to the array is `NULL`)