

# Analysis of Algorithms - Assignment 1

Zhenrui Zheng

Chinese University of Hong Kong, Shenzhen  
225040512@link.cuhk.edu.cn

## Contents

<b>Problem 1: Array Selection Algorithm</b>	<b>1</b>
1.1 Optimized Selection Algorithm . . . . .	1
1.2 Recurrence Relation Analysis . . . . .	2
1.3 Complexity Analysis . . . . .	3
<b>Problem 2: Search in Bitonic Sequence</b>	<b>4</b>
<b>Problem 3: Master Theorem</b>	<b>6</b>
<b>Problem 4: Recursion Tree</b>	<b>8</b>
<b>Problem 5: Password Generation Counting</b>	<b>10</b>
5.1 Recurrence Relation Analysis . . . . .	10
5.2 Pseudocode . . . . .	10
<b>Problem 6: Ancient Pyramid Treasure Hunt</b>	<b>11</b>
6.1 State Definition . . . . .	11
6.2 State Transition & Recurrence Relation . . . . .	11
6.3 Boundary Case & Base Case . . . . .	12

# Problem 1: Array Selection Algorithm

## 1.1 Optimized Selection Algorithm

The algorithm described below is the **Median-of-Medians** algorithm, which finds the  $k$ -th smallest element in an unsorted array of  $n$  elements in a recursive style.

Given an array  $A$  of  $n$  distinct elements and an integer  $k$  (where  $1 \leq k \leq n$ ), the following algorithm returns the  $k$ -th smallest element in  $A$ :

1. **Divide into Groups:**

Divide the  $n$  elements of array  $A$  into  $\lceil \frac{n}{5} \rceil$  groups. Each group will contain 5 elements, except possibly the last group, which may contain between 1 and 5 elements.

2. **Find Group Medians:**

For each of the  $\lceil \frac{n}{5} \rceil$  groups, sort the elements within the group and find the median. This can be done with a constant number of comparisons, since each group has at most 5 elements.

3. **Find Median-of-Medians (Pivot Selection):**

Create a new array  $M$  consisting of all the medians found in Step 2. The size of  $M$  is  $\lceil \frac{n}{5} \rceil$ . Recursively apply the algorithm to  $M$  to find its **median**<sup>2</sup>. Let this element be  $x$ , which will serve as the pivot.

4. **Partition:**

Partition the original array  $A$  into three subarrays based on the pivot  $x$ :

- $L = \{a \in A \mid a < x\}$  (elements strictly less than  $x$ )
- $E = \{a \in A \mid a = x\}$  (elements equal to  $x$ )
- $G = \{a \in A \mid a > x\}$  (elements strictly greater than  $x$ )

Let  $|L|$ ,  $|E|$ , and  $|G|$  denote the number of elements in these subarrays, respectively.

5. **Recursive Search:**

Determine which subarray contains the  $k$ -th smallest element and recursively apply the algorithm to that subarray:

- If  $k \leq |L|$ , the  $k$ -th smallest element is in  $L$ . Recursively apply the algorithm to find the  $k$ -th smallest element in  $L$ .
- If  $|L| < k \leq |L| + |E|$ , the  $k$ -th smallest element is  $x$ . Return  $x$ .
- Otherwise,  $k > |L| + |E|$ , the  $k$ -th smallest element is in  $G$ . Recursively apply the algorithm to find the  $(k - |L| - |E|)$ -th smallest element in  $G$ .

---

<sup>1</sup>Some might be curious why is 5 here; this is because 5 is the smallest group size that satisfies linear worst-case complexity. A more detailed discussion can be found in the justification section.

<sup>2</sup>It's worth noting that the **median** here is not necessarily the median of array  $A$ .

## 1.2 Recurrence Relation Analysis

Let  $T(n)$  be the worst-case number of comparisons required by the algorithm described above for an input of size  $n$ .

**1. Divide into Groups:**

This step involves no comparisons, just array manipulation.  $O(1)$ .

**2. Find Group Medians:**

- There are  $\lceil \frac{n}{5} \rceil$  groups.
- Each group has at most 5 elements. Find median of at most 5 elements requires a constant number of comparisons<sup>3</sup>.
- Thus, finding all group medians requires  $O(\lceil \frac{n}{5} \rceil \cdot 1) = O(n)$  comparisons.

**3. Find Median-of-Medians (Pivot Selection):**

We recursively apply the algorithm to the array  $M$  of size  $\lceil \frac{n}{5} \rceil$  to find its median (i.e., the  $\lceil \frac{\lceil \frac{n}{5} \rceil}{2} \rceil$ -th smallest element), which contributes  $T(\lceil \frac{n}{5} \rceil)$  to the recurrence.

**4. Partition:**

Partitioning the  $n$  elements around based on pivot  $x$  takes  $O(n)$  comparisons.

**5. Recursive Search:**

- Consider the pivot  $x$ . It is the median of the  $\lceil \frac{n}{5} \rceil$  group medians.
- At least half of these group medians are less than or equal to  $x$ .  
That is, at least  $\lceil \frac{\lceil \frac{n}{5} \rceil}{2} \rceil$  elements are  $\leq x$ .
- Each such group median  $m_i$  comes from a group of 5 elements, and at least 3 elements in that group are  $\leq m_i$ .
- Therefore, at least  $3 \cdot \lceil \frac{\lceil \frac{n}{5} \rceil}{2} \rceil$  elements in the original array  $A$  are  $\leq x$ .
  - Since  $\lceil \frac{n}{x} \rceil \geq \frac{n}{x}$ , we have  $\lceil \frac{\lceil \frac{n}{5} \rceil}{2} \rceil \geq \frac{n}{10}$
  - Thus, at least  $\frac{3n}{10}$  elements are  $\leq x$ .
  - Which implies that  $|L| + |E| \geq \frac{3n}{10}$ .
- Similarly, at least  $\frac{3n}{10}$  elements are  $\geq x$ . Which implies that  $|G| + |E| \geq \frac{3n}{10}$ .
- For these bounds, we can deduce the maximum size of the recursive call:
  - $|L| = |A| - |G| - |E| \leq n - \frac{3n}{10} = \frac{7n}{10}$ .
  - $|G| = |A| - |L| - |E| \leq n - \frac{3n}{10} = \frac{7n}{10}$ .
  - Therefore, the maximum size of the recursive call is on a subarray of size at most  $\frac{7n}{10}$ . Which contributes  $T(\frac{7n}{10})$  to the recurrence.

Combining all above, we have the recurrence relation for the worst-case comparison number:

$$T(n) \leq T(\lceil \frac{n}{5} \rceil) + T(\frac{7n}{10}) + O(n)$$

For simplicity, ignoring ceilings, we have:

$$T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$$

---

<sup>3</sup>It can be proven that using tournament sort, the median of 5 elements can be found in the worst case of 6 comparisons.

### 1.3 Complexity Analysis

The complexity of the above **Median-of-Medians** algorithm can be analyzed using substitution method. We can first re-write the recurrence relation as:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + cn$$

for some constant  $c > 0$ .

Assume that  $T(n) \leq dn$  for some constant  $d > 0$  and for all  $n \geq 1$ , we need to find a  $d$  such that this assumption holds. Substitute  $dn$  into the recurrence:

$$\begin{aligned} T(n) &\leq d\frac{n}{5} + d\frac{7n}{10} + cn \\ &= \frac{9dn}{10} + cn \leq dn \end{aligned}$$

Which gives  $d \geq 10c$ . Thus, we can choose  $d = 10c$  to make the assumption holds. For base case ( $n \leq 5$ ), since it takes constant time, we can always choose  $d$  to be large enough to ensure the solution holds.

To this end, we have proved that the worst-case time complexity of the **Median-of-Medians** algorithm is  $O(n)$ .

### Justification

In this additional section, we discuss why the group size is 5.

In the previous section, we proved that for a group size of 5, at most, a subproblem of size  $\frac{7n}{10}$  needs to be solved. Now, let's assume the group size is  $k$ . This can be rephrased as: among  $\lceil \frac{n}{k} \rceil$  medians,  $\lceil \frac{\lceil \frac{n}{k} \rceil}{2} \rceil$  are smaller than the pivot. where, in each group,  $\lceil \frac{k}{2} \rceil$  elements are smaller or equal than the pivot. This totals  $\lceil \frac{\lceil \frac{n}{k} \rceil}{2} \rceil \lceil \frac{k}{2} \rceil \geq \frac{n^4}{4}$ , meaning the maximum size of the subproblem is  $n - \frac{n}{4} = \frac{3n}{4}$ . In the recurrence relation, we need to handle subproblems  $T(\frac{n}{k})$  and  $T(\frac{3n}{4})$ . For this recurrence relation to resolve to  $O(n)$ , it needs to be ensured that  $\frac{n}{k} + \frac{3n}{4} < n$ , that is,  $k > 4$ .

---

<sup>4</sup>This argument is still imprecise, but we can test  $k = 2 \sim 4$  to get the same result.

## Problem 2: Search in Bitonic Sequence

The running time of the **Search in Bitonic** algorithm can be broken down as follows:

### 1. Divide:

- Dividing the array into two halves involves calculating a middle index, which takes  $O(1)$  time.
- To determine if a subarray is bitonic or monotonic, we can compare a few key elements  $(a_1, a_2, a_{n-1}, a_n)$ , which takes  $O(1)$  time. So analyzing the monotonicity of both halves takes also  $O(1)$  time.

### 2. Conquer:

- **Scenario 1:** One half is bitonic (size  $N/2$ ), the other is monotonic (size  $N/2$ ). The recursive search on the bitonic half contributes  $T(N/2)$ . The binary search on the monotonic half takes  $O(\log(N/2)) = O(\log N)$  time. Total time for this scenario:  $T(N/2) + O(\log N)$ .
- **Scenario 2:** Both halves are monotonic (each size  $N/2$ ). Total time for this scenario:  $O(\log N)$ .

### 3. Combine:

Combining the answers from the left and right halves takes  $O(1)$  time.

The worst-case scenario occurs when the algorithm always has to recurse on a bitonic half. Therefore, the worst-case recurrence relation is:

$$T(N) = T\left(\frac{N}{2}\right) + O(\log N)$$

For simplicity, we can re-write it as:

$$T(N) = T\left(\frac{N}{2}\right) + c \log N$$

For some constant  $c > 0$ . This recurrence can be solved by unrolling it:

$$\begin{aligned} T(N) &= T\left(\frac{N}{2}\right) + c \log N \\ &= T\left(\frac{N}{4}\right) + c \log N + c \log \frac{N}{2} \\ &= T\left(\frac{N}{8}\right) + c \log N + c \log \frac{N}{2} + c \log \frac{N}{4} \\ &\dots \end{aligned}$$

This pattern continues until the base case,  $T(1)$  or  $T(2)$ , which takes  $O(1)$  time. The recursion depth is  $k = \log N$ . Thus, we have:

$$\begin{aligned}
 T(N) &= T\left(\frac{N}{2^k}\right) + c \sum_{i=0}^{k-1} \log\left(\frac{N}{2^i}\right) \\
 &= T(1) + c \sum_{i=0}^{\log N - 1} (\log N - i) \\
 &= O(1) + c \frac{\log N (\log N + 1)}{2} \\
 &= O(1) + O(\log^2 N) = O(\log^2 N)
 \end{aligned}$$

Therefore, the worst-case time complexity of the **Search in Bitonic** algorithm is  $O(\log^2 N)$ .

## Problem 3: Master Theorem

1.  $T(n) = 3T(n/2) + n^2$ 
  - $a = 3, b = 2, f(n) = n^2$
  - $f(n) = \Theta(n^2) \rightarrow d = 2$
  - $a = 3 < 4 = 2^2 = b^d$
  - $T(n) = \Theta(n^d) = \Theta(n^2)$  (case 1)

2.  $T(n) = 5T(n/4) + n(\log n)^2$

Let  $T_L(n) = 5T_L(n/4) + n$

- $a = 5, b = 4, f(n) = n$
- $f(n) = \Theta(n) \rightarrow d = 1$
- $a = 5 > 4^1 = b^d$
- $T_L(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_4 5})$  (case 3)

Let  $T_U(n) = 5T_U(n/4) + n^{\log_4 5}$

- $a = 5, b = 4, f(n) = n^{\log_4 5}$
- $f(n) = \Theta(n^{\log_4 5}) \rightarrow d = \log_4 5$
- $a = 5 = 4^{\log_4 5} = b^d$
- $T_U(n) = \Theta(n^d \log n) = \Theta(n^{\log_4 5} \log n)$  (case 2)

Since we have  $T_L(n) < T(n) < T_U(n)$ , we can give the following asymptotic bound:

$$T(n) = \Omega(n^{\log_4 5}) \text{ and } T(n) = O(n^{\log_4 5} \log n)$$

Well, actually we can also derive an asymptotically tight bound for  $T(n)$  by substitution and unrolling the recurrence:

$$\begin{aligned}
 T(n) &= 5T(n/4) + n(\log n)^2 \\
 n &\mapsto 4^k \\
 T(4^k) &= 5T(4^{k-1}) + 4^k(\log 4^k)^2 \\
 &= 5T(4^{k-1}) + 4^k(2k)^2 \\
 \frac{T(4^k)}{4^k} &= \frac{5}{4} \frac{T(4^{k-1})}{4^{k-1}} + (2k)^2 \\
 S(k) &:= \frac{T(4^k)}{4^k} \\
 S(k) &= \frac{5}{4} S(k-1) + (2k)^2 \\
 &= \left(\frac{5}{4}\right)^k S(0) + \sum_{i=1}^k \left(\frac{5}{4}\right)^{k-i} (2i)^2 \\
 &= \left(\frac{5}{4}\right)^k T(1) + 4 \left(\frac{5}{4}\right)^k \sum_{i=1}^k \left(\frac{4}{5}\right)^i i^2
 \end{aligned}$$

$$\begin{aligned}
 0 < x < 1 &\rightarrow \sum_{i=1}^{\infty} x^i i^2 = \frac{x(1+x)}{(1-x)^3} \rightarrow 0 < \sum_{i=1}^k \left(\frac{4}{5}\right)^i i^2 < \frac{\frac{4}{5}(1+\frac{4}{5})}{(1-\frac{4}{5})^3} = 180 \\
 &\rightarrow S(k) = \left(\frac{5}{4}\right)^k [T(1) + 4 \sum_{i=1}^k \left(\frac{4}{5}\right)^i i^2] = \Theta\left(\left(\frac{5}{4}\right)^k\right) \\
 T(n) &= nS\left(\frac{\log n}{2}\right) = \Theta\left(n\left(\frac{5}{4}\right)^{\frac{\log n}{2}}\right) = \Theta\left(n\left(\frac{\sqrt{5}}{2}\right)^{\log n}\right)
 \end{aligned}$$

3.  $T(n) = 3T\left(\frac{n}{3} - 3\right) + \frac{n}{3}$

Let  $T_L(n) = 3T_L\left(\frac{n}{4}\right) + \frac{n}{3}$ <sup>1</sup>

- $a = 3, b = 4, f(n) = \frac{n}{3}$
- $f(n) = \Theta(n) \rightarrow d = 1$
- $a = 3 < 4^1 = b^d$
- $T_L(n) = \Theta(n^d) = \Theta(n)$  (case 1)

Let  $T_U(n) = 3T_U\left(\frac{n}{3}\right) + \frac{n}{3}$

- $a = 3, b = 3, f(n) = \frac{n}{3}$
- $f(n) = \Theta(n) \rightarrow d = 1$
- $a = 3 = 3^1 = b^d$
- $T_U(n) = \Theta(n^d \log n) = \Theta(n \log n)$  (case 2)

Since we have  $T_L(n) < T(n) < T_U(n)$ , we can give the following asymptotic bound:

$$T(n) = \Omega(n) \text{ and } T(n) = O(n \log n)$$

4.  $T(n) = 2T(\sqrt{n}) + \log n$

By making the substitution  $n \mapsto 2^k$  and  $S(k) = T(2^k)$ , the recurrence relation can be re-write as:

$$\begin{aligned}
 T(2^k) &= 2T(2^{k-1}) + \log 2^k \\
 S(k) &= 2S(k-1) + k
 \end{aligned}$$

By unrolling the recurrence, we have:

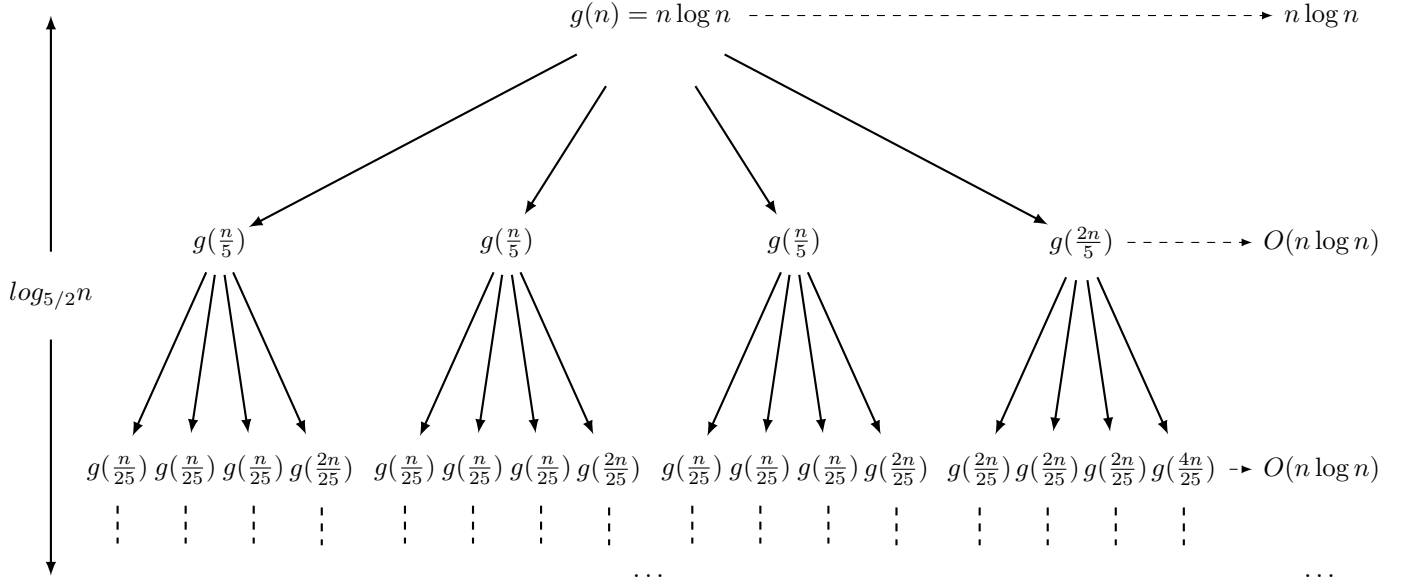
$$\begin{aligned}
 S(k) &= 2S(k-1) + k \\
 &= 2^k S(0) + \sum_{i=1}^k 2^{k-i} i \\
 &= 2^k T(1) + 2^k \sum_{i=1}^k \frac{i}{2^i} \\
 0 < x < 1 &\rightarrow \sum_{i=1}^{\infty} x^i i = \frac{x}{(1-x)^2} \rightarrow 0 < \sum_{i=1}^k \frac{i}{2^i} < \frac{0.5}{0.5^2} = 2 \\
 &\rightarrow S(k) = \Theta(2^k) \\
 T(n) &= S\left(\frac{\log n}{2}\right) = \Theta\left(2^{\frac{\log n}{2}}\right) = \Theta(\sqrt{n})
 \end{aligned}$$

---

<sup>1</sup>Since for sufficiently large  $n$  ( $n \geq 36$ ),  $\frac{n}{4} \leq \frac{n}{3} - 3$  is guaranteed, it can therefore be considered not to affect the asymptotic properties.



## Problem 4: Recursion Tree



Total:  $O(n \log^2 n)$

The derivation of the cost for each layer is as follows: Let the set  $M = \{m_1, m_2, \dots\}$  denote the set of subproblem sizes for each layer. It can be proven that  $\sum_{i=1}^{|M|} m_i \leq n$  (this is because for any problem decomposition, the subproblem sizes are  $3n/5 + 2n/5 = n$ ).

Since we have  $g(n) + g(m) \leq g(n + m)$  for  $g(n) = n \log n$  and any  $n, m \geq 2$ , one can prove that  $\sum_{i=1}^{|M|} (m_i \log m_i) \leq (\sum_{i=1}^{|M|} m_i) \log (\sum_{i=1}^{|M|} m_i) \leq n \log n$  holds if all  $m_i \geq 2$ .

Intuitively, the solution to the recurrence relation is at most the number of layers multiplied by the cost of each layer, i.e.,  $O(\log_{5/2} n \times n \log n) = O(n \log^2 n)$ . Here we use the substitution method to prove that this is indeed an asymptotic upper bound for the recurrence solution:

Assume that  $T(n) \leq dn \log^2 n$  holds for some constant  $d > 0$  and for all  $n \geq 1$ , we need to find a  $d$  such that this assumption holds. Substitute it into the recurrence relation:

$$\begin{aligned}
 T(n) &= 3T\left(\frac{n}{5}\right) + T\left(\frac{2n}{5}\right) + n \log n \\
 &\leq 3d \frac{n}{5} \log^2 \frac{n}{5} + d \frac{2n}{5} \log^2 \frac{2n}{5} + n \log n \\
 &= 3d \frac{n}{5} (\log^2 n - 2 \log n \log 5 + \log^2 5) \\
 &\quad + d \frac{2n}{5} (\log^2 n - 2 \log n \log \frac{5}{2} + \log^2 \frac{5}{2}) + n \log n \\
 &= dn \log^2 n + n \log n \left[1 - \frac{2d}{5} (5 \log 5 - 2 \log 2)\right] + \frac{dn}{5} (3 \log^2 5 + 2 \log^2 \frac{5}{2})
 \end{aligned}$$

To ensure the  $\text{RHS} \leq dn \log^2 n$ , we need to have:

$$\begin{aligned} n \log n \left[ 1 - \frac{2d}{5}(5 \log 5 - 2 \log 2) \right] + \frac{dn}{5} (3 \log^2 5 + 2 \log^2 \frac{5}{2}) &\leq 0 \\ \log n \left[ 1 - \frac{2d}{5}(5 \log 5 - 2 \log 2) \right] + \frac{d}{5} (3 \log^2 5 + 2 \log^2 \frac{5}{2}) &\leq 0 \quad (\text{Divide by } n) \end{aligned}$$

In order for this inequality to hold for sufficiently large  $n$ , we must make the coefficient of  $\log n$  negative or zero:

$$\begin{aligned} 1 - \frac{2d}{5}(5 \log 5 - 2 \log 2) &\leq 0 \\ \frac{2d}{5}(5 \log 5 - 2 \log 2) &\geq 1 \\ d &\geq \frac{5}{2(5 \log 5 - 2 \log 2)} \end{aligned}$$

Therefore, we can choose  $d = \frac{5}{2(5 \log 5 - 2 \log 2)}$  to ensure the solution holds. For base case  $(n \leq 1)^1$ , since it takes constant time, we can always choose  $d$  to be large enough to ensure the solution  $T(n) \leq dn \log^2 n$  holds.

To this end, we have proved that the asymptotic complexity of the recurrence relation is  $O(n \log^2 n)$ .

---

<sup>1</sup>Here, we do not need to guarantee  $\frac{n}{5} \geq 2$ , because the inequality  $g(n) + g(m) \leq g(n + m)$  was not used when solving the recurrence relation using the substitution method.

# Problem 5: Password Generation Counting

## 5.1 Recurrence Relation Analysis

Let  $T(n)$  be the **number of operations** required to count the **number of valid passwords**  $S(n)$  of digital sum  $n$ . The following algorithm gives a recursive solution to this problem:

- If  $n = 1$ , or  $n$  has been visited before, return  $S(n)$ . It's trivial that  $S(1) = 1$ .
- Otherwise, recursively call the algorithm to count the number of valid passwords of digital sum  $n - 1$ , and return the sum  $S(n) = \sum_{i=1}^{\min(9, n-1)} S(n - i)$ .

The recurrence relation can be written as:

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

This is because in each step  $n$ , we only need to call the algorithm once on  $n - 1$ , which contributes  $T(n - 1)$  to the recurrence, The remaining operation is just a single summation of at most ten elements  $\sum_{i=1}^{\min(9, n-1)} S(i)$  which takes constant time, because  $S(1)$  through  $S(n - 1)$  have all been pre-calculated after calling the algorithm for  $n - 1$ .

The base case is  $T(1) = c$ , since there is only one valid password of digital sum 1, which is '1'.

The total number of operations  $T(n) = O(n)$  can be immediately derived by unrolling the recurrence relation.

## 5.2 Pseudocode

---

### Algorithm 1 Count Passwords with Sum $S$

---

```

1: Input: Target sum  $n$ 
2: Output: Total number of different passwords
3: Global/Shared: 'S' array of size  $(n + 1)$ , initialized with 0,  $S[1] \leftarrow 1$ 
4: function COUNTPASSWORDSRECURSIVE( $n$ )
5:   if  $n < 1$  then
6:     return 0
7:   end if
8:   if  $S[n] > 0$  then
9:     return  $S[n]$ 
10:  end if
11:   $ans \leftarrow 0$ 
12:  for  $i$  from 1 to 9 do
13:     $ans \leftarrow ans + \text{COUNTPASSWORDSRECURSIVE}(n - i)$ 
14:  end for
15:   $S[n] \leftarrow ans$ 
16:  return  $ans$ 
17: end function

```

---

▷ Boundary Case

▷ Pre-calculated

▷ Store result

# Problem 6: Ancient Pyramid Treasure Hunt

## 6.1 State Definition

First, it can be proven that any valid path from the bottom-left corner to the top-right corner is also a valid path from the top-right corner to the bottom-left corner; therefore, we can transform the problem into finding the maximum number of non-duplicate treasures that can be collected by two people starting simultaneously from the bottom-left corner and arriving at the top-right corner.

A direct solution would be to record the current positions of both people and the corresponding number of (non-duplicate) treasures collected, which would have an  $O(n^4)$  complexity; A straightforward observation is that when the position of one person is determined, the number of possible positions for the other person is only  $O(n)$  (instead of  $O(n^2)$ ). This is because when both people start at the same time, they would have taken the same number of steps, leading to  $x_1 + y_1 = x_2 + y_2$ .

Therefore, we can use a three-dimensional state array  $dp[x_1][y_1][x_2]$  to record number of unique treasures obtained, and calculate  $y_2$  state-wisely during transitions. Since each state will be computed at most once, the time complexity of this algorithm is  $O(n^3)$ .

## 6.2 State Transition & Recurrence Relation

For any valid state  $(x_1, y_1, x_2, y_2)$ , since each person can only move from the left or from below, there are four possible preceding states:

- $(x_1 - 1, y_1, x_2 - 1, y_2)$
- $(x_1, y_1 - 1, x_2 - 1, y_2)$
- $(x_1 - 1, y_1, x_2, y_2 - 1)$
- $(x_1, y_1 - 1, x_2, y_2 - 1)$

Each person moves to the position corresponding to the current state (they might overlap) and collects unique treasures. Therefore, the state transition equation can be written as:

$$\begin{aligned} dp[x_1][y_1][x_2] = \max( \\ & dp[x_1 - 1][y_1][x_2 - 1], \\ & dp[x_1][y_1 - 1][x_2 - 1], \\ & dp[x_1 - 1][y_1][x_2], \\ & dp[x_1][y_1 - 1][x_2] \\ & ) + \text{unique treasures collected} \end{aligned}$$

Where the last dimension  $y_2$  was compressed. Next, one can traverse the states in a temporal order to ensure that when any state is reached, its subproblems have already been visited.

alternatively, one can use a recursive solution, starting from the top-right corner  $(n, n, n)$ , and whenever a state  $(x_1, y_1, x_2)$  is accessed, either return  $dp[x_1][y_1][x_2]$  (if this state has already been visited) or recursively visit its predecessor states, combine the answers, and record the result to  $dp[x_1][y_1][x_2]$ .

Finally, invalid states should be handled. For iterative solutions, states where (any one person) is at a wall can be skipped during the loop; for recursive solutions, recursion can return immediately when an invalid state is accessed.

### 6.3 Boundary Case & Base Case

Boundary conditions and base cases can be simply handled by array initialization:

- Initialize the  $dp$  array to  $-\infty$ .
- Set  $dp[0][0][0] \leftarrow 0$ .

These impossible (but not wall) cases will always be negative during calculation and will not contribute to the final answer. One might think that some positions reachable from the bottom-left corner might not be reachable from the top-right corner, but it can be proven that these positions will also not contribute to the final answer (because only complete valid paths from bottom-left to top-right will).