

# 面试常考算法模板

## 万能算法小抄 Cheat Sheet V5.0

---

### 前言

- ◆ 版权归属：九章算法（杭州）科技有限公司
- ◆ 九章算法是 IT 求职领域领先的互联网在线教育平台。已帮助 30000+求职者获得了 Google、Facebook、阿里巴巴、腾讯、字节跳动等顶尖互联网公司的工作机会。已开设算法、大数据、人工智能、系统设计、简历提升、模拟面试等方面的课程和服务，致力于提供高品质、全方位的技术培训和就业指导。
- ◆ 可以原文转载和分享，转载时需保留此版权信息，不得对内容进行增删和修改
- ◆ 本文作者：九章算法令狐冲
- ◆ 官方网站：<http://www.jiuzhang.com/>

# 目录

前言.....	1
二分法 Binary Search.....	3
双指针 Two Pointers.....	8
排序算法 Sorting.....	17
二叉树分治 Binary Tree Divide & Conquer.....	23
二叉搜索树非递归 BST Iterator.....	26
宽度优先搜索 BFS.....	30
深度优先搜索 DFS.....	40
动态规划 Dynamic Programming.....	42
堆 Heap.....	47
并查集 Union Find.....	52
字典树 Trie.....	61
LRU 缓存.....	66

## 求职面试必备课程

《[九章算法基础班 Java+Python](#)》 《[九章算法班](#)》 《[九章算法面试高频题冲刺班](#)》  
《[动态规划专题班](#)》 《[面向对象设计 OOD](#)》 《[系统架构设计 System Design](#)》  
《[数据分析就业班](#)》 《[Twitter 后端系统-Python 项目实战](#)》

# 二分法 Binary Search

## 使用条件

- 排序数组 (30–40%是二分)
- 当面试官要求你找一个比  $O(n)$  更小的时间复杂度算法的时候(99%)
- 找到数组中的一个分割位置, 使得左半部分满足某个条件, 右半部分不满足(100%)
- 找到一个最大/最小的值使得某个条件被满足(90%)

## 复杂度

- 时间复杂度:  $O(\log n)$
- 空间复杂度:  $O(1)$

## 领扣例题

- [LintCode 14. 二分查找\(在排序的数据集上进行二分\)](#)
- [LintCode 460. 在排序数组中找最接近的 K 个数 \(在未排序的数据集上进行二分\)](#)
- [LintCode 437. 书籍复印\(在答案集上进行二分 \)](#)

## 代码模版

Java

```
1. 1.int binarySearch(int[] nums, int target) {  
2.   
3. 2. // corner case 处理
```

```
4.  
5. 3. if (nums == null || nums.length == 0) {  
6.  
7. 4. return -1;  
8.  
9. 5. }  
10.  
11. 6.  
12.  
13. 7. int start = 0, end = nums.length - 1;  
14.  
15. 8.  
16.  
17. 9. // 要点 1: start + 1 < end  
18.  
19. 10. while (start + 1 < end) {  
20.  
21. 11. // 要点 2: start + (end - start) / 2  
22.  
23. 12. int mid = start + (end - start) / 2;  
24.  
25. 13. // 要点 3: =, <, > 分开讨论, mid 不 +1 也不 -1  
26.  
27. 14. if (nums[mid] == target) {  
28.  
29. 15. return mid;  
30.  
31. 16. } else if (nums[mid] < target) {  
32.  
33. 17. start = mid;  
34.  
35. 18. } else {  
36.
```

```
37. 19.         end = mid;
38.
39. 20.     }
40.
41. 21. }
42.
43. 22.
44.
45. 23. // 要点 4: 循环结束后, 单独处理 start 和 end
46.
47. 24. if (nums[start] == target) {
48.
49. 25.     return start;
50.
51. 26. }
52.
53. 27. if (nums[end] == target) {
54.
55. 28.     return end;
56.
57. 29. }
58.
59. 30. return -1;
60.
61. 31.}
```

## Python

```
1.
2. 1.def binary_search(self, nums, target):
3.
4. 2. # corner case 处理
5.
```

```
6. 3. # 这里等价于 nums is None or len(nums) == 0
7.
8. 4. if not nums:
9.
10. 5.     return -1
11.
12. 6.
13.
14. 7.     start, end = 0, len(nums) - 1
15.
16. 8.
17.
18. 9. # 用 start + 1 < end 而不是 start < end 的目的是为了避免死循环
19.
20. 10. # 在 first position of target 的情况下不会出现死循环
21.
22. 11. # 但是在 last position of target 的情况下会出现死循环
23.
24. 12. # 样例: nums=[1, 1] target = 1
25.
26. 13. # 为了统一模板, 我们就都采用 start + 1 < end, 就保证不会出现死循环
27.
28. 14. while start + 1 < end:
29.
30. 15.     # python 没有 overflow 的问题, 直接 // 2 就可以了
31.
32. 16.     # java 和 C++ 最好写成 mid = start + (end - start) / 2
33.
34. 17.     # 防止在 start = 2^31 - 1, end = 2^31 - 1 的情况下出现加法 overflow
35.
36. 18.     mid = (start + end) // 2
37.
38. 19.     # >, =, < 的逻辑先分开写, 然后在看看 = 的情况是否能合并到其他分支里
```

39.

40. 20. `if` `nums[mid] < target`:

41.

42. 21. `start = mid`

43.

44. 22. `elif` `nums[mid] == target`:

45.

46. 23. `end = mid`

47.

48. 24. `else`:

49.

50. 25. `end = mid`

51.

52. 26.

53.

54. 27. `# 因为上面的循环退出条件是 start + 1 < end`

55.

56. 28. `# 因此这里循环结束的时候, start 和 end 的关系是相邻关系 (1 和 2, 3 和 4 这种)`

57.

58. 29. `# 因此需要再单独判断 start 和 end 这两个数谁是我们的答案`

59.

60. 30. `# 如果是找 first position of target 就先看 start, 否则就先看 end`

61.

62. 31. `if` `nums[start] == target`:

63.

64. 32. `return start`

65.

66. 33. `if` `nums[end] == target`:

67.

68. 34. `return end`

69.

70. 35. `return -1`

# 双指针 Two Pointers

## 使用条件

- 滑动窗口 (90%)
- 时间复杂度要求  $O(n)$  (80%是双指针)
- 要求原地操作，只可以使用交换，不能使用额外空间 (80%)
- 有子数组 subarray / 子字符串 substring 的关键词 (50%)
- 有回文 Palindrome 关键词(50%)

## 复杂度

- 时间复杂度:  $O(n)$ 
  - 时间复杂度与最内层循环主体的执行次数有关
  - 与有多少重循环无关
- 空间复杂度:  $O(1)$ 
  - 只需要分配两个指针的额外内存

## 领扣例题

- [LintCode 1879. 两数之和 VII\(同向双指针\)](#)
- [LintCode1712.和相同的二元子数组\(相向双指针\)](#)
- [LintCode627. 最长回文串 \(背向双指针\)](#)
- [LintCode 64: 合并有序数组](#)



## 代码模版

Java

```
1. 1. // 相向双指针(partition in quicksort)
2.
3. 2. public void partition(int[] A, int start, int end) {
4.
5. 3. if (start >= end) {
6.
7. 4. return;
8.
9. 5. }
10.
11. 6. int left = start, right = end;
12.
13. 7. // key point 1: pivot is the value, not the index
14.
15. 8. int pivot = A[(start + end) / 2];
16.
17. 9. // key point 2: every time you compare left & right, it should be
18.
19. 10. // left <= right not left < right
20.
21. 11. while (left <= right) {
22.
23. 12. while (left <= right && A[left] < pivot) {
24.
25. 13. left++;
26.
27. 14. }
28.
29. 15. while (left <= right && A[right] > pivot) {
30.
```

```
31. 16.         right--;  
32.  
33. 17.         }  
34.  
35. 18.         if (left <= right) {  
36.  
37. 19.             int temp = A[left];  
38.  
39. 20.             A[left] = A[right];  
40.  
41. 21.             A[right] = temp;  
42.  
43. 22.             left++;  
44.  
45. 23.             right--;  
46.  
47. 24.         }  
48.  
49. 25.     }  
50.  
51. 26.}  
52.  
53. 27.  
54.  
55. 28.// 背向双指针  
56.  
57. 29.left = position;  
58.  
59. 30.right = position + 1;  
60.  
61. 31.while (left >= 0 && right < length) {  
62.  
63. 32.    if (可以停下来了) {
```

```
64.
65. 33. break;
66.
67. 34. }
68.
69. 35. left--;
70.
71. 36. right++;
72.
73. 37.}
74.
75. 38.
76.
77. 39.// 同向双指针
78.
79. 40.int j = 0;
80.
81. 41.for (int i = 0; i < n; i++) {
82.
83. 42. // 不满足则循环到满足搭配为止
84.
85. 43. while (j < n && i 到 j 之间不满足条件) {
86.
87. 44. j += 1;
88.
89. 45. }
90.
91. 46. if (i 到 j 之间满足条件) {
92.
93. 47. 处理 i, j 这次搭配
94.
95. 48. }
96.
```

```
97. 49.}
98.
99. 50.
100.
101. 51.// 合并双指针
102.
103. 52.ArrayList<Integer> merge(ArrayList<Integer> list1, ArrayList<Integer> list2) {
104.
105. 53. // 需要 new 一个新的 list, 而不是在 list1 或者 list2 上直接改动
106.
107. 54. ArrayList<Integer> newList = new ArrayList<Integer>();
108.
109. 55.
110.
111. 56. int i = 0, j = 0;
112.
113. 57. while (i < list1.size() && j < list2.size()) {
114.
115. 58. if (list1.get(i) < list2.get(j)) {
116.
117. 59. newList.add(list1.get(i));
118.
119. 60. i++;
120.
121. 61. } else {
122.
123. 62. newList.add(list2.get(j));
124.
125. 63. j++;
126.
127. 64. }
128.
129. 65. }
```

```
130.
131. 66.
132.
133. 67. // 合并上下的数到 newList 里
134.
135. 68. // 无需用 if (i < list1.size()), 直接 while 即可
136.
137. 69. while (i < list1.size()) {
138.
139. 70.     newList.add(list1.get(i));
140.
141. 71.     i++;
142.
143. 72. }
144.
145. 73. while (j < list2.size()) {
146.
147. 74.     newList.add(list2.get(j));
148.
149. 75.     j++;
150.
151. 76. }
152.
153. 77.
154.
155. 78. return newList;
156.
157. 79.}
```

## Python

```
1. 1.# 相向双指针(partition in quicksort)
2.
```

```
3. 2.def patition(self, A, start, end):
4.
5. 3.  if start >= end:
6.
7. 4.      return
8.
9. 5.      left, right = start, end
10.
11. 6.      # key point 1: pivot is the value, not the index
12.
13. 7.      pivot = A[(start + end) // 2];
14.
15. 8.      # key point 2: every time you compare left & right, it should be
16.
17. 9.      # left <= right not left < right
18.
19. 10.     while left <= right:
20.
21. 11.         while left <= right and A[left] < pivot:
22.
23. 12.             left += 1
24.
25. 13.         while left <= right and A[right] > pivot:
26.
27. 14.             right -= 1
28.
29. 15.         if left <= right:
30.
31. 16.             A[left], A[right] = A[right], A[left]
32.
33. 17.             left += 1
34.
35. 18.             right -= 1
```

36.

37. 19.

38.

39. 20.# 背向双指针

40.

41. 21.left = position

42.

43. 22.right = position + 1

44.

45. 23.while left >= 0 and right < len(s):

46.

47. 24. if left 和 right 可以停下来了:

48.

49. 25. break

50.

51. 26. left -= 1

52.

53. 27. right += 1

54.

55. 28.

56.

57. 29.# 同向双指针

58.

59. 30.j = 0

60.

61. 31.for i in range(n):

62.

63. 32. # 不满足则循环到满足搭配为止

64.

65. 33. while j < n and i 到 j 之间不满足条件:

66.

67. 34. j += 1

68.

```
69. 35. if i 到 j 之间满足条件:
70.
71. 36.     处理 i 到 j 这段区间
72.
73. 37.
74.
75. 38.# 合并双指针
76.
77. 39.def merge(list1, list2):
78.
79. 40.     new_list = []
80.
81. 41.     i, j = 0, 0
82.
83. 42.
84.
85. 43.     # 合并的过程只能操作 i, j 的移动, 不要去用 list1.pop(0) 之类的操作
86.
87. 44.     # 因为 pop(0) 是 O(n) 的时间复杂度
88.
89. 45.     while i < len(list1) and j < len(list2):
90.
91. 46.         if list1[i] < list2[j]:
92.
93. 47.             new_list.append(list1[i])
94.
95. 48.             i += 1
96.
97. 49.         else:
98.
99. 50.             new_list.append(list2[j])
100.
101. 51.             j += 1
```



```
102.
103. 52.
104.
105. 53. # 合并剩下的数到 new_list 里
106.
107. 54. # 不要用 new_list.extend(list1[i:]) 之类的方法
108.
109. 55. # 因为 list1[i:] 会产生额外空间耗费
110.
111. 56. while i < len(list1):
112.
113. 57.     new_list.append(list1[i])
114.
115. 58.     i += 1
116.
117. 59. while j < len(list2):
118.
119. 60.     new_list.append(list2[j])
120.
121. 61.     j += 1
122.
123. 62.
124.
125. 63. return new_list
```

## 排序算法 Sorting

### 使用条件

## 复杂度

- 时间复杂度：
  - 快速排序(期望复杂度)：  $O(n\log n)$
  - 归并排序(最坏复杂度)：  $O(n\log n)$
- 空间复杂度：
  - 快速排序：  $O(1)$
  - 归并排序：  $O(n)$

## 领扣例题

- [LintCode 464. 整数排序 II](#)

## 代码模板

Java

```
1. // quick sort
2. public class Solution {
3.     /**
4.      * @param A an integer array
5.      * @return void
6.      */
7.     public void sortIntegers(int[] A) {
8.         quickSort(A, 0, A.length - 1);
9.     }
10.
11.     private void quickSort(int[] A, int start, int end) {
12.         if (start >= end) {
```

```
13.         return;
14.     }
15.
16.     int left = start, right = end;
17.     // key point 1: pivot is the value, not the index
18.     int pivot = A[(start + end) / 2];
19.
20.     // key point 2: every time you compare left & right, it should be
21.     // left <= right not left < right
22.     while (left <= right) {
23.         while (left <= right && A[left] < pivot) {
24.             left++;
25.         }
26.         while (left <= right && A[right] > pivot) {
27.             right--;
28.         }
29.         if (left <= right) {
30.             int temp = A[left];
31.             A[left] = A[right];
32.             A[right] = temp;
33.
34.             left++;
35.             right--;
36.         }
37.     }
38.
39.     quickSort(A, start, right);
40.     quickSort(A, left, end);
41. }
42. }
43. // merge sort
44. public class Solution {
45.     public void sortIntegers(int[] A) {
```

```
46.     if (A == null || A.length == 0) {
47.         return;
48.     }
49.     int[] temp = new int[A.length];
50.     mergeSort(A, 0, A.length - 1, temp);
51. }
52.
53. private void mergeSort(int[] A, int start, int end, int[] temp) {
54.     if (start >= end) {
55.         return;
56.     }
57.     // 处理左半区间
58.     mergeSort(A, start, (start + end) / 2, temp);
59.     // 处理右半区间
60.     mergeSort(A, (start + end) / 2 + 1, end, temp);
61.     // 合并排序数组
62.     merge(A, start, end, temp);
63. }
64.
65. private void merge(int[] A, int start, int end, int[] temp) {
66.     int middle = (start + end) / 2;
67.     int leftIndex = start;
68.     int rightIndex = middle + 1;
69.     int index = start;
70.     while (leftIndex <= middle && rightIndex <= end) {
71.         if (A[leftIndex] < A[rightIndex]) {
72.             temp[index++] = A[leftIndex++];
73.         } else {
74.             temp[index++] = A[rightIndex++];
75.         }
76.     }
77.     while (leftIndex <= middle) {
78.         temp[index++] = A[leftIndex++];
```

```
79.     }
80.     while (rightIndex <= end) {
81.         temp[index++] = A[rightIndex++];
82.     }
83.     for (int i = start; i <= end; i++) {
84.         A[i] = temp[i];
85.     }
86. }
87. }
```

## Python

```
1.     # quick sort
2.     class Solution:
3.         # @param {int[]} A an integer array
4.         # @return nothing
5.         def sortIntegers(self, A):
6.             # Write your code here
7.             self.quickSort(A, 0, len(A) - 1)
8.
9.         def quickSort(self, A, start, end):
10.            if start >= end:
11.                return
12.
13.            left, right = start, end
14.            # key point 1: pivot is the value, not the index
15.            pivot = A[(start + end) // 2];
16.
17.            # key point 2: every time you compare left & right, it should be
18.            # left <= right not left < right
19.            while left <= right:
20.                while left <= right and A[left] < pivot:
21.                    left += 1
```

```
22.
23.     while left <= right and A[right] > pivot:
24.         right -= 1
25.
26.     if left <= right:
27.         A[left], A[right] = A[right], A[left]
28.
29.         left += 1
30.         right -= 1
31.
32.     self.quickSort(A, start, right)
33.     self.quickSort(A, left, end)
34. # merge sort
35. class Solution:
36.     def sortIntegers(self, A):
37.         if not A:
38.             return A
39.
40.         temp = [0] * len(A)
41.         self.merge_sort(A, 0, len(A) - 1, temp)
42.
43.     def merge_sort(self, A, start, end, temp):
44.         if start >= end:
45.             return
46.
47.         # 处理左半区间
48.         self.merge_sort(A, start, (start + end) // 2, temp)
49.         # 处理右半区间
50.         self.merge_sort(A, (start + end) // 2 + 1, end, temp)
51.         # 合并排序数组
52.         self.merge(A, start, end, temp)
53.
54.     def merge(self, A, start, end, temp):
```

```
55.     middle = (start + end) // 2
56.     left_index = start
57.     right_index = middle + 1
58.     index = start
59.
60.     while left_index <= middle and right_index <= end:
61.         if A[left_index] < A[right_index]:
62.             temp[index] = A[left_index]
63.             index += 1
64.             left_index += 1
65.         else:
66.             temp[index] = A[right_index]
67.             index += 1
68.             right_index += 1
69.
70.     while left_index <= middle:
71.         temp[index] = A[left_index]
72.         index += 1
73.         left_index += 1
74.
75.     while right_index <= end:
76.         temp[index] = A[right_index]
77.         index += 1
78.         right_index += 1
79.
80.     for i in range(start, end + 1):
81.         A[i] = temp[i]
```

## 二叉树分治 Binary Tree Divide & Conquer

### 使用条件

- 二叉树相关的问题 (99%)
- 可以一分为二去分别处理之后再合并结果 (100%)
- 数组相关的问题 (10%)

## 复杂度

时间复杂度  $O(n)$

空间复杂度  $O(n)$  (含递归调用的栈空间最大耗费)

## 领扣例题

- [LintCode 1534. 将二叉搜索树转换为已排序的双向链接列表](#)
- [LintCode 94. 二叉树中的最大路径和](#)
- [LintCode 95. 验证二叉查找树](#)

## 代码模板

Java

```
1. 1. public ResultType divideConquer(TreeNode node) {  
2.  
3. 2. // 递归出口  
4.  
5. 3. // 一般处理 node == null 就够了  
6.  
7. 4. // 大部分情况不需要处理 node == leaf  
8.  
9. 5. if (node == null) {
```



```
10.  
11. 6. return ...;  
12.  
13. 7. }  
14.  
15. 8. // 处理左子树  
16.  
17. 9. ResultType leftResult = divideConquer(node.left);  
18.  
19. 10. // 处理右子树  
20.  
21. 11. ResultType rightResult = divideConquer(node.right);  
22.  
23. 12. //合并答案  
24.  
25. 13. ResultType result = merge leftResult and rightResult  
26.  
27. 14. return result;  
28.  
29. 15.}
```

## Python

```
1. 1.def divide_conquer(root):  
2.  
3. 2. # 递归出口  
4.  
5. 3. # 一般处理 node == null 就够了  
6.  
7. 4. # 大部分情况不需要处理 node == leaf  
8.  
9. 5. if root is None:  
10.
```

```
11. 6. return ...
12.
13. 7. # 处理左子树
14.
15. 8. left_result = divide_conquer(node.left)
16.
17. 9. # 处理右子树
18.
19. 10. right_result = divide_conquer(node.right)
20.
21. 11. # 合并答案
22.
23. 12. result = merge left_result and right_result to get merged result
24.
25. 13. return result
```

## 二叉搜索树非递归 BST Iterator

### 使用条件

- 用非递归的方式（Non-recursion / Iteration）实现二叉树的中序遍历
- 常用于 BST 但不仅仅可以用于 BST

### 复杂度

时间复杂度  $O(n)$

空间复杂度  $O(n)$

## 领扣例题

- [LintCode 67. 二叉树的中序遍历](#)
- [LintCode 902. 二叉搜索树的第 k 大元素](#)

## 代码模板

Java

```
1. 1. List<TreeNode> inorderTraversal(TreeNode root) {  
2.  
3. 2. List<TreeNode> inorder = new ArrayList<>();  
4.  
5. 3. if (root == null) {  
6.  
7. 4. return inorder;  
8.  
9. 5. }  
10.  
11. 6. // 创建一个 dummy node, 右指针指向 root  
12.  
13. 7. // 放到 stack 里, 此时栈顶 dummy 就是 iterator 的当前位置  
14.  
15. 8. TreeNode dummy = new TreeNode(0);  
16.  
17. 9. dummy.right = root;  
18.  
19. 10. Stack<TreeNode> stack = new Stack<>();  
20.  
21. 11. stack.push(dummy);  
22.  
23. 12.
```

```
24.  
25. 13. // 每次将 iterator 挪到下一个点  
26.  
27. 14. // 就是调整 stack 使得栈顶是下一个点  
28.  
29. 15. while (!stack.isEmpty()) {  
30.  
31. 16.     TreeNode node = stack.pop();  
32.  
33. 17.     if (node.right != null) {  
34.  
35. 18.         node = node.right;  
36.  
37. 19.         while (node != null) {  
38.  
39. 20.             stack.push(node);  
40.  
41. 21.             node = node.left;  
42.  
43. 22.         }  
44.  
45. 23.     }  
46.  
47. 24.     if (!stack.isEmpty()) {  
48.  
49. 25.         inorder.add(stack.peek());  
50.  
51. 26.     }  
52.  
53. 27. }  
54.  
55. 28. return inorder;  
56.
```

57. 29.}

## Python

```
1.     1.def inorder_traversal(root):
2.
3.     2. if root is None:
4.
5.     3.     return []
6.
7.     4.
8.
9.     5. # 创建一个 dummy node, 右指针指向 root
10.
11.    6. # 并放到 stack 里, 此时 stack 的栈顶 dummy
12.
13.    7. # 是 iterator 的当前位置
14.
15.    8. dummy = TreeNode(0)
16.
17.    9. dummy.right = root
18.
19.    10. stack = [dummy]
20.
21.    11.
22.
23.    12. inorder = []
24.
25.    13. # 每次将 iterator 挪到下一个点
26.
27.    14. # 也就是调整 stack 使得栈顶到下一个点
28.
29.    15. while stack:
30.
```

```
31. 16.     node = stack.pop()
32.
33. 17.     if node.right:
34.
35. 18.         node = node.right
36.
37. 19.         while node:
38.
39. 20.             stack.append(node)
40.
41. 21.             node = node.left
42.
43. 22.         if stack:
44.
45. 23.             inorder.append(stack[-1])
46.
47. 24.     return inorder
```

## 宽度优先搜索 BFS

### 使用条件

- 12. 拓扑排序(100%)
- 13. 出现连通块的关键词(100%)
- 14. 分层遍历(100%)
- 15. 简单图最短路径(100%)
- 16. 给定一个变换规则，从初始状态变到终止状态最少几步(100%)

## 复杂度

- 时间复杂度：  $O(n + m)$ 
  - $n$  是点数,  $m$  是边数
- 空间复杂度：  $O(n)$

## 领扣例题

- [LintCode 974. 01 矩阵\(分层遍历\)](#)
- [LintCode 431. 找无向图的连通块](#)
- [LintCode 127. 拓扑排序](#)

## 代码模版

Java

```
1. 1.ReturnType bfs(Node startNode) {
2.
3. 2. // BFS 必须要用队列 queue, 别用栈 stack!
4.
5. 3. Queue<Node> queue = new ArrayDeque<>();
6.
7. 4. // hashmap 有两个作用, 一个是记录一个点是否被丢进过队列了, 避免重复访问
8.
9. 5. // 另外一个记录 startNode 到其他所有节点的最短距离
10.
11. 6. // 如果只求连通性的话, 可以换成 HashSet 就行
12.
13. 7. // node 做 key 的时候比较的是内存地址
```

```
14.
15. 8. Map<Node, Integer> distance = new HashMap<>();
16.
17. 9.
18.
19. 10. // 把起点放进队列和哈希表里，如果有多个起点，都放进去
20.
21. 11. queue.offer(startNode);
22.
23. 12. distance.put(startNode, 0); // or 1 if necessary
24.
25. 13.
26.
27. 14. // while 队列不空，不停的从队列里拿出一个点，拓展邻居节点放到队列中
28.
29. 15. while (!queue.isEmpty()) {
30.
31. 16. Node node = queue.poll();
32.
33. 17. // 如果有明确的终点可以在这里加终点的判断
34.
35. 18. if (node 是终点) {
36.
37. 19. break or return something;
38.
39. 20. }
40.
41. 21. for (Node neighbor : node.getNeighbors()) {
42.
43. 22. if (distance.containsKey(neighbor)) {
44.
45. 23. continue;
46.
```



```
47. 24.     }  
48.  
49. 25.     queue.offer(neighbor);  
50.  
51. 26.     distance.put(neighbor, distance.get(node) + 1);  
52.  
53. 27.     }  
54.  
55. 28. }  
56.  
57. 29. // 如果需要返回所有点离起点的距离, 就 return hashmap  
58.  
59. 30. return distance;  
60.  
61. 31. // 如果需要返回所有连通的节点, 就 return HashMap 里的所有点  
62.  
63. 32. return distance.keySet();  
64.  
65. 33. // 如果需要返回离终点的最短距离  
66.  
67. 34. return distance.get(endNode);  
68.  
69. 35.}
```

## Python

```
1. 1.def bfs(start_node):  
2.  
3. 2. # BFS 必须要用队列 queue, 别用栈 stack!  
4.  
5. 3. # distance(dict) 有两个作用, 一个是记录一个点是否被丢进过队列了, 避免重复访问  
6.
```

```
7. 4. # 另外一个记录 start_node 到其他所有节点的最短距离
8.
9. 5. # 如果只求连通性的话, 可以换成 set 就行
10.
11. 6. # node 做 key 的时候比较的是内存地址
12.
13. 7. queue = collections.deque([start_node])
14.
15. 8. distance = {start_node: 0}
16.
17. 9.
18.
19. 10. # while 队列不空, 不停的从队列里拿出一个点, 拓展邻居节点放到队列中
20.
21. 11. while queue:
22.
23. 12.     node = queue.popleft()
24.
25. 13.     # 如果有明确的终点可以在这里加终点的判断
26.
27. 14.     if node 是终点:
28.
29. 15.         break or return something
30.
31. 16.     for neighbor in node.get_neighbors():
32.
33. 17.         if neighbor in distance:
34.
35. 18.             continue
36.
37. 19.         queue.append(neighbor)
38.
39. 20.         distance[neighbor] = distance[node] + 1
```

```
40.  
41. 21.  
42.  
43. 22. # 如果需要返回所有点离起点的距离, 就 return hashmap  
44.  
45. 23. return distance  
46.  
47. 24. # 如果需要返回所有连通的节点, 就 return HashMap 里的所有点  
48.  
49. 25. return distance.keys()  
50.  
51. 26. # 如果需要返回离终点的最短距离  
52.  
53. 27. return distance[end_node]
```

## Java 拓扑排序 BFS 模板

```
1. 1.List<Node> topologicalSort(List<Node> nodes) {  
2.  
3. 2. // 统计所有点的入度信息, 放入 hashmap 里  
4.  
5. 3. Map<Node, Integer> indegrees = getIndegrees(nodes);  
6.  
7. 4.  
8.  
9. 5. // 将所有入度为 0 的点放到队列中  
10.  
11. 6. Queue<Node> queue = new ArrayDeque<>();  
12.  
13. 7. for (Node node : nodes) {  
14.  
15. 8. if (indegrees.get(node) == 0) {  
16.
```

```
17. 9.         queue.offer(node);
18.
19. 10.     }
20.
21. 11. }
22.
23. 12.
24.
25. 13. List<Node> topoOrder = new ArrayList<>();
26.
27. 14. while (!queue.isEmpty()) {
28.
29. 15.     Node node = queue.poll();
30.
31. 16.     topoOrder.add(node);
32.
33. 17.     for (Node neighbor : node.getNeighbors()) {
34.
35. 18.         // 入度减一
36.
37. 19.         indegrees.put(neighbor, indegrees.get(neighbor) - 1);
38.
39. 20.         // 入度减到 0 说明不再依赖任何点，可以被放到队列（拓扑序）里了
40.
41. 21.         if (indegrees.get(neighbor) == 0) {
42.
43. 22.             queue.offer(neighbor);
44.
45. 23.         }
46.
47. 24.     }
48.
49. 25. }
```

```
50.
51. 26.
52.
53. 27. // 如果 queue 是空的时候，图中还有点没有被挖出来，说明存在环
54.
55. 28. // 有环就没有拓扑序
56.
57. 29. if (topoOrder.size() != nodes.size()) {
58.
59. 30.     return 没有拓扑序;
60.
61. 31. }
62.
63. 32. return topoOrder;
64.
65. 33.}
66.
67. 34.
68.
69. 35. Map<Node, Integer> getIndegrees(List<Node> nodes) {
70.
71. 36.     Map<Node, Integer> counter = new HashMap<>();
72.
73. 37.     for (Node node : nodes) {
74.
75. 38.         counter.put(node, 0);
76.
77. 39.     }
78.
79. 40.     for (Node node : nodes) {
80.
81. 41.         for (Node neighbor : node.getNeighbors()) {
82.
```

```
83. 42.         counter.put(neighbor, counter.get(neighbor) + 1);
84.
85. 43.     }
86.
87. 44. }
88.
89. 45. return counter;
90.
91. 46. }
```

## Python

```
1.     1.def get_indegrees(nodes):
2.
3.     2.     counter = {node: 0 for node in nodes}
4.
5.     3.     for node in nodes:
6.
7.     4.         for neighbor in node.get_neighbors():
8.
9.     5.             counter[neighbor] += 1
10.
11.     6.     return counter
12.
13.     7.
14.
15.     8.def topological_sort(nodes):
16.
17.     9.     # 统计入度
18.
19.     10.     indegrees = get_indegrees(nodes)
20.
21.     11.     # 所有入度为 0 的点都放到队列里
22.
```

```
23. 12. queue = collections.deque([
24.
25. 13.     node
26.
27. 14.     for node in nodes
28.
29. 15.     if indegrees[node] == 0
30.
31. 16. ])
32.
33. 17. # 用 BFS 算法一个个把点从图里挖出来
34.
35. 18. topo_order = []
36.
37. 19. while queue:
38.
39. 20.     node = queue.popleft()
40.
41. 21.     topo_order.append(node)
42.
43. 22.     for neighbor in node.get_neighbors():
44.
45. 23.         indegrees[neighbor] -= 1
46.
47. 24.         if indegrees[neighbor] == 0:
48.
49. 25.             queue.append(neighbor)
50.
51. 26. # 判断是否有循环依赖
52.
53. 27. if len(topo_order) != len(nodes):
54.
55. 28.     return 有循环依赖(环),没有拓扑序
```

56.

57. 29. `return topo_order`

## 深度优先搜索 DFS

### 使用条件

- 找满足某个条件的所有方案 (99%)
- 二叉树 Binary Tree 的问题 (90%)
- 组合问题(95%)
  - 问题模型：求出所有满足条件的“组合”
  - 判断条件：组合中的元素是顺序无关的
- 排列问题 (95%)
  - 问题模型：求出所有满足条件的“排列”
  - 判断条件：组合中的元素是顺序“相关”的。

### 不要用 DFS 的场景

- 17. 连通块问题（一定要用 BFS，否则 StackOverflow）
- 18. 拓扑排序（一定要用 BFS，否则 StackOverflow）
- 19. 一切 BFS 可以解决的问题

### 复杂度



- 时间复杂度：  $O(\text{方案个数} * \text{构造每个方案的时间})$ 
  - 树的遍历：  $O(n)$
  - 排列问题：  $O(n! * n)$
  - 组合问题：  $O(2^n * n)$

## 领扣例题

- [LintCode 67.二叉树的中序遍历\(遍历树\)](#)
- [LintCode 652.因式分解\(枚举所有情况\)](#)

## 代码模版

Java

```
1. 1. public ReturnType dfs(参数列表) {  
2.  
3. 2. if (递归出口) {  
4.  
5. 3. 记录答案;  
6.  
7. 4. return;  
8.  
9. 5. }  
10.  
11. 6. for (所有的拆解可能性) {  
12.  
13. 7. 修改所有的参数  
14.  
15. 8. dfs(参数列表);  
16.
```

17. 9. 还原所有被修改过的参数
- 18.
19. 10. }
- 20.
21. 11. `return` something 如果需要的话, 很多时候不需要 `return` 值除了分治的写法
- 22.
23. 12.}

## Python

1. 1.def dfs(参数列表):
- 2.
3. 2. `if` 递归出口:
- 4.
5. 3. 记录答案
- 6.
7. 4. `return`
- 8.
9. 5. `for` 所有的拆解可能性:
- 10.
11. 6. 修改所有的参数
- 12.
13. 7. dfs(参数列表)
- 14.
15. 8. 还原所有被修改过的参数
- 16.
17. 9. `return` something 如果需要的话, 很多时候不需要 `return` 值除了分治的写法

# 动态规划 Dynamic Programming

## 使用条件

- 使用场景：
  - 求方案总数(90%)
  - 求最值(80%)
  - 求可行性(80%)
- 不适用的场景：
  - 找所有具体的方案（准确率 99%）
  - 输入数据无序(除了背包问题外，准确率 60%~70%)
  - 暴力算法已经是多项式时间复杂度（准确率 80%）
- 动态规划四要素(对比递归的四要素)：
  - 状态 (State) -- 递归的定义
  - 方程 (Function) -- 递归的拆解
  - 初始化 (Initialization) -- 递归的出口
  - 答案 (Answer) -- 递归的调用
- 几种常见的动态规划：
- 背包型
  - 给出  $n$  个物品及其大小,问是否能挑选出一些物品装满大小为  $m$  的背包
  - 题目中通常有“和”与“差”的概念，数值会被放到状态中
  - 通常是二维的状态数组，前  $i$  个组成和为  $j$  状态数组的大小需要开  $(n + 1) * (m + 1)$
  - 几种背包类型：
    - 01 背包
      - 状态 state

$dp[i][j]$  表示前  $i$  个数里挑若干个是否能组成和为  $j$

方程 function

$dp[i][j] = dp[i - 1][j]$  or  $dp[i - 1][j - A[i - 1]]$  如果  $j \geq A[i - 1]$

$dp[i][j] = dp[i - 1][j]$  如果  $j < A[i - 1]$

第  $i$  个数的下标是  $i - 1$ ，所以用的是  $A[i - 1]$  而不是  $A[i]$

初始化 initialization

$dp[0][0] = \text{true}$

$dp[0][1...m] = \text{false}$

答案 answer

使得  $dp[n][v]$ ,  $0 \leq v \leq m$  为 true 的最大  $v$

- 多重背包

- 状态 state

$dp[i][j]$  表示前  $i$  个物品挑出一些放到  $j$  的背包里的最大价值和

方程 function

$dp[i][j] = \max(dp[i - 1][j - \text{count} * A[i - 1]] + \text{count} * V[i - 1])$

其中  $0 \leq \text{count} \leq j / A[i - 1]$

初始化 initialization

$dp[0][0..m] = 0$

答案 answer

$dp[n][m]$

- 区间型

- 题目中有 subarray / substring 的信息

- 大区间依赖小区间
- 用  $dp[i][j]$  表示数组/字符串中  $i, j$  这一段区间的最优值/可行性/方案总数
- 状态 state

$dp[i][j]$  表示数组/字符串中  $i, j$  这一段区间的最优值/可行性/方案总数

方程 function

$dp[i][j] = \max/\min/\text{sum}/\text{or}(dp[i, j \text{ 之内更小的若干区间}])$

- 匹配型

- 通常给出两个字符串
- 两个字符串的匹配值依赖于两个字符串前缀的匹配值
- 字符串长度为  $n, m$  则需要开  $(n + 1) \times (m + 1)$  的状态数组
- 要初始化  $dp[i][0]$  与  $dp[0][i]$
- 通常都可以用滚动数组进行空间优化
- 状态 state

$dp[i][j]$  表示第一个字符串的前  $i$  个字符与第二个字符串的前  $j$  个字符怎么样怎么样

(max/min/sum/or)

- 划分型

- 是前缀型动态规划的一种, 有前缀的思想
- 如果指定了要划分为几个部分:
  - $dp[i][j]$  表示前  $i$  个数/字符划分为  $j$  个 部分的最优值/方案数/可行性
- 如果没有指定划分为几个部分:

- $dp[i]$  表示前  $i$  个数/字符划分为若干个部分的最优值/方案数/可行性

- 状态 state

指定了要划分为几个部分:  $dp[i][j]$  表示前  $i$  个数/字符划分为  $j$  个部分的最优值/方案数/可行性

没有指定划分为几个部分:  $dp[i]$  表示前  $i$  个数/字符划分为若干个部分的最优值/方案数/可行性

- 接龙型

- 通常会给一个接龙规则, 问你最长的龙有多长
- 状态表示通常为:  $dp[i]$  表示以坐标为  $i$  的元素结尾的最长龙的长度
- 方程通常是:  $dp[i] = \max\{dp[j] + 1\}$ ,  $j$  的后面可以接上  $i$
- LIS 的二分做法选择性的掌握, 但并不是所有的接龙型 DP 都可以用二分来优化
- 状态 state

状态表示通常为:  $dp[i]$  表示以坐标为  $i$  的元素结尾的最长龙的长度

方程 function

$dp[i] = \max\{dp[j] + 1\}$ ,  $j$  的后面可以接上  $i$

## 复杂度

- 时间复杂度:

- $O(\text{状态总数} * \text{每个状态的处理耗费})$
- 等于  $O(\text{状态总数} * \text{决策数})$

- 空间复杂度:

- $O(\text{状态总数})$  (不使用滚动数组优化)
- $O(\text{状态总数} / n)$  (使用滚动数组优化,  $n$  是被滚动掉的那一个维度)

## 领扣例题

- [LintCode563. 背包问题 V\(背包型\)](#)
- [LintCode76. 最长上升子序列\(接龙型\)](#)
- [LintCode 476. 石子归并 V\(区间型\)](#)
- [LintCode 192. 通配符匹配 \(匹配型\)](#)
- [LintCode107. 单词拆分\(划分型\)](#)

## 堆 Heap

### 使用条件

- 20. 找最大值或者最小值(60%)
- 21. 找第  $k$  大(pop  $k$  次 复杂度  $O(n\log k)$ )(50%)
- 22. 要求  $\log n$  时间对数据进行操作(40%)

### 堆不能解决的问题

- 23. 查询比某个数大的最小值/最接近的值 (平衡排序二叉树 Balanced BST 才可以解决)

24. 找某段区间的最大值最小值（线段树 SegmentTree 可以解决）

25.  $O(n)$ 找第  $k$  大（使用快排中的 partition 操作）

## 领扣例题

- [LintCode 1274. 查找和最小的 K 对数字](#)
- [LintCode 919. 会议室 II](#)
- [LintCode 1512. 雇佣 K 个人的最低费用](#)

## 代码模板

Java 带删除特定元素功能的堆

```
1. 1.class ValueIndexPair {
2.
3. 2. int val, index;
4.
5. 3. public ValueIndexPair(int val, int index) {
6.
7. 4. this.val = val;
8.
9. 5. this.index = index;
10.
11. 6. }
12.
13. 7.}
14.
15. 8.class Heap {
16.
17. 9. private Queue<ValueIndexPair> minheap;
18.
```



```
19. 10. private Set<Integer> deleteSet;
20.
21. 11. public Heap() {
22.
23. 12.     minheap = new PriorityQueue<>((p1, p2) -> (p1.val - p2.val));
24.
25. 13.     deleteSet = new HashSet<>();
26.
27. 14. }
28.
29. 15.
30.
31. 16. public void push(int index, int val) {
32.
33. 17.     minheap.add(new ValueIndexPair(val, index));
34.
35. 18. }
36.
37. 19.
38.
39. 20. private void lazyDeletion() {
40.
41. 21.     while (minheap.size() != 0 && deleteSet.contains(minheap.peek().index)) {
42.
43. 22.         ValueIndexPair pair = minheap.poll();
44.
45. 23.         deleteSet.remove(pair.index);
46.
47. 24.     }
48.
49. 25. }
50.
51. 26.
```

52.

53. 27. `public ValueIndexPair top() {`

54.

55. 28. `lazyDeletion();`

56.

57. 29. `return minheap.peak();`

58.

59. 30. `}`

60.

61. 31.

62.

63. 32. `public void pop() {`

64.

65. 33. `lazyDeletion();`

66.

67. 34. `minheap.poll();`

68.

69. 35. `}`

70.

71. 36.

72.

73. 37. `public void delete(int index) {`

74.

75. 38. `deleteSet.add(index);`

76.

77. 39. `}`

78.

79. 40.

80.

81. 41. `public boolean isEmpty() {`

82.

83. 42. `return minheap.size() == 0;`

84.

85. 43. }

86.

87. 44.}

## Python 带删除特定元素功能的堆

```
1. 1.from heapq import heappush, heappop
2.
3. 2.
4.
5. 3.class Heap:
6.
7. 4.
8.
9. 5. def __init__(self):
10.
11. 6.     self.minheap = []
12.
13. 7.     self.deleted_set = set()
14.
15. 8.
16.
17. 9. def push(self, index, val):
18.
19. 10.     heappush(self.minheap, (val, index))
20.
21. 11.
22.
23. 12. def _lazy_deletion(self):
24.
25. 13.     while self.minheap and self.minheap[0][1] in self.deleted_set:
26.
27. 14.         heappop(self.minheap)
```

```
28.  
29. 15.  
30.  
31. 16. def top(self):  
32.  
33. 17.     self._lazy_deletion()  
34.  
35. 18.     return self.minheap[0]  
36.  
37. 19.  
38.  
39. 20. def pop(self):  
40.  
41. 21.     self._lazy_deletion()  
42.  
43. 22.     heappop(self.minheap)  
44.  
45. 23.  
46.  
47. 24. def delete(self, index):  
48.  
49. 25.     self.deleted_set.add(index)  
50.  
51. 26.  
52.  
53. 27. def is_empty(self):  
54.  
55. 28.     return not bool(self.minheap)
```

## 并查集 Union Find

## 使用条件

- 需要查询图的连通状况的问题
- 需要支持快速合并两个集合的问题

## 复杂度

- 时间复杂度 union  $O(1)$ , find  $O(1)$
- 空间复杂度  $O(n)$

## 领扣例题

- [LintCode 1070. 账号合并](#)
- [LintCode 1014. 打砖块](#)
- [LintCode 1813. 构造二叉树](#)

## 代码模板

Java

```
1. 1.class UnionFind {  
2.  
3. 2. private Map<Integer, Integer> father;  
4.  
5. 3. private Map<Integer, Integer> sizeOfSet;  
6.  
7. 4. private int numOfSet = 0;  
8.  
9. 5. public UnionFind() {
```

```
10.
11. 6. // 初始化父指针, 集合大小, 集合数量
12.
13. 7. father = new HashMap<Integer, Integer>();
14.
15. 8. sizeOfSet = new HashMap<Integer, Integer>();
16.
17. 9. numOfSet = 0;
18.
19. 10. }
20.
21. 11.
22.
23. 12. public void add(int x) {
24.
25. 13. // 点如果已经出现, 操作无效
26.
27. 14. if (father.containsKey(x)) {
28.
29. 15. return;
30.
31. 16. }
32.
33. 17. // 初始化点的父亲为 空对象 null
34.
35. 18. // 初始化该点所在集合大小为 1
36.
37. 19. // 集合数量增加 1
38.
39. 20. father.put(x, null);
40.
41. 21. sizeOfSet.put(x, 1);
42.
```

```
43. 22.    numOfSet++;
44.
45. 23.    }
46.
47. 24.
48.
49. 25.    public void merge(int x, int y) {
50.
51. 26.        // 找到两个节点的根
52.
53. 27.        int rootX = find(x);
54.
55. 28.        int rootY = find(y);
56.
57. 29.        // 如果根不是同一个则连接
58.
59. 30.        if (rootX != rootY) {
60.
61. 31.            // 将一个点的根变成新的根
62.
63. 32.            // 集合数量减少 1
64.
65. 33.            // 计算新的根所在集合大小
66.
67. 34.            father.put(rootX, rootY);
68.
69. 35.            numOfSet--;
70.
71. 36.            sizeOfSet.put(rootY, sizeOfSet.get(rootX) + sizeOfSet.get(rootY));
72.
73. 37.        }
74.
75. 38.    }
```

```
76.  
77. 39.  
78.  
79. 40. public int find(int x) {  
80.  
81. 41. // 指针 root 指向被查找的点 x  
82.  
83. 42. // 不断找到 root 的父亲  
84.  
85. 43. // 直到 root 指向 x 的根节点  
86.  
87. 44. int root = x;  
88.  
89. 45. while (father.get(root) != null) {  
90.  
91. 46. root = father.get(root);  
92.  
93. 47. }  
94.  
95. 48. // 将路径上所有点指向根节点 root  
96.  
97. 49. while (x != root) {  
98.  
99. 50. // 暂存 x 原本的父亲  
100.  
101. 51. // 将 x 指向根节点  
102.  
103. 52. // x 指针上移至 x 的父节点  
104.  
105. 53. int originalFather = father.get(x);  
106.  
107. 54. father.put(x, root);  
108.
```



109. 55.        x = originalFather;

110.

111. 56.        }

112.

113. 57.

114.

115. 58.        return root;

116.

117. 59.        }

118.

119. 60.

120.

121. 61.        public boolean isConnected(int x, int y) {

122.

123. 62.        // 两个节点连通 等价于 两个节点的根相同

124.

125. 63.        return find(x) == find(y);

126.

127. 64.        }

128.

129. 65.

130.

131. 66.        public int getNumOfSet() {

132.

133. 67.        // 获得集合数量

134.

135. 68.        return numOfSet;

136.

137. 69.        }

138.

139. 70.

140.

141. 71.        public int getSizeOfSet(int x) {

```
142.  
143. 72.    // 获得某个点所在集合大小  
144.  
145. 73.    return sizeOfSet.get(find(x));  
146.  
147. 74.    }  
148.  
149. 75.}
```

## Python

```
1.    1.class UnionFind:  
2.  
3.    2.    def __init__(self):  
4.  
5.    3.    # 初始化父指针, 集合大小, 集合数量  
6.  
7.    4.    self.father = {}  
8.  
9.    5.    self.size_of_set = {}  
10.  
11.    6.    self.num_of_set = 0  
12.  
13.    7.  
14.  
15.    8.    def add(self, x):  
16.  
17.    9.    # 点如果出现, 操作无效  
18.  
19.    10.    if x in self.father:  
20.  
21.    11.    return  
22.  
23.    12.    # 初始化点的父亲为 空对象 None
```

```
24.
25. 13.      # 初始化该点所在集合大小为 1
26.
27. 14.      # 集合数量增加 1
28.
29. 15.      self.father[x] = None
30.
31. 16.      self.num_of_set += 1
32.
33. 17.      self.size_of_set[x] = 1
34.
35. 18.
36.
37. 19.  def merge(self, x, y):
38.
39. 20.      # 找到两个节点的根
40.
41. 21.      root_x, root_y = self.find(x), self.find(y)
42.
43. 22.      # 如果根不是同一个则连接
44.
45. 23.      if root_x != root_y:
46.
47. 24.          # 将一个点的根变成新的根
48.
49. 25.          # 集合数量减少 1
50.
51. 26.          # 计算新的根所在集合大小
52.
53. 27.          self.father[root_x] = root_y
54.
55. 28.          self.num_of_set -= 1
56.
```

57. 29. self.size\_of\_set[root\_y] += self.size\_of\_set[root\_x]

58.

59. 30.

60.

61. 31. def find(self, x):

62.

63. 32. # 指针 root 指向被查找的点 x

64.

65. 33. # 不断找到 root 的父亲

66.

67. 34. # 直到 root 指向 x 的根节点

68.

69. 35. root = x

70.

71. 36. while self.father[root] != None:

72.

73. 37. root = self.father[root]

74.

75. 38. # 将路径上所有点指向根节点 root

76.

77. 39. while x != root:

78.

79. 40. # 暂存 x 原本的父亲

80.

81. 41. # 将 x 指向根节点

82.

83. 42. # x 指针上移至 x 的父节点

84.

85. 43. original\_father = self.father[x]

86.

87. 44. self.father[x] = root

88.

89. 45. x = original\_father

```
90.
91. 46. return root
92.
93. 47.
94.
95. 48. def is_connected(self, x, y):
96.
97. 49. # 两个节点连通 等价于 两个节点的根相同
98.
99. 50. return self.find(x) == self.find(y)
100.
101. 51.
102.
103. 52. def get_num_of_set(self):
104.
105. 53. # 获得集合数量
106.
107. 54. return self.num_of_set
108.
109. 55.
110.
111. 56. def get_size_of_set(self, x):
112.
113. 57. # 获得某个点所在集合大小
114.
115. 58. return self.size_of_set[self.find(x)]
```

## 字典树 Trie

### 使用条件

- 需要查询包含某个前缀的单词/字符串是否存在
- 字符矩阵中找单词的问题

## 复杂度

- 时间复杂度  $O(L)$  增删查改
- 空间复杂度  $O(N * L)$   $N$  是单词数,  $L$  是单词长度

## 领扣例题

- [LintCode 1221. 连接词](#)
- [LintCode 1624. 最大距离](#)
- [LintCode 1090. 映射配对之和](#)

## 代码模板

Java

```
1. 1.class TrieNode {
2.
3. 2. // 儿子节点
4.
5. 3. public Map<Character, TrieNode> children;
6.
7. 4. // 根节点到该节点是否是一个单词
8.
9. 5. public boolean isWord;
10.
11. 6. // 根节点到该节点的单词是什么
```

```
12.
13. 7. public String word;
14.
15. 8. public TrieNode() {
16.
17. 9.     sons = new HashMap<Character, TrieNode>();
18.
19. 10.     isWord = false;
20.
21. 11.     word = null;
22.
23. 12. }
24.
25. 13.}
26.
27. 14.
28.
29. 15. public class Trie {
30.
31. 16.     private TrieNode root;
32.
33. 17.     public Trie() {
34.
35. 18.         root = new TrieNode();
36.
37. 19.     }
38.
39. 20.
40.
41. 21.     public TrieNode getRoot() {
42.
43. 22.         return root;
44.
```

```
45. 23. }
46.
47. 24.
48.
49. 25. // 插入单词
50.
51. 26. public void insert(String word) {
52.
53. 27.     TrieNode node = root;
54.
55. 28.     for (int i = 0; i < word.length(); i++) {
56.
57. 29.         char letter = word.charAt(i);
58.
59. 30.         if (!node.sons.containsKey(letter)) {
60.
61. 31.             node.sons.put(letter, new TrieNode());
62.
63. 32.         }
64.
65. 33.         node = node.sons.get(letter);
66.
67. 34.     }
68.
69. 35.     node.isWord = true;
70.
71. 36.     node.word = word;
72.
73. 37. }
74.
75. 38.
76.
77. 39. // 判断单词 word 是不是在字典树中
```



```
78.
79. 40. public boolean hasWord(String word) {
80.
81. 41.     int L = word.length();
82.
83. 42.     TrieNode node = root;
84.
85. 43.     for (int i = 0; i < L; i++) {
86.
87. 44.         char letter = word.charAt(i);
88.
89. 45.         if (!node.sons.containsKey(letter)) {
90.
91. 46.             return false;
92.
93. 47.         }
94.
95. 48.         node = node.sons.get(letter);
96.
97. 49.     }
98.
99. 50.
100.
101. 51.     return node.isWord;
102.
103. 52. }
104.
105. 53.
106.
107. 54. // 判断前缀 prefix 是不是在字典树中
108.
109. 55. public boolean hasPrefix(String prefix) {
110.
```

```
111. 56.    int L = prefix.length();
112.
113. 57.    TrieNode node = root;
114.
115. 58.    for (int i = 0; i < L; i++) {
116.
117. 59.        char letter = prefix.charAt(i);
118.
119. 60.        if (!node.sons.containsKey(letter)) {
120.
121. 61.            return false;
122.
123. 62.        }
124.
125. 63.        node = node.sons.get(letter);
126.
127. 64.    }
128.
129. 65.    return true;
130.
131. 66. }
132.
133. 67. }
```

## LRU 缓存

### 使用条件

## 复杂度

- 时间复杂度 get  $O(1)$ , set  $O(1)$
- 空间复杂度  $O(n)$

## 领扣例题

- [LintCode 134. LRU 缓存](#)

## 代码模板

Java

```
1. // 用链表存放 cache, 表尾的点是 most recently, 表头的点是 least recently used
2. public class LRUCache {
3.     // 单链表节点
4.     class ListNode {
5.         public int key, val;
6.         public ListNode next;
7.
8.         public ListNode(int key, int val) {
9.             this.key = key;
10.            this.val = val;
11.            this.next = null;
12.        }
13.    }
14.}
```

```
15. // cache 的最大容量
16. private int capacity;
17. // cache 当前存储的容量
18. private int size;
19. // 单链表的 dummy 头
20. private ListNode dummy;
21. // 单链表尾
22. private ListNode tail;
23. // key => 数据节点之前的节点
24. private Map<Integer, ListNode> keyToPrev;
25.
26. // 构造函数
27. public LRUCache(int capacity) {
28.     this.capacity = capacity;
29.     this.keyToPrev = new HashMap<Integer, ListNode>();
30.     // dummy 点的 key 和 value 随意
31.     this.dummy = new ListNode(0, 0);
32.     this.tail = this.dummy;
33. }
34.
35. // 将 key 节点移动到尾部
36. private void moveToTail(int key) {
37.     ListNode prev = keyToPrev.get(key);
38.     ListNode curt = prev.next;
39.
40.     // 如果 key 节点已经再尾部, 无需移动
41.     if (tail == curt) {
42.         return;
43.     }
44.
```

```
45.    // 从链表中删除 key 节点
46.    prev.next = prev.next.next;
47.    tail.next = curt;
48.    curt.next = null;
49.
50.    // 分两种情况更新当前节点下一个节点对应的前导节点为 prev
51.    if (prev.next != null) {
52.        keyToPrev.put(prev.next.key, prev);
53.    }
54.    keyToPrev.put(curt.key, tail);
55.
56.    tail = curt;
57. }
58.
59. public int get(int key) {
60.    // 如果这个 key 根本不存在于缓存, 返回 -1
61.    if (!keyToPrev.containsKey(key)) {
62.        return -1;
63.    }
64.
65.    // 这个 key 刚刚被访问过, 因此 key 节点应当被移动到链表尾部
66.    moveToTail(key);
67.
68.    // key 节点被移动到链表尾部, 返回尾部的节点值, 即 tail.val
69.    return tail.val;
70. }
71.
72. public void set(int key, int value) {
73.    // 如果 key 已经存在, 更新 keyNode 的 value
74.    if (get(key) != -1) {
```

```
75.     ListNode prev = keyToPrev.get(key);
76.     prev.next.val = value;
77.     return;
78. }
79.
80.     // 如果 key 不存在于 cache 且 cache 未超上限
81.     // 再结尾存入新的节点
82.     if (size < capacity) {
83.         size++;
84.         ListNode curt = new ListNode(key, value);
85.         tail.next = curt;
86.         keyToPrev.put(key, tail);
87.
88.         tail = curt;
89.         return;
90.     }
91.
92.     // 如果超过上限, 删除链表头, 继续保存。此处可与上边合并
93.     ListNode first = dummy.next;
94.     keyToPrev.remove(first.key);
95.
96.     first.key = key;
97.     first.val = value;
98.     keyToPrev.put(key, dummy);
99.
100.     moveToTail(key);
101. }
102. }
```

```
1. # 单链表节点
2. class ListNode:
3.
4.     def __init__(self, key=None, value=None, next=None):
5.         self.key = key
6.         self.value = value
7.         self.next = next
8.
9. # 用链表存放 cache, 表尾的点是 most recently, 表头的点是 least recently used
10. class LRUCache:
11.
12.     def __init__(self, capacity):
13.         # key => 数据节点之前的节点
14.         self.key_to_prev = {}
15.         # 单链表 dummy 头节点
16.         self.dummy = ListNode()
17.         # 单链表尾节点
18.         self.tail = self.dummy
19.         # cache 的最大容量
20.         self.capacity = capacity
21.
22.     # 把一个点插入到链表尾部
23.     def push_back(self, node):
24.         self.key_to_prev[node.key] = self.tail
25.         self.tail.next = node
26.         self.tail = node
27.
28.     def pop_front(self):
29.         # 删除头部
30.         head = self.dummy.next
```

```
31.     del self.key_to_prev[head.key]
32.     self.dummy.next = head.next
33.     self.key_to_prev[head.next.key] = self.dummy
34.
35.     # change "prev->node->next...->tail"
36.     # to "prev->next->...->tail->node"
37.     def kick(self, prev):    #将数据移动至尾部
38.         node = prev.next
39.         if node == self.tail:
40.             return
41.
42.         # remove the current node from linked list
43.         prev.next = node.next
44.         # update the previous node in hash map
45.         self.key_to_prev[node.next.key] = prev
46.         node.next = None
47.
48.         self.push_back(node)
49.
50.     def get(self, key):
51.         # 如果这个 key 根本不存在于缓存, 返回 -1
52.         if key not in self.key_to_prev:
53.             return -1
54.
55.         prev = self.key_to_prev[key]
56.         current = prev.next
57.
58.         # 这个 key 刚刚被访问过, 因此 key 节点应当被移动到链表尾部
59.         self.kick(prev)
60.         return current.value
```



```
61.
62. def set(self, key, value):
63.     if key in self.key_to_prev:
64.         self.kick(self.key_to_prev[key])
65.         self.key_to_prev[key].next.value = value
66.         return
67.
68.     #如果 key 不存在, 则存入新节点
69.     self.push_back(LinkedListNode(key, value))
70.     #如果缓存超出上限, 删除头部节点
71.     if len(self.key_to_prev) > self.capacity:
72.         self.pop_front()
```