**算法设计与分析第五章作业**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **姓名** | **王靳** | **班级** | **计科十班** | **学号** | 220111012 |
| **第1题** |  | | | | |
| **第2题** |  | | | | |
| **第3题** |  | | | | |
| **第4题** |  | | | | |
| **第5题** |  | | | | |
| **总分** |  | | | | |
| **备注** | 作业提交截止时间： 2023年10月25日24:00，超过提交截至时间的作业视为无效。确因网络等特殊原因无法及时提交作业的学生，应至少提前1小时与助教联系沟通。作业提交邮箱：[hitsz\_algorithm@126.com。](mailto:hitsz_algorithm@126.com。)作业文件名命名方式： 第x章-x班-姓名-学号（例，第1章-1班-张三-220110101.docx）； 邮件主题为：第x章作业, x班，姓名，学号（例，第1章作业，1班，张三，220110101）。缺少这些信息的作业将被酌情扣分。 | | | | |

1. **（30分）**假定我们不再一直选择最早结束的活动，而是选择最晚开始的活动，前提仍然是与之前选出的所有活动兼容。描述如何利用这一方法设计贪心算法，并证明算法会产生最优解。



1. = 设计
2. == 想法
3. 我创建了一个 "自定义类型", 自定义类型中, 保存有: 索引(i), 开始时间(s), 结束时间(f), 然后我定义了一个比较的方法, 因为heapq需要用到这个, 这个就有点像C++中的重载运算符(<)之类的, 然后我还定义了对象的打印的方法, 类似于C++中重载(<<)
4. 定义了一个函数`is\_compatible`, 用来判断是否兼容的
5. 然后定义了一个辅助函数, 传入已经打包好的`act\_arr`,
6. 将`act\_arr`堆化, 其中, (剩余的)最晚结束的事件永远在堆顶
7. 先将最晚开始的那个元素, 放到传入到result中,
8. 然后进入循环, 循环的终止条件是: 堆空.
9. 循环不变式: 我们可以保证, result中的所有元素都是 "优化子结构"(看后面的证明), 然后我们取出来result中, 最近一次加入的元素last, 让last与堆顶i进行比较, 如果(last, i)兼容, 那么就将i加入到result中(优化子结构扩大), 否则, 抛弃i
10. == 代码实现
11. ```py
12. import heapq
13. import random
14. class activity:
15. def \_\_init\_\_(self, s: int, f: int, i: int) -> None:
16. self.s = s
17. self.f = f
18. self.i = i  *# 这个i表示标号*
19. def \_\_lt\_\_(self, other) -> bool:
20. return self.s > other.s
21. def \_\_str\_\_(self) -> str:
22. return f"i: {self.i}\ts: {self.s}\tf: {self.f}"
23. def is\_compatible(a: activity, b: activity) -> bool:
24. if a.s <= b.s and a.f <= b.s:
25. *# a在b之前, 并且a与b相容*
26. return True
27. if b.s <= a.s and b.f <= a.s:
28. return True
29. return False
30. def solution\_utils(act\_arr: list) -> list:
31. result = []
32. heapq.heapify(act\_arr)  *# 建堆*
33. last = heapq.heappop(act\_arr)
34. result.append(last)
35. while act\_arr:
36. i = heapq.heappop(act\_arr)
37. last = result[-1]
38. if is\_compatible(last, i):
39. result.append(i)
40. result.reverse()
41. return result
42. def solution(s\_arr: list, f\_arr: list) -> list:
43. act\_len: int = len(s\_arr)
44. act\_arr: list = [
45. activity(s, t, i) for (s, t, i) in zip(s\_arr, f\_arr, range(act\_len))
46. ]
47. random.shuffle(act\_arr)
48. result = solution\_utils(act\_arr)
49. return result
50. if \_\_name\_\_ == "\_\_main\_\_":
51. s\_arr: list = [3, 1, 5, 2, 5, 3, 8, 6, 8, 12]
52. f\_arr: list = [6, 4, 7, 5, 9, 8, 11, 10, 12, 14]
53. result = solution(s\_arr, f\_arr)
54. for i in result:
55. print(i)
56. print("hello world")
57. ```
58. = 证明
59. == 优化子结构
60. === 引理1 (base case)
61. 某个优化解包含 "活动n"(其中, "活动n"是最晚开始的活动)
62. + 平凡的: 如果一个优化解直接包含 "活动n"
63. + 构造: 如果一个优化解并不直接包含 "活动n", 那么有两种情况:
64. - "活动k"(优化解中的最后一个元素) 与 "活动n"不兼容: (构造法)  那么从优化解中去掉 "活动k", 并加上 "活动n", 这并不影响活动的数量, 因此这个操作是合法的
65. - "活动k"(优化解中的最优一个元素) 与 "活动n"兼容: 这个是伪命题, 因为可以加上 "活动n", 成为一个 "更优" 的解, 因此该情况下并不是一个 "优化解"
66. === 引理2 (递推)
67. 设 $S={1, 2, ..., n}$ 是 $n$ 个活动集合, $[s\_i, f\_i]$ 是活动的起始和终止时间, 且$s\_1 <= s\_2 <= ... <= s\_n$, 设$A$是$S$的 "活动安排问题" 的一个 "优化解", 且 包含 "活动n", 则 $A' = A - {n}$ 是 $S' = {i in S | s\_i <= s\_n}$ 的 "活动安排问题" 的 "优化解"
68. - 显然, $A'$ 中的活动是 "相容" 的
69. - 即证明: $A'$ 是 $S'$问题中, 最大的
70. (反证法) 若 $ exists B' and abs(B) > abs(A)$, 则有 $B = B' union {n} => abs(B) = abs(B') + 1 > abs(A') + 1 = abs(A)$
71. 然而, 这与 $A$ 是 "优化解" 的假设相悖
72. 故: "活动选择问题" 具有 "优化子结构"
73. == 贪心选择性
74. 设 $S = {1, 2, ..., n}$ 是 $n$ 个 "活动" 的集合, 其中 $s\_1 <= s\_2 <= ... <= s\_n$, let $l\_i$ 是集合 $S\_i = {j in S | f\_j <= s\_(l\_i + 1)}$ 中 具有 "最晚" 开始时间 $s\_{l\_i}$ 的活动, 设 $A$ 是 $S$ 包含 "活动n" 的 "优化解", 则$A = union\_(i=1)^(k) {l\_i}$
75. (归纳法)
76. - 当 $abs(A) = 1$ 时, 由 "引理1", 命题成立
77. - 假设 $abs(A) < k$时, 命题(即 $A$ 是 "优化解")成立 (归纳假设)
78. - 当 $abs(A) = k$时, 由 "引理2", $A=A\_(n-1) union {n}$, 其中 $A\_(n-1)$ 是 $S\_(n-1) = {j in S | f\_j <= s\_n}$ 的 "优化解", 由假设, $A\_(n-1)$ 是 "优化解", 并且 "引理1" 指出, $exists A\_n$ 包含 "活动n", 故 $A\_n = A\_(n-1) union {n}$ 是一个 "优化解", 得证

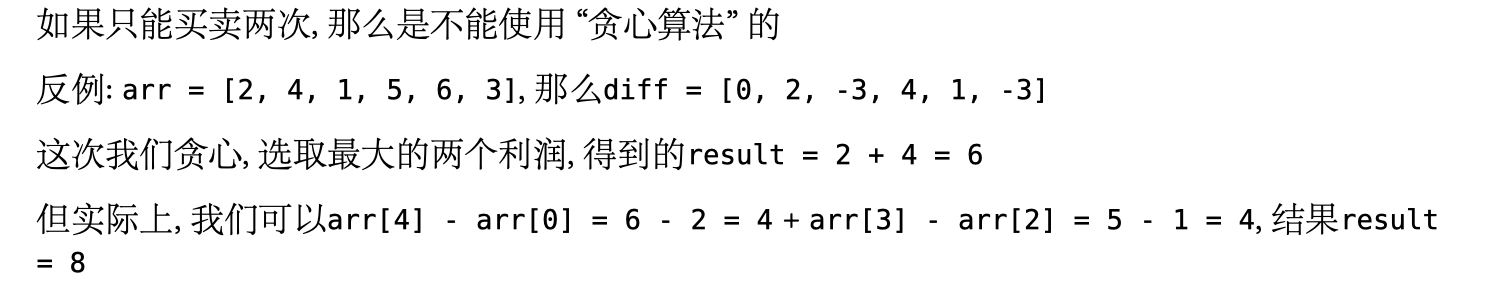
**2、（30分）**考虑给定一个数组prices，包含n个元素，它的第i个元素是一支给定股票第i天的价格。

1）设计一个时间复杂度为 O(n) 的贪心算法来计算这n天买卖这支股票可以获得的最大利润，文字描述清楚思路即可（20分）。**注意，你可以尽可能地完成更多的交易，即可以多次买卖这支股票，但是你不能同时参与多笔交易（你必须在再次购买前出售掉之前购买的股票）。**

****

1. def solution\_utils(diff: list) -> int:
2. result = 0
3. diff\_len = len(diff)
4. for i in range(diff\_len):
5. if diff[i] >= 0:
6. result += diff[i]
7. return result
8. def solution(arr: list) -> int:
9. *# 买卖股票最佳时机*
10. arr\_len = len(arr)
11. diff = [0] \* arr\_len  *# 第0天, 利润为0*
12. for i in range(1, arr\_len):
13. diff[i] = arr[i] - arr[i - 1]
14. result = solution\_utils(diff)
15. return result
16. if \_\_name\_\_ == "\_\_main\_\_":
17. print([0] \* 10)
18. print("hello world")
19. arr = [7, 1, 5, 10, 3, 6, 4]
20. result = solution(arr)
21. print(result)

2）假设这n天内只允许两次交易，此问题是否还能使用贪心算法解决？如果可以请描述算法，否则设计出反例的prices数组使得贪心算法不能保证得到最优解。（10分）



示例：  
prices = [7, 1, 5, 3, 6, 4]  
可获得的最大利润为7。

**3、（40分）编程题：**

题目描述：

给定一个长度为 n 的整数数组 nums，初始位置为 nums[0]。

每个元素 nums[i] 表示从索引 i 向前跳转的最大长度。换句话说，如果你在 nums[i] 处，你可以跳转到任意 nums[i + j] 处:

0 <= j <= nums[i] ，且i + j < n。

求解：返回到达 nums[n - 1] 的最小跳跃次数。生成的测试用例可以到达 nums[n - 1]。

示例 1:

输入: nums = [2,3,1,1,4]

输出: 2

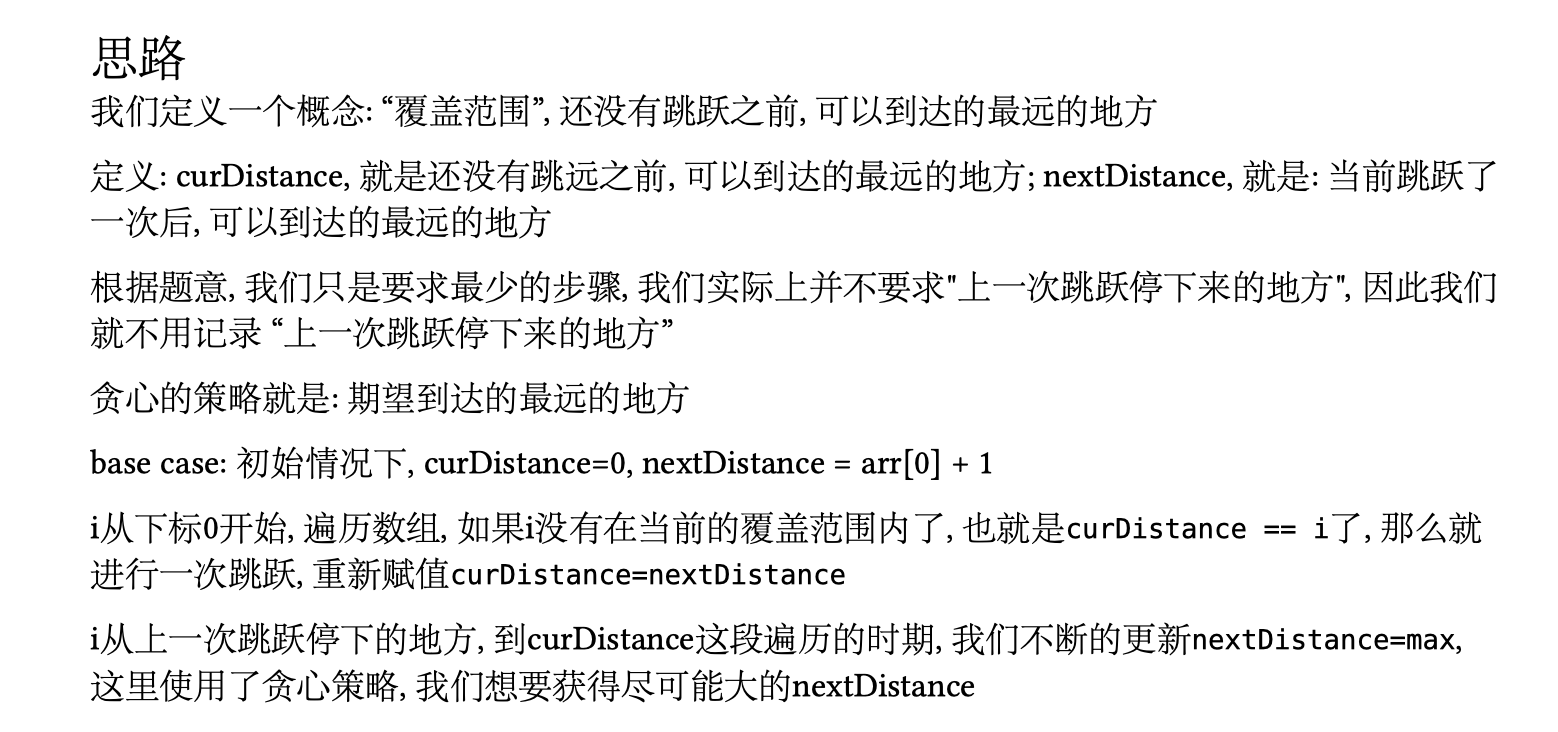
解释: 跳到最后一个位置的最小跳跃数是 2。

# 从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。

示例 2:

输入: nums = [2,3,0,1,4]

输出: 2



1. def solution(arr: list) -> int:
2. arr\_len: int = len(arr)
3. if arr\_len == 0:
4. return -1
5. if arr\_len == 1:
6. return 0  *# 不用跳跃了*
7. result: int = 0
8. curDistance: int = 0  *# 当前步覆盖范围*
9. nextDistance: int = arr[0] + 1  *# 下一步覆盖范围*
10. for i in range(arr\_len):
11. nextDistance = max(arr[i] + 1, nextDistance)
12. if i == curDistance:
13. result += 1
14. curDistance = nextDistance
15. if nextDistance >= arr\_len - 1:
16. break
17. return result
18. if \_\_name\_\_ == "\_\_main\_\_":
19. arr = [2, 3, 1, 1, 4]
20. result = solution(arr)
21. print(result)
22. print("hello world")

提示:

1 <= nums.length <= 104

0 <= nums[i] <= 1000

题目保证可以到达 nums[n-1]

要求：运用贪心思想作答，请写出分析过程（20分），并用一种语言（最好是C、C++、Python）实现你的思路（20分），上交作业时请将代码一并提交，代码粘贴在交作业的word里面，复杂度尽可能低。