期中大作业

范潇　2254298

2024 年 5 月 16 日

题目 1. (时间复杂度分析) 分析快速排序的平均时间复杂性；并证明快速排序的最坏时间复杂性是 $O(n^2)$。

解答.

Algorithm 1: Quick-Sort(A,l,r)

```
if l<r then
    p = randomized-Partition(A,l,r)
    Quick-Sort (A,l,p-1)
    Quick-Sort (A,p+1,r)
end if
```

Function randomized-Partition(A,l,r)

```
// 默认以最后一个元素作为 pivot;l,r 分别为子序列的左右端点下标
// 返回 pivot 的最终下标
pivot = randint (l,r)// [l..r] 中随机抽取一个数字
Swap (A[r],A[pivot])
i = l - 1
for j = l to r-1 do
    if Compare(A[j],A[r]) then // A[j]<A[r]
        i = i + 1
        Swap (A[j],A[i])
    end if
end for
Swap (A[i+1],A[r])
return i+1// 最终 A[i+1] 左侧的元素均小于它，右侧的大于等于它
```

快速排序的时间复杂度主要由三部分组成：1）比较次数 $f(n)$，即调用 Compare 函数的次数；2）交换元素次数 $g(n)$，即调用 Swap 函数的次数；3）其他语句所消耗的时间 $h(n)$。由以上伪代码易知，$f(n)$ 既是 $g(n)$ 的渐进上界，也是 $h(n)$ 的渐进上界。因此，想要分析快速排序的平均时间复杂度，只需分析调用 Compare 函数的平均次数的渐进界即可。而所谓“平均”，是指由于 randint 函数造成的各个可能情况

的平均。调用 Compare 函数的平均次数便等于单次运行 Quick-Sort 时调用 Compare 函数的次数的期望

$$E[\sum_{i=1}^{n}\sum_{j=1}^{n} I_{ij}] = \sum_{i=1}^{n}\sum_{j=1}^{n} E[I_{ij}]$$

这里 I_{ij} 是指事件“调用 Compare(a_i, a_j)”发生的次数。

显然

$$E[I_{ii}] = 0, i = 1, \cdots, n$$

由于当 I_{ij} 发生时，a_i, a_j 其中一者被选为了枢纽 pivot，不会参与到后续的递归中，因此

$$I_{ij} + I_{ji} \le 1, i, j = 1, \cdots, n, i \ne j$$

所以 I 可以视为示性函数。同时，由于在 randomized-Partition 中，会根据元素和枢纽的相对大小进行重新排序，比枢纽小的移动到其左侧，否则到其右侧，因此，如果 $a_k(i < k < j)$ 被设为枢纽，则之后 Compare(a_i, a_j) 不再可能被调用，因为 a_i, a_j 已经被划分到两个不重叠的区间中。同时，对于长度大于等于 2 的区间,其中至少会有一个元素在某一时刻被选为枢纽。并且枢纽的选取是等概率的,$a_i, \cdots, a_k, \cdots, a_j$ 的地位是相同的，所以其中任何一者都可能成为它们之中最先被选为枢纽的元素，概率为 $\frac{1}{j-i+1}$，而只有当 a_i 或 a_j 被最先选取时，Compare(a_i, a_j) 或 Compare(a_j, a_i) 才会发生。因此

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=1}^{n} E[I_{ij}] &= \sum_{i}\sum_{j\ne i} E[I_{ij}] \\
&= \sum_{i}\sum_{j\ne i} Pr\{I_{ij}\} \\
&= \sum_{i}\sum_{j\ne i} Pr\{\text{先选择了} a_j\} \\
&= \sum_{i}\sum_{j\ne i} \frac{1}{|j-i+1|} \\
&= \sum_{i}\sum_{j>i} \frac{2}{j-i+1}
\end{aligned}$$

又因为

$$\sum_{i}\sum_{j>i} \frac{2}{j-i+1} = \sum_{i=1}^{n}\sum_{k=1}^{n-i} \frac{2}{k+1} \le n\sum_{k=1}^{n} \frac{2}{k} = O(n \lg n)$$

所以快速排序的平均时间复杂度为 $O(n \lg n)$

下面用数学归纳法证明最坏情况下，快速排序的渐进上界为 $O(n^2)$。当 $l - r + 1 = n$ 时，由于循环变量由 l 递增到 $r-1$，所以 randomized-Partition 的时间复杂度为 $\Theta(n)$。又由于 Quick-Sort 函数体内递归调用自身时传入的参数分别为 $l, p-1$ 和 $p+1, r$，设最坏时间复杂度为 $T(n)$，有

$$T(n) = \max_{0 \le q \le n-1}\{T(q) + T(n-1-q) + \Theta(n)\}$$

设当 $n \le k-1$ 时，$T(n) \le cn^2$，这里取足够大的 c 使得 $n=1$ 时成立。则

$$
\begin{aligned}
T(k) &\le \max_{0 \le q \le k-1} \{cq^2 + c(k-1-q)^2 + \Theta(k)\} \\
&= \max_{0 \le q \le k-1} \{2cq^2 - 2(k-1)q + c(k-1)^2 + \Theta(k)\} \\
&\le c(k^2 - 2k + 1) + dk \\
&\le ck^2
\end{aligned}
$$

其中常数 d 足够大。因此快速排序的最坏时间复杂度 $T(n) = O(n^2)$。

题目 2. (选择问题) 给定线性序集中 n 个元素和一个整数 k，$1 \le k \le n$，要求找出这 n 个元素中第 k 小的元素，(这里给定的线性集是无序的)。下面三种是可行的方法：

1. 基于堆的选择：不需要对全部 n 个元素排序，只需要维护 k 个元素的最大堆，即用容量为 k 的最大堆存储最小的 k 个数，总费时 $O(k + (n-k) \cdot \log k)$
2. 随机划分线性选择：在最坏的情况下时间复杂度为 $O(n^2)$，平均情况下期望时间复杂度为 $O(n)$。
3. 利用中位数的线性时间选择：选择中位数的中位数作为划分的基准，在最坏情况下时间复杂度为 $O(n)$ 。

请给出以上三种方法的算法描述，用你熟悉的编程语言实现上述三种方法。并通过实际用例测试，绘出三种算法的运行时间随 k 和 n 变化情况的对比图（表），特别是 n 较大时方法（2）和（3）的对比。

解答. 在基于堆的选择中，维护一个大小为 k 的大顶堆，因此堆顶元素至少为第 k 大的元素。遍历未在堆中的元素，如果小于堆顶元素，则说明堆顶元素至少为第 $k+1$ 大的元素，可以移除，用当前元素替代。否则，说明当前元素至少为第 $k+1$ 大的元素，可以舍弃。伪代码如下。

Algorithm 2: heapSelect(nums,k)

```
// nums 下标从 0 开始
heap = build(nums[0..k-1])// 初始化大顶堆，大小为 k
i = k
while i<len (nums) do
    if nums[i]<maxItem(heap) then replace(heap,nums[i]) // 如果当前项小于堆中的最大项，则
      用当前项替换
    i += 1
end while
```

Function heapify(heap,i)

```
if lchild(i)<n and heap[lchild(i)] > heap[i] then
    Swap (heap[i],heap[lchild])
end if
if rchild(i)<n and heap[rchild(i)] > heap[i] then
    Swap (heap[i],heap[rchild])
end if
```

Function build(nums)

```
heap = nums.copy()
i = len (nums)
while i>0 do
    heapify(nums,i)
    i -= 1
end while
return heap
```

Function maxItem(heap)

```
return heap[1]
```

Function replace(heap,newItem)

```
heap[1] = newItem
heapify(heap,1)
```

下面给出的是随机划分线性选择的伪代码。

Algorithm 3: RandomizeSelect(nums,k,left,right)

```
// nums 下标从 0 开始
if l<r then
    pivot = RandomizePartition (nums,left,right)// 返回枢纽在 nums[left..right] 中的排名
    if pivot == k then  return nums[left+pivot-1]
    if pivot>k then
        return  RandomizeSelect(nums,k,left,left+pivot-2)
    else
        return  RandomizeSelect(nums,k-pivot,left+pivot,right)
    end if
end if
```

```
Function RandomizePartition(nums,k,left,right)
l = left - 1// 目前确定小于枢纽的最后一个下标
r = left// 待确定的第一个下标
pivot = randint (left,right)
Swap (nums[pivot],nums[right])// 随机选取枢纽
while r<right do
    if nums[r]<nums[right] then // 当前元素小于枢纽
        l += 1
        Swap (nums[r],nums[l])
    end if
    r += 1
end while
Swap (nums[l+1],nums[right])// 放置枢纽
return l+2-left// 返回的是枢纽的位序，从 1 开始
```

下面给出的是利用中位数的线性时间选择的伪代码。

```
Algorithm 4: linearSelect(nums,k)
l = len (nums)
while l mod 5 ≠ 0 do // 使长度为 5 的整数
    if k == l then return nums[l]
    nums.popMax()// 移除最大值
    l -= 1
end while
medians = getMedianPerFiveElement()// 5 个元素一组求中位数
mid = linearSelect(medians,⌊(len (medians)+1)/2⌋)// 返回中位数
pos = partion(nums,nums.index(mid)) + 1// +1 得到从 1 开始的位序
if pos == k then
    return nums[pos-1]
else if pos>k then
    select(nums[0..pos-2],k)
else
    return select(nums[pos..end],k-pos)
end if
```

我利用 python 语言完成了算法的实现。环境为:

1. Pycharm 2023.3.5

2. Python 3.11.7

3. numpy 1.24.3

4. pandas 1.5.3

我在该实验中测试了分别测试了数据量为 10、100、1k、10k、20k、30k、40k、50k、60k、70k、80k、90k 和 100k 的样例。每个样例中，k 的选取方式分别为“最小值”、“下四分位”、“中位数”、“上四分位”、“最大值”。

实验结果如下图所示。从中可以得到以下结论：

1. 当 $|k - \frac{n}{2}|$ 较大时，heapSelect 的性能较好。当 $|k - \frac{n}{2}|$ 较小时，和 linearSelect 和 randomSelect 相比常数项过高。

2. linearSelect 的常数项和 randomSelect 的常数项相比较高

3. randomSelect 的性能波动较为明显，而 linearSelect 相比更加稳定。

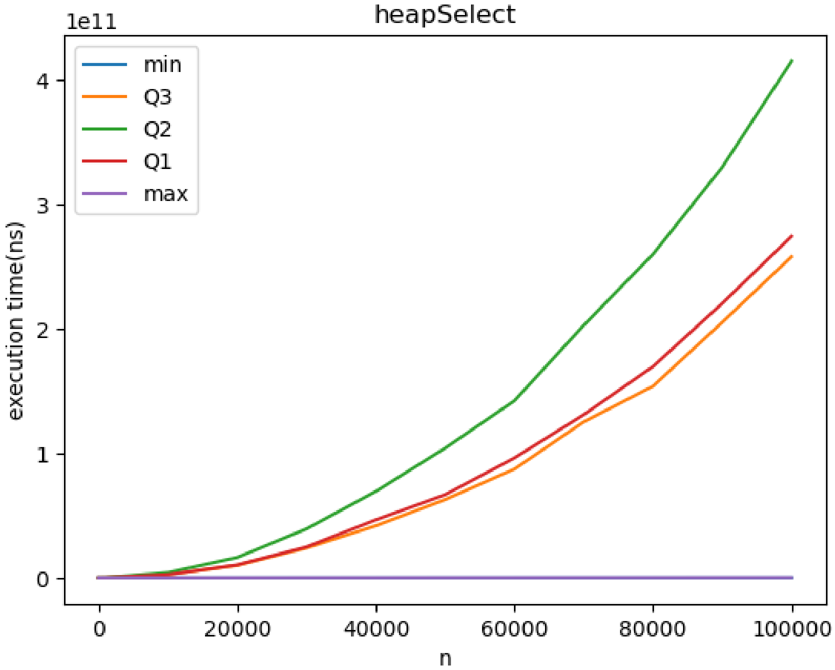

图 1: heapSelect 时间复杂度

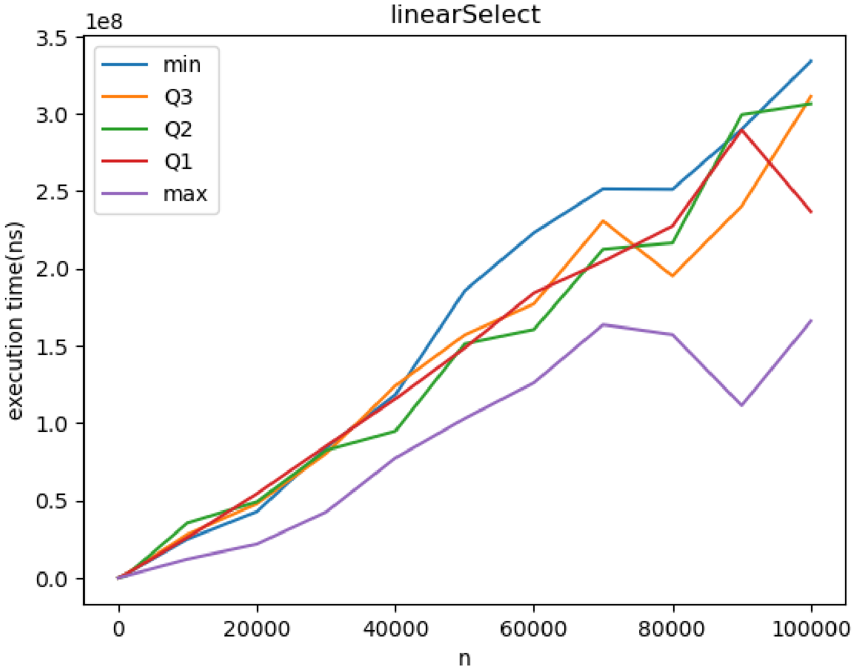

图 2: linearSelect 时间复杂度

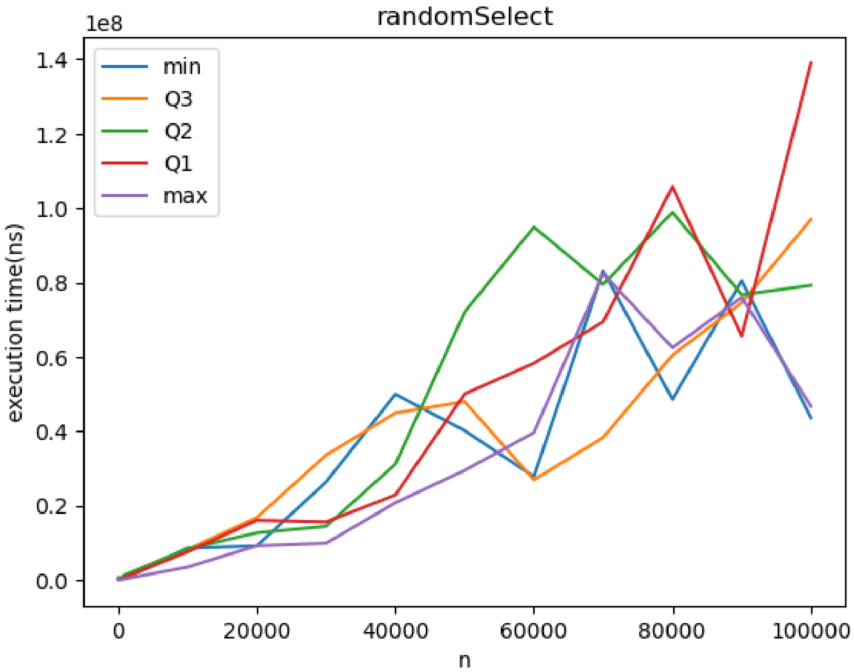

图 3: randomSelect 时间复杂度

题目 3. (动态规划) 给定一列值 $< v_1, v_2, \cdots, v_n >, v_i > 0, i = 1, \cdots, n$，求集合 $\{1, 2, \cdots, n\}$ 的一个二划分 $\{A, B\}$，使得 $\sum_{i\in B} v_i - \sum_{i\in A} v_i > 0$ 且取最小值。输出分为 4 行，分别为

1. $\sum_{i\in A} v_i$
2. A 中元素（升序）
3. $\sum_{i\in B} v_i$
4. B 中元素（升序）

输入输出示例如下:

Input:

2 1 3 1 5 2 3 4

Output:

10

1 3 5

11

2 4 6 7 8

要求首先通过暴力方法求解。然后使用动态规划方法进行求解，要求先从动态规划的角度对该题目进行分析，并写出递归属性。同时还应涉及用于存储中间量的数组。对于两种方式，都要求分析时间复杂度，并通过实验来检验分析结果。

解答. 暴力方法即枚举 $\{1, 2, \cdots, n\}$ 的所有可能的二划分。由于对于 $i, 1 \le i \le n$，要么它属于 A，要么它属于 B，而这可以用 0 和 1 来表示。又因为当我们确定完所有元素的归属后，也就确定了一个划分，所以可以用长度为 n 的 01 串来编码划分。通过递增的方式来遍历所有长度为 n 的 01 串，并通过位运算将信息提取出来。遍历的 01 串数量为 2^n，对于每个 01 串，要计算它对应的差值，需要 $\Theta(n)$ 次计算，所以暴力方法的时间复杂度为 $\Theta(n2^n)$，其中 n 为输出数组的长度。伪代码如下。

Function extract(i,j)

```
return (i»j)&1/* 提取出第 j 位 */
```

Function extractSolution(i,n)

```
A = []
for j in [0..n-1] do
    if extract(i,j)==1 then A.append(j)
end for
return A
```

Algorithm 5: brute-force(values)

```
l = len (values)
MAX = 1«l// 所有长度为 n 的 01 串小于该值
i = 0// 01 串
ans = 0// 记录最佳的 01 串
bestdiff = sum (values)// 记录最佳的差值
while i<MAX do
    diff = 0// 当前 01 串对应的差值
    for j in [0..l-1] do
        if  extract(i,j) == 1
         then // 提取出第 j 位
            diff -= values[j]
        else
            diff += values[j]
        end if
    end for
    if diff ≥0 and diff<bestdiff then // 更新最佳 01 串
        bestdiff = diff
        ans = i
    end if
    i += 1
end while
return extractSolution(ans,l)
```

我利用 python 语言完成了算法的实现。环境为：Python 3.11.7。

结果实验得到如下运行时间：

输入长度 n	5	10	15	20	25
运行时间 (ns)	37100	1202700	56254000	2161726300	86466016500

从图4可以看出，实验结果符合由分析得到的时间复杂度。

想要使得差值 $\sum_{i\in B} v_i - \sum_{i\in A} v_i$ 最小，等同于"使得 $\sum_i v_i - 2\sum_{i\in A} v_i > 0$ 或 $\frac{1}{2}\sum_i v_i - \sum_{i\in A} v_i > 0$ 最小"。由于输入数组中的元素都是正整数，可以继续转换为"使得 $\lfloor \frac{1}{2}\sum_{i\le n} v_i \rfloor \ge \sum_{i\in A} v_i$ 且 $\sum_{i\in A} v_i$ 尽可能地大 $(A \subseteq \{1,2,\cdots,n\})$"。

问题"使得 $\sum_{i\in A} v_i \le x$ 且 $\sum_{i\in A} v_i$ 尽可能地大 $(A \subseteq \{1,2,\cdots,y\})$"的最优解对应的 $\sum_{i\in A} v_i$ 记为 $m[x,y]$，则有

$$m[i,j] = \begin{cases} m[i,j-1] & v_j > i \\ \max\{m[i-v_j,j-1]+v_j, m[i,j-1]\} & else \end{cases}$$

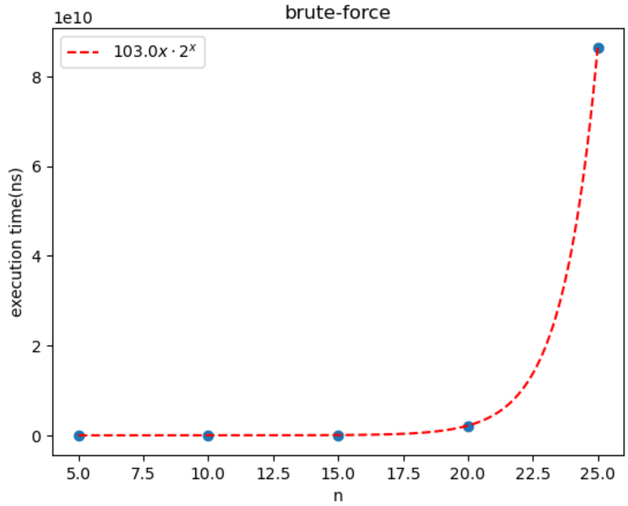

图 4: 暴力方法时间复杂度

显然, 当 $v_j > i$ 时, m[i,j] 中不可能包含 v_j, 因为这违反了不等式约束, 从而 m[i,j] 与 m[i,j-1] 对应的合法的解相同, 最优解也相同, 从而 m[i,j]=m[i,j-1]。当 $v_j \le i$ 时, 由反证法易得, 若 $j \notin A$, 则 $A \subseteq \{1, 2, \cdots, j-1\}$, 对应的最优解为 $m[i, j-1]$ 的最优解; 若 $j \in A$, 则对应的最优解为 $\{j\} \cup A'$, 其中 A' 对应 $m[i-v_j, j-1]$ 的最优解。

记 $M = \lfloor \frac{1}{2} \sum_{i \le n} v_i \rfloor$。原问题的最优解对应的和便是 $m[M, n]$。想要计算得到 $m[M, n]$, 共需计算 $M \cdot n$ 个值, 所以时间复杂度为 $\Theta(sn)$, 其中 $s = \sum_{i=1}^{n} v_i$, n 为输入元素的个数。

为了回溯还原最优解, 需要保存过程中计算得到的 $m[i, j]$ 值。回溯过程从 m[M,n] 开始, 若 $m[x, y] == m[x, y-1]$, 则说明 $m[x, y]$ 的最优解的构成和 $m[x, y-1]$ 相同。否则, 说明 $m[x, y]$ 对应的最优解中, $v_y \in A$, 剩余部分由是 $m[x - v_y, y-1]$ 对应的最优解。回溯过程所需要的时间复杂度为 $\Theta(s + n)$

当 $j = 0$ 时, 说明 $m[i, j]$ 对应的最优解中 $A = \varnothing$, 所以 $m[i, 0] = 0$。当 $i = 0$ 时, 由于 $v_k > 0$, 最优解中 $A = \varnothing$, 从而 $m[0, j] = 0$。

伪代码以及实验结果如下, 可以看出, 实验结果符合由分析得出的时间复杂度。

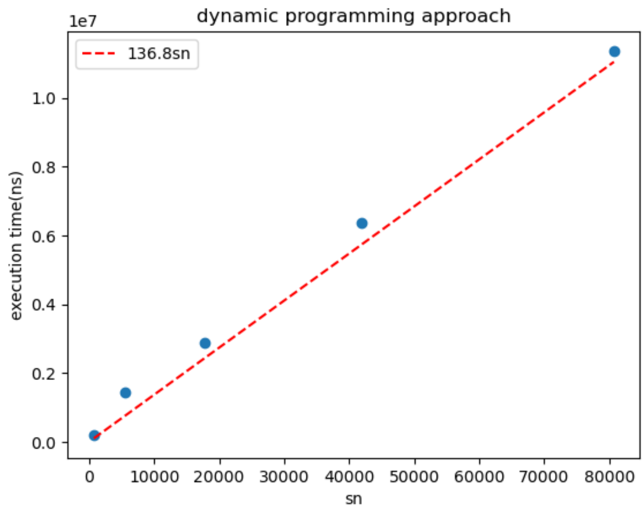

图 5: 动态规划方法时间复杂度

Algorithm 6: DynamicProgrammingApproach(values)

```
// values 下标从 1 开始
M = ⌊sum (values)/2⌋
l = len (values)
初始化一个 (M+1)×(l+1) 的全零数组 dp
for i in [1..M] do
    for j in [1..l] do
        if values[j]>i then
            dp[i][j] = dp[i][j-1]
        else
            dp[i][j] = max(dp[i][j-1],dp[i-values[j]][j-1]+values[j])
        end if
    end for
end for
A=[]// 开始回溯
x = M
y = l
while x and y do
    while y and dp[x][y] == dp[x][y-1] do y -= 1
    if y==0 then break
    A.append(y)
    x -= values[y-1]
    y -= 1
end while
return A
```