下面将解释你提到的几种排序算法的最坏情况和平均情况（期望）时间复杂度的区别：

### 1. 插入排序（Insertion Sort）

- \*\*最坏情况时间复杂度\*\*：\(O(n^2)\)

- \*\*平均情况时间复杂度\*\*：\(O(n^2)\)

插入排序在最坏情况下（即输入数组完全逆序）和平均情况下都表现为二次时间复杂度。这是因为每次插入都可能需要比较并移动之前的所有元素。

### 2. 归并排序（Merge Sort）

- \*\*最坏情况时间复杂度\*\*：\(O(n \log n)\)

- \*\*平均情况时间复杂度\*\*：\(O(n \log n)\)

归并排序无论在最坏情况还是平均情况下，时间复杂度都是 \(O(n \log n)\)，因为归并排序总是遵循分而治之的策略，将数组分成两半，然后排序合并。

### 3. 堆排序（Heapsort）

- \*\*最坏情况时间复杂度\*\*：\(O(n \log n)\)

- \*\*平均情况时间复杂度\*\*：\(O(n \log n)\)

堆排序在建立堆和从堆中移除元素时，每个操作都具有对数时间复杂度，因此总的时间复杂度是 \(O(n \log n)\)，无论是在最坏情况还是平均情况下。

### 4. 快速排序（Quicksort）

- \*\*最坏情况时间复杂度\*\*：\(O(n^2)\)

- \*\*平均情况时间复杂度\*\*：\(O(n \log n)\)（期望）

快速排序的最坏情况出现在每次分区操作都分得非常不平均时，例如每次都选到了最小或最大的元素作为枢轴。但在平均情况下，由于分区相对均匀，快速排序通常表现出较好的 \(O(n \log n)\) 时间复杂度。

### 5. 计数排序（Counting Sort）

- \*\*最坏情况时间复杂度\*\*：\(O(k + n)\)

- \*\*平均情况时间复杂度\*\*：\(O(k + n)\)

计数排序的时间复杂度依赖于输入中数据的范围 \(k\) 和元素数量 \(n\)，适用于数据范围小而数据量大的场景。

### 6. 基数排序（Radix Sort）

- \*\*最坏情况时间复杂度\*\*：\(O(d \cdot (n + k))\)

- \*\*平均情况时间复杂度\*\*：\(O(d \cdot (n + k))\)

基数排序的复杂度依赖于位数 \(d\)（决定了需要进行多少次分配和收集的过程），数据范围 \(k\) 和元素数量 \(n\)。

### 7. 桶排序（Bucket Sort）

- \*\*最坏情况时间复杂度\*\*：\(O(n^2)\)

- \*\*平均情况时间复杂度\*\*：\(O(n)\)（平均情况）

桶排序的性能取决于输入数据的分布。在最理想的情况下，数据均匀分布到每个桶中，每个桶使用快速排序或插入排序都能快速完成。但如果数据极不均匀，某个桶中的元素数量可能接近 \(n\)，则单个桶的排序时间可能达到 \(O(n^2)\)。

**Bucket sort**, or **bin sort**, is a [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm" \o "Sorting algorithm) that works by distributing the elements of an [array](https://en.wikipedia.org/wiki/Array_data_structure" \o "Array data structure) into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. It is a [distribution sort](https://en.wikipedia.org/wiki/Distribution_sort" \o "Distribution sort), a generalization of [pigeonhole sort](https://en.wikipedia.org/wiki/Pigeonhole_sort" \o "Pigeonhole sort) that allows multiple keys per bucket, and is a cousin of [radix sort](https://en.wikipedia.org/wiki/Radix_sort" \o "Radix sort) in the most-to-least significant digit flavor. Bucket sort can be implemented with comparisons and therefore can also be considered a [comparison sort](https://en.wikipedia.org/wiki/Comparison_sort" \o "Comparison sort) algorithm. The [computational complexity](https://en.wikipedia.org/wiki/Analysis_of_algorithms" \o "Analysis of algorithms) depends on the algorithm used to sort each bucket, the number of buckets to use, and whether the input is uniformly distributed.

Bucket sort works as follows:

1. Set up an array of initially empty "buckets".
2. **Scatter**: Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket.
4. **Gather**: Visit the buckets in order and put all elements back into the original array.

## Pseudocode[[edit](https://en.wikipedia.org/w/index.php?title=Bucket_sort&action=edit&section=1" \o "Edit section: Pseudocode)]

**function** bucketSort(array, k) **is**

buckets ← new array of k empty lists

M ← 1 + the maximum key value in the array

**for** i = 0 **to** length(array) **do**

insert *array[i]* into *buckets[floor(k × array[i] / M)]*

**for** i = 0 **to** k **do**

nextSort(buckets[i])

**return** the concatenation of buckets[0], ...., buckets[k]

基数排序（英语：Radix sort）是一种非比较型整数排序算法，其原理是将整数按位数切割成不同的数字，然后按每个位数分别比较。 由于整数也可以表达字符串（比如名字或日期）和特定格式的浮点数，所以基数排序也不是只能使用于整数。 它是这样实现的：将所有待比较数值（正整数）统一为同样的数位长度，数位较短的数前面补零。

### 练习 5.1 概率计数

\*\*a. 证明在执行 \( n \) 次 INCREMENT 操作后计数器所表示的预期值恰好是 \( n \)\*\*

\*\*解答：\*\*

考虑我们对计数器进行 \( n \) 次 INCREMENT 操作。假设计数器的当前值是 \( i \)，每次 INCREMENT 操作增加 1 的概率是 \( \frac{1}{n\_{i+1} - n\_i} \)，保持不变的概率是 \( \frac{n\_{i+1} - 1}{n\_{i+1} - n\_i} \)。

我们定义随机变量 \( X \) 表示计数器的值。初始值 \( X\_0 = 0 \)。每次 INCREMENT 操作后，计数器的变化可以表示为：

\[ X\_{i+1} = X\_i + I\_i \]

其中 \( I\_i \) 是指示变量，表示第 \( i \) 次 INCREMENT 操作是否使计数器增加 1：

\[ I\_i = \begin{cases}

1, & \text{以概率 } \frac{1}{n\_{i+1} - n\_i} \\

0, & \text{以概率 } \frac{n\_{i+1} - 1}{n\_{i+1} - n\_i}

\end{cases} \]

因此，指示变量 \( I\_i \) 的期望值为：

\[ E[I\_i] = \frac{1}{n\_{i+1} - n\_i} \]

由于我们进行 \( n \) 次 INCREMENT 操作，总期望值为：

\[ E[X] = \sum\_{i=0}^{n-1} E[I\_i] = \sum\_{i=0}^{n-1} 1 = n \]

所以，在执行 \( n \) 次 INCREMENT 操作后，计数器所表示的预期值恰好是 \( n \)。

\*\*b. 计数器所表示计数的方差分析取决于 \( n\_i \) 的顺序。让我们考虑一个简单的情况：对于所有 \( i \geq 0 \)，\( n\_i = 100i \)。在执行 \( n \) 次 INCREMENT 操作后，估计寄存器所表示的值的方差。\*\*

\*\*解答：\*\*

对于 \( n\_i = 100i \)，我们有：

\[ P(I\_i = 1) = \frac{1}{n\_{i+1} - n\_i} = \frac{1}{100} \]

\[ P(I\_i = 0) = \frac{99}{100} \]

指示变量 \( I\_i \) 的方差为：

\[ \text{Var}(I\_i) = E[I\_i^2] - (E[I\_i])^2 \]

由于 \( I\_i \) 只能取值 0 或 1，且 \( E[I\_i] = \frac{1}{100} \)，我们有：

\[ E[I\_i^2] = E[I\_i] = \frac{1}{100} \]

\[ \text{Var}(I\_i) = \frac{1}{100} - \left(\frac{1}{100}\right)^2 = \frac{1}{100} - \frac{1}{10000} = \frac{99}{10000} \]

由于 \( n \) 次 INCREMENT 操作是独立的，总方差为：

\[ \text{Var}(X) = \sum\_{i=0}^{n-1} \text{Var}(I\_i) = n \cdot \frac{99}{10000} = \frac{99n}{10000} \]

### 练习 5.2 搜索无序数组

\*\*a. 编写 RANDOM-SEARCH 过程的伪代码以实现上述策略。\*\*

\*\*解答：\*\*

```python

def RANDOM\_SEARCH(A, x):

n = len(A)

indices = set()

while len(indices) < n:

i = RANDOM(0, n-1)

if A[i] == x:

return i

indices.add(i)

return -1 # 表示未找到

```

\*\*b. 假设只有一个索引 \( i \) 使得 \( A[i] = x \)。在找到 \( x \) 和 RANDOM-SEARCH 终止之前，必须选取的 \( A \) 中的索引的预期数量是多少？\*\*

\*\*解答：\*\*

只有一个索引 \( i \) 使得 \( A[i] = x \)，找到 \( x \) 的概率是 \( \frac{1}{n} \)。期望选取的索引数量是：

\[ E[\text{选取次数}] = \sum\_{k=1}^{n} k \cdot \frac{1}{n} = n \]

\*\*c. 将你的解决方案推广到部分 (b)，假设有 \( k \geq 1 \) 个索引 \( i \) 使得 \( A[i] = x \)。在找到 \( x \) 和 RANDOM-SEARCH 终止之前，必须选取的 \( A \) 中的索引的预期数量是多少？\*\*

\*\*解答：\*\*

找到 \( x \) 的概率是 \( \frac{k}{n} \)。期望选取的索引数量是：

\[ E[\text{选取次数}] = \sum\_{k=1}^{n} k \cdot \frac{k}{n} = \frac{n}{k} \]

\*\*d. 假设没有索引 \( i \) 使得 \( A[i] = x \)。在检查完 \( A \) 的所有元素和 RANDOM-SEARCH 终止之前，必须选取的 \( A \) 中的索引的预期数量是多少？\*\*

\*\*解答：\*\*

没有索引 \( i \) 使得 \( A[i] = x \)，必须检查所有元素才能终止。期望选取的索引数量是 \( n \)。

\*\*e. 假设只有一个索引 \( i \) 使得 \( A[i] = x \)。DETERMINISTIC-SEARCH 的平均运行时间是多少？DETERMINISTIC-SEARCH 的最坏运行时间是多少？\*\*

\*\*解答：\*\*

平均运行时间：

\[ E[\text{运行时间}] = \frac{1 + 2 + \cdots + n}{n} = \frac{n+1}{2} \]

最坏运行时间：

最坏情况下，\( x \) 是数组中的最后一个元素：

\[ \text{最坏运行时间} = n \]

\*\*f. 将你的解决方案推广到部分 (e)，假设有 \( k \geq 1 \) 个索引 \( i \) 使得 \( A[i] = x \)。DETERMINISTIC-SEARCH 的平均运行时间是多少？DETERMINISTIC-SEARCH 的最坏运行时间是多少？\*\*

\*\*解答：\*\*

平均运行时间：

\[ E[\text{运行时间}] = \frac{1 + 2 + \cdots + n}{k} = \frac{n+1}{2} \]

最坏运行时间：

最坏情况下，所有 \( k \) 个索引都在数组的末尾：

\[ \text{最坏运行时间} = n \]

\*\*g. 假设没有索引 \( i \) 使得 \( A[i] = x \)。DETERMINISTIC-SEARCH 的平均运行时间是多少？DETERMINISTIC-SEARCH 的最坏运行时间是多少？\*\*

\*\*解答：\*\*

没有索引 \( i \) 使得 \( A[i] = x \)，必须检查所有元素才能终止。

平均运行时间：

\[ E[\text{运行时间}] = n \]

最坏运行时间：

\[ \text{最坏运行时间} = n \]

\*\*h. 让 \( k \) 为索引 \( i \) 的数量，使得 \( A[i] = x \)，给出 SCRAMBLE-SEARCH 在 \( k = 0 \) 和 \( k = 1 \) 情况下的最坏情况和预期运行时间。概括您的解决方案以处理 \( k \geq 1 \) 的情况。\*\*

\*\*解答：\*\*

SCRAMBLE-SEARCH 先随机排列输入数组，然后在结果排列数组上运行确定性线性搜索。

对于 \( k = 0 \)：

最坏运行时间：\( n \)

预期运行时间：\( n \)

对于 \( k = 1 \)：

最坏运行时间：\( n \)

预期运行时间：

\[ E[\text{运行时间}] = \frac{n+1}{2} \]

对于 \( k \geq 1 \)：

最坏运行时间

：\( n \)

预期运行时间：

对于 \( k \geq 1 \)，在随机排列的数组中，找到 \( x \) 的期望位置是 \( \frac{n+1}{2} \)。

所以预期运行时间为：

\[ E[\text{运行时间}] = \frac{n+1}{2} \]

### 练习 5.2-3

\*\*问题\*\*：

你把球扔进 \( b \) 个箱子里，直到某个箱子里有两个球。每次抛球都是独立的，每个球落入任何一个箱子的概率都相等。抛球的预期次数是多少？

\*\*解答\*\*：

设 \( X \) 表示抛球次数。我们希望找到 \( E[X] \)。

每次抛球后，某个箱子里有两个球的概率是：

\[ P(\text{某个箱子里有两个球}) = \frac{1}{b} \]

设 \( T \) 表示抛球次数。我们需要找到 \( E[T] \)。

考虑每次抛球后的情况：

1. 第一次抛球：任意箱子都有 0 个球，抛球成功（没有重复）。

2. 第二次抛球：任意箱子都有 1 个球，抛球成功（没有重复）。

3. 第 \( k \) 次抛球：前 \( k-1 \) 次抛球没有重复，成功。

在每次抛球之前，至少有一个箱子是空的。因此，成功抛球的期望次数是：

\[ E[X] = \sum\_{k=1}^{b} \frac{1}{1 - \frac{k-1}{b}} = b \sum\_{k=1}^{b} \frac{1}{b - (k-1)} = b \sum\_{k=1}^{b} \frac{1}{k} \]

这和调和级数有关：

\[ E[X] \approx b \ln(b) + \gamma b \]

其中 \( \gamma \) 是欧拉-马歇罗尼常数，大约为 0.577。

所以，预期抛球次数为 \( b \ln(b) + \gamma b \)。

### 练习 5.2-4

\*\*问题\*\*：

对于生日悖论的分析，生日相互独立很重要，还是两两独立就足够了？证明你的答案。

\*\*解答\*\*：

对于生日悖论的分析，两两独立就足够了。

证明：

我们关心的是两个人的生日是否相同。因此，我们可以定义指示变量 \( X\_{ij} \)，表示第 \( i \) 个人和第 \( j \) 个人的生日是否相同。

\[ X\_{ij} = \begin{cases}

1, & \text{如果 } i \text{ 和 } j \text{ 的生日相同} \\

0, & \text{否则}

\end{cases} \]

总的相同生日对数 \( X \):

\[ X = \sum\_{1 \leq i < j \leq n} X\_{ij} \]

期望值 \( E[X] \):

\[ E[X] = \sum\_{1 \leq i < j \leq n} E[X\_{ij}] \]

我们只需要每对生日是否独立。

因为每对 \( X\_{ij} \) 是独立的，所以我们有：

\[ E[X\_{ij}] = \frac{1}{365} \]

所以，期望值为：

\[ E[X] = \sum\_{1 \leq i < j \leq n} \frac{1}{365} = \frac{\binom{n}{2}}{365} = \frac{n(n-1)}{2 \cdot 365} \]

所以，对于生日悖论的分析，两两独立就足够了。

### 练习 5.2-5

\*\*问题\*\*：

为了使有三个人的生日相同，应该邀请多少人参加聚会？

\*\*解答\*\*：

设 \( n \) 是需要邀请的人的数量。我们希望找到 \( n \)，使得至少有三个人的生日相同。

定义事件 \( E\_i \) 表示第 \( i \) 个人的生日和前 \( i-1 \) 个人的生日不同。

我们可以用泊松分布近似：

\[ P(\text{有三个人的生日相同}) \approx 1 - P(X = 0) - P(X = 1) - P(X = 2) \]

其中 \( X \) 是生日相同的人数。设 \( \lambda = \frac{n(n-1)(n-2)}{6 \cdot 365^2} \)，则

\[ P(X = 0) = e^{-\lambda} \]

\[ P(X = 1) = \lambda e^{-\lambda} \]

\[ P(X = 2) = \frac{\lambda^2}{2} e^{-\lambda} \]

所以，

\[ P(X \geq 3) = 1 - e^{-\lambda} - \lambda e^{-\lambda} - \frac{\lambda^2}{2} e^{-\lambda} \]

我们需要这个概率大于 1/2：

\[ 1 - e^{-\lambda} - \lambda e^{-\lambda} - \frac{\lambda^2}{2} e^{-\lambda} > 1/2 \]

解这个不等式大约得出 \( \lambda \approx 1.68 \)。因此，

\[ \frac{n(n-1)(n-2)}{6 \cdot 365^2} \approx 1.68 \]

\[ n \approx \sqrt[3]{1.68 \cdot 6 \cdot 365^2} \approx 87 \]

所以，应该邀请大约 87 人。

### 练习 5.2-6

\*\*问题\*\*：

在大小为 \( n \) 的集合中，\( k \) 字符串（定义见第 1179 页）形成 \( k \) 排列的概率是多少？这个问题与生日悖论有什么关系？

\*\*解答\*\*：

假设 \( S \) 是大小为 \( n \) 的集合，我们希望找到 \( k \) 字符串形成 \( k \) 排列的概率。

假设有 \( k \) 个位置，每个位置可以从 \( n \) 个元素中选择。对于每个位置，选择一个元素的概率是均匀的。

\*\*计算\*\*：

总的排列数是 \( n! \)，形成 \( k \) 排列的概率为：

\[ P(\text{形成 } k \text{ 排列}) = \frac{k!}{n!} \]

与生日悖论的关系：

在生日悖论中，我们关心的是在 \( n \) 个元素中是否有两个元素的生日相同。与此类似，我们关心的是在 \( k \) 个位置中是否有两个位置选择了相同的元素。

### 练习 5.2-7

\*\*问题\*\*：

你将 \( n \) 个球扔进 \( n \) 个箱子里，每次抛球都是独立的，球落入任何一个箱子的概率都相同。空箱子的预期数量是多少？只有一个球的箱子的预期数量是多少？

\*\*解答\*\*：

1. \*\*空箱子的预期数量\*\*：

定义事件 \( E\_i \) 表示第 \( i \) 个箱子是空的。

\[ P(E\_i) = \left(1 - \frac{1}{n}\right)^n \approx e^{-1} \]

空箱子的预期数量为：

\[ E[\text{空箱子数}] = n \cdot P(E\_i) \approx n \cdot e^{-1} = \frac{n}{e} \]

2. \*\*只有一个球的箱子的预期数量\*\*：

定义事件 \( F\_i \) 表示第 \( i \) 个箱子只有一个球。

\[ P(F\_i) = \binom{n}{1} \left(\frac{1}{n}\right) \left(1 - \frac{1}{n}\right)^{n-1} \approx n \cdot \frac{1}{n} \cdot e^{-1} = e^{-1} \]

只有一个球的箱子的预期数量为：

\[ E[\text{只有一个球的箱子数}] = n \cdot P(F\_i) \approx n \cdot e^{-1} = \frac{n}{e} \]

### 练习 5.2-8

\*\*问题\*\*：

通过证明在 \( n \) 枚公平硬币中，出现长度为 \( \lg n - 2 \lg \lg n \) 的连续正面的连续概率至少为 \( 1 - 1/n \)，可以进一步确定连续长度的下限。

\*\*解答\*\*：

设 \( X \) 表示连续正面的最大长度。我们希望找到 \( X \) 的下限，使得出现长度为 \( \lg n - 2 \lg \lg n \) 的连续正面的概率至少为 \( 1 - 1/n \)。

首先，我们考虑在 \( n \) 次

练习

6.2-1

使用图 6.2 作为模型，说明 MAX-HEAPIFY .A; 3/ 对数组 A D h27; 17; 3; 16; 13; 10; 1; 5; 7; 12; 4; 8; 9; 0i 的操作。

6.2-2

说明 n 节点堆根的每个子节点都是包含最多 2n=3 个节点的子树的根。最小常数 ˛ 是多少，使得每个子树最多有 ˛n 个节点？这对递归 (6.1) 及其解决方案有何影响？

6.2-3

从过程 MAX-HEAPIFY 开始，为过程 MIN-HEAPIFY .A; i / 编写伪代码，该过程对最小堆执行相应的操作。 MIN-HEAPIFY 的运行时间与 MAXHEAPIFY 的运行时间相比如何？

6.2-4

当元素 AŒi� 大于其子元素时，调用 MAX-HEAPIFY .A; i / 会产生什么效果？

6.2-5

当 i > A:heap-size=2 时，调用 MAX-HEAPIFY .A; i / 会产生什么效果？

6.2-6

MAX-HEAPIFY 的代码在常数因子方面非常高效，除了第 10 行中的递归调用，某些编译器可能会为此生成低效的代码。编写一个高效的 MAX-HEAPIFY，使用迭代控制构造（循环）而不是递归。

6.3 构建堆 167

6.2-7

证明 MAX-HEAPIFY 在大小为 n 的堆上的最坏情况运行时间为 �.lg n/。（提示：对于具有 n 个节点的堆，给出节点值，使 MAXHEAPIFY 在从根到叶的简单路径上的每个节点上递归调用。）

练习

6.3-1

以图 6.3 为例，说明 BUILD-MAX-HEAP 对数组 A D h5; 3; 17; 10; 84; 19; 6; 22; 9i 的操作。

6.3-2

证明 Û

n=2hC1



¹ 对于 0 හ h හ blg nc，1=2。

6.3-3

为什么 BUILD-MAX-HEAP 第 2 行的循环索引 i 从 bn=2c 减少到 1，而不是从 1 增加到 bn=2c？

6.3-4

证明在任何 n 元素堆中，最多有 Û

n=2hC1



个高度为 h 的节点

### 练习 6.2-1

\*\*使用图 6.2 作为模型，说明 MAX-HEAPIFY(A, 3) 对数组 A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0] 的操作。\*\*

\*\*解答：\*\*

首先，数组 A 对应的二叉堆结构如下：

```

27

/ \

17 3

/ \ / \

16 13 10 1

/ \ / \ / \ / \

5 7 12 4 8 9 0

```

调用 `MAX-HEAPIFY(A, 3)`，即对根节点为 3 的子树进行堆化：

1. 根节点是 3，它的左子节点是 10，右子节点是 1。

2. 10 是最大的子节点，将 10 和 3 交换。

交换后的堆：

```

27

/ \

17 10

/ \ / \

16 13 3 1

/ \ / \ / \ / \

5 7 12 4 8 9 0

```

继续对新位置的 3 进行堆化：

1. 根节点是 3，它的左子节点是 8，右子节点是 9。

2. 9 是最大的子节点，将 9 和 3 交换。

交换后的堆：

```

27

/ \

17 10

/ \ / \

16 13 9 1

/ \ / \ / \ / \

5 7 12 4 8 3 0

```

最终结果：

`MAX-HEAPIFY(A, 3)` 完成后的数组 A 为：[27, 17, 10, 16, 13, 9, 1, 5, 7, 12, 4, 8, 3, 0]

### 练习 6.2-2

\*\*说明 n 节点堆根的每个子节点都是包含最多 2n/3 个节点的子树的根。最小常数 α 是多少，使得每个子树最多有 αn 个节点？这对递归 (6.1) 及其解决方案有何影响？\*\*

\*\*解答：\*\*

对于一个包含 \( n \) 个节点的堆：

- 根节点为 1 个。

- 左子树和右子树的节点数之和为 \( n - 1 \) 个。

假设左子树和右子树的节点数分别为 \( l \) 和 \( r \)，且 \( l \geq r \)：

\[ l + r = n - 1 \]

\[ l \approx \frac{2n}{3} \]

\[ r \approx \frac{n}{3} \]

所以：

\[ l \leq \frac{2n}{3} \]

最小常数 α 是 2/3。

对递归 (6.1) 的影响：

\[ T(n) = T(\frac{2n}{3}) + O(\log n) \]

该递归关系表示，在最坏情况下，每次递归调用会处理最多 \( \frac{2n}{3} \) 个节点，最终的时间复杂度为 \( O(\log n) \)。

### 练习 6.2-3

\*\*从过程 MAX-HEAPIFY 开始，为过程 MIN-HEAPIFY(A, i) 编写伪代码，该过程对最小堆执行相应的操作。 MIN-HEAPIFY 的运行时间与 MAX-HEAPIFY 的运行时间相比如何？\*\*

\*\*解答：\*\*

MIN-HEAPIFY 的伪代码如下：

```python

def MIN\_HEAPIFY(A, i):

n = len(A)

smallest = i

left = 2 \* i + 1

right = 2 \* i + 2

if left < n and A[left] < A[smallest]:

smallest = left

if right < n and A[right] < A[smallest]:

smallest = right

if smallest != i:

A[i], A[smallest] = A[smallest], A[i]

MIN\_HEAPIFY(A, smallest)

```

MIN-HEAPIFY 和 MAX-HEAPIFY 的运行时间是相同的，因为它们都执行 \( O(\log n) \) 次交换和比较操作。

### 练习 6.2-4

\*\*当元素 A[i] 大于其子元素时，调用 MAX-HEAPIFY(A, i) 会产生什么效果？\*\*

\*\*解答：\*\*

当元素 \( A[i] \) 大于其子元素时，调用 `MAX-HEAPIFY(A, i)` 将不会改变堆，因为堆已经满足最大堆的性质，不需要任何交换操作。

### 练习 6.2-5

\*\*当 \( i > A.\text{heap-size}/2 \) 时，调用 MAX-HEAPIFY(A, i) 会产生什么效果？\*\*

\*\*解答：\*\*

当 \( i > A.\text{heap-size}/2 \) 时，节点 \( i \) 是叶节点。对于叶节点，调用 `MAX-HEAPIFY(A, i)` 将不会进行任何操作，因为叶节点没有子节点，不需要进行堆化操作。

### 练习 6.2-6

\*\*MAX-HEAPIFY 的代码在常数因子方面非常高效，除了第 10 行中的递归调用，某些编译器可能会为此生成低效的代码。编写一个高效的 MAX-HEAPIFY，使用迭代控制构造（循环）而不是递归。\*\*

\*\*解答：\*\*

下面是迭代版本的 `MAX-HEAPIFY`：

```python

def MAX\_HEAPIFY(A, i):

n = len(A)

largest = i

while True:

left = 2 \* i + 1

right = 2 \* i + 2

if left < n and A[left] > A[largest]:

largest = left

if right < n and A[right] > A[largest]:

largest = right

if largest != i:

A[i], A[largest] = A[largest], A[i]

i = largest

else:

break

```

### 练习 6.2-7

\*\*证明 MAX-HEAPIFY 在大小为 \( n \) 的堆上的最坏情况运行时间为 \( O(\log n) \)。（提示：对于具有 \( n \) 个节点的堆，给出节点值，使 MAX-HEAPIFY 在从根到叶的简单路径上的每个节点上递归调用。）\*\*

\*\*解答：\*\*

为了证明 `MAX-HEAPIFY` 的最坏情况运行时间为 \( O(\log n) \)，我们需要构造一个堆，使得每次递归调用都发生在根到叶的路径上。

考虑一个完全二叉树，其中根节点为最大值，其子节点逐渐减小。例如，对于堆 `[16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]`，其结构如下：

```

16

/ \

15 14

/ \ / \

13 12 11 10

/ \ / \ / \ / \

9 8 7 6 5 4 3 2

```

如果我们调用 `MAX-HEAPIFY` 从根节点开始（即 \( i = 0 \)），则每次递归调用都会处理根到叶的路径上的每个节点。

在完全二叉树中，根到叶的路径长度为 \( \log n \)。因此，在最坏情况下，`MAX-HEAPIFY` 需要进行 \( \log n \) 次递归调用或交换操作。

因此，`MAX-HEAPIFY` 的最坏情况运行时间为 \( O(\log n) \)。

为了更好地理解和解答这些关于堆排序的问题，让我们一步步来看：

### 练习 6.3-1：BUILD-MAX-HEAP 过程说明

首先，我们需要明确 `BUILD-MAX-HEAP` 的功能是将一个数组重新排列成一个最大堆。最大堆是一种满足每个节点的值都大于或等于其子节点值的二叉堆。

对于数组 A = [5, 3, 17, 10, 84, 19, 6, 22, 9]，`BUILD-MAX-HEAP` 的操作如下：

1. \*\*初始化\*\*：堆的构建从最后一个非叶子节点开始，这个节点在数组中的索引是 `n/2`，对于长度为 9 的数组，这个索引是 `9/2 = 4.5`，向下取整得 4。

2. \*\*逐步堆化\*\*：从索引 4 开始，向下直到索引 1，对每个节点执行 `MAX-HEAPIFY`，以确保节点的值大于其子节点的值。

- 当 `i = 4`，节点值为 10，它的子节点是 22 和 9。`MAX-HEAPIFY` 将 22 与 10 交换，因为 22 大于 10。

- 当 `i = 3`，节点值为 17，它的子节点是 6 和 19。19 与 17 交换。

- 当 `i = 2`，现在是 84，子节点是 [10, 19]（由前面步骤的交换所得）。84 已经是最大，不需要交换。

- 当 `i = 1`，节点是 3，它的子节点是 [84, 17]。84 与 3 交换。

最终的最大堆数组是 [84, 22, 19, 10, 3, 17, 6, 5, 9]。

### 练习 6.3-2：证明

关于 `n = 2^(h+1) - 1` 对于 `0 <= h <= floor(log(n))` 的正确性。这个表达式基于完全二叉树的性质，其中高度 `h` 的完全二叉树有 `2^(h+1) - 1` 个节点。这个公式计算从树的顶部到给定高度 `h` 的节点总数。此处要证明的是，对于堆中任何高度 `h`，最多有 `floor(n / (2^(h+1)))` 个节点处于高度 `h`。

### 练习 6.3-3：为什么从 `floor(n/2)` 开始

`BUILD-MAX-HEAP` 中的循环从 `floor(n/2)` 开始向下到 1，而不是从 1 到 `floor(n/2)`，是因为堆的构建需要从底层向上层逐步构建堆的性质。底部的子树先满足最大堆的条件，然后才是它们的父节点。这样从底向上可以保证当处理任意节点作为根的子树时，其子树已经是最大堆。

### 练习 6.3-4：证明高度为 h 的节点数量

在任何 n 元素堆中，高度为 h 的节点最多有 `floor(n / (2^(h+1)))` 个。这是因为在高度 h 的每个节点都有 `2^(h+1)` 个子节点（在完全二叉树中），因此所有高度 h 的节点加起来就是总节点数 n 除以 `2^(h+1)`。

这些练习是堆排序和堆数据结构的核心理解部分，涉及堆的构建和维护过程，以及堆的结构性质。如果需要进一

# utting a Rod | DP-13

**Last Updated :**30 May, 2024

Given a rod of length n inches and an array of prices that includes prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example, if the length of the rod is 8 and the values of different pieces are given as the following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)

length | 1 2 3 4 5 6 7 8   
--------------------------------------------  
price | 1 5 8 9 10 17 17 20

And if the prices are as follows, then the maximum obtainable value is 24 (by cutting in eight pieces of length 1)

length | 1 2 3 4 5 6 7 8   
--------------------------------------------  
price | 3 5 8 9 10 17 17 20

Recommended Problem

Rod Cutting

[Arrays](https://practice.geeksforgeeks.org/explore?page=1&category[]=Arrays&sortBy=submissions)

[Dynamic Programming](https://practice.geeksforgeeks.org/explore?page=1&category[]=Dynamic Programming&sortBy=submissions)

+2 more

[Google](https://practice.geeksforgeeks.org/explore?page=1&company[]=Google&sortBy=submissions)

[Solve Problem](https://www.geeksforgeeks.org/problems/rod-cutting0840/1?itm_source=geeksforgeeks&itm_medium=article&itm_campaign=bottom_sticky_on_article" \o "Permalink to Rod Cutting)

Submission count: 1.1L

**Method 1:**A naive solution to this problem is to generate all configurations of different pieces and find the highest-priced configuration. This solution is exponential in terms of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem and can efficiently be solved using Dynamic Programming.

**1) Optimal Substructure:**

We can get the best price by making a cut at different positions and comparing the values obtained after a cut. We can recursively call the same function for a piece obtained after a cut.  
Let cutRod(n) be the required (best possible price) value for a rod of length n. cutRod(n) can be written as follows.  
cutRod(n) = max(price[i] + cutRod(n-i-1)) for all i in {0, 1 .. n-1}

# A recursive solution for Rod cutting problem

# Returns the best obtainable price for a rod of length n

# and price[] as prices of different pieces

def cutRod(price, index, n):

# base case

if index == 0:

return n\*price[0]

#v if n is 0 we cannot cut the rod anymore.

if (n ==0):

return 0

# At any index we have 2 options either

# cut the rod of this length or not cut

# it

notCut = cutRod(price,index - 1,n)

cut = float("-inf")

rod\_length = index + 1

if (rod\_length <= n):

cut = price[index]+cutRod(price,index,n - rod\_length)

return max(notCut, cut)

# Driver program to test above functions

arr = [ 1, 5, 8, 9, 10, 17, 17, 20 ]

size = len(arr)

print("Maximum Obtainable Value is ",cutRod(arr, size - 1, size))

# This code is contributed by Vivek Maddeshiya

# Matrix Chain Multiplication | DP-8

**Last Updated :**20 Dec, 2022

Given the dimension of a sequence of matrices in an array **arr[]**, where the dimension of the **ith** matrix is **(arr[i-1] \* arr[i])**, the task is to find the most efficient way to multiply these matrices together such that the total number of element multiplications is minimum.

**Examples:**

***Input:****arr[] = {40, 20, 30, 10, 30}****Output:****26000****Explanation:****There are 4 matrices of dimensions 40×20, 20×30, 30×10, 10×30.  
Let the input 4 matrices be A, B, C and D.  
The minimum number of  multiplications are obtained by   
putting parenthesis in following way (A(BC))D.  
The minimum is 20\*30\*10 + 40\*20\*10 + 40\*10\*30*

***Input:****arr[] = {1, 2, 3, 4, 3}****Output:****30****Explanation:****There are 4 matrices of dimensions 1×2, 2×3, 3×4, 4×3.   
Let the input 4 matrices be A, B, C and D.    
The minimum number of multiplications are obtained by   
putting parenthesis in following way ((AB)C)D.  
The minimum number is 1\*2\*3 + 1\*3\*4 + 1\*4\*3 = 30*

**[311. Sparse Matrix Multiplication](https://leetcode.com/problems/sparse-matrix-multiplication/)**

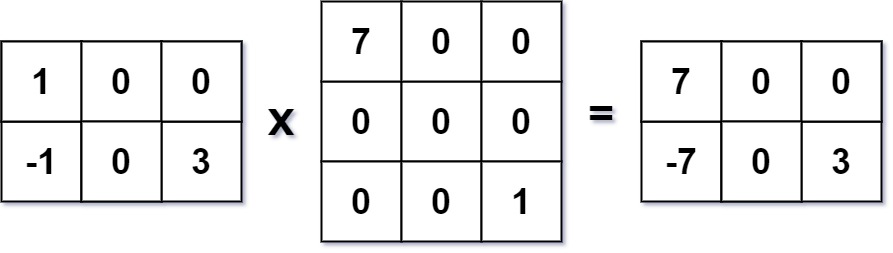
Medium

Topics

Companies

Given two [sparse matrices](https://en.wikipedia.org/wiki/Sparse_matrix" \t "https://leetcode.com/problems/sparse-matrix-multiplication/description/_blank) mat1 of size m x k and mat2 of size k x n, return the result of mat1 x mat2. You may assume that multiplication is always possible.

****Example 1:****



**Input:** mat1 = [[1,0,0],[-1,0,3]], mat2 = [[7,0,0],[0,0,0],[0,0,1]]**Output:** [[7,0,0],[-7,0,3]]

****Example 2:****

**Input:** mat1 = [[0]], mat2 = [[0]]**Output:** [[0]]

In [computer science](https://en.wikipedia.org/wiki/Computer_science" \o "Computer science), **amortized analysis** is a method for [analyzing](https://en.wikipedia.org/wiki/Analysis_of_algorithms" \o "Analysis of algorithms) a given algorithm's [complexity](https://en.wikipedia.org/wiki/Computational_complexity" \o "Computational complexity), or how much of a resource, especially time or memory, it takes to [execute](https://en.wikipedia.org/wiki/Execution_(computing)" \o "Execution (computing)). The motivation for amortized analysis is that looking at the worst-case run time can be too pessimistic. Instead, amortized analysis averages the running times of operations in a sequence over that sequence.[[1]](https://en.wikipedia.org/wiki/Amortized_analysis" \l "cite_note-tarjan-1): 306  As a conclusion: "Amortized analysis is a useful tool that complements other techniques such as [worst-case](https://en.wikipedia.org/wiki/Worst-case_execution_time" \o "Worst-case execution time) and [average-case](https://en.wikipedia.org/wiki/Average-case_complexity" \o "Average-case complexity) analysis."[[2]](https://en.wikipedia.org/wiki/Amortized_analysis" \l "cite_note-fiebrink-2): 14

For a given operation of an algorithm, certain situations (e.g., input parametrizations or data structure contents) may imply a significant cost in resources, whereas other situations may not be as costly. The amortized analysis considers both the costly and less costly operations together over the whole sequence of operations. This may include accounting for different types of input, length of the input, and other factors that affect its performance.[[](https://en.wikipedia.org/wiki/Amortized_analysis" \l "cite_note-fiebrink-2)

What is a Minimum Spanning Tree? A minimum spanning tree (MST) is a subset of the edges of a connected, edge-weighted graph that connects all the vertices together without any cycles and with the minimum possible total edge weight. It is a way of finding the most economical way to connect a set of vertices.A **minimum spanning tree** (**MST**) or **minimum weight spanning tree** is a subset of the edges of a [connected](https://en.wikipedia.org/wiki/Connected_graph" \o "Connected graph), edge-weighted undirected graph that connects all the [vertices](https://en.wikipedia.org/wiki/Vertex_(graph_theory)" \o "Vertex (graph theory)) together, without any [cycles](https://en.wikipedia.org/wiki/Cycle_(graph_theory)" \o "Cycle (graph theory)) and with the minimum possible total edge weight.[[1]](https://en.wikipedia.org/wiki/Minimum_spanning_tree" \l "cite_note-Numpy_and_Scipy_Documentation_%E2%80%94_Numpy_and_Scipy_documentation-1) That is, it is a [spanning tree](https://en.wikipedia.org/wiki/Spanning_tree" \o "Spanning tree) whose sum of edge weights is as small as possible.[[2]](https://en.wikipedia.org/wiki/Minimum_spanning_tree" \l "cite_note-NetworkX_2.6.2_documentation-2) More generally, any edge-weighted undirected graph (not necessarily connected) has a **minimum spanning forest**, which is a union of the minimum spanning trees for its [connected components](https://en.wikipedia.org/wiki/Connected_component_(graph_theory)" \o "Connected component (graph theory)).

There are many use cases for minimum spanning trees. One example is a telecommunications company trying to lay cable in a new neighborhood. If it is constrained to bury the cable only along certain paths (e.g. roads), then there would be a graph containing the points (e.g. houses) connected by those paths. Some of the paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. Currency is an acceptable unit for edge weight – there is no requirement for edge lengths to obey normal rules of geometry such as the [triangle inequality](https://en.wikipedia.org/wiki/Triangle_inequality" \o "Triangle inequality). A *spanning tree* for that graph would be a subset of those paths that has no cycles but still connects every house; there might be several spanning trees possible. A *minimum spanning tree* would be one with the lowest total cost, representing the least expensive path for laying the cable.