1.在计算机科学与社会生活中，经常涉及到要求前K个元素即top-k的问题，请给出不同策解决这一问题，并对比分析。

问题分析：

TOP-k的意思是要得到最有价值，或者最大的k个值，常用于淘宝推荐，网络安全中的恶意行为检测等等。我们的目的其实就是返回值最大的前k个数。

算法1

本质上就是对一个序列进行排序。最经典的方法就是冒泡排序，其时间复杂度为O(n^2)。在前面的作业中提到过几种冒泡排序的优化，使其在最优情况下可以达到O(n)。我们可以利用其达到我们的目的，在此不再赘述。

算法2

我们可以利用快速排序算法对序列进行排序。其算法思路为：

1. 首先选择一个标记元素，通常选择中间的元素，将数组分成左右两部分。
2. 使用两个指针 i 和 j 分别指向左半部分的起始位置和右半部分的结束位置。
3. 在每次循环中，将 i 向右移动，直到找到第一个小于等于标记元素的元素；将 j 向左移动，直到找到第一个大于等于标记元素的元素。
4. 如果i 大于等于 j，则交换 i 和j的位置，将 i 向右移动一位，将 j向左移动一位。
5. 重复步骤 3 和 4，直到i 大于j。
6. 接着将左半部分和右半部分分别递归调用快速排序函数，直到每个部分的元素只剩下一个或零个。
7. 最后将排好序的两个部分合并起来，得到最终的排序结果。

简要概括为每一次循环都让标记元素左边的元素大，右边的元素小。并不断迭代将规模变小以达到整个序列为从大到小排列。

代码实现

```cpp
#include <iostream>
#include <vector>
#include <chrono>  // 用于计算程序运行时间
using namespace std;

// 快速排序函数
void quickSort(vector<int>& nums, int left, int right) {
    // 左右指针重合或交叉，排序完成
    if (left >= right) {
        return;
    }
    // 选择标记元素，通常为中间的元素
    int pivot = nums[(left + right) / 2];
    int i = left, j = right;
    // 进行划分，将左边的元素都小于标记，右边的元素都大于标记
    while (i <= j) {
        // 找到第一个大于等于标记元素的元素
        while (nums[i] > pivot) {
            i++;
        }
        // 找到第一个小于等于标记元素的元素
        while (nums[j] < pivot) {
            j--;
        }
        // 如果 i <= j，则交换 i 和 j 的位置，继续移动 i 和 j
        if (i <= j) {
            swap(nums[i], nums[j]);
            i++;
            j--;
        }
    }
    // 对左半部分和右半部分分别递归调用快速排序函数
    quickSort(nums, left, j);
    quickSort(nums, i, right);
}

int main() {
    vector<int> nums = { 3,4,5,6,7,1,2 };
    quickSort(nums, 0, nums.size() - 1);
    // 输出前三个元素
    for (int i = 0; i < 3; i++) {
        cout << nums[i] << endl;
    }
    return 0;
}
```

分析

快速排序算法的时间复杂度为O(nlogn)，但在最坏情况下可能达到O(n^2)。这是因为数组本身就是有序或者完全相反的情况下，每次划分只能排除一个元素，这种情况下递归深度达到了 n，每次需要遍历 n 个元素，因此时间复杂度为 O(n^2)。

算法3

堆算法可以用来解决 TOP-K 问题，具体算法如下：

1. 构建大小为 k 的小根堆 heap，堆中元素为前 k 个数。
2. 遍历数组中剩余的元素，如果元素小于堆顶元素，则替换堆顶元素为该元素，并进行堆调整。
3. 遍历完数组后，堆中的元素即为前 k 大的数。

代码实现

```cpp
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

// 定义函数topK，输入为一个整数向量nums和一个整数k，返回一个整数向量
vector<int> topK(vector<int>& nums, int k) {
// 定义一个小根堆minHeap，存储最大的k个数
priority_queue<int, vector<int>, greater<int>> minHeap;
// 遍历nums的前k个数，加入minHeap中
for (int i = 0; i < k; ++i) {
minHeap.push(nums[i]);
}
// 遍历nums剩余的数，如果比minHeap中的最小值大，将最小值出堆，加入当前值
for (int i = k; i < nums.size(); ++i) {
if (nums[i] > minHeap.top()) {
minHeap.pop();
minHeap.push(nums[i]);
}
}
// 定义结果向量result，将minHeap中的k个最大值依次出堆加入result
vector<int> result;
while (!minHeap.empty()) {
result.push_back(minHeap.top());
minHeap.pop();
}
// 返回结果向量result
return result;
}

int main() {
// 定义一个测试向量nums和整数k
vector<int> nums = { 3, 5, 2, 6, 1, 8, 7, 9, 4 };
int k = 3;
// 调用topK函数，将结果存储在向量result中
vector<int> result = topK(nums, k);
// 输出结果向量result中的元素
for (int i = 0; i < result.size(); ++i) {
cout << result[i] << " ";
}
cout << endl;
return 0;
}
```

分析：

该算法的时间复杂度为 O(nlogk)，其中 n 是数组的长度，因为堆中最多只有 k 个元素，每次堆调整的时间复杂度为 O(logk)，因此总的时间复杂度为 O(nlogk)。

2.请用递归方式实现堆排序，并进行性能分析

算法思路

使用递归方式实现的堆排序算法，具体流程如下：

1. 首先构建最大堆，从最后一个非叶子节点开始进行堆化操作，使其满足堆的性质（父节点大于等于子节点）。
2. 接着从堆顶取出最大值，将其移动到数组末尾，并对剩余元素进行堆化操作，使其仍然满足堆的性质。
3. 重复上述操作直到整个数组有序。

代码实现

```cpp
#include <iostream>
using namespace std;

void heapify(int arr[], int n, int i) {
    // 将根节点初始化为最大值
    int largest = i;

    // 获取左右子节点的索引
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    // 如果左子节点大于根节点，将其设为最大值
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // 如果右子节点比当前最大值还要大，将其设为最大值
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // 如果最大值不是根节点，交换根节点和最大值
    if (largest != i) {
        swap(arr[i], arr[largest]);
        // 递归地对子树进行堆化操作
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    // 构建最大堆
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // 依次将元素取出并对剩余元素进行堆化操作
    for (int i = n - 1; i > 0; i--) {
        // 将当前的根节点（即最大值）移动到数组末尾
        swap(arr[0], arr[i]);
        // 对剩余元素进行堆化操作
        heapify(arr, i, 0);
    }
}

int main() {
    int arr[] = { 3, 6, 1, 8, 2, 4, 9, 5, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "原始序列: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
```

```
    heapSort(arr, n);

    cout << "新序列 ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;

    return 0;
}
```

性能分析

堆排序的时间复杂度为 O(n log n)，其中 n 为要排序的元素个数。具体来说，构建最大堆的时间复杂度为 O(n)，每次取出最大值并进行堆化操作的时间复杂度为 O(log n)，需要执行 n-1 次，因此总的时间复杂度为 O(n log n)。空间复杂度为 O(1)，因为只需要使用常数个临时变量。

需要注意的是，递归方式实现的堆排序虽然代码简洁易懂，但是由于递归调用的开销比较大，实际执行效率可能不如迭代方式实现的堆排序。因此，在实际应用中需要根据具体情况选择合适的实现方式。