Answer

Introduction

为了方便贵司面试官对我的代码进行评估，我将代码提交到了github上，地址为：find_stones

测试结果通过github action生成，可在github page查看

Local Test

```
git clone https://github.com/windzu/find_stones_solution.git && \
cd find_stones_solution && \
mkdir build && \
cd build && \
cmake .. && \
make && \
./test/unit_test
```

Question A

> Formally describe the question

给定一个数组 `nums` 和一个数 `k`，从 `nums` 中找到满足 `nums[j]-nums[i] == k` 条件的第一对 `(nums[j]-nums[i])`，然后返回一对值所对应的下标 `(i, j)` (可以是 None),具体条件如下 :

- `i != j`
- `0 <= i, j < len(nums)`
- `nums[j] - nums[i]) == k`

注意：

- nums 数组中的元素为数值类型，可能为整数或浮点数，可能重复，数组长度可能为 0，数值范围可能为double上下限
- k 为数值类型，可能为整数或浮点数，数值范围可能为double上下限

example1:

```
Input: nums = [3,1,4,1,5], k = 2
Output: [1,0] # nums[0]-nums[1]==2
```

example2:

```
Input: nums = [1,2,3,4], k = 1
Output: [0,1] # nums[1]-nums[0]==1
```

Question B

1. Code

> 以下代码为片选，完整代码请参考 solution

hash表法

```
// 构建hash表 visited，key为nums[i]，value为i
// 步骤如下：
// 1. 遍历nums，对于每个nums[i]，计算nums[i]+k 或者 nums[i]-k，如果存在于visited中，则返回它们的下标对
// 2. 如果不存在，则将nums[i]加入visited中
// 3. 如果遍历结束，仍未找到，则返回None

std::unordered_map<T, int> visited;
for (int i = 0; i < nums.size(); i++) {
    if (visited.count(nums[i] - k)) {
        return std::vector<int>{visited[nums[i] - k], i};
    }
    if (visited.count(nums[i] + k)) {
        return std::vector<int>{i, visited[nums[i] + k]};
    }

    visited.emplace(nums[i], i);
}
```

双指针法

> 采用双指针法原因是无法规避浮点数的精度问题，而被迫采用专门用于解决浮点数精度问题

```cpp
// 首先拷贝一份数组以及每个数值对应在原数组中的index，然后对数组进行排序
std::vector<std::pair<T, int>> pairs;
for (int i = 0; i < nums.size(); i++) {
    pairs.emplace_back(nums[i], i);
}
std::sort(pairs.begin(), pairs.end());

// 双指针法寻找满足条件的数值对,读取对应的下标并返回
// 因为精度问题，所以使用fabs并结合预设的允许误差eps，判断是否满足条件
int i = 0, j = 1;
while (i < pairs.size() && j < pairs.size()) {
    if (i != j && fabs(pairs[j].first - pairs[i].first - k) < eps) {
        return std::vector<int>{pairs[i].second, pairs[j].second};
    } else if (i != j && pairs[j].first - pairs[i].first < k) {
    j++;
    } else {
        i++;
    }
}
```

2. Space Complexity & Time Complexity

整数类型

分析：

- 此种方法因为仅遍历一次数组，且访问hash表的时间复杂度为O(1)，因此时间复杂度为O(n)
- 此种方法需要额外的hash表，因此空间复杂度为O(n)

结论：

- 时间复杂度：O(n)
- 空间复杂度：O(n)

浮点类型

分析：

- 双链表法需要先对数组进行排序，时间复杂度为O(nlogn)，然后遍历一次数组，时间复杂度为O(n)，因此总时间复杂度为O(nlogn)
- 此种方法需要额外的数组，因此空间复杂度为O(n)

结论：

- 时间复杂度：O(nlogn)

- 空间复杂度：O(n)

3. Analysis & Optimization

本题的核心是**查找**，但是查找往往需要排序，通常复杂度为O(nlogn) 为了实现O(n)的时间复杂度，本题解法使用hash表来存储数组中的元素

- key为数组元素
- value为数组元素对应的下标

这样在遍历数组的时候，只需要O(1)的时间复杂度就可以找到满足条件的元素，从而实现O(n)的时间复杂度

但是，如果数组中的元素是浮点数，那么hash表就无法使用了 这是因为浮点数的精度问题

- 浮点数本身就是不精确的，其随着位数的增加，精度会越来越低
- 两个浮点数经过运算后的结果会有一个与目标值极小的且随机的浮点误差
- 采用浮点数做key，无法保证经过运算后的值可以在hash表中被找到，从而失去作为查找表的作用

在经过多种尝试无果后，最终决定针对浮点数情况使用`排序+双指针`的方式来实现，时间复杂度为O(nlogn)

4. Test Case

> 详情请参考 test

通过googletest进行测试，测试用例的编写遵循以下原则：

- 覆盖数值边界条件
- 覆盖各种数值类型
- 覆盖极端情况

以下是部分测试用例：

```cpp
TEST(test_q1, int_case0) {
  // int arrays are ordered
  Solution<int> s;
  std::vector<int> nums = {1, 2, 3, 4, 5};
  int k = 2;
  std::vector<int> expected = {0, 2};
  std::vector<int> res = s.find_k_diff_pair(nums, k);
  EXPECT_EQ(res, expected);
}

TEST(test_q1, int_case1) {
  // negative elements
  Solution<int> s;
  std::vector<int> nums = {-1, 2, 1, 4, -5};
  int k = 2;
  std::vector<int> expected = {0, 2};
  std::vector<int> res = s.find_k_diff_pair(nums, k);
  EXPECT_EQ(res, expected);
}

TEST(test_q1, double_case0) {
  // double arrays are not ordered
  Solution<double> s;
  std::vector<double> nums = {2.2, -1.1, 1, -0.5, -1.1};
  double k = 3.3;
  std::vector<int> expected = {1, 0};
  std::vector<int> res = s.find_k_diff_pair(nums, k);
  EXPECT_EQ(res, expected);
}

TEST(test_q3, int_case2) {
  // k=0
  Solution<int> s;
  std::vector<int> nums = {3, 3, 3, 1, 1};
  int k = 0;
  std::vector<std::vector<int>> expected = {{0, 1}, {0, 2}, {1, 2}, {3, 4}};
  std::vector<std::vector<int>> res = s.find_k_diff_pairs(nums, k);
  // sort the result
  std::sort(res.begin(), res.end());
  EXPECT_EQ(res, expected);
}
```

以下是测试用例的部分结果：

```
...
[----------] 5 tests from test_q1
[ RUN      ] test_q1.int_case0
[       OK ] test_q1.int_case0 (0 ms)
[ RUN      ] test_q1.int_case1
[       OK ] test_q1.int_case1 (0 ms)
[ RUN      ] test_q1.int_case2
[       OK ] test_q1.int_case2 (0 ms)
[ RUN      ] test_q1.float_case0
[       OK ] test_q1.float_case0 (0 ms)
[ RUN      ] test_q1.double_case0
[       OK ] test_q1.double_case0 (0 ms)
[----------] 5 tests from test_q1 (0 ms total)

[----------] 3 tests from test_q3
[ RUN      ] test_q3.int_case0
[       OK ] test_q3.int_case0 (0 ms)
[ RUN      ] test_q3.int_case1
[       OK ] test_q3.int_case1 (0 ms)
[ RUN      ] test_q3.int_case2
[       OK ] test_q3.int_case2 (0 ms)
[----------] 3 tests from test_q3 (0 ms total)
...
```

Question C

> 以下代码为片选，完整代码请参考 solution

hash表法

```cpp
// 通过hash表构建一个map，变量名 visited ，key为数组元素，value为数组元素对应的下标
// 结构如下：
// {
//   nums[i]: [i1, i2, ...],
//   nums[j]: [j1, j2, ...],
//   ...
// }
std::unordered_map<T, std::vector<int>> visited;
for (int i = 0; i < nums.size(); i++) {
    if (visited.count(nums[i])) {
    visited[nums[i]].emplace_back(i);
    } else {
    visited.emplace(nums[i], std::vector<int>{i});
    }
}

// 当k为0时是一个特殊情况，其结果为visited中相同元素的下标排列组合
// 针对这种情况，进行单独处理
// 例如: Input: nums = [3, 3, 3, 1, 1], k = 0
// visited 内容为：
// {
//   3: [0, 1, 2],
//   1: [3, 4],
// }
// Output: [[0,1],[0,2],[1,2],[3,4]]
if (k == 0) {
    std::vector<std::vector<int>> res_vec;
    for (auto& [key, value] : visited) {
    if (value.size() >= 2) {
        for (int i = 0; i < value.size(); i++) {
            for (int j = i + 1; j < value.size(); j++) {
                res_vec.emplace_back(std::vector<int>{value[i], value[j]});
            }
        }
    }
}

// when k != 0, which is a general case,
// 当k不为0时，是一般情况
// 针对通用情况的处理步骤如下：
// 1. 遍历visited中的每一个元素，找到所有满足条件的元素的index将其按照如下格式存储在 res 中
//  res 结构如下：
//  {
//      i1: {j1, j2, ...},
//      i2: {j1, j2, ...},
//      ...
//  }
```

```
//  上述 i1 所对应的value 是所有满足条件的元素的下标的集合，(i1,j1), (i1,j2), ...都是满足条件的元素
//  这样的写法可以保证不会出现重复的元素
// 2. 遍历res，将所有的元素按照题目要求的格式存储在res_vec中
//  res_vec 结构如下：
//  {
//      {i1, j1},
//      {i1, j2},
//      ...
//  }
//
// Tricks:
//  如果在遍历visited的过程中，发现某个元素分别可以找到-k 和 +k 的元素，那么这个元素在此次遍历后就不需要再遍历了，可以将其从visited中删除
std::unordered_map<int, std::unordered_set<int>> res;
for (auto& [key, value] : visited) {
    int used_count = 0;
    if (visited.count(key + k)) {
        used_count++;
        for (auto& i : value) {
            for (auto& j : visited[key + k]) {
                if (res.count(i)) {
                    res[i].emplace(j);
                }else {
                    res.emplace(i, std::unordered_set<int>{j});
                }
            }
        }
    }
    if (visited.count(key - k)) {
        used_count++;
        for (auto& i : visited[key - k]) {
            for (auto& j : value) {
                if (res.count(i)) {
                    res[i].emplace(j);
                } else {
                    res.emplace(i, std::unordered_set<int>{j});
                }
            }
        }
    }
    if (used_count == 2) {
        visited.erase(key);
    }
}
```

整数类型

分析：

- 采用的是hash表进行的遍历，时间复杂度为O(n)，然后为了生成所有的组合，需要遍历所有的元素，时间复杂度为O(R)，所以总的时间复杂度为O(max(R, N))
- 因为遍历数组时候将其均存储在了hash表中，所以空间复杂度为O(N)，因为需要存储所有的元素对，所以空间复杂度为O(R)，所以总的时间复杂度为O(max(R, N))

结论：

- 时间复杂度：O(max(R, N))
- 空间复杂度：O(max(R, N))

浮点类型

暂时没有想到更好的方法，暂时没有实现

One More Thing

为了快速实现，我同时也实现了一个python版本的代码，代码地址为：solution

可以通过以下命令进行测试：

```
cd python && \
python3 solution.py
```