CS101 Algorithms and Data Structures
Fall 2022
Homework 3

Due date: 23:59, October 12th, 2022

1. Please write your solutions in English.
2. Submit your solutions to gradescope.com.
3. Set your FULL name to your Chinese name and your STUDENT ID correctly in Account Settings.
4. If you want to submit a handwritten version, scan it clearly. `CamScanner` is recommended.
5. When submitting, match your solutions to the problems correctly.
6. No late submission will be accepted.
7. Violations to any of the above may result in zero points.

1 4 10 18 15 5 7 5 14

1. (8 points) Which Sort?

Given a sequence

$$A = \langle 4, 10, 18, 15, 5, 1, 5, 14, 7, 7 \rangle,$$

we have performed some different sorting algorithms on it, during which some intermediate results are printed. Note that the steps you see below are **not** necessarily consecutive steps in the algorithm, but they are guaranteed to be in the correct order.

For each group of steps, guess ($\surd$) what the algorithm is. The algorithm might be one among the following choices:

- Insertion-sort, implemented in the way that avoids swapping elements 右往左
- Bubble-sort, which stops immediately when no swap happens during one iteration 左往右.
- Merge-sort
- Quick-sort, with pivot chosen to be A_l when partitioning a subarray $A_l, \cdots, A_r$.

(a) (2')

$$\langle 4, 10, 18, 15, 5, 1, 5, 14, 7, 7 \rangle,$$
$$\langle 4, 10, 18, 5, 15, 1, 5, 14, 7, 7 \rangle,$$
$$\langle 4, 10, 5, 15, 18, 1, 5, 14, 7, 7 \rangle,$$
$$\langle 4, 5, 10, 15, 18, 1, 5, 14, 7, 7 \rangle.$$

4 10

1 7

○ Insertion-sort ○ Bubble-sort ✓ Merge-sort ○ Quick-sort

(b) (2')

$$\langle 4, 10, 15, 18, 5, 1, 5, 14, 7, 7 \rangle,$$
$$\langle 4, 5, 10, 15, 18, 1, 5, 14, 7, 7 \rangle,$$
$$\langle 1, 4, 5, 10, 15, 18, 5, 14, 7, 7 \rangle,$$
$$\langle 1, 4, 5, 5, 10, 15, 18, 14, 7, 7 \rangle.$$

✓ Insertion-sort ○ Bubble-sort ○ Merge-sort ○ Quick-sort

(c) (2')

$$\langle 4, 10, 15, 5, 1, 5, 14, 7, 7, 18 \rangle,$$
$$\langle 4, 10, 5, 1, 5, 14, 7, 7, 15, 18 \rangle,$$
$$\langle 4, 5, 1, 5, 10, 7, 7, 14, 15, 18 \rangle,$$
$$\langle 4, 1, 5, 5, 7, 7, 10, 14, 15, 18 \rangle.$$

○ Insertion-sort ✓ Bubble-sort ○ Merge-sort ○ Quick-sort

(d) (2')

$$\langle 1, 4, 18, 15, 5, 7, 5, 14, 7, 10 \rangle,$$
$$\langle 1, 4, 18, 15, 5, 7, 5, 14, 7, 10 \rangle,$$
$$\langle 1, 4, 10, 15, 5, 7, 5, 14, 7, 18 \rangle,$$
$$\langle 1, 4, 7, 5, 7, 5, 10, 14, 15, 18 \rangle.$$

○ Insertion-sort ○ Bubble-sort ○ Merge-sort ✓ Quick-sort

2. (6 points) Best Sort

There is no such thing as a generally 'best' sorting algorithm on all kinds of problems. For each of the following situations, choose ($\surd$) the most suitable sorting algorithm. Your choice should be the one that satisfies all the special constraints and is most efficient.

(a) (2') Sorting an array of coordinates of points $\langle(x_1, y_1), \cdots, (x_n, y_n)\rangle$ on a 2d plane in ascending order of the x coordinate, while preserving the original order of the y coordinate for any pair of elements $(x_i, y_i), (x_j, y_j)$ with $x_i = x_j$. nlogn

◯ Insertion-sort ◯ Bubble-sort ✓ Merge-sort ◯ ~~Quick-sort~~ 逆序对个数.

$\theta(n+d)$.

(b) (2') Sorting an array that is *almost* sorted with only $n/2$ inversions due to some kind of perturbation.

✓ Insertion-sort ◯ Bubble-sort ◯ Merge-sort nlog. ◯ Quick-sort nlog .

(c) (2') Sorting an array on an embedded system with quite limited memory. You may only use $\Theta(1)$ extra space, but a higher time cost is acceptable.

✓ Insertion-sort ◯ Quick-sort ◯ ~~Merge-sort~~

1个指针

3. (6 points) Multiple Choices

Each question has **one or more** correct answer(s). Select all the correct answer(s). For each question, you will get 0 points if you select one or more wrong answers, but you will get 1 point if you select a non-empty subset of the correct answers.

Write your answers in the following table.

(a)	(b)	(c)
CD	D	ACD

$\Theta(k)$

(a) (2') Which of the following statements are true?

A. In the k-th iteration of insertion-sort, finding a correct position for a new element to be inserted at takes $\Theta(k)$ time. If we use *binary-search* instead (which takes $\Theta(\log k)$ time), it is possible to optimize the total running time to $\Theta(n \log n)$.

B. Traditional implementations of merge-sort need $\Theta(n \log n)$ time when the input sequence is sorted or reversely sorted, but it is possible to make it $\Theta(n)$ on such input while still $\Theta(n \log n)$ on average case.

C. Insertion-sort takes $\Theta(n)$ time if the number of inversions in the input sequence is $\Theta(n)$. $n+d$

D. The running time of a comparison-based algorithm could be $\Omega(n)$.

(b) (2') Which of the following implementations of quick-sort take $\Theta(n \log n)$ time in **worst case**?

A. Randomized quick-sort, i.e. choose an element from $\{a_l, \cdots, a_r\}$ randomly as the pivot when partitioning the subarray $\langle a_l, \cdots, a_r \rangle$.

B. When partitioning the subarray $\langle a_l, \cdots, a_r \rangle$ (assuming $r - l \geqslant 2$), choose the median of $\{a_l, a_m, a_r\}$ as the pivot, where $m = \lfloor (l + r)/2 \rfloor$.

C. When partitioning the subarray $\langle a_l, \cdots, a_r \rangle$ (assuming $r - l \geqslant 2$), choose the median of $\{a_x, a_y, a_z\}$ as the pivot, where x, y, z are three different indices chosen randomly from $\{l, l + 1, \cdots, r\}$.

D. None of the above.

(c) (2') Which of the following situations are **true** for an array of n random numbers?

A. The number of inversions in this array can be found by applying a recursive algorithm adapted from merge-sort in $\Theta(n \log n)$ time.

B. It is expected to have $O(n \log n)$ inversions.

C. If it has exactly $n(n-1)/2$ inversions, it can be sorted in $O(n)$ time. $\theta(n)$.

D. If the array is $\langle 6, 4, 5, 2, 8 \rangle$, there are 5 inversions.

4. (5 points) k-th Minimal Value

Given an array $\langle a_1, \cdots, a_n \rangle$ of length n with *distinct* elements and an integer $k \in [1, n]$, we will design an algorithm to find the k-th minimal value of a. We say a_x is the k-th minimal value of a if there are exactly $k-1$ elements in a that are less than a_x, i.e.

$$|\{i \mid a_i < a_x\}| = k - 1.$$

Consider making use of the '**partition**' procedure in quick-sort. The function has the signature

```
int partition(int a[], int l, int r);
```

which processes the subarray $\langle a_l, \cdots, a_r \rangle$. It will choose a pivot from the subarray, place all the elements that are less than the pivot before it, and place all the elements that are greater than the pivot after it. After that, the index of the pivot is returned.

Our algorithm to find the k-th minimal value is implemented below.

```
// returns the k-th minimal value in the subarray a[l],...,a[r].
int kth_min(int a[], int l, int r, int k) {
  auto pos = partition(a, l, r), num = pos - l + 1;
  if (num == k)
    return a[pos];
  else if (num > k)
    return kth_min(a, l, pos -1, k);
  else
    return kth_min(a, pos +1, r, k-num);
}
```

By calling `kth_min(a, 1, n, k)` we will get the answer.

(a) (2') Fill in the blanks in the code snippet above.

(b) (3') What's the time complexity of our algorithm in the **worst case**? Please answer in the form of $\Theta(\cdot)$ and fully justify your answer.

Solution: k is given, if each time the element we choose is the smallest

we need $(n-1+1)+(n-2+1)+\cdots+(n-k-1+1) = \frac{(n+n-k)k}{2} = nk-\frac{k^2}{2}$

the time complexity is $\Theta(nk)$

5. (2 points) Discovery

(a) (2') Is C++ STL `std::sort` stable or not? If not, is there any stable sort function provided by the standard library?

Solution: When there're less than 32 elements, use insertion sort, which is stable. When there're more than 32 elements, use quick sort, which is unstable. There is a function stable_sort() provided by stl which is stable.

(b) (0') Suppose A is an array of size n. If we can find the median value of A within $O(n)$ time, it is possible to make quick-sort $\Theta(n \log n)$ in worst case. STFW (Search The Friendly Web) about how to find the median value in $O(n)$ time.

(c) (0') It is known that some sorting algorithms, like quick-sort, need to swap elements. Run the following code, change the value of n and see how the output changes.

```
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <vector>

namespace std {
  template <>
  inline void swap<int>(int &lhs, int &rhs) noexcept {
    auto tmp = lhs;
    lhs = rhs;
    rhs = tmp;
    std::cout << "swap is called.\n";
  }
} // namespace std

int main() {
  std::srand(19260817);
  constexpr int n = 10;
  std::vector<int> vec;
  for (auto i = 0; i != n; ++i)
    vec.push_back(std::rand());
  std::sort(vec.begin(), vec.end());
  return 0;
}
```

From your observation, the `swap` function is never called when $n \leqslant$ ____. What algorithm(s) does `std::sort` use?

Solution: