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. **(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
- Bubble-sort, which stops immediately when no swap happens during one iteration
- Merge-sort
- Quick-sort, with pivot chosen to be A_l when partitioning a subarray $A_l, \cdots, A_r$.

(a) (2')

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

○ Insertion-sort ○ Bubble-sort $\surd$ **Merge-sort** ○ Quick-sort

(b) (2')

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

$\surd$ **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 $\surd$ **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 $\surd$ **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 $x_i = x_j$.

◯ Insertion-sort ◯ Bubble-sort $\surd$ **Merge-sort** ◯ Quick-sort

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

$\surd$ **Insertion-sort** ◯ Bubble-sort ◯ Merge-sort ◯ Quick-sort

(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.

$\surd$ **Insertion-sort** ◯ Quick-sort ◯ Merge-sort

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)
BCD	D	ACD

(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 $\Theta(n)$.

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.

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: let $T(n)$ be the time complexity with n elements in the array
for the worst case
1 if $k \le \frac{n}{2}$,the partition always choose the pivot at the end of the sub-array
$T(n) = T(n-1) + \Theta(n)$
$T(n-1) = T(n-2) + \Theta(n-1)$
...
and $T(k) = 1$
so $T(n) = T(k) + n + (n-1) + ... + (k+1) = \frac{n^2+n-k^2-k-2}{2} = \Theta(\frac{3}{8}n^2 + \frac{n}{4} + 1)$
$T(n) = \Theta(n^2)$
2 if $k > \frac{n}{2}$,the partition always choose the pivot at the begin of the sub-array
$T(n) = T(n-1) + \Theta(n)$
$T(n-1) = T(n-2) + \Theta(n-1)$
...

and $T(n-k+1)=1$
so $T(n)=T(n-k+1)+n+(n-1)+...+(n-k)=nk-\frac{k^2}{2}=\Theta(\frac{n^2}{2})$
$T(n)=\Theta(n^2)$
so above all, the worst case of the time complexity is $\Theta(n^2)$

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:** $C++$ STL $std :: sort$ is not stable.
> but the $std :: stable\_sort$ 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.

> **Solution:**
> In the book $Introductionotoalgorithms$, and by searching the friendly internet, I found an algorithm called $BFPRT$, the name is from five of its inventors.
> the algorithm is mainly decided by the following steps.
>
> - divide the array into groups every five adjacent numbers
> - for each group of numbers, we find the median of the five numbers and form a median array of the median of all groups
> - the median of each group forms a new array
> - recursively do the above steps
> - finally there reamin only one number, the medium number of the whole array.
>
> $T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + c \cdot n$
> and it is proved on book and internet that $T(n) \leq 10c \cdot n$
> so $T(n) = \Theta(n)$
> actually, when I was learning algorithms before, I hearded that we can use stl in C++, the $std :: nth\_element(...)$ can find the k_th minimal element with in $\Theta(n)$ time.
> the median element can be easily find by using it, so I just want to write it before scearching
> However, when I scearch the code of it clearly, I thought, and some analysis on the internet also say that the time complexity of $std :: nth_element$ is $\Theta(n)$ on average, instead of the worst case, the worst case might turn into $\Theta(n^2)$
> so it might not be the best way to find the medium number, the $DFPRT$ algorithm can get the median in $\Theta(n)$

(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>
```

```cpp
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: the `swap` function is never called when $n \leqslant 16$.

the std::sort is based on the introspective sorting. It is combined with many different type of sorting algorithms, and it will execute different onces if it reach some different situations.

the threshold of the std::sort is 16.

- the length of the array need to be sorted is less than or equal to the threshold the std::sort will execute the insertion-sort.
- the length of the array need to be sorted is more than the threshold,

  there is another value $depth\_limit$, which usually set as $2 * \log N$, N is the length of the array need to be sorted

  - the time of the recursion is less than or equal to the $depth\_limit$ it will execute the recursive quick-sort.
  - the time of the recursion more than the $depth\_limit$ it will execute heap-sort.