Lab01-Algorithm Analysis

CS214-Algorithm and Complexity, Xiaofeng Gao, Spring 2019.

∗ If there is any problem, please contact TA Mingran Peng. Also please use English in homework.

∗ Name:Kylin Chen Student ID:517030910155 Email: k1017856853@icloud.com

1. Read Algorithm **??** and Algorithm **??** carefully.

Algorithm 1: SelectionSort

Input: An array $A[1, \cdots, n]$
Output: $A[1, \cdots, n]$ sorted nonincreasingly

```
1  i ← 1;
2  for i ← 1 to n − 1 do
3      max ← A[i]; pos ← i;
4      for j ← i + 1 to n do
5          if A[j] > max then
6              max ← A[j];
7              pos ← j;
8      swap A[pos] and A[i];
```

Algorithm 2: CocktailSort

Input: An array $A[1, \cdots, n]$
Output: $A[1, \cdots, n]$ sorted nonincreasingly

```
1  i ← 1; j ← n; sorted ← false;
2  while not sorted do
3      sorted ← true;
4      for k ← i to j − 1 do
5          if A[k] < A[k + 1] then
6              swap A[k] and A[k + 1];
7              sorted ← false;
8      j ← j − 1;
9      for k ← j downto i + 1 do
10         if A[k − 1] < A[k] then
11             swap A[k − 1] and A[k];
12             sorted ← false;
13     i ← i + 1;
```

Fill in the blanks and explain your answers. You need to answer when the best case and the worst case happen. (Hint: if it's both $O(g)$ and $\Omega(g)$, just answer $\Theta(g)$)

Algorithm	Time Complexity	Space Complexity
InsertionSort	$O(n^2)$, $\Omega(n)$	$\Theta(1)$
CocktailSort	$O(n^2)$, $\Omega(n)$	$\Theta(1)$
SelectionSort	$\Theta(n^2)$	$\Theta(1)$

Solution. For a comparison algorithm, it costs the most time to operate comparision. Therefore, we assume each comparision operation takes $O(1)$ time.

Claim 1(CocktailSort): The number of comparisons carried out by Algorithm CocktailSort is at least

$$(n-1)+(n-2)=2n-3$$

and at most

$$\sum_{i=1}^{n-1} i=(n-1)+(n-2)+\cdots+3+2+1=\frac{n(n-1)}{2}$$

.

1) The best case is that **case 1.** The input array already sorted nonincreasingly (in this case, the bool value*sorted* is true, and the comparisons only last two rounds.) or **case 2.** Only the largest and smallest elements are not sorted (for example, array[2,5,1,3,4] have only the smallest elements 1 and largest 5 nonsorted), if we thinking about exchanging two elements' value takes $O(1)$, **case 1** will be the best, and the time complexity is $\Omega(n)$.
2) The worst case is that the input array already sorted decreasingly, in this case, the bool value*sorted* always keeps false until all comparisions down.Therefore, the worst time complexity is $O(n^2)$.
3) Whether input is sorted or not, CocktailSort Algorithm only takes constant extra space, so its space complexity will be $\Theta(1)$.

Claim 2(SelectionSort): For SelectionSort, comparision statements run until the end of the loop. The number of all comparision steps is

$$\sum_{i=2}^{n-1} i = (n-1) + (n-2) + \cdots + 3 + 2 = \frac{(n+1)(n-2)}{2}$$

it means the time complexity of SelectionSort is both $O(n^2)$ and $\Omega(n^2)$, just represents $\Theta(n^2)$.

If we assume each exchanging operation takes $O(1)$, the best case happens when the input array is well sorted nonincreasingly, and worst case happens when input array is in reverse order(sorted increasingly).

As for space complexity, SelectionSort only takes constant extra space, so it's $\Theta(1)$.

2. Let us assume that you have learned two type of data structures: **Stack** and **Queue**. **Stack** has two operations: *push* and *pop*, while **Queue** also has two operations: *enqueue* and *dequeue*.

   Now you have two **Stacks**, how can you use them to simulate a **Queue**?

   (a) Briefly explain your approach. (Pseudo code is not needed.)
   (b) Give the time complexity of *enqueue* and *dequeue* operations of the simulated **Queue**. Use **potential function** for amortized analysis.

Solution.

(a) There are two approaches to achieve Queue simulation, which corresponding two types of algorithm complexity. They can be summed up as follows:
- Now that we have two stacks, we define them as A and B. For any *enqueue* operation, we *push* the element in stack A. For any *dequeue* operation, we devide it into two cases to discuss:
  (1. If stack B is empty, we *pop* one element from stack A, then we *push* the just-popped element into stack B, repeat this operation until stack A is empty. finally, we *pop* one element from B as *dequeue* function returning value.
  (2. If stack B is not empty, we *pop* the stack B and get one element as *deque* function returning value.
- Equally, we define these provided two stacks as A and B. For any *dequeue* operation, we just *pop* one element from stack A as its returning value. For any *enqueue* operation, we have some basic work to do:

(1. Repeatedly *pop* one element from stack A and *push* it into stack B until A is empty.
(2. For the *enqueue* element, we *push* it into empty stack A, then stack A has only one element.
(3. Repeatedly *pop* one element from stack B and *push* it into stack A until B is empty.

(b) In the solution (a), we give two methods to achieve $Stack - Queue$ simulation, which have two types of algorithm complexity. We will give the analysis and summarizing one by one.

- For the first method, we have two parameter, we define the number of elements in stack A is N_A as well as number of B is N_B.

  Potential Function: Let $\phi(S)$ denotes the double number of items in stack A. That is to say, $\phi(S) = 2 \times N_A$.
  Correctness: $\phi(S_i) \geq 0 = \phi(S_0)$ for any i; $\phi(S_i) - \phi(S_0) \geq 0$ for any i.
  Operation Complexity:
  - **Enqueue:** $\hat{C}_i = C_i + \phi(S_i) - \phi(S_{i-1}) = 1 + 2 \times (N_{A(i)} - N_{A(i-1)}) = 1 + 2 = 3$;

    Dequeue:
    [1.] If the stack B is not empty, $N_{A(i)} = N_{A(i-1)}$.
    That is to say,
    $$\phi(S_i) - \phi(S_{i-1}) = 0$$
    Therefore, we can get the cost :
    $$\hat{C}_i = C_i + \phi(S_i) - \phi(S_{i-1}) = 1 + 0 = 1$$
    [2.] If the stack B is not empty, We can get these formula:
    $$\phi(S_{i-1}) = 2 \times N_{A(i-1)}$$
    $$\phi(S_i) = 0$$
    $$C_i = C_{pop} + C_{push} = N_{A(i-1)} + N_{A(i-1)} = 2 \times N_{A(i-1)}$$
    Therefore, we can get the cost:
    $$\hat{C}_i = C_i + \phi(S_i) - \phi(S_{i-1}) = 1 + 0 = 1$$

  Key Observation:
  $$\#enqueue \geq \#dequeue$$

  Amortized Analysis: Thus, starting from an empty *Queue*, any sequence of n_1 *enqueue*, n_2 *dequeue* operations takes at most
  $$T(n) = \sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \hat{C}_i = 3n_1 + n_2 \leq 4n_1$$
  Here $n = n_1 + n_2$.
  If we assume $n_1 \approx n_2$, we can get average step cost:
  $$\overline{C_i} = \frac{3n_1 + n_2}{n_1 + n_2} \approx 2$$

  Extra-analysis: On the other hand, we can find that operation *enqueue* always takes $O(1)$. However, in order to execute operation *dequeue*, we need *pop* it from stack A and *push* it into stack B, so it means we should take extra two steps to achieve the goal. Therefore, operation *dequeue* takes $O(3)$ averagely.

- For the second method, we can find stack B only provide assistance because B is always empty after every step. We assume the number of elements in stack A is N.
  Potential Function: since the *dequeue* only takes 1 or constant steps, we think its potential function is C (C is a constant). But for *enqueue* operation, we assume N is its potential function. That is,

$$\phi(S) = \begin{cases} N & , enqueue, \\ C & , dequeue. \end{cases}$$

  Correctness: $\phi(S_i) \geq 0 = \phi(S_0)$ for any i.
  Operation Complexity:

  Enqueue: $\phi(S_i) - \phi(S_{i-1}) = 1$; $\hat{C}_i = C_i + \phi(S_i) - \phi(S_{i-1}) = 2$;
  * **Dequeue:** $\phi(S_i) - \phi(S_{i-1}) = 0$; $\hat{C}_i = C_i + \phi(S_i) - \phi(S_{i-1}) = 1$

  Key Observation: #Enqueue $\geq$ #Dequeue
  Thus, starting from an empty *Queue*, any sequence of n_1 *enqueue*, n_2 *dequeue* operations takes at most

$$T(n) = \sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \hat{C}_i = 2 \times n_1 + 1 \times n_2 \leq 3n_1$$

  Here $n = n_1 + n_2$.
  Extra-analysis: On the other hand, we can find that operation *dequeue* always takes $O(1)$. However, in order to execute operation *dequeue*, we need *pop* all elements from stack A and *push* them into stack B in order, and repeat it again, so it means we should take extra 4 steps to achieve the goal. Therefore, operation *dequeue* takes $O(6)$ averagely.