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: Weiqi Feng Student ID: 517021910752 Email: fengweiqi@sjtu.edu.cn

1. Read Algorithm 1 and Algorithm 2 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 blank 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.

- Analysis of *Cocktail Sort* :
  (a) An illustration of *Cocktail Sort*

  Before the analysis of time complexity and space complexity, we firstly show how *Cocktail Sort* works. Each iteration of *Cocktail Sort* is broken up into following **two stages**∗. †

†Rahul Agrawal (2019) Cocktail Sort. [Online] Available from: https://www.geeksforgeeks.org/cocktail-sort [Accessed 06/03/19].

– The first stage loops through the array from left position i to position $j-1$. During the inner **for** loop, adjacent items are compared and if value on the left is smaller than the value on the right, values are swapped.
At the end of first inner *for* loop, the smallest number will reside at the position j.
– The second stage loops through the array in the opposite direction - starting from the position $j-1$, and moving back to the the position $i+1$. Similarly, adjacent items are compared and are swapped if meeting the requirement.
At the end of second inner *for* loop, the largest number will reside at the position i.

Here is an illustration of *Cocktail Sort* as depicted in Figure 1.

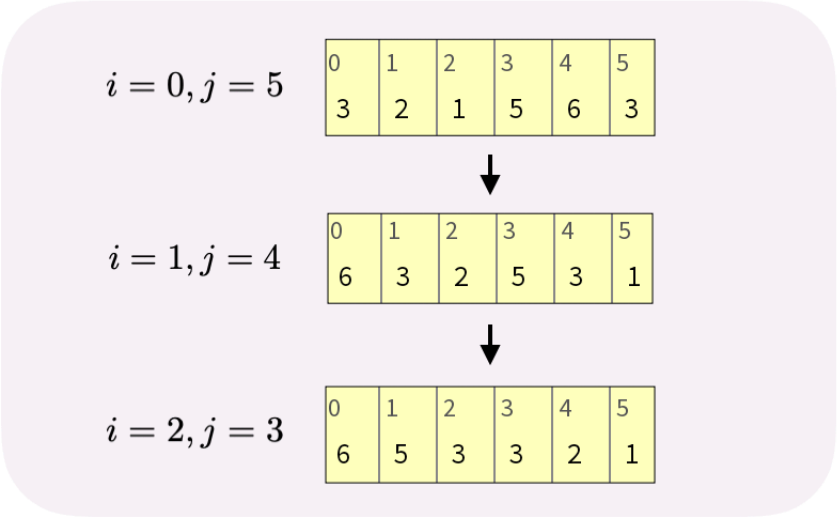

Figure 1: An illustration of *Cocktail Sort*

(b) Time Complexity

In the analysis of time complexity of *Cocktail Sort*, we use the number of iterations as a good indicative of the running time.

– **The best case is that array A is already sorted nonincreasingly .**
In the first **while** loop, there is no index k which satisfies $A[k] < A[k+1]$. Thus *sorted* is set to be **true** in step 3 and remains it value. There are totally $2n-3$ iterations of the inner **for** loop in the first **while** iteration. In the second **while** loop, the *sorted* value is **true** which contradicts with the loop condition and the loop ends.
We can conclude that the time complexity of *Cocktail Sort* is $\Omega(n)$
– **The worst case is that array A is in an increasing order.**
Firstly, we can easily derive that there are $((j-i)+(j-i-1))$ inner iterations in each **while** loop.
So that the total iterations can be calculated as follows:

$$\sum_{i=1,j=n}^{A \text{ is sorted}} (2j - 2i - 1)$$

After each **while** loop, we decrease the value of j and increase the value of i both by 1. Thus we can get the equation between i and j:

$$j = n + 1 - i$$

For the worst case, the while loop stops until i reaches $\lfloor \frac{n}{2} \rfloor$.
So we can derive the total iterations:

$$\begin{aligned}\sum_{i=1}^{i=\lfloor \frac{n}{2} \rfloor} (2n + 1 - 4i) &= \sum_{i=1}^{i=\lfloor \frac{n}{2} \rfloor} (n + 1 - 2i) + \sum_{i=1}^{i=\lfloor \frac{n}{2} \rfloor} (n - 2i) \\ &. = \sum_{i=1}^{n} (n - i) \\ &= \frac{n(n-1)}{2}\end{aligned}$$

So that the time complexity of *Cocktail Sort* is $O(n^2)$.

(c) Space Complexity

In Algorithm *Cocktail Sort*, only constant number of auxiliary memory cells are used to hold the value of $sorted, i, j, k, sorted$.
So we can conclude that the space complexity is $\Theta(1)$ for all cases.

- Analysis of *Selection Sort*:

(a) Time complexity

In the analysis of time complexity of *selection sort*, we use the number of iterations as a good indicative of the running time.
The inner **for** loop is executed repeatedly for the following values of n:

$$n - 1, n - 2, n - 3, \ldots, 1.$$

Thus, the total number of Step 6 is executed is

$$\sum_{i=1}^{n-1} (n - i).$$

Since

$$\sum_{i=1}^{n-1} (n - i) = \frac{n(n-1)}{2},$$

we conclude that Step 6 is executed $\Theta(n^2)$ times. Note that the number of iterations is independent on the input array A.
We conclude that *Selection Sort* is $\Theta(n^2)$ for both the best case and the worst case.

(b) Space Complexity

In Algorithm *Selection Sort*, only constant number of memory cells are used to hold the value of max, pos, i, j.
So we can conclude that the space complexity is $\Theta(1)$ for all cases.

□

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 eplain your approach. (Pusudo code is not needed.)

   (b) Give out the time complexity of *enque* and *deque* operations of the simulated **Queue**. Use **potential function** for amortized analysis.

Solution.

(a)

enqueue(x):

(1) push x to **Stack1**.

dequeue():

(1) If both **Stacks** are empty, then raise an **Queue-empty-exception**.

(2) If **Stack2** is empty, push everything from **Stack1** to **Stack2** While **Stack1** is not empty.

(3) Pop the top of **Stack2** and return it.

Here is an illustration of *enqueue* and *dequeue* operation in Figure 2.

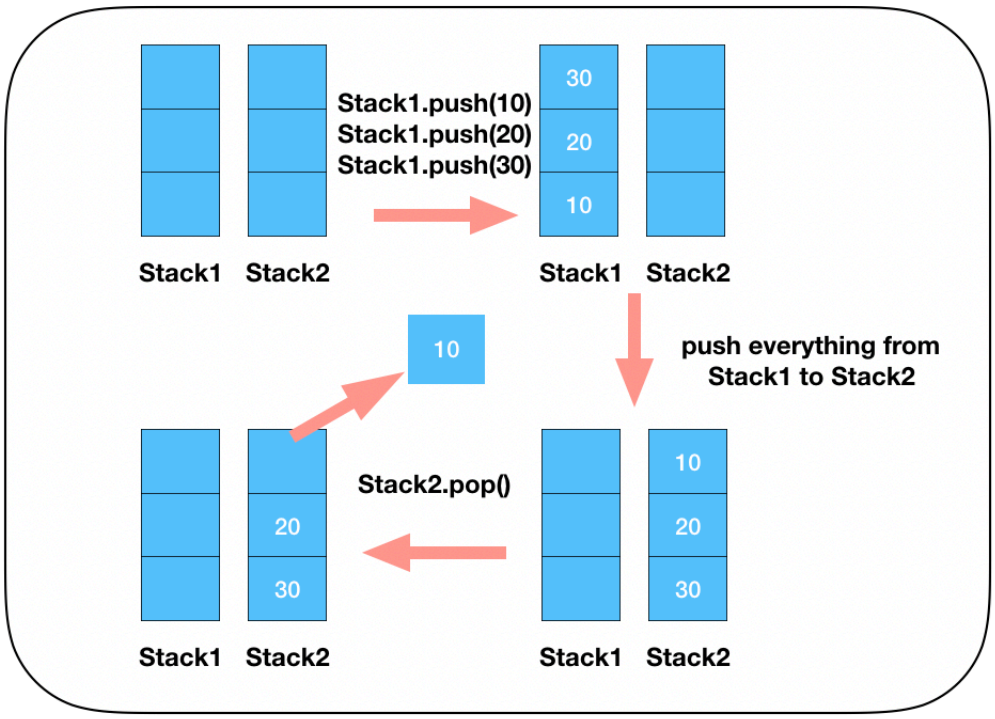

Figure 2: How *enqueue* and *dequeue* works on a simple example.

(b)

Considering we have n *enqueue* and *dequeue* operations. And we want to use **potential function** method for amortized analysis.

Firstly, we consider the cost C_{enq} for *enqueue* and C_{deq} for *dequeue*.

$C_{enq} = 1$ and C_{deq} is dependent on whether **Stack2** is empty.

$$C_{deq} = \begin{cases} 1 & \textbf{Stack2} \text{ is not empty} \\ 2size(\textbf{Stack1}) + 1 & \textbf{Stack2} \text{ is empty} \end{cases}$$

When we want to do the *dequeue* operation and **Stack2** is empty, we need to first *pop* all elements from **Stack1**, push all elements to **Stack2** and pop the top of **Stack2**. The total cost $C_{deq} = 2size(\textbf{Stack1}) + 1$. Another two cases are naive and both cost only 1.

We define a potential function $\Phi(S) : S \to R$, where S is the state of **Stack1** and **Stack2**.

$$\Phi(S_i) = 2size(\textbf{Stack1})$$

Correctness:

$$\Phi(S_i) = 2size(\textbf{Stack1}) \geq 0 = \Phi(S_0)$$

Then amortized cost setting:

$$\hat{C}_i = C_i + \Phi(S_i) - \Phi(S_{i-1})$$

.

The amortized cost of *enqueue*:

$$\begin{aligned} \hat{C}_{enq} &= 1 + \Phi(S_i) - \Phi(S_{i-1}) \\ &= 1 + 2(size(\textbf{Stack1}) + 1) - 2size(\textbf{Stack1}) \\ &= 3 \end{aligned}$$

The amortized cost of *dequeue* when **Stack2** is not empty:

$$\begin{aligned} \hat{C}_{enq} &= 1 + \Phi(S_i) - \Phi(S_{i-1}) \\ &= 1 + 2size(\textbf{Stack1}) - 2size(\textbf{Stack1}) \\ &= 1 \end{aligned}$$

The amortized cost of dequeue when **Stack2** is empty:

$$\begin{aligned} \hat{C}_{enq} &= 2size(\textbf{Stack1}) + 1 + \Phi(S_i) - \Phi(S_{i-1}) \\ &= 2size(\textbf{Stack1}) + 1 + 0 - 2size(\textbf{Stack1}) \\ &= 1 \end{aligned}$$

Then the amortized cost of *enqueue* and *dequeue* are 3 and 1 respectively.

Finally, we can derive that

$$\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \hat{C}_i \leq \sum_{i=1}^{n} 3 = 3n.$$

The amortized cost of each operation can be $\frac{T(n)}{n} = O(1)$.

□

Remark: You need to include your .pdf and .tex files in your uploaded .rar or .zip file.