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:Lynn Xiao Student ID:________ Email: ___________

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

$i \leftarrow 1$;
for $i \leftarrow 1$ ***to*** $n-1$ **do**
  $max \leftarrow A[i]; pos \leftarrow i$;
  for $j \leftarrow i+1$ ***to*** n **do**
    if $A[j] > max$ **then**
      $max \leftarrow A[j]$;
      $pos \leftarrow j$;
  swap $A[pos]$ and $A[i]$;

Algorithm 2: CocktailSort

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

$i \leftarrow 1$; $j \leftarrow n$;$sorted \leftarrow false$;
while ***not*** $sorted$ **do**
  $sorted \leftarrow true$;
  for $k \leftarrow i$ ***to*** $j-1$ **do**
    if $A[k] < A[k+1]$ **then**
      swap $A[k]$ and $A[k+1]$;
      $sorted \leftarrow false$;
  $j \leftarrow j-1$;
  for $k \leftarrow j$ ***downto*** $i+1$ **do**
    if $A[k-1] < A[k]$ **then**
      swap $A[k-1]$ and $A[k]$;
      $sorted \leftarrow false$;
  $i \leftarrow i+1$;

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.
(i) Analysis of $InsertionSort$

The best case: the array has been sorted non-increasingly. We only need to do $n-1$ times of comparison. So the time complexity is $\Omega(n)$.

The worst case: the array is in an increasing order.So the times of comparison will be $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$.

space complexity: the extra space only contains the space of constant number of variables, so the space complexity is $\Theta(1)$.

(ii) Analysis of $CocktailSort$.

We will first show how the algorithm works. Each iteration of $CocktailSort$ is broken up into following two steps.

After the first loop range from i to $j-1$, the smallest number will be put at the position $j-1$. After the second loop range from j down-to $i+1$, the largest number will be put at position i. So this algorithm works properly to sort the array non-increasingly.

The best case: the array has been sorted non-increasingly. We only need to do $2n-3$ times of comparison. So the time complexity is $\Omega(n)$.

The worst case: the array is in an increasing order. We can get the times of comparison is $\frac{n(n-1)}{2}$. So the time complexity is $\Omega(n)$.

space complexity: the extra space only contains the space of constant number of variables, so the space complexity is $\Theta(1)$.

(iii) Analysis of *SelectionSort* Regardless of the order of the array, *SelectionSort* will take constant times of comparison which is $\frac{n(n-1)}{2}$.

Also it is obviously that the space complexity is $\Theta(1)$. □

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)

Assume the two stacks are **stack1** and **stack2**.

(i) $enqueue(x)$:push x to **stack1**;

(ii) $dequeue(x)$: If both two stacks are empty, then raise an exception.

If **stack2** is empty, then pop all the numbers and push them to **stack2**. Finally pop the top of **stack2**.

(b)

Considering we have n operations, including *enqueue* and *dequeue*.

Let C_{enq} be the cost for *enqueue* and C_{deq} be the cost for *dequeue*. Then $C_{eq} = 1$ then C_{dq} is dependent on whether **stack2** is empty. When **stack2** is empty, C_{dq} is 1 and when **stack2** is not empty, C_{dq} is $2size($**stack1**$)$+1.

So we can define a potential function, $\Phi(S_i) = 2size($**stack1**$)$. It is easy to know that $\Phi(S_i) \geq \Phi(S_0)$.

Then amortized cost setting $\hat{C}_i = C_i + \Phi(S_i) - \Phi(S_{i-1})$.

After the operation of *enqueue*, the size of **Stack1** will increase 1.

So the amortized cost of *enqueue* is:

$\hat{C}_{enq} = C_i + \Phi(S_i) - \Phi(S_{i-1}) = 3$

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

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

because the size of **Stack1** will remain.

The amortized cost of *dequeue* when **Stack2** is empty is
$\hat{C}_{deq} = C_i + \Phi(S_i) - \Phi(S_{i-1}) = 2size(\mathbf{Stack1}) + 1 + 0 - 2size(\mathbf{Stack1}) = 1$
Finally, we have $\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \hat{C}_i = 3n$.
So the amortized cost of each operation can be $O(1)$. $\square$

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