Lab02-DC & Greedy

CS2308-Algorithm and Complexity, Xiaofeng Gao, Spring 2023.

∗ Contact TA Jiasen Li for any questions. Include both your .pdf and .tex files in the uploaded .rar or .zip file.
∗ Name: 张祥东 Time Spent on this Lab:7-8 h Your Total Credits this Semester(本学期你的总学分): 24.5

1. (Divide & Conquer: Counting Inversions) Let $A = [a_1, a_2, \cdots, a_n]$, $a_i \in \mathbb{Z}$ be an array (with repeatable elements). (a_i, a_j) is an *inversion* if and only if $i < j$ and $a_i > a_j$. Using the divide-and-conquer to find the number of inversions for A in $O(n \log n)$. Write your pseudo-code and analyze its time complexity.

   Solution. Here comes the pseudo-code

Algorithm 1: $Count\_inversion(A[\cdot], left, right)$

```
Input: An array $A[1, 2, \ldots, n]$ of n elements,the left index,the right index
Output: The number of inversions for array $A[1, 2, \ldots, n]$ range from left to the right
if $left == right$ then
    return 0;
$middle \leftarrow (left + right)/2$;
$answer \leftarrow 0$;
$s \leftarrow left$;
$t \leftarrow middle + 1$;
$count \leftarrow 0$;
$index \leftarrow left$;
$answer \leftarrow answer + Count\_inversion(A[\cdot], left, middle)$;
$answer \leftarrow answer + Count\_inversion(A[\cdot], middle + 1, right)$;
while $s \leq middle$ and $t \leq right$ do
    if $A[s] \leq A[t]$ then
        $B[index] \leftarrow A[s]$;
        $s \leftarrow s + 1$;
        $answer \leftarrow answer + count$;
    else
        $B[index] \leftarrow A[t]$;
        $t \leftarrow t + 1$;
        $count \leftarrow count + 1$;
    $index \leftarrow index + 1$;
if $s \leq middle$ then
    $B[index \ldots right] \leftarrow A[s \ldots middle]$;
else
    $B[index \ldots right] \leftarrow A[t \ldots right]$;
$A[left \ldots right] \leftarrow B[left \ldots right]$;
return answer;
```

By calling function $Count\_inversion(A[\cdot], 1, n)$,we can get the number of inversions of $A[\cdot]$.

In general,we use a Divide-and-Conquer algorithm,dividing the count task into two parts in each step and after these two parts are solved,we merge the result.

Let $T(n)$ denote the Time needed for figuring out the number of inversions for an array of size n,then the problem can be divided into the following parts:

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

Let us explain the above equation.The eqution can be depicted as:

- First,assume we get the left array part's and the right array part's inversion number,the time needed being $2T(\frac{n}{2})$.
- For every element in the left part,we can count how many inversion parteners they have in the right part.Namely for every element in the left part,we count the number of cases in right part that is smaller than it.**And we've kept the left and the right part ordered before,so we just need a $Merge$ to get the number needed. It's $O(n)$**

So for an array of size n,the inversions in it can be calculated by

$$\begin{aligned} Inversion(n) =& Inversion(left\ part) + Inversion(right\ part) \\ &+ Inversion\ between\ left\ part\ and\ right\ part \end{aligned}$$

Above all,we prove the correctness of our algorithm and obtain

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

Using master theorem,because

$$1 = \log_2 2$$

The time complexity for this Divide-and-Conquer algorithm is

$$O(n \log n)$$

*It's noteworthy that in the Line 12 in the pseude-code,**we use $\leq$ but not** $<$ to prevent counting the element in the right part that equals to the current element,guaranteeing we only count the element that is smaller than the current element.

***Other feasible solutions:**

- **Solution 1.**
  - Build a value segment tree.
  - Traverse the array from $n \to 1$,and add every number into the value segment tree,in the meantime,counting the number smaller than the current number which is located at the right of it.
    *One time query could be done by value segment tree in O($\log n$)
  - So
    $$O(n * \log n) = O(n \log n)$$
    This solution is feasible in $O(n \log n)$
- **Solution 2.**
  - Record the value and corresponding index as a new pair array B.
  - Sort B by the value.
  - Add the element in B sequentially to a **fenwick Tree**.
  - After adding one element,query on the **fenwick Tree** for how many numbers have been added before in the right of the current number.
    *One time query could be done by fenwick Tree in $O(\log n)$

– So

$$O(n * \log n) = O(n \log n)$$

This solution is feasible in $O(n \log n)$

□

2. (Divide & Conquer: Linear Selection) To select the i^{th} smallest element in an array $A = [a_1, a_2, \cdots, a_n]$, $a_i \in \mathbb{Z}$, a naïve solution is using QuickSort and returns the i^{th} element. However, after dividing the array into two parts using the pivot, one part can be deleted. Write the pseudo-code of this modified algorithm and calculate its average case time complexity.

Solution. Here comes the pseude-code for the modified algorithm

Algorithm 2: $Modefied\_QuickSort(A[\cdot], l, r, i)$

```
Input: An array $A[1, \ldots, n]$ and i which denote the $i^{th}$ smallest element's index needed
Output: The value of $i^{th}$ smallest element
$s \leftarrow l$;
$pivot \leftarrow A[r]$;
for $t \leftarrow l$ to $r - 1$ do
    if $A[t] \leq pivot$ then
        Swap($A[s]$,$A[t]$);
        $s \leftarrow s + 1$;
Swap($A[s]$,$A[r]$);
if $s == i$ then
    return $A[s]$;
else if $i \leq s$ then
    return $Modified\_QuickSort(A[\cdot], l, s - 1, i)$;
else
    return $Modified\_QuickSort(A[\cdot], s + 1, r, i)$;
```

We take $T(n, k)$ to denote the time needed for finding the k^{th} smallest element in an array of size k.

For the $A[\cdot]$ of size n, after one time division,it could be divided into three parts of n situations:

$$(0, 1, n-1), (1, 1, n-2), \ldots, (k-2, 1, n-k+1), (k-1, 1, n-k), (k, 1, n-k-1), \ldots, (n-1, 1, 0)$$

We'll discard the part that doesn't contain the i^{th} element,so the corresponding time needed for these n situations should be

$$T(n-1, k-1), T(n-2, k-2), \ldots, T(n-k+1, 1), O(1), T(k, k), \ldots, T(n-1, k)$$

The probability for these cases are the same to each other,it's $\frac{1}{n}$

So we obtain

$$T(n, k) = \frac{1}{n}(T(n-1, k-1) + T(n-2, k-2) + \cdots + T(n-k+1, 1) + O(1) + T(k, k) + \cdots + T(n-1, k))$$

We can easily get

$$\begin{aligned} nT(n,k) =& T(n-1,k-1) + T(n-2,k-2) + \cdots + T(n-k+1,1) \\ &+ O(1) + T(k,k) + \cdots + T(n-1,k) \end{aligned} \tag{1}$$

$$\begin{aligned} (n-1)T(n-1,k) =& T(n-2,k-2) + \cdots + T(n-k+1,1) \\ &+ O(1) + T(k,k) + \cdots + T(n-2,k) \end{aligned} \tag{2}$$

So

$$(1) - (2) \rightarrow nT(n,k) = nT(n-1,k) + T(n-1,k-1)$$

Obviously,there exits

$$T(n,k) = T(n,k-1) = \cdots = T(n,1)$$

Because from the aspect of average and our common sense,the value of k should not influence the time needed for finding the target element.

So take the $T(n)$ to record the time needed for finding the k^{th} smallest element in an array of size n. Here comes:

$$T(n) = \frac{n+1}{n}T(n-1) = \cdots = \frac{n+1}{n}\frac{n}{n-1}\cdots\frac{3}{2}T(1) = \frac{n+1}{2}T(1)$$

And $T(1) = 1$ So the $T(n) = \frac{n+1}{2}$,the average time complexity is

$$O(n)$$

□

3. (Greedy with Sorting) Given an array $A[n]$ of strings $s_1, s_2, \cdots, s_n$, where $len(s_1) + \cdots + len(s_n) = m$. We can reorder this array and concatenate all n strings to obtain a string S of length m. (There are $n!$ possible S's with repeatable strings). Design a greedy algorithm with at most $O(n \log n)$ comparisons to find the string S with the minimum lexicographic order. For example: if $A = [\text{“}a\text{”}, \text{“}ba\text{”}, \text{“}bb\text{”}]$, then there are 6 possible strings S, denoted as $S_1 = \text{“}ababb\text{”}$, $S_2 = \text{“}abbba\text{”}$, $S_3 = \text{“}baabb\text{”}$, $S_4 = \text{“}babba\text{”}$, $S_5 = \text{“}bbaba\text{”}$, and $S_6 = \text{“}bbbaa\text{”}$. The one with the minimum lexicographic order is $S_1 = \text{“}ababb\text{”}$. Write the pseudo-code of your algorithm, prove its correctness and analyze the time complexity. If you have no idea how to complete this task, you can refer to the following definitions and the hint.

   Definition 1 (Lexicographic Order*). *A relation to compare strings, simply denoted as “$\leq$”.*

   For instance, when comparing string “*aba*” with string “*abb*”, we will compare them character by character from left to right, and output a result “*aba*”$\leq$“*abb*”.

   Definition 2 (Concatenated Order). *Define a function $\lessdot$ to compare concatenated strings, where $s_i \lessdot s_j$ returns the Boolean result of whether inequality $s_i \circ s_j \leq s_j \circ s_i$ holds.*

   For instance, if $s_i = \text{“}aba\text{”}$, and $s_j = \text{“}ab\text{”}$, then $s_i \lessdot s_j$ holds since “$abaab$” $\leq$ “$ababa$”, although apparently $s_j \leq s_i$. Hint: Design your algorithm according to Def. 2 and then try to prove its correctness by the algebraic attributions of an relation. E.g., given an optimal solution $S^* = s_1 \circ s_2 \circ \cdots \circ s_n$, then $\forall i \in \{1, \cdots, n-1\}$, $s_i \lessdot s_{i+1}$.

*You can refer more details from: https://baike.baidu.com/item/%E5%AD%97%E5%85%B8%E5%BA%8F/7786229.

Solution. We have the relation $\lessdot$ that satisfies

$$s_i \lessdot s_j \iff s_i \circ s_j \le s_j \circ s_i$$

Algorithm 3: Find the minimum lexicographic order sequence

```
    Input: $A[n]$ of strings $s_1, s_2, \ldots, s_n$
    Output: $S^*$ with the minimum lexicographic order
1   sort the array A with the relation $\lessdot$;
2   $S^* \leftarrow empty$;
3   for $i \leftarrow 1$ to $n$ do
4       $S^* \leftarrow S^* \circ A[i]$;
5   return $S^*$;
```

Let us prove the relation is a partial relation:

- We first claim that the a,b,c used here all denote strings.
- Reflexivity.Because

  $$s_i \circ s_i = s_i \circ s_i$$

  We get

  $$s_i \lessdot s_i$$

- Antisymmetry. In the relation $\lessdot$,the definiation of euqal should be

  $$a \overset{\rightarrow}{=} b \iff ab = ba$$

  We assume that $string\ a \neq string\ b\ and\ a \lessdot b\ and\ b \lessdot a$
  So here comes

  $$ab \le ba\ and\ ba \le ab$$

  It's obvious that $ab = ba$. So

  $$a \overset{\rightarrow}{=} b$$

  We obtain

  $$a \lessdot b\ and\ b \lessdot a \rightarrow a \overset{\rightarrow}{=} b$$

  The antisymmetry is satisfied.

- Transitivity.Prove

  $$a \lessdot b, b \lessdot c \rightarrow a \lessdot c$$

  $$\begin{aligned} a \lessdot c &\iff ac \le ca \\ &\iff acb \le cab \end{aligned}$$

  We have

  $$ab \le ba\ and\ bc \le cb$$

  By scaling down and up them,we get

  $$abc \le acb \le cab \le cba$$

  Namely

  $$abc \le cba$$

  Erase b from it and the inequation is still right,so here comes:

  $$ac \le ca$$

  We obtain

  $$a \lessdot c$$

Above all,the order is a partial order.

Next,we claim that the final optimal string $S^* = s_1 \circ s_2 \circ \cdots \circ s_n$ is of feature that $s_1 \lessdot s_2 \lessdot \cdots \lessdot s_n$. Prove comes as follows:

Record my anwer as S_G.

The optimal string as S^*,and the element in S^* satisfies the $S^*[i] = S_G[i]$ until $i = k$

So here comes

$$S_G = s_1 \circ \ldots s_k \circ s_{k+1} \circ \ldots s_n$$

$$S^* = s_1 \circ \ldots s_k^{'} \circ s_{k+1}^{'} \circ \ldots s_n^{'}$$

When contrasting them,we can erase the first $k-1$ element in them.

$$S^* \leq S_G \rightarrow s_k^{'} \circ s_{k+1}^{'} \circ \ldots s_n^{'} \leq s_k \circ s_{k+1} \circ \ldots s_n$$

Obviously,$s_k \neq s_k^{'}$,and s_k is euqals to one element in the sequence $s_{k+1}^{'}, \ldots, s_n^{'}$,taking the index of this element is m.

$$s_k^{'}, \ldots, s_{m-1}^{'}, s_m^{'}, \ldots, s_n^{'}$$

Construct a sequence $S^{'}$ of

$$s_k^{'}, \ldots, s_m^{'}, s_{m-1}^{'}, \ldots, s_n^{'}$$

Because

$$s_m^{'} = s_k$$

so

$$s_m^{'} \lessdot s_{m-1}^{'}$$

$$s_m^{'} \circ s_{m-1}^{'} \leq s_{m-1}^{'} \circ s_m^{'}$$

$$s_k^{'}, \ldots, s_m^{'}, s_{m-1}^{'}, \ldots, s_n^{'} \leq s_k^{'}, \ldots, s_{m-1}^{'}, s_m^{'}, \ldots, s_n^{'}$$

So we can swap the

$$s_m^{'} \; and \; s_{m-1}^{'}$$

And the post S^* is no worse than the previous S^*

Do this operation until the post part of S^* be

$$s_m^{'}, s_k^{'} \ldots, s_{m-1}^{'}, \ldots, s_n^{'}$$

This sequence is no worse than the initial S^*.

Due to $s_m^{'} = s_k$, the k is not valid now and should keep plus one till the k is n + 1 which prove the k does not exist and the

$$S_G = S^*$$

Thus,the prove is completed.

As for the time complexity

For Sort,it's

$$O(n \log n)$$

For the *For operation* in the algorithm,it's

$$O(n)$$

Above all,the time complexity is

$$O(n \log n)$$

□

4. (Circular Interval Scheduling) An array of circular intervals is defined as $A = [s_1, s_2, \cdots, s_n]$, where each $s_i = [a_i, b_i]$, and a_i, b_i are non-negative integers $< k$. Here, if $a_i > b_i$ then the circular interval $[a_i, b_i]$ is regarded as a continuous interval $[a_i, k) \cup [0, b_i]$. Design an algorithm to select a maximum subset of A within $O(n^2)$ time with no overlap intervals. Write its pseudo-code and prove its correctness.

   Hint: Enumerate the beginning interval to apply the algorithm for the non-circular interval scheduling problem and optimize its complexity to $O(n^2)$. In fact, you can further improve the time complexity into $O(n \log n)$ (but it is tough to achieve for beginners and does not count for the grading of this homework).

Solution. Here comes the pseudo-code

Algorithm 4: Find the the maximum compatible subset of A

```
Input: $A = [s_1, s_2, \dots, s_n]$
Output: the maximum compatible subset of A
$S \leftarrow empty$ for $i \leftarrow 1$ to $n$ do
    $[a_i, b_i] = s_i$;
    if $a_i \le b_i$ then
        $S = S \cup [a_i, b_i]$;
        $a_i \leftarrow a_i + k$;$b_i \leftarrow b_i + k$;
        $S = S \cup [a_i, b_i]$;
    else
        $a_i \leftarrow a_i + k$;
        $S = S \cup [a_i, b_i]$;
$m \leftarrow sizeof(S)$;
sort $S[\cdot]$ by the finish time;
$ans\_S \leftarrow empty$; $optimal\_num \leftarrow 0$;
$cur\_S \leftarrow empty$; $count \leftarrow 0$;
for $i \leftarrow 1$ to $m$ do
    $cur\_S \leftarrow empty$;
    $cur\_S = cur\_S \cup S[i]$;
    for $j \leftarrow i + 1$ to $i + n - 1$ do
        if S[i] is compatible with cur_S then
            $cur\_S = cur\_S \cup S[i]$;
            $count \leftarrow count + 1$;
    if $optimal\_num < count$ then
        $optimal\_num \leftarrow count$;
        $ans\_S \leftarrow cur\_S$;
return $ans\_S$;
```

Definition 3. $Opt_{element} \doteq$ *the optimal maximum compatible subset that takes the element as the first element.*

Definition 4. $Alg_{element} \doteq$ *the maximum compatible subset that takes the element as the first element found by my Greedy algorithm.*

***note: the element refers to one interval in the set.**

To make it clear,my greedy algorithm first extend the element and sort them by the finish time.Then I try every element in the interval set and find the minimum finish time that is compatible to the current set and add it into the current set.Finally,I choose the best one from all sets that I've gotten.

In detail,$Alg_{element}$ is selected by taking the **element** as the first interval,and **try next** $n-1$ **intervals which've been already sorted by the finish time and add the ones to the** $Alg_{element}$ **that are compatible with the previous** $Alg_{element}$;

Prove $Opt_{element} = Alg_{element}$:

Assume that

$$Opt_{element}[i] = Alg_{element}[i]$$

until $i = k$,then the $Alg_{element}[k]$ finish earlier than the $Opt_{element}[k]$.

Then if we replace the $Opt_{element}[k]$ with $Alg_{element}[k]$ in the $Opt_{element}$,the result is no worse than the previous one.

So the k should keep plus one until the k = n + 1.Namely the k doesn't exist.

Thus,the

$$Opt_{element} = Alg_{element}$$

is correct.

And for any compatible subset,if it takes *element* as the first interval,then the result is no better than the $Opt_{element}$.

Above all,to choose the overall best solution,we only need to choose from the

$$Opt_{element} \; for \; element \; that \; traverses \; interval \; set$$

Because $Opt_{element} = Alg_{element}$,simply choosing the best one from the $Alg_{element}$ is OK.

Thus,the correctness prove of my greedy algorithm is completed.

Note that in my pseudo-code,the m is no bigger than $2n$,so the overall time complexity is

$$O(2n * n)$$

It's

$$O(n^2)$$

***Here comes a $O(n \log n)$ solution.**

I claim here,that the algorithm is not one hundred percent right for I may think wrongly on some aspects that I didn't figure out,though I can prove it from my perscpective.I write it for exercising myself.

At first,we give a definition of a struct

$$node = \begin{cases} finish\ time, \\ index, \\ last\ index, \\ previous\ index, \\ last\ previous\ index, \\ count \end{cases}$$

Algorithm 5: Find the the maximum compatible subset of A

```
    Input: $A = [s_1, s_2, \dots, s_n]$
    Output: the maximum compatible subset of A
 1  $S \leftarrow empty$;
 2  for $i \leftarrow 1$ to $n$ do
 3      $[a_i, b_i] = s_i$;
 4      if $a_i \le b_i$ then
 5          $S = S \cup [a_i, b_i]$;
 6          $a_i \leftarrow a_i + k$;$b_i \leftarrow b_i + k$;
 7          $S = S \cup [a_i, b_i]$;
 8      else
 9          $a_i \leftarrow a_i + k$;
10          $S = S \cup [a_i, b_i]$;
11  $m \leftarrow sizeof(S)$;
12  answer $\leftarrow 0$;
13  sort $S[\cdot]$ by the finish time;
14  pq denote a priority queue which contains nodes,and compares elements by the relation $<$
      of the $finish\ time$ variable in the node struct.
15  $index \leftarrow m$;
16  for $i \leftarrow m$ to 1 do
17      $[a_i, b_i] = s_i$;
18      while $index >= 1$ and $index > i$ and $b_i < s[index].a_i$ do
19          new element $\leftarrow$ nodes[index];
20          $index \leftarrow index - 1$;
21          pq.push(new element);
22      $element \leftarrow pq.top()$
23      now count $\leftarrow$ $element's\ count$;
24      while element's last index != 0 and element's last index's start time - $k \in [a_i, b_i]$ or
          element's last index's finish time - $k \in [a_i, b_i]$ do
25          now count $\leftarrow$ nowcount - 1;
26          element's last index $\leftarrow$ element's last previous index
27          element's last previous index $\leftarrow$ nodes[element's last previous]'s previous index.
28      if nodes[element's index]'s previous index = 0 or $a_i <$ nodes[element's index]'s
          previous index then
29          nodes[element's index]'s previous index = i;//change the previous element
30
          $$new\ element = \begin{cases} finish\ time = b_i, \\ index = i, \\ last\ index = element's\ last\ index, \\ last\ previous\ index = element's\ last\ previous\ index, \\ count = new\ count\ +\ 1 \end{cases}$$
31      answer $\leftarrow$ max(count,answer);
32      nodes[i] $\leftarrow$ new element
33  return answer; //If the concrete container is needed,use a new array to record
```

*整体思路，从后往前记，前驱只会有一个（即开始时间最小的那个，这是关键）, 两个while和

一个for是独立的，上界O(n),对pq的操作在$O(n \log n)$，排序是$O(n \log n)$
*所以时间复杂度O($n \log n$). $\square$