CS222 Homework 2

Exercises for Algorithm Design and Analysis by Li Jiang, 2016 Autumn Semester

5140309507 林禹臣 yuchenlin@sjtu.edu.cn

1. There are two sorted arrays nums1 and nums2 of size m and n respectively.

   Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

   Example 1:

   nums1 = [1, 3]

   nums2 = [2]

   The median is 2.0

   Example 2:

   nums1 = [1, 2]

   nums2 = [3, 4]

   The median is (2 + 3)/2 = 2.5

   Input:

   int nums1[]; int m;

   int nums2[]; int n;

   Output:

   double median.

   Solution. My solution here is based on a divide-and-conquer idea so that it can be a $O(log(m+n))$ algorithm.

   The most naive intuition for solving this problem is first to merge sort all m+n elements and then calculate the median. But this will cause an $O(m+n)$ time complexity. The reason why this intuition is a bad idea is that it does not consider very much about how to avoid many unnecessary comparison.

   We now consider a more general problem: *find the k-th smallest element of the union of this two sorted arrays.*

   Proposition: If $a+b=k(a,b>0)$ and $nums1[a-1]<nums2[b-1]$, we can say that the k-th smallest element (T) must be not in $nums1[0:a]$.

   By contradiction, if T is in nums1[0:a], then $nums1[a-1]=T$, since $a<k$. Also, because $a+b=k$, T is the maximum value of $\{nums1[0:a-1],\ nums2[0:b-1]\}$, and thus $\underline{T>=nums2[b-1]}$. However, it causes a conflict that $\underline{T=nums1[a-1]<nums2[b-1]}$.

   Algorithm 1 and Algorithm 2 show the process. Evidently, the time complexity is $O(log(\frac{m+n}{2}))$.

   $\square$

Algorithm 1: $findK$: Find the k-th smallest element of the union of the two sorted arrays.

Input: int $nums1[]$, int m, int $nums2[]$, int n, int k

Output: double k-th smallest element

// ensure that $m \leq n$

if $m > n$ **then**

  return *findK(* $nums2[:]$, n, $nums1[:]$, m, k *);*

// two base cases

if $m == 0$ **then**

  return $nums2[k-1]$;

if $k == 1$ **then**

  return $min\{nums1[0], nums2[0]\}$;

// ensure that $index\_1 + index\_2 == k$ and $index\_2 >= index\_1$

$index\_1 = min\{k/2, m\}$;

$index\_2 = k - index\_1$;

// reduce the problem to a half-smaller one

if $nums1[index\_1 - 1] < nums2[index\_2 - 1]$ **then**

  return *findK(* $nums1[index\_1 :]$, $m - index\_1$, $nums2[:]$, n, $index\_2$ *);*

else if $nums1[index\_1 - 1] > nums2[index\_2 - 1]$ **then**

  return *findK(* $nums1[:]$, m, $nums2[index\_2 :]$, $n - index\_2$, $index\_1$ *);*

else

  return $nums1[index\_1]$

Algorithm 2: $findMedian$: Find the median of the union of the two sorted arrays.

Input: int $nums1[]$,int m, int $nums2[]$,int n

Output: double the median

$all = m + n$; **if** *all is even* **then**

  return $findK($ $nums1$, m, $nums2$, n $, (all/2) + 1)$;

else

  return ($findK($ $nums1$, m, $nums2$, n $, all/2) +$
  $findK($ $nums1$, m, $nums2$, n $, (all/2) + 1)$) / 2

2. Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

   For example, given the array [-2,1,-3,4,-1,2,1,-5,4], the contiguous subarray [4,-1,2,1] has the largest sum = 6.

   Input:

   int A[]: the input array.

   int N: length of A.

   Output:

   return the largest sum.

   Solution. I have two solutions here. The first one is using Divide and Conquer strategy, which gives us an $O(nlgn)$ time complexity. While the other is an online solution, which means its time complexity is just $O(n)$.

   Solution 1: Divide and Conquer

   As usual, we have 3 steps when we are using this strategy. Say we are computing the largest sum from x to y of array $A[]$.

   First, we compute the large sum of its left half part by calling this function with $A, x, \frac{(x+y)}{2} - 1$. Suppose the result is L. Second, we do the same thing to the right half part by calling the function with $A, \frac{(x+y)}{2}, y$ and say the result is R. Third, we find the largest sum of the sub-array including the middle node. Say the result is M.

   Then we can return the maximum of L,R and M. See Algorithm 3. We can simply call $getLargestSum(A,\ 0,\ N)$ and get the result. The time complexity of this solution is evidently $O(nlgn)$ since $T(n) = 2T(\frac{n}{2}) + O(n)$

Algorithm 3: *getLargestSum*: Find the largest sum of a contiguous sub-array of $A[x : y)$.

Input: int $A[]$,int x, int y
Output: int the largest sum
if $y - x == 1$ **then**
  return A[x];
$mid = [(x + y)/2]$;
$L = getLargestSum(A,\ x,\ mid)$;
$R = getLargestSum(A,\ mid,\ y)$;
$tmp = 0$;
$Ml = 0$;
for i *from* $mid - 1$ *to* x **do**
  $tmp+ = A[i]$;
  $Ml = max\{Ml,\ tmp\}$
$tmp = 0$;
$Mr = 0$;
for i *from* mid *to* $y - 1$ **do**
  $tmp+ = A[i]$;
  $Mr = max\{Mr,\ tmp\}$;
$M = Ml + Mr$;
return $max\{L,\ R\ , M\}$;

Solution 2: Online Processing Algorithm

Actually, we have an intuition that if we calculate the sum from the first node to the last one, we can reset the sum to 0 every time when our current sum becomes negative. It is because that if we go on, then we can say why not we just keep the next one(say P)? No matter what number P is, P must be greater than the $currentSum + P$ when $currentSum$ is negative. Thus, we can have a most simple and most efficient($O(n)$) algorithm. See Algorithm 4.

Algorithm 4: *getLargestSum*: Find the largest sum of a contiguous sub-array of $A[0 : N)$.

Input: int A[],int N

Output: int the largest sum

$currentSum = 0$;

$maxSum = 0$;

for $i \in [\ 0, N\ )$ **do**

  $currentSum += A[i]$;

  $maxSum = max\{maxSum,\ currentSum\}$;

  if $currentSum < 0$ **then**

    $currentSum = 0$;

return $maxSum$;

□

3. Given a non-empty array containing only positive integers, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

   Note:

   Each of the array element will not exceed 100.

   The array size will not exceed 200.

   Example 1:

   Input: [1, 5, 11, 5]

   Output: true

   Explanation: The array can be partitioned as [1, 5, 5] and [11].

   Example 2:

   Input: [1, 2, 3, 5]

   Output: false

   Explanation: The array cannot be partitioned into equal sum subsets.

   Input:

   int A[]: the input array.

   int N: length of A.

   Output:

   return true or false.

Solution. I cannot figure out a good Divide-and-Conquer Strategy for this problem. However, from my historical programming experience, this problem is a typical Dynamic Programming problem. Thus, I tried to implement a DP solution.
First, we must know that the sum of the array which is very useful. If the sum S is odd then we can simply return False. If S is even, then the problem would be transformed to a typical problem that whether we can select some certain elements from the array so that the sum of them is exactly $S/2$.
Using dynamic programming strategy means we need to define a *state* first. Here we say a state $f(i,j)$ means if our bag's capacity is j kg and we can select from i items then we can at most fill the bag with $f(i,j)$ kg. (In this sense, $A[i]$ means the weight of item i.)
Thus, we can just return whether $f(N, S/2) == S/2$; If it can be, then that is to say we can select some certain items from the all N items and make their sum is exactly $S/2$.

Here comes the state transform function:
$f(0,0) = 0;$
$f(i,j) = max\{\ f(i-1,j),\ f(i-1, j-A[i]) + A[i]\};$

See Algorithm 5. Evidently, the time complexity is $O(S * N)$ where S is the sum of these numbers.

Algorithm 5: *judgeEqual*

Input: int $A[]$,int N
Output: boolean whether we can separate the set accordingly
S = the sum of all the elements;
if S *is odd* **then**
  return $False$;
$f = new\ int[N][S/2+1]$;
for $i \in [1,\ n)$ **do**
  for $j \in [A[i],\ S/2]$ **do**
    $f(i,j) = max\{\ f(i-1,j),\ f(i-1, j-A[i]) + A[i]\}$;
return $f(N, S/2) == S/2$;

Another more simple and efficient algorithm is to consider this problem as a typical 0-1 bagging problem and use bit-set to solve the problem. We define the state $f(i) = 1$ means that i can be a sum of a certain subset of original set. According to the description, we can know that the largest sum will be $200 * 100 = 40000$, and thus the target will be up to $20000 + 1$. Then our goal is to find whether f(target)==1 or not.
We can use left-shift as a plus on all states and use bit-or to save all the changes.
For example, $1, 2, 5, 2$, firstly the bitset will be 000001,and then when we consider 1, we plus 1 on all the positive numbers and save the changes, so the bitset becomes 000011, and then 001111,101111(which means 5 can be generated and we can stop here.). See Algorithm 6. If we consider left-shift can be a basic operation, then the time complexity will be $O(n)$;

□

Algorithm 6: *judgeEqualWithBitset*

Input: int $A[]$,int N

Output: boolean whether we can separate the set accordingly

$S =$ the sum of all the elements;
if S *is odd* **then**
  return $False$;
$f = bitset(200001)$;
$f(0) = 1$;
for $i \in [0,\ n)$ **do**
  $f = f \mid (f << A[i])$;
return $f(S/2) == 1$;