Algorithm Design - Array

Nicolò Felicioni[1]

Dipartimento di Elettronica e Informazione
Politecnico di Milano

nicolo . felicioni @ polimi . it

May 25, 2021

[1]Based on Nicholas Mainardi's material.

Exact-Sum

Exact Sum Problem

Given an array and a value x, find if there are 2 elements $a[i], a[j]$ such that $a[i] + a[j] = x$

- Naive idea: check all the possible pairs to see if they sum up to x
- Complexity is $O(n^2)$

An Alternative Strategy

- When problems are slower that $O(n \log(n))$, we can always try to observe how to solve the same problem on a sorted array
- If we can solve it with at most $O(n \log(n))$ complexity, we manage to improve the solution
- The general idea is that sorted array are more structured, thus we can employ this property to design better algorithms

Exact-Sum

Exact-Sum on a Sorted Array

- Let's identify the property introduced by a sorted array
- If $a[i] + a[j] > x \Rightarrow \forall z \geq j(a[i] + a[z] > x)$, and the same for i
- Conversely, if $a[i] + a[j] < x \Rightarrow \forall z \leq i(a[z] + a[j] < x)$, and the same for j
- These 2 properties means that:
  - if $a[i] + a[j] < x$, then we need to increase either i or j
  - if $a[i] + a[j] > x$, then we need to decrease either i or j
- Which one of these indexes shall we modify?
- Intuition: it would be nice if one of them is always increased while the other one is always decreased, since there is no arbitrary choice
- For instance, if $i = 1$ and $j = n$, then we can either increase i or decrease j, depending on the outcome of $a[1] + a[n] > x$

Exact-Sum

Exact-Sum on a Sorted Array

- Can we extend this idea to different pairs of i, j?

```
EXACT-SUM$(A, x)$
1  $i \leftarrow 1$
2  $j \leftarrow A.length$
3  while $i < j$
4      do if $A[i] + A[j] = x$
5          then return $True$
6          else if $A[i] + A[j] > x$
7              then $j \leftarrow j - 1$
8              else $i \leftarrow i + 1$
9  return $False$
```

- Why does it work? During the first iteration, we can immediately discard either i or j depending on the sum

Exact-Sum

Exact-Sum on a Sorted Array

- At the next iteration, we can still discard either i or j because the same properties still holds! Let's look at all the cases:
  1. $i = 2$, $j = n$ and $a[2] + a[n] < x$. Then, we can immediately discard $i = 2$, since $\forall z \leq n(a[2] + a[z] < x)$. Thus, we increase i
  2. $i = 2$, $j = n$ and $a[2] + a[n] > x$. Then, we can discard $j = n$, since $\forall z \geq 2(a[z] + a[n] > x)$, while $a[1] + a[n] < x$, otherwise we would not be in this case. Thus, we decrease j
  3. $i = 1$, $j = n - 1$ and $a[1] + a[n-1] < x$. Then, we can discard $i = 1$, since $\forall z \leq n - 1(a[1] + a[z] < x)$, while $a[1] + a[n] > x$, otherwise we would not be in this case. Thus, we increase i
  4. $i = 1$, $j = n - 1$ and $a[1] + a[n-1] > x$. Then, we can discard $j = n - 1$, since $\forall z \geq 1(a[z] + a[n-1] > x)$. Thus, we decrease j

Exact-Sum

Exact-Sum on a Sorted Array

- In conclusion, it seems that at each iteration, all $z < i$ and all $t > j$ have already been discarded, thus leaving only one possible choice.
- We already see that this is true in the first 2 iterations, let's generalize it via an inductive proof
- There are 2 cases for the inductive step:
  1. $a[i] + a[j] > x$. In this case, we know that $\forall z \geq i(a[z] + a[j] > x)$, while all $z < i$ have already been discarded, thus we know $\neg\exists z < i(a[z] + a[j] = x)$. Therefore, we can discard the current value j, and decrease it by 1
  2. $a[i] + a[j] < x$. In this case, we know that $\forall z \leq j(a[i] + a[z] < x)$, while all $z > j$ have already been discarded, thus we know $\neg\exists z > j(a[i] + a[z] = x)$. Therefore, we can discard the current value i, and increase it by 1

Exact-Sum

Exact-Sum on a Sorted Array

- In conclusion, at each iteration we can discard either i or j depending on the outcome of the comparison, thus the algorithm works
- Time complexity? $O(n)$, since i and j come closer by 1 at each iteration, starting from a distance of $n - 1$

Complete Exact-Sum

- Since we need to perform sorting before being able to run this algorithm, the time complexity of the whole EXACT-SUM is $O(n \log(n))$

Sorting Arrays on Attributes

Problem

We have an array where each element has a color (red, green and blue) and some additional data. We want to sort, with $O(k)$ space complexity, this array according to the color of its elements, where red $<$ green $<$ blue. The algorithm may be non-adaptive, i.e., order of red, green and blue elements may not be preserved

- We can use every sorting algorithm with $O(1)$ space complexity (e.g INSERTION-SORT)
- With SHELL-SORT we can do it in $O(n^{\frac{3}{2}})$
- Can we do it better?
- Well, the domain of the attributes to be used for sorting is finite, thus we can use an algorithm like COUNTING-SORT

Sorting Arrays on Attributes

Algorithm Design

- We cannot use COUNTING-SORT as is. Indeed:
  - Either it uses an additional array to write the results
  - Or since each bucket used to count occurrences refers to a particular value, it simply writes this value on the original array for the number of times written in the bucket $\Rightarrow$ since there are additional data attached to the attributes referred by buckets, we cannot know these data when writing the elements, but we need to copy them from the array elements
- Therefore, we need to exploit information of the occurrences counters to overwrite the array in place, including additional data which cannot be inferred by the bucket itself
- What information in each bucket?

Sorting Arrays on Attributes

Algorithm Design

- Each bucket stores the position where we can write an element having a sorting attribute equal to the one referred by the bucket (e.g if red bucket is 3, we can write a red element in slot 3)
- Thus, we know the final position of the element $\Rightarrow$ same writing mechanism of CYCLE-SORT
- Thus, the algorithm first counts the occurrences for each color, determining the positions of the array where we can write a given colored element
- Then, it places each element in its color writing position, decrementing it for that color after the insertion
- The next element to be placed is the one being overwritten by previous iteration

Sorting Arrays on Attributes

Pseudocode

```
COLORING-SORT($A$)
 1  $Colors[3] \leftarrow [0, 0, 0]$ ▷ Store occurrences of each color
 2  for $i \leftarrow 1$ to $A.length$
 3      do if $A[i].color = red$
 4          then $Colors[1] \leftarrow Colors[1] + 1$
 5          else if $A[i].color = green$
 6              then $Colors[2] \leftarrow Colors[2] + 1$
 7              else if $A[i].color = blue$
 8                  then $Colors[3] \leftarrow Colors[3] + 1$
 9  $Colors[2] \leftarrow Colors[2] + Colors[1]$
10  $Colors[3] \leftarrow Colors[3] + Colors[2]$
11  $Start[3] \leftarrow [1, Colors[1] + 1, Colors[2] + 1]$ ▷ Store the first available position for each color, used later on
12  for $i \leftarrow 1$ to 3
13      do $item \leftarrow A[Start[i]]$
14        while $Colors[i] \geq Start[i]$ ▷ Avoid infinite loop on writing A[start[i]]
15            do if $item.color = red$
16                then SWAP($item$, $A[Colors[1]]$)
17                    $Colors[1] \leftarrow Colors[1] - 1$
18                else if $item.color = green$
19                    then SWAP($item$, $A[Colors[2]]$)
20                        $Colors[2] \leftarrow Colors[2] - 1$
21                    else if $item.color = blue$
22                        then SWAP($item$, $A[Colors[3]]$)
23                            $Colors[3] \leftarrow Colors[3] - 1$
```

Time complexity? $O(n)$, since each element is written only once to its final position, as in CYCLE-SORT

Equal Sum Partitioning Problem

Problem

Given an array, we want to figure out if it can be partitioned in a set of pairs all having the same sum

- Even the naive solution seems not so trivial
- Computing all possible partitions is not that easy by itself!

Reasoning on the problem

- First of all, the array can be partitioned in pairs only if its length is even
- If the length is odd, we can already provide a negative answer
- Similarly, if the sum of the elements of the array is not a multiple of $\frac{n}{2}$, then the partition cannot exists
- Does a sorted array help?

Equal Sum Partitioning Problem

Equal Sum Partitioning Problem on a Sorted Array

- Suppose the array can be partitioned.
- Consider $A[1]$ and its paired element j: $A[1]$ is the minimum element of the array $\Rightarrow A[j]$ is the maximum $\Rightarrow j = A.length$
  - Indeed, if $A[j]$ is not the maximum, then it means that there is another element $A[h] > A[j]$, which is paired with another element $A[k] \geq A[i] \Rightarrow$ impossible that $A[i] + A[j] = A[k] + A[h]$
- By considering the sub-array $A[2 \ldots N-1]$, with the same line of reasoning we must derive that $A[2]$ is paired with $A[A.length - 1]$ in a sorted array, and so on
- Therefore, in a sorted array, there is only one possible partition which needs to be tested, the one made by pairs $A[i], A[A.length + 1 - i]$

Equal Sum Partitioning Problem

Algorithm Design

- First, we check that the length is even and the sum is a multiple of $\frac{n}{2}$
- Then, we sort the array
- Eventually, we check if all the pairs $A[i], A[n-i+1]$ have the same sum

Complexity Analysis

- Checking the pairs can be done in $O(n)$
- Thus the complexity is dominated by sorting $\Rightarrow O(n\log(n))$

Equal Sum Partitioning Problem

A Variant

Suppose we want to partition the array in set of 3 elements, instead of 2. As an additional constraint, if the partition exists, then each element of the array belongs exactly to a unique triple whose sum is the target value[2]. How does the algorithm change?

- We lose the nice property observed on a sorted array for pairs
- Therefore, we need another way to provide an answer
- Again, the array can be partitioned in set of 3 elements only if its length n is a multiple of 3
- Moreover, the array can be partitioned in set of 3 elements only if the sum of the elements is a multiple of $\frac{n}{3}$!
- Therefore, we can easily compute the value of the sum of each set in the partition. How?

[2]Otherwise we get the 3-partition problem, which is NP-complete

Equal Sum Partitioning Problem

Determining the sum of the sets

- Compute the sum of all the elements of the array S
- Verify if S is divisible by $\frac{n}{3}$
- In this case, the sum of each set is exactly $s = \frac{S}{\frac{n}{3}} = \frac{3S}{n}$

Algorithm Design

- How can we use the computed value s to find the partitions?
- Suppose we choose an element $A[i]$. In order to find a set, it is sufficient to find a pair $A[j], A[z]$ s.t. $A[i] + A[j] + A[z] = s \Rightarrow A[j] + A[z] = s - A[i]$
- But $s - A[i]$, since we choose $A[i]$, is a known value, thus we need to find a pair which sums up to a known value $s - A[i]$. Which algorithm do we know for this?

Equal Sum Partitioning Problem

Algorithm Design

- $\text{EXACT-SUM}(A, x)$! In particular, we know a quite efficient linear version which works on a sorted array
- Thus, we sort the array and then we exploit this algorithm: we know that the solution provided is the right one because it is unique (recall additional constraint in the problem statement)
- However, we need to find all the sets, not just one, thus different calls to EXACT-SUM for each chosen $A[i]$
- Therefore, if EXACT-SUM successfully returns a pair, we need to erase it from the array altogether with the chosen element $A[i]$, to avoid overlapping among identified sets
- Erasure can be performed by moving the identified triple to the beginning of the array and shift, not swap, all the others to preserve sorting property of the remaining elements

Equal Sum Partitioning Problem

Pseudocode

```
3-SUM-PARTITIONING($A$)
if $A.length$ mod $3 \neq 0$
   then return $False$
$S \leftarrow 0$
for $i \leftarrow 1$ to $A.length$
      do $S \leftarrow S + A[i]$
if $S$ mod $\frac{A.length}{3} \neq 0$
   then return $False$
$s \leftarrow \frac{3S}{A.length}$
MERGE-SORT($A$)
for $start \leftarrow 1$ to $A.length$ by 3
      do $(found, j, z) \leftarrow$ EXACT-SUM($A[start+1, \ldots, A.length], s - A[start]$)
         if $!found$
            then return $False$
         $j \leftarrow j + start + 1$
         $z \leftarrow z + start + 1$
         $el_j \leftarrow A[j]$
         $el_z \leftarrow A[z]$
         $offset \leftarrow 0$
         for $i \leftarrow A.length$ downto $start + 1 + offset$
               do if $i - offset = z \vee i - offset = j$
                     then $offset \leftarrow offset + 1$
                  $A[i] \leftarrow A[i - offset]$
         $A[start+1] \leftarrow el_j$
         $A[start+2] \leftarrow el_z$
return $True$
```

Equal Sum Partitioning Problem

Complexity Analysis

- Computing S costs $O(n)$
- Then, MERGE-SORT costs $O(n\log(n))$
- The **for** cycle at line 10 is performed $\frac{n}{3}$ times at most (when the array can be partitioned)
- Each EXACT-SUM costs $O(n-3i)$ on a sorted array
- The **for** cycle at lines 19 – 22 costs $O(n-3i)$
- Thus, the outer **for** body is $O(n-3i)$, repeated by $\frac{n}{3}$ times $\Rightarrow \sum_{i=1}^{\frac{n}{3}} n-3i \Rightarrow O(n^2)$
- In conclusion, the complexity is dominated by the outer **for** loop, thus it is $O(n^2)$

Number of Inversions

Number of Inversions Problem

Given an array, we want to compute the number of inversions, that is the pairs i, j where $i < j$ and $a[i] > a[j]$.

- Naive idea: check all the possible pairs and count the ones which are an inversion
- Complexity is $O(n^2)$

Trying to Improve

- Can we think about the same problem on a sorted array?
- On a sorted array there are no inversions!
- We cannot sort the array, since the problem depends on the position of the elements in the array, but ...

Number of Inversions

Algorithm Design

- There is for sure a relation with sorting
  - A sorted array has 0 inversions
  - Conversely, a reversely sorted array has $\frac{n(n+1)}{2}$ inversions, since for an element in position i, all $a[j], j > i$ are smaller than $a[i]$
- A possible solution: what if the number of inversions for an element is the difference between the indexes in the sorted array and in the original one (if this difference is > 0)?
- Consider $a = [99, 4, 88, 7, 5, -3, 1, 34, 11]$. In the sorted array, 4 is in position 3, thus according to our idea the number of inversions should be $3 - 2 = 1$. But ...
- The number of inversions is 2, since $4 > -3$ and $4 > 1$
- In this example, the problem resides in 99 being moved after 4 in sorted array, and this move "undoes" one inversion of 4

Number of Inversions

Algorithm Design

- However, the same idea works on a particular instance of the problem: when the array is split into 2 sorted sub-arrays!
- Indeed, consider separately elements of the first and the second sub-array:
    - for elements in the first sub-array, there are no elements on their left which are moved on their right by sorting, since elements on their left are smaller. Therefore, the difference between their positions in the sorted and unsorted arrays is given only by elements moved on their left because they are smaller, i.e. the inversions
    - for elements in the second sub-array, there are no inversions, since all the elements on their right are bigger, thus the difference between their positions in the sorted and unsorted arrays is always negative, in turn yielding a number of inversions equal to 0

Number of Inversions

Algorithm Design

- How to exploit this particular case for the whole algorithm?
- On a sub-array with 2 elements, the inversions can be computed with one comparison, and sorted accordingly
- If we consider 2 of these sub-arrays, we get a sub-array split in 2 sorted sub-arrays $\Rightarrow$ we can apply our algorithm on it!
- Each of the inversions computed at each level represents a pair of elements which are in the wrong order in the unsorted array $\Rightarrow$ the total number of inversions is the sum of all the inversions computed on each sub-array, until we get to the whole array being sorted
- Wait! Doesn't it remind a well known sorting algorithm?

Number of Inversions

Algorithm Design

- It is the same divide-et-impera scheme of $\mathrm{MERGE\text{-}SORT}$!
- Basically, we are sorting with this algorithm and we count the number of inversions at each MERGE operation by computing differences between the indexes in sorted and unsorted sub-arrays!
- How to modify MERGE? Consider A to be the sub-array to be merged, we build an array B s.t. $\forall i \leq A.length(B[i] = i)$
- Then,all the swaps performed by MERGE on A are mirrored on $B \Rightarrow$ at the end of MERGE, $B[i]$ stores the index of $A[i]$ in the unsorted sub-array
- Thus, the inversions in each MERGE can be counted as $\sum_{i=1}^{B.length} \mathrm{MAX}(0, i - B[i])$
- Thus, same complexities: $O(n\log(n))$ time, $O(n)$ space

Maximum Sub-array

Problem

Given an array, we want to find the sub-array which maximizes the sum of its elements. For instance, if $a = [192, 169, 130, 128, -450, -340, 1280, 340, -34]$, the sub-array $[1280, 340]$ has the maximum sum of its elements

- Naive solution is not so efficient ...
- How many possible sub-arrays? $\Rightarrow O(n^2)$

Does Sorting Help?

- We cannot sort since we lose the position of elements, and thus we cannot correctly identify sub-arrays
- No benefit from sorting!

Maximum Sub-Array

Algorithm Design

- Why is the problem difficult?
- Because of negative values!
- Otherwise, the maximum sub-array is just the array itself!
- Thus, as soon as there are no negative values in a sub-array, no need to test for its sub-arrays since they are surely not the maximum ones
- How to handle negative values? We may split the array in positive sub-arrays and check which one has the maximum sum
- Does it work?

Maximum Sub-Array

Algorithm Design - 2

- Consider $a = [3, 2, 4, 10, -12, 9, 5]$
- With our strategy, the maximum sub-array is $[3, 2, 4, 10]$, whose sum equal to 19
- However, the sum of all elements of the array is 21!
- We cannot simply discard negative values
- Basically, if a sub-array is preceded by a sub-array with a positive sum, surely their concatenation has a higher sum than it
- Conversely, if a sub-array is followed by a sub-array with a positive sum, surely their concatenation has a higher sum than it
- Let's put this idea in an iterative version on each element ...

Maximum Sub-Array

Algorithm Design - 3

- Idea for the algorithm: iterate through the array and add each element to the sum of the current sub-array until this sum is positive
- Until we get a positive sum, it is worth to merge this sub-array with following ones instead of splitting them
- If the sum becomes negative, it is zeroed and its computation starts again from the next element (indeed if the sum becomes negative the current element is necessarily negative)
- Computing the maximum among the sums encountered during the iterations on elements of the array allows us to identify the maximum sub-array
- In this way, if the suffix of a sub-array decreases its sum, this suffix is not appended to the sub-array, maximizing the sum

Maximum Sub-Array

Pseudocode

```
MAXIMUM-SUBARRAY($A$)
1   $max_{sum} \leftarrow curr_{sum} \leftarrow A[1]$
2   $max_{begin} \leftarrow curr_{begin} \leftarrow max_{end} \leftarrow 1$
3   if $curr_{sum} \leq 0$
4       then $curr_{sum} \leftarrow 0$
5            $curr_{begin} \leftarrow 2$
6   for $i \leftarrow 2$ to $A.length$
7       do $curr_{sum} \leftarrow curr_{sum} + A[i]$
8          if $curr_{sum} > max_{sum}$
9             then $max_{sum} \leftarrow curr_{sum}$
10                 $max_{begin} \leftarrow curr_{begin}$
11                 $max_{end} \leftarrow i$
12         if $curr_{sum} \leq 0$
13            then $curr_{sum} \leftarrow 0$
14                 $curr_{begin} \leftarrow i + 1$
15  return $(max_{begin}, max_{end}, max_{sum})$
```