# Adv Data Struct & Algorithm: Homework 2

Zimian Li

April 25, 2018

1. A GRE combined score is an iteger between 260 and 340. You are given a sorted array with the scores of all the n students that took the exam in 2017. Design and analyze a sublinear (i.e., in o(n)) algorithm that reports the number of people that received each score x, for $260 \leq x \leq 340$.

   The algorithm is as below. FindUpper is like binary search, thus apparently the time complexity of function FindUpper is $\Theta(log(n))$. The function CountScores executes 80 times of FindUpper, so its time complexity is also $\Theta(log(n))$.
   I also compared it with a naive brute force algorithm with $\Theta(n)$ time cost. I just counted the times of loops in these 2 algorithms. As it shows below, when n is small, the brute force algorithm has a smaller number of loops and runs faster. That is because CountScores has a big constant factor. When n is big enough, it's easy to find the time cost of my algorithm grows much slower than the brute force one.

```
--------------------------------Problem 1--------------------------------
Input size:  10
Loops: my algorithm -  199 , brute force -  10
Time: my algorithm -  0.09272 ms , brute force -  0.00246 ms
------------------------------------------------------------------------
Input size:  1000
Loops: my algorithm -  640 , brute force -  1000
Time: my algorithm -  0.41436 ms , brute force -  0.18092 ms
------------------------------------------------------------------------
Input size:  100000
Loops: my algorithm -  1164 , brute force -  100000
Time: my algorithm -  0.83898 ms , brute force -  21.86915 ms
------------------------------------------------------------------------
```

Figure 1: (p1)

**Algorithm 1** Count Scores

```
1: procedure FindUpper(A, x, p, r)
2:     mid = (p+r)/2
3:     if p == r then
4:         if A[mid] == x then
5:             return mid
6:         else
7:             return None
8:         end if
9:     else
10:        if A[mid] == x and A[mid+1] > A[mid] then
11:            return mid
12:        end if
13:        if A[mid] > x then
14:            return FindUpper(A, x, p, mid-1)
15:        end if
16:        if (A[mid] < x) or ((A[mid] == x) and (A[mid+1] == x) then
17:            return FindUpper(A, x, mid+1, r)
18:        end if
19:    end if
20: end procedure
21: procedure CountScores(A)
22:    initialize a hashmap Result
23:    lower = 0, upper = 0
24:    for x ←260 to 340 do
25:        upper = FindUpper(A, x, lower, len(A)-1)
26:        if upper == None then
27:            Result[x] = 0
28:        else
29:            Result[x] = upper - lower + 1
30:            lower = upper + 1
31:        end if
32:    end for
33:    return Result
34: end procedure
```

2. Let S be a set of n points in $\mathbb{R}^2$. The bounding box of S is the smallest upright rectangle containing S.

   (a) Describe and analyze a divide-and-conquer algorithm to compute the bounding box of n points in the plane using fewer than 3n element to element comparisons. Make sure that your solution handles correctly all values of n.

**Algorithm 2** Find the bounding box

```
1: procedure FindBoundingBox(A, p, r)
2:     minX, maxX, minY, maxY = 0
3:     if r - p == 1 then
4:         if A[p][0] < A[r][0] then
5:             minX = A[p][0]
6:             maxX = A[r][0]
7:         else
8:             minX = A[r][0]
9:             maxX = A[p][0]
10:        end if
11:        if A[p][1] < A[r][1] then
12:            minY = A[p][1]
13:            maxY = A[r][1]
14:        else
15:            minY = A[r][1]
16:            maxY = A[p][1]
17:        end if
18:        return minX, maxX, minY, maxY
19:    end if
20:    if p == r then
21:        return A[p][0], A[p][1], A[p][0], A[p][1]
22:    end if
23:    mid = (p+r)/2
24:    if mid is even then
25:        mid += 1
26:    end if
27:    minX, maxX, minY, maxY = FindBoundingBox(A, p, mid)
28:    minX1, maxX1, minY1, maxY1 = FindBoundingBox(A, mid+1, r)
29:    minX = min(minX, minX1)
30:    minY = min(minY, minY1)
31:    maxX = max(maxX, maxX1)
32:    maxY = max(maxY, maxY1)
33:    return minX, maxX, minY, maxY
34: end procedure
```

Actually, for an array with $n$ points $\{(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)\}$, finding an upright bounding box is equal to find the minimal and maximal values of $x$ and $y$.

The number of comparisons of my algorithm depends on the parity of $n$. If $n$ is even, then the number of comparisons is $3n - 4$, and if $n$ is odd, it's $3n - 3$, both fewer than $3n$.

I also compared it with a naive brute force algorithm as below. The brute force one takes around $4n$ times of comparisons.

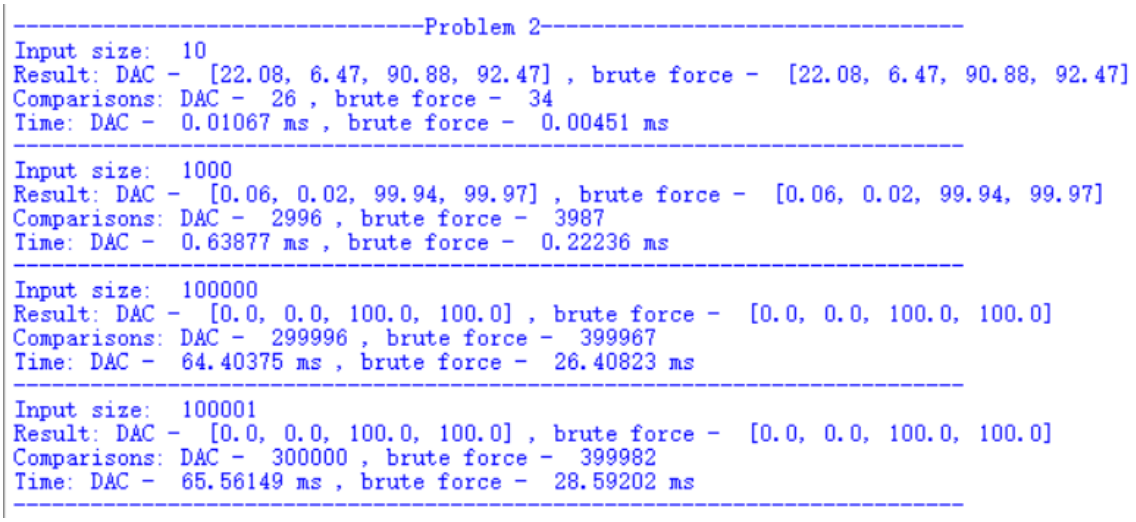


Figure 2: (p2)

(b) Can an incremental approach based on adding one point at a time improve or match the performance? How about adding several points at a time? Explain.

No, it can't. If we add one point at a time, this point would have to compare with current min and max values of x and y, so at last it need less than $4n$ times of comparisons. (But on the other hand in best case it only need $2n$ times of comparisons.)
But if adding several points at a time, it can match the performance, actually, if adding 2 points at a time, it could be same as the DAC algorithm, these 2 points compares first, then separately compares with min and max values of x and y. And 2 points one time is the best case, if adding more than 2 points at a time, we can't get a fewer times of comparisons.

3. You are given a list A of n computational tasks that can be executed in parallel. Each task is specified by the amount of time (an integer) required to execute the task. Your goal is to assign a list of consecutive tasks to each of $p \leq n$ processors so as to minimize the total parallel processing time, i.e., the max over all p processors of the time required by each processor to solve its assigned tasks. For example, if A $= \{7, 4, 3, 5\}$ and p $= 3$ an optimal partition is $\{(7), (4, 3), (5)\}$ with a total parallel time of 7.

**Algorithm 3** Find the optimal partition

```
1: procedure minProc(A, t)
2:     initialize a list Procs
3:     cap = 0
4:     for i ← 0 to len(A) do
5:         cap += A[i]
6:         if i == len(A) - 1 then
7:             Procs.append(i)
8:         else
9:             if cap + A[i+1] > t then
10:                cap = 0
11:                Procs.append(i)
12:            end if
13:        end if
14:    end for
15:    return Procs
16: end procedure
17: procedure minTime(A, p)
18:    approx = max(ceil(sum(A)/p), the max value in A)
19:    procs = minProc(A, approx)
20:    while len(procs) > p do
21:        approx += 1
22:        procs = minProc(A, approx)
23:    end while
24:    return procs
25: end procedure
```

The first part of this algorithm is that given t to get a minimal p. It's a greedy algorithm, and the time complexity is $\Theta(n)$. However, the goal is to get a minimal t with a given p. So we need to find a lower bound of t and then iterate the first function many times until meeting the right p.

The time complexity of the whole algorithm is not easy to get. I can find a lower bound that is max(ceil(sum(A)/p), the max value in A). What about upper bound? It's easy to find $T(minTime(A,p)) \leq T(minTime(A,p-1)) \leq ... \leq T(minTime(A,1)) = sum(A)$. Hence, $T(minTime(A,p)) \leq (sum(A) - sum(A)/p) \cdot \Theta(n) \in O((p-1)/p \cdot sum(A) \cdot n)$. Other than $n$ and $p$, it's related with the content of the input array.

```
--------------------------------Problem 3--------------------------------
Input size: n =  10 , p =  4
Result: greedy -  [2, 6, 8, 9] , brute force -  [1, 6, 8, 9]
Loops: greedy -  280 , brute force -  161
Time: greedy -  0.09682 ms , brute force -  0.25641 ms
------------------------------------------------------------------------
Input size: n =  100 , p =  4
Result: greedy -  [25, 50, 77, 99] , brute force -  [25, 50, 77, 99]
Loops: greedy -  800 , brute force -  166646
Time: greedy -  0.20841 ms , brute force -  1406.34864 ms
------------------------------------------------------------------------
Input size: n =  100 , p =  5
Result: greedy -  [19, 38, 59, 79, 99] , brute force -  [19, 38, 59, 78, 99]
Loops: greedy -  1300 , brute force -  4082920
Time: greedy -  0.33231 ms , brute force -  33626.75046 ms
------------------------------------------------------------------------
```

Figure 3: (p3)

Compared with a brute force algorithm of which time complexity is $O(\binom{n-1}{p-1} \cdot n)$, when the input size gets bigger, it has a much better performance than the brute force one.

4. The problem of finding local maxima of a function $f : \mathbb{R} \to \mathbb{R}$ finds numerous applications in fields as varied as machine learning and anthropology. We are interested here in the discrete version of this problem, where $f$ is represented as an array of number, A location in A is a local maximum if its value is greater than or equal to that of its immediate neighbors. As an example, given an elevation raster of a terrain, an archaeologist may be interested in locating the location of mounds, as they may correspond to a promising excavation sites.

   (a) In the 1D version of this problem, you are given an array A of n numbers. A[k] is a local maximum if $A[k-1] \leq A[k]$ and $A[k] \geq A[k+1]$ (adjusting accordingly, if A[x] only has one neighbor). For example, the array (9, 7, 7, 2, 1, 3, 7, 5, 4, 7, 3, 3, 4) has four local maxima (at location 1, 3, 7, and 10).

      i. Prove that a local maximum must exist.
      **Proof:** Assume there's no local maximum in the array A. Then for any element in the array A except head and tail, it has to meet $A[k-1] < A[k] < A[k+1], or A[k-1] > A[k] > A[k+1], or A[k] < A[k-1] and A[k] < A[k+1]$. And for the head element $A[0]$, it has to meet $A[0] < A[1]$, then for $A[1]$, it has already greater than $A[0]$, so we can conclude $A[1] < A[2]$, the consider $A[2]$, and repeat this process to the end, then we can find for the tail $A[n-1]$, $A[n-1] > A[n-2]$, so $A[n-1]$ is a local maxima, that's a contradiction! Hence, a local maximum must exist.

      ii. Describe and analyze an agorithm that finds a local maximum in $o(n)$ time.

**Algorithm 4** Find a local maximum in 1D array

```
1: procedure FINDLOCALMAX1D(A, p, r)
2:     mid = (p+r)/2
3:     if mid == 0 and A[mid] ≥ A[mid+1] then
4:         return mid
5:     else
6:         if mid == len(A)-1 and A[mid] ≥ A[mid-1] then
7:             return mid
8:         end if
9:         if A[mid] ≥ A[mid-1] and A[mid] ≥ A[mid+1] then
10:            return mid
11:        end if
12:        if A[mid] <A[mid-1] then
13:            return findLocalMax1D(A, p, mid-1)
14:        end if
15:        return findLocalMax1D(A, mid+1, r)
16:    end if
17: end procedure
```

It's similar with binary search, takes $\Theta(logn)$ time.

```
-------------------------Problem 4 (1D cases)-------------------------
Input size:  10
Result: my algorithm -  0 , brute force -  0
Loops: my algorithm -  3 , brute force -  1
Time: my algorithm -  0.00369 ms , brute force -  0.00164 ms
----------------------------------------------------------------------
Input size:  1000
Result: my algorithm -  249 , brute force -  1
Loops: my algorithm -  2 , brute force -  2
Time: my algorithm -  0.00328 ms , brute force -  0.00287 ms
----------------------------------------------------------------------
Input size:  100000
Result: my algorithm -  1267 , brute force -  1
Loops: my algorithm -  10 , brute force -  2
Time: my algorithm -  0.0119 ms , brute force -  0.00492 ms
----------------------------------------------------------------------
```

Figure 4: (p41)

Compared with a naive brute force algorithm, I found the brute force algorithm is also fast, which is incredible! Actually it makes sense. Consider an array with n distinct numbers, for all position except head and tail, the possibility of that an arbitrary element is a local maximum is 1/3, for head and tail, it's 1/2. So if we search from the head, the expected number of search times is less than 3! If there are same numbers, it just decreases this expected value. Hence, in average case, the brute force algorithm is a $\Theta(1)$ algorithm, it's really fast. It's the same for my binary search. But in the worst case, the brute force algorithm would take $\Theta(n^2)$, not that much good.

(b) In 2D, you are given a $n \times n$ array A of numbers. Then A[i, j] is a local maimum if it is greater than or equal to its four neighbors: A[i - 1, j], A[i + 1, j], A[i, j - 1], A[i, j + 1]. Describe and analyze an algorithm that finds a local maimum in $o(n^2)$ time.

**Algorithm 5** Find a local maximum in 2D array

```
1: procedure findLocalMax2D(A, p, r)
2:     mid = (p+r)/2
3:     maxy = max(A[mid])
4:     if mid == 0 and A[mid][maxy] ≥ A[mid+1][maxy] then
5:         return mid, maxy
6:     else
7:         if mid == len(A)-1 and A[mid][maxy] ≥ A[mid-1][maxy] then
8:             return mid, maxy
9:         end if
10:        if A[mid][maxy] ≥ A[mid-1][maxy] and A[mid][maxy] ≥ A[mid+1][maxy] then
11:            return mid, maxy
12:        end if
13:        if A[mid][maxy] <A[mid-1][maxy] then
14:            return findLocalMax2D(A, p, mid-1)
15:        end if
16:        return findLocalMax2D(A, mid+1, r)
17:    end if
18: end procedure
```

For this algorithm, I think I need to prove its correctness.

**Proof of Correctness:** In this algorithm, It's equal to do a binary search among the maximal values of all rows. I need to prove at least one of them is a local maximum.

Assume none of them is a local maximum. Note them as $\{A[0][k_0], A[1][k_1], ..., A[n-1][k_{n-1}]\}$. Then all of them need to meet $A[i][k_i] < A[i-1][k_i] or A[i][k_i] < A[i+1][k_i]$. Let $A[m][k_m]$ be the maximal one of all, thus I can get $A[m-1][k_m] > A[m][k_m] or A[m+1][k_m] > A[m][k_m]$. Suppose $A[m-1][k_m] > A[m][k_m]$, because $A[m-1][k_{m-1}]$ is the max of $A[m-1]$, $A[m-1][k_{m-1}] > A[m-1][k_m] \implies A[m-1][k_{m-1}] > A[m][k_m]$. But $A[m][k_m]$ is the maximal value of all, that's a contradiction!

So at least one of them is a local maximum, and through binary search, I can find it.

```
-------------------------Problem 4 (2D cases)-------------------------
Input size:  10
Result: my algorithm -  [4, 4] , brute force -  [0, 0]
Loops: my algorithm -  10 , brute force -  1
Time: my algorithm -  0.00451 ms , brute force -  0.00738 ms
----------------------------------------------------------------------
Input size:  100
Result: my algorithm -  [49, 18] , brute force -  [0, 3]
Loops: my algorithm -  100 , brute force -  4
Time: my algorithm -  0.01149 ms , brute force -  0.00574 ms
----------------------------------------------------------------------
Input size:  1000
Result: my algorithm -  [499, 427] , brute force -  [0, 0]
Loops: my algorithm -  1000 , brute force -  1
Time: my algorithm -  0.08985 ms , brute force -  0.00328 ms
----------------------------------------------------------------------
```

Figure 5: (p42)

The time complexity of this algorithm is $\Theta(nlog(n))$. And in average case, it's worse than the brute force algorithm, for the expected value of search times of brute force algorithm is less than $5 \in \Theta(1)$, but mine is $\Theta(n)$. In worst case, brute force is $\Theta(n^2)$, and mine is $\Theta(nlog(n))$.

5. Implement two of the algorithms (from 1, 2a, 3, 4b) in Python, Java, or C++. Compare the performance with a reasonable naive brute force algorithm, Include, in each case, a driver function that takes at least the input size n as a parameter, generates a random test case of that size, and reports the answers and running time of both your algorithm and the brute force solution. You must do this part of the assignment strictly individually.

   I implemented them all. Just check the attached python codes.