# CS5010 Algorithm Assignment 1

## *Q1 ConvexMax*

**A convex polygon is defined as a polygon where all its internal angles are less than 180 degrees and no edges cross each other. You can assume the vertex with the smallest x-coordinate (assuming origin at bottom left) is the first coordinate and other vertices are numbered in a counter-clockwise direction. Figure 1 shows an example of such a polygon where V[1] is the first polygon. For simplicity, you can also assume all polygon vertices have distinct x and y co-ordinates. Given such a polygon, write an efficient algorithm to find:**

**(a)The polygon vertex with the maximum x-coordinate. Also provide the running time of the algorithm in the tightest bound possible. (15+5)**

**(b)The polygon vertex with the maximum y-coordinate. Also provide the running time of the algorithm in the tightest bound possible. (15+5)**

### *Answer:*

(a)

- Since Every vertex of the Convex has distinguished X-index and Y-index, in order to find the maximum x-coordinate, brute force method has a O(n) complexity.
- Form all x-coordinate into a list (assuming each vertex is list contains 2 elements,x and y):

  ```
  x_cor = [i[0] for i in vertex]
  ```

  If the x_cor is an ordinary array, finding the biggest number in x_cor has a O(n) complexity.

  However, this is a convex polygon, every internal angles are less than 180. x_cor[0] is supposed be the min element, no matter how it spin, as long as x_cor[0] is the smallest element, this is a bitonic sequence.
- A bitonic sequence means it is increasing then decreasing, or firstly decreasing then increasing.
- So we can find the maximum in following algorithm in Python:

  ```
  def findMaximum(arr, left, right):
  if left == right:
      return arr[left]

  if right == left + 1 and arr[left] >= arr[right]:
      return arr[left];

  if right == left + 1 and arr[left] < arr[right]:
      return arr[right]
  ```

```
    mid = (left + right)//2   #left + (right - left)

    if arr[mid] > arr[mid + 1] and arr[mid] > arr[mid - 1]:
        return arr[mid]
    if arr[mid] > arr[mid + 1] and arr[mid] < arr[mid - 1]:
        return findMaximum(arr, left, mid-1)
    else:
        return findMaximum(arr, mid + 1, right)
```

- **The Time Complexity is *O(log n)***

(b)

- We can still use the algorithm we applied from the previous problem. Except for the biggest point, any other point is increasing first then decreasing.

  Form all y-coordinate into a list:

  ```
  y_cor = [i[1] for i in vertex]
  ```

  Make y_cor[n] = y_cor[0] which makes the array a loop (Not to say how to implement, just assume the array is a loop). Then we have Four kinds of situations since we do not know any rule about the first node's y-coordinate.

  - Loop is up and up, which means y_cor[0] < y_cor[1] and y_cor[0] < y_cor[n-1]. Here y_cor[0] is the minimum.
  - Loop is down and down, which means y_cor[0] > y_cor[1] and y_cor[0] > y_cor[n-1]. Here y_cor[0] is the maximum.
  - Loop is up and down, which means y_cor[0] < y_cor[1] and y_cor[0] > y_cor[n-1]. Here y_cor goes up from y_cor[0] to Maximum and then goes down to Minimum and go up to y_cor[0].
  - Loop is down and up, which means y_cor[0] > y_cor[1] and y_cor[0] < y_cor[n-1]. Here y_cor goes down from y_cor[0] to Minimum and then goes up to Maximum and go down to y_cor[0].

  The first two situation are bitonic sequence, which can be treated the same as before, has a time complexity of O(log n)

  Then we check the last two situation which are alike in some way.

  Assume the Maximum has a index **m**, the Minimum has a index **n**, currently we are on index **i**.

  - If y_cor[i−1] < y_cor[i] < y_cor[i+1] and y_cor[i]>y_cor[0], then i < m.
  - If y_cor[i−1] < y_cor[i] > y_cor[i+1], then i = m.
  - If y_cor[i−1] > y_cor[i] > y_cor[i+1], then m < i < n.
  - If y_cor[i-1] > y_cor[i] < y_cor[i+1] then i = n.

  - If y_cor[i-1] < y_cor[i] < y_cor[i+1] and y_cor[i] < y_cor[0], then i > n.

  The above search would enable us to cut the sequence with 2 or 3 comparisons each time.

  Thus, we will have an algorithm with O(log n) time complexity.

- **The Time Complexity is *O(log n)***

## *Q2 FastProduct*

**You are given two binary strings representing two integers X and Y. Provide an efficient algorithm to multiply these two integers and return the product of the integers.**

**Please make sure the running time of your algorithm is faster than Theta($n^2$). Also provide the running time of your algorithm in the tightest bound possible.**

### *Answer:*

- As standard procedure, the product of two binary number has a time complexity of O($n^2$).

- Now we use the Divide and Conquer, we divide the given numbers into two parts, left and right.

  Say we have X and Y, so we now have:

  **$X = Xl * 2^{n/2} + Xr$**

  **$Y = Yl * 2^{n/2} + Yr$**

- The product of X and Y now is:

  **$XY = (Xl * 2^{n/2} + Xr) * (Yl * 2^{n/2} + Yr)$**

  = **$2^n XlYl + 2^{n/2}(XlYr + XrYl) + XrYr$**

  In the meantime,

  **XlYr + XrYl = (Xl+Xr)(Yl+Yr) - XlYl - XrYr**

  So we have,

  **$XY = 2^n XlYl + 2^{n/2}[(Xl+Xr)(Yl+Yr)-XlYl-XrYr] + XrYr$**

- With above trick, the recurrence becomes

  T(n) = 3T(n/2) + O(n)

- According to the Master Method, we have

  $T(n) = O(n^{3/2})$

## *Q3 Recurrence Relation*

**Find the tightest possible asymptotic bounds for T(n) in each of the following recurrence relations:**

**(a) $T(n) = 2 * T(n/3) + n\log_2(n)$**

**(b) $T(n) = 3 * T(n/5) + \log_2^2(n)$**

*Answer:*

(a)

- a = 2, b = 3, according to Master Method:

  **$C = \log_3^2$**

  **$f(n) = n\log_2(n)$**

  Then we compare the $n^C$ and f(n), then to determine the T(n)

- **$n^C = n^{\log_3^2}$**

- So $n^{\log_3^2} < n$, and $\log_2(n) > 0$ when $n > 1$, so

  **$n^C < n < n\log_2(n)$**

- $T(n) = Theta( n\log_2(n) )$

(b)

- Repeat the procedures we did in part a.

  **$C = \log_5^3$**

  **$f(n) = \log_2 n * \log_2 n$**

- **$n^C = n^{\log_5^3}$**

- So we compare $n^{\log_5^3}$ and $\log_2 n * \log_2 n$, in order to simplify the computation, we get

  $\log_5^3 = 0.682$

  sqrt(0.682) = 0.826

  So we compare $n^{0.826}$ and $\log_2 n$.

- Compute the gradients of both number, then we get

  $(n^{0.826})^` = 0.826 * n^{-0.176}$

  $(\log n)^` = \ln 2 * n^{-1} < c * n^{-0.176}$

- So we have $n^{\log_5^3} > \log_2^2$

- T(n) = Theta($n^{\log_5 3}$)