Arthur Dent
42

Assignment 1

COMP 3804
October 5, 2022

Question 1

Consider the following (not so great) algorithm defined on a set S of n numbers; assume that $n = 3k + 1$, for some non-negative integer k.
For i = 1 to b(n/3)c do
STEP 1: in linear time (in the size of S), find and then delete both min- and max-element from S
STEP 2: Again, in linear time, find and then delete from S (as updated by STEP 1) (just) the max-element.

1. First, what does this algorithm do more precisely? (It might help you, to first think about what the algorithm does if you were to omit Step 2. Do not write this as part of your solution.)
   Let us run the algorithm on the following sets: $\{1, 2, 3, 4\}$, $\{1, 2, 3, 4, 5, 6, 7\}$, and $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
   Set k=1: (loop once)
   Step 1: $\{1, 2, 3, \cancel{4}\}$
   Step 2: $\{1, 2, \cancel{3}, \cancel{4}\}$
   Set k=2: (loop twice)
   Step 1: $\{1, 2, 3, 4, 5, 6, \cancel{7}\}$
   Step 2: $\{1, 2, 3, 4, 5, \cancel{6}, \cancel{7}\}$
   Step 1: $\{1, \cancel{2}, 3, 4, \cancel{5}, \cancel{6}, \cancel{7}\}$
   Step 2: $\{1, \cancel{2}, 3, \cancel{4}, \cancel{5}, \cancel{6}, \cancel{7}\}$
   Set k=3: (loop three times)
   Step 1: $\{1, 2, 3, 4, 5, 6, 7, 8, 9, \cancel{10}\}$
   Step 2: $\{1, 2, 3, 4, 5, 6, 7, 8, \cancel{9}, \cancel{10}\}$
   Step 1: $\{1, \cancel{2}, 3, 4, 5, 6, 7, \cancel{8}, \cancel{9}, \cancel{10}\}$
   Step 2: $\{1, \cancel{2}, 3, 4, 5, 6, \cancel{7}, \cancel{8}, \cancel{9}, \cancel{10}\}$
   Step 1: $\{1, \cancel{2}, \cancel{3}, 4, 5, \cancel{6}, \cancel{7}, \cancel{8}, \cancel{9}, \cancel{10}\}$
   Step 2: $\{1, \cancel{2}, \cancel{3}, 4, \cancel{5}, \cancel{6}, \cancel{7}, \cancel{8}, \cancel{9}, \cancel{10}\}$

   We can see that when $k = 1$, the second smallest is returned, when $k = 2$ the third, and when $k = 3$, the fourth. From this we conclude that the algorithm returns the $k + 1$ smallest element in the set.

2. Then, write this algorithm in more detailed pseudo-code (filling in the min and max finding, in particular). Also make sure that you take care of the boundary/terminating conditions.

Algorithm 1 Find $k+1$ smallest

Input: Set: S
Output: element: $k+1$

```
1: if |S| = 1 then
2:     return S
3: for i = 1 to ⌊n/3⌋ do
4:     minimum ← ∞
5:     maximum ← −∞
6:     for each element ∈ S do
7:         if element < minimum then
8:             minimum ← element
9:         if element > maximum then
10:            maximum ← element
11:    S ← S \ minimum
12:    S ← S \ maximum
13:    maximum ← −∞
14:    for each element ∈ S do
15:        if element > maximum then
16:            maximum ← element
17:    S ← S \ maximum
18: return S
```

Note: Since we are assuming that $n = 3k + 1$ we will always end up with exactly one element left in the set upon completion of the for loop. Remember we need to check for edge cases (there is one when $k = 0$, Therefore there is 1 element in the set).

3. What is the time complexity of this (bad) algorithm?
The for loop executes $\lfloor \frac{n}{3} \rfloor$ times, which is $O(n)$.
Setting the max and min initial values takes $O(1)$ time.
We then look for the largest and smallest elements by examining all elements in the set. This takes $O(n)$ time.
Removal of these elements takes $O(1)$ time.
We then find the next largest (step 2), this also takes $O(n)$ time.
Removal again is $O(1)$
Inside the for loop the min/max search takes the longest, so the overall time complexity is $O(n) * O(n) = O(n^2)$

4. Then, prove that this algorithm correctly finds that element of a set of n elements as determined by you in Part I.
   To prove we use a loop invariant:

   Invariant:
   At the beginning of the i^{th} iteration, $\{k+1\} \in S$ **Initialization:**
   When starting, the Set contains all n elements, which contains the k+1-smallest element of the set.

   Maintenance:
   At the beginning of iteration i, the $i-1$ smallest elements have been removed and the largest $2i-2$ elements have been removed. Since $i \leq k$ as $\lfloor \frac{n}{3} \rfloor = k$, and $n-(2i-2) > k+1$ the $k+1$ smallest element is still in the set. Assuming the Invariant is True, we enter the outer loop. The first inner loop removes the $(2i-1)^{th}$ largest and i^{th} smallest elements from the set. The second loop removes the $2i^{th}$ largest element from the set. As $i < \lfloor \frac{n}{3} \rfloor + 1 < n - 2i$ the $k+1$ smallest element is still in the set.

   Termination:
   At the start if the $\lfloor \frac{n}{3} \rfloor^{th}$ step the $\lfloor \frac{n}{3} \rfloor - 1$ smallest and $\lfloor \frac{n}{3} \rfloor - 2$ largest elements have been removed leaving exactly 4 elements left in the set. At the end of the loop 3 more elements have been removed leaving the $k+1$ smallest element as $\lfloor \frac{n}{3} \rfloor = k < k+1 < 2 * \lfloor \frac{n}{3} \rfloor = n - 2k + 2$

Question 2

Implement the two algorithms presented in class for computing Finonacci numbers, fib1 and fib2 and the direct way using the Golden Ratio.

Each time you compute a Fibonacci number, measure how many milliseconds it takes, and print this timing information. There is a Java function, System.currentTimeMillis(), which returns a long result; call it once before you do the computation, and a second time after you do the computation; subtract to get the elapsed time. Be careful to measure only the computation time; any changes to the GUI should be made either before or after you start measuring. Do not do anything else on your computer. From time to time, your computer will do garbage collection, this may explain strange timing results.

What is the maximum value of n for which you can compute Fibonacci? How long does it take for each n that you can compute? Hand in your code and the timing results.

Algorithm 2 fib1

Input: int: n
Output: int: *number*, **float:** *time*

1: $start \leftarrow$ time.time()
2: $number \leftarrow$ fib1help(n)
3: $end \leftarrow$ time.time()
4: **return** ($number$, $end - start$)

Algorithm 3 fib1help

Input: int: n
Output: int: *number*

1: **if** $n \leq 2$ **then**
2: **return** 1
3: **return** (fib1help($n-1$) + fib1help($n-2$))

Depending on which language you implemented this in you would encounter one of the two following errors:

1. If you declared your data type to store your Fibonacci number as an integer, F_{47} would produce an integer overflow. If you used a long long, you would not get an overflow until F_{94}. If you chose Python as your programming language, Python handles the datatype in software and which will prevent the number from overflowing, and you would then only encounter the second type of error.

2. The other error you may encounter is a Stack Overflow. For Python, this is somewhere around 1,000 recursive calls, in Java, it is dependent on how much memory is assigned to the virtual memory for the instance running.

Algorithm 4 fib2

Input: int: n
Output: int: *number*, **float:** *time*

1: $start \leftarrow$ time.time()
2: **if** $n \leq 2$ **then**
3: **return** (1, time.time() - *start*)
4: $FibArray \leftarrow [1] * n$
5: **for** $\{i = 3, i \leq n, ++i\}$ **do**
6: *FibArray[i]* $\leftarrow$ *FibArray[i-1]* + *FibArray[i-2]*
7: $stop \leftarrow$ time.time()
8: **return** ($FibArray[n]$, *stop* - *start*)

The first error you should run into is an integer (or long long) overflow error. With Python, you would run out of memory first.

Algorithm 5 GoldenRatio

Input: int: n
Output: int: *number*, **float:** *time*

1: *start* ← time.time()
2: *phi* ← $(0.5 - \frac{\sqrt{5}}{2})$
3: *rho* ← $(0.5 + \frac{\sqrt{5}}{2})$
4: *fibNum* ← (*phi****n* + *rho****n*) / $\sqrt{5}$
5: *stop* ← time.time()
6: **return** (*FibNum*, *stop* - *start*)

The first error you are likely to run into is a math error as $\sqrt{5}$ is an irrational number and computers do not like those.

Question 3

For the following we will assume that $T(1) = 1$. (If required, you can also assume that $T(k) = O(1)$, for any constant k.) For 1-3, "solve" means provide a big-oh bound.

1. Solve, by unfolding also known as the iterative method (or repeatedly applying the recurrence relation):

   (a) $T(n) = T(n-2) + n$

$$\begin{aligned} T(n) &= T(n-2) + n \\ &= T(n-4) + (n-2) + n = T(n-4) + 2n - 2 \\ &= T(n-6) + (n-4) + 2n - 2 = T(n-6) + 3n - 6 \\ &= T(1) + n * n - 2 * n \\ &= O(n^2) \end{aligned}$$

(b) $T(n) = 2T(n/4) + n$

$$\begin{aligned}
T(n) &= 2T(n/4) + n \\
&= 2T(n/16) + 2 * n/4 + n = 2T(n/16) + 3n/2 \\
&= 2T(n/64) + 4 * n/16 + 3n/2 = 2T(n/64) + 7n/4 \\
&= 2T(n/256) + 8 * n/64 + 7n/4 = 2T(n/256) + 15n/8 \\
&= 2T(n/4^{\log n}) + 2^{(\log n)-1} * n/4^{\log n} + O(n) \\
&= 2T(k) + n/2^{\log n+2} + O(n) \\
&= O(n)
\end{aligned}$$

Note: We are reducing the size of the recursion by half each time hence the O(n) (recall the sum of $1/2^i \approx 2$)

2. Now, solve by applying the recursion tree method:

(a) $T(n) = T(n-2) + n$

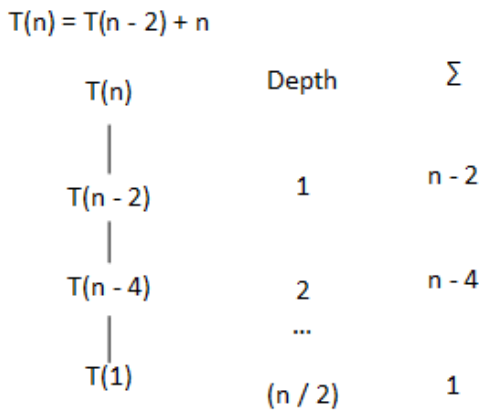

Figure 1: Recursion tree for $T(n) = 4T(\frac{n}{4}) + n$

(b) $T(n) = 4T(n/4) + n$

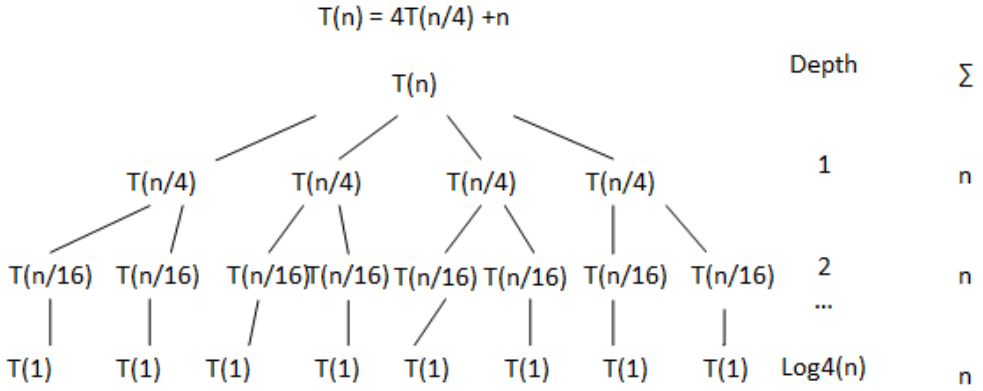

Figure 2: Recursion tree for $T(n) = 4T(\frac{n}{4}) + n$

(c) $T(n) = 2T(n/4) + n$

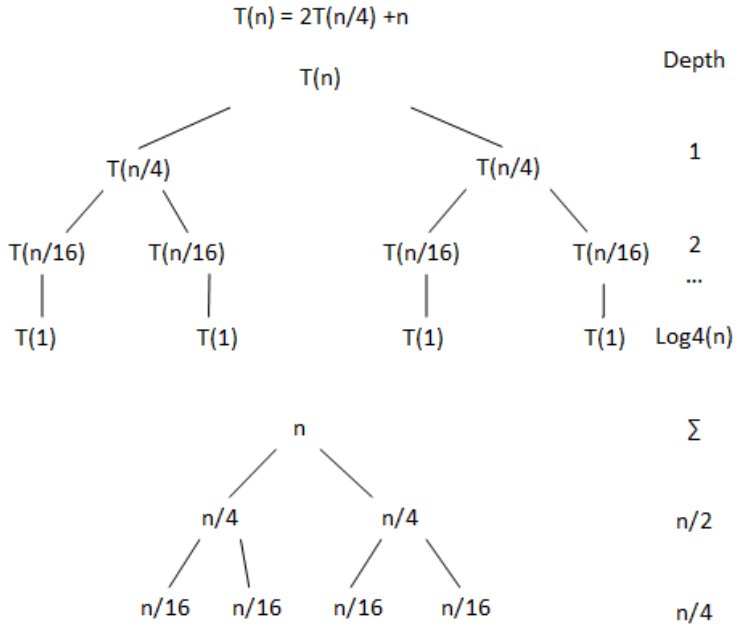

Figure 3: Recursion tree for $T(n) = 2T(\frac{n}{4}) + n$

3. Next, solve by guessing the solution and then proving it using induction:

   (a) $T(n) = 8T(n/8) + n$

   Guess: $T(n) = cn \log n$
   Base Case:
   $T(2) = c * 2 * \log 2 = 2c$
   Prove for n
   $T(n) = 8 * c\frac{n}{8} \log \frac{n}{8} + n$
   $T(n) = cn \log n - cn \log 8 + n$
   $T(n) = cn \log n + n - 3cn$
   $T(n) < cn \log n$

   (b) $T(n) = T(n/2) + \log n$

   Guess: $T(n) = c \log n$
   Base Case:
   $T(4) = c \log \frac{4}{2} + \log 4$
   $T(4) = c + 2$
   Prove for n :
   $T(n) = c \log \frac{n}{2} + \log n$
   $T(n) = c \log n - c + \log n$
   $T(n) < c \log n,\ c \geq \log n$

   (c) $T(n) = T(n/2^n) + 1$

   Guess: $T(n) = \log n$
   Base Case:
   $T(4) = 2 + 1$
   Prove for n :
   $T(n) = \log \frac{n}{2^n} + 1$
   $T(n) = \log n - \log 2^n + 1$
   $T(n) < \log n,\ n \geq 1$

   Note: There are a couple of ways you can look at this question: either the upper bound is $O(1)$ as: $\lim_{n\to\infty} \frac{n}{2^n} \to 0$ or there is no solution as $\frac{n}{2^n}$ has no integer solutions when $n \in \mathbb{N}^+$.

4. Finally, apply the Master Theorem (in its general form), where possible, to solve the following recurrence relations and give a -bound for each of them. If the master's theorem is not applicable, state why and then solve the recurrence using another method that we learned in class.

   (a) $T(n) = T(n-4) + n$ Cannot be solved using Master's Theorem, as it is not of the correct form. We will then guess that $T(n) \leq cn^2$ and solve by induction:
   Assume True for $n-4$, prove for n:

$$\begin{aligned} T(n) &\leq c(n-4)^2 + n \\ &\leq cn^2 - 8cn + 16c + n \\ &\leq cn^2 + n(1-8c) + 16c \\ &\leq cn^2, c > \frac{1}{8} \end{aligned}$$

   (b) $T(n) = T(9n/3) + n^2$
   $T(n) = T(9n/3) + n^2 = T(3n) + n^2$, the sub-problem is increasing in size and therefore does not terminate and has no solution.

   (c) $T(n) = T(16n/4) + n^2$
   $T(n) = T(16n/4) + n^2 = T(4n) + n^2$, the sub-problem is increasing in size and therefore does not terminate and has no solution.

   (d) $T(n) = 7T(n/3) + n^2$

$$\begin{aligned} &a = 7,\ b = 3\ c_{crit} = \log_3 7 \approx 1.77 \\ &f(n) = O(n^2),\ c = 2 \\ &\therefore \text{Case 3 as } c > c_{crit} \\ &T(n) = \Theta(n^2) \end{aligned}$$

(e) $T(n) = 2^n T(n/2) + O(1)$
Master's Theorem does not apply as $a = 2^n$ is not a constant. We will solve by Tree decomposition:

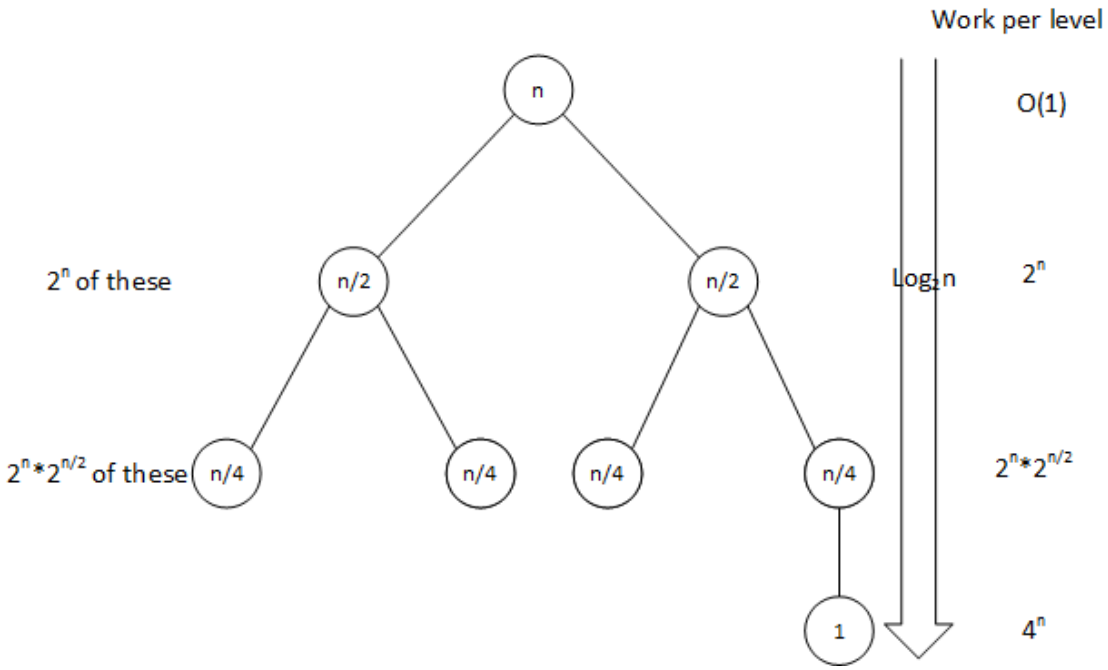

Figure 4: Recursion Tree for $T(n) = 2^n T(n/2) + O(1)$

(f) $T(n) = cT(n/c) + n$ for some positive integer c

$$a = c,\ b = c,\ f(n) = n$$
$$n^{\log_c c} = n = \Theta(n \log^k n),\ k = 0$$
Case 2a
$$T(n) = \Theta(n^{\log_c c} * \log^{k+1} n)$$
$$T(n) = \Theta(n \log n)$$

(g) $T(n) = 2T(n/4) + n^{0.6}$

$$a = 2,\ b = 4,\ f(n) = n^{0.6}$$
$$n^{\log_4 2} = n^{\frac{1}{2}} < f(n)$$
Case 3
$$T(n) = \Theta(n^{0.6})$$

5.

What can you say about the time complexity of an algorithm whose running time is given by this recurrence $T(n) = 2T(n) + O(\log^2 n)$?

This algorithm will never terminate, as the recurrence is called on n with no reduction in size.

(b) Which method might be most appropriate to solve the following recurrence? Use it to give its solution. $T(n) = T(n/4) + T(3n/4) + n$
Since this has two recursions, we cannot apply master's theorem. Guessing for induction and unfolding looks hard, so we will use a recursion tree:

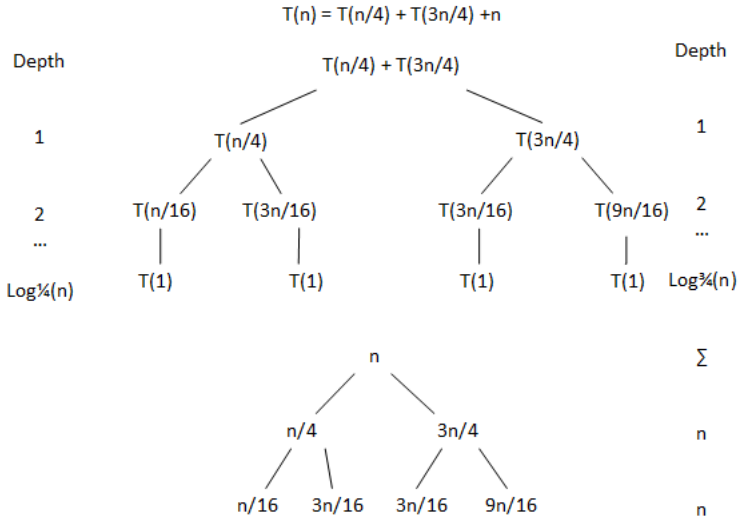

Figure 5: Recursion tree for $T(n) = T(\frac{n}{4}) + T(\frac{3n}{4}) + n$

Question 4

DT claims that he has a data structure that he can create in O(n) time and on which he can perform the operation DeleteMin using o(logn) comparisons. (note the little-oh)

(a) What could you say about the number of comparisons used by the most efficient sorting algorithm designed based on DT's claim? Argue.

If DT can perform DeleteMin using $o(\log n)$ comparisons then we can sort n elements by doing the following:

1. Build DT's data structure with all n elements, which he claims takes $O(n)$ time.

2. Perform DeleteMin n times to remove all n elements in sorted order and add them to an array. Since each removal requires at most $*o(\log n)$ comparisons, the total number of comparisons made for all n elements is $o(n * \log n)$.

Since Step 2 dominates the time complexity, we can say that the entire process of sorting n elements is $o(n \log n)$.

(b) Based (a) argue that DT must be lying.

It has been proven that you cannot sort faster than $\Omega(n \log n)$ in the worst case. If DT was telling the truth, then he could sort (in the worst case) faster than possible as $o(n \log n) < \Omega(n \log n)$. Therefore, DT must be lying.

Question 5

Informally, a convex polygon is a (non self-intersecting) polygon in which each interior angle is less than 180°.

- Draw a convex polygon based on that definition

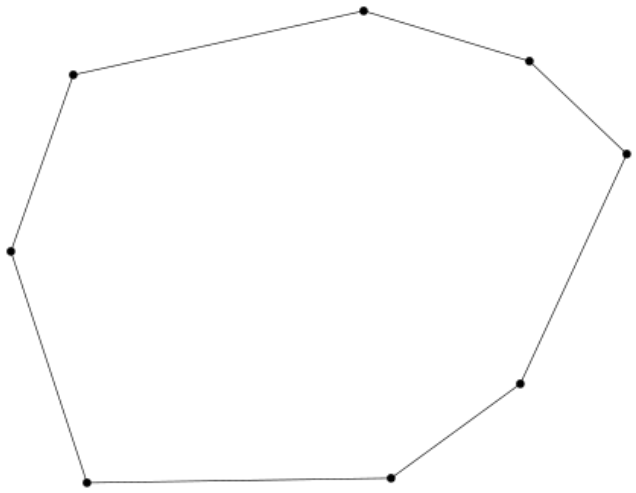

Figure 6: A Convex Polygon

- Draw a self-intersecting polygon

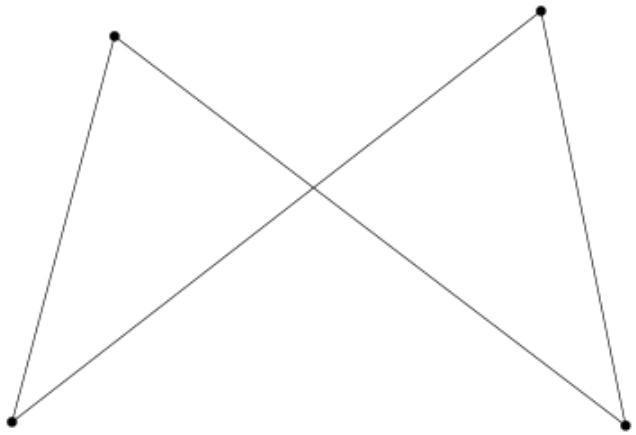

Figure 7: A Self Intersecting Polygon

The convex hull of a set of points S is the smallest convex polygon containing all points of S (no point of S is outside the convex hull; the vertices of the polygon are points from S). (Without handing this in, draw examples for yourself.)

- A property of the convex hull is that, for each edge e of the convex hull polygon, all points of S are on one side of the (infinite) line drawn through e (some points may be on the line). Draw this and hand in with your solution.

  Now, design a simple algorithm that uses this property. Prove its correctness and establish its time complexity.
  We will make one assumption: That the points are in general position, i.e. no three points are colinear. (We can relax this requirement by measuring the length of each line and taking the shortest if there is more than one).

Algorithm 6 Convex1

Input: Set of points: S
Output: Set of points: *hull*

```
1: S ← Sort(S, X_Min)
2: hull ← S[0]
3: while True do
4:     for point ∈ S\{hull} do
5:         line ← Create_Line(point, hull[−1])
6:         if One_Side(line, S) then
7:             hull ← hull + point
8:             continue
9:     return hull
```

The One_Side() function returns True if all points are on, or on one side of the line.The logic of the rest is then fairly straight forward: We know (up to) four points on the hull, the points with have: min_x, max_x, min_y, and max_y. Using this as a starting point we then search for its neighbour. By removing the found points from the set we eliminate the possibility of getting trapped in an infinite loop. We will prove by way of loop invariant:

Invariant:
At the beginning of the i^{th} iteration, *hull* contains i convex-hull points and S contains $k - i$ convex-hull points, where k is the total number of convex-hull points.

Initialization:
Before entering the for loop, *hull* contains 1 convex-hull point and S contains $k - 1$ convex-hull points.

Maintenance:
Assuming the Invariant is True, At the start of the i^{th} iteration ($i < k-1$), *hull* contains i convex-hull points and S contains $k - i$ points. One of the $k - i$ points is connected to $hull[-1]$ by an edge. One_Side() will identify the point on the convex-hull that is connected to $hull[-1]$ by an edge and add it to *hull*. *hull* now contains $i + 1$ points.

Termination:
At the beginning of the k^{th} iteration, *hull* contains all k points. As the for loop only creates lines with $S \backslash \{hull\}$, One_Side() will never return True and the for loop ends. Algorithm Convex1 then returns *hull* and terminates.

- Let us see if we can do better. You are given two linear-time algorithms which you can just use without having to give pseudocode nor prove their correctness. (Like use as a black box)
  The first one is a median finding algorithm. The median of a set of n numbers is the element for which $\lfloor \frac{n}{2} \rfloor$ are less than or equal to it and the remaining elements are larger than it.
  The second one computes the convex hull of two point sets for which we are given their convex hulls. The points sets here are assumed to be separated by a vertical line.

Now use this to design a D&C algorithm to find convex hull of a set on n points. Prove its correctness and establish its time complexity.

Algorithm 7 Convex2

Input: Set of points: S
Output: Set of points: *hull*

1: **if** $|S| \leq 3$ **then**
2: **return** S
3: $median \leftarrow$ Find_Median(S)
4: $left \leftarrow$ Get_Left(S, $median$)
5: $right \leftarrow$ Get_Right(S, $median$)
6: $left\_hull \leftarrow$ Convex2($left$)
7: $right\_hull \leftarrow$ Convex2($right$)
8: **return** Merge_Hulls($left$, $right$)

We will prove this by induction:

Base Case:

Any point set consisting of at most 3 points forms a convex hull. Lines 1 & 2 returns any set of size 3 or smaller.

Inductive Assumption:

Convex2 correctly finds a convex-hull for $n - 1$ points.

Inductive Step:

First we assume $n > 3$, then we evaluate the Algorithm line by line: From the problem statement Find_Median() correctly identifies the median of the n points. Get_Left() will correctly identify all points to the left (and including) the median by evaluating $point.x \leq median.x$. Similarly Get_Right() evaluates $point.x > median.x$. By the inductive assumption Convex2 correctly creates the left_hull and right_hull sets. Finally by the problem statement Merge_Hulls() will correctly merge left_hull and right_hull and returns the convex-hull.

Time Complexity:

We will evaluate line by line:

Checking the size of S is done in constant time

From the problem statement Find_Median() takes linear time

Finding left takes $O(n)$

Finding right takes $O(n)$

There are $O(\log n)$ recursive calls to Convex2

From the problem statement Merge_Hulls takes $O(n)$ time

Therefore each call to Convex2 takes $O(n)$ time and with $O(\log n)$ calls, the overall time complexity is $O(n \log n)$