CS 161 **Problem Set 2**
Fall 2025 **Due:** Friday October 10 at 11:59pm on Gradescope.

Style guide and expectations: Please see the top of the "Homework" page on the course webpage for guidance on what we are looking for in homework solutions. We will grade according to these standards.

Make sure to look at the "**We are expecting**" blocks below each problem to see what we will be grading for in each problem!

Collaboration policy: You may do the HW in groups of size up to three. Please submit one HW for your whole group on Gradescope. (Note that there is an option to submit as a group). See the "Policies" section of the course website for more on the collaboration policy.

LLM policy: Check out the course webpage for best practices on how to productively use LLMs on homework, if you use them at all.

Exercises

We recommend you do the exercises on your own before collaborating with your group. The point is to check your understanding.

---

1. **(6 pt.)** Compute the *best* (that is, the smallest) upper-bound that you can on the following recurrence relations. You may use any method we've seen in class (Master Theorem, Substitution Method, algebra, etc.)

   (a) **(2 pt.)** $T(n) = 2T(\lfloor n/2 \rfloor) + O(\sqrt{n})$, where $T(0) = T(1) = 1$.
   (b) **(2 pt.)** $T(n) = T(n-2) + 1$, where $T(0) = T(1) = 1$.
   (c) **(2 pt.)** $T(n) = 6T(\lfloor n/4 \rfloor) + n^2$ for $n \geq 4$, and $T(n) = 1$ for $n < 4$.

   [**We are expecting:** *For each part, a statement of your answer of the form "$T(n) = O(\_\_\_\_)$", as well as a short justification. You don't need to give a rigorous proof, but your justification should be convincing to the grader. If you use the master theorem, state the values of a, b, d.*]

SOLUTION:

In the solutions below, when using the master theorem, recall that we define the constants a, b, d to fit the recurrence relation form of $T(n) = aT(\frac{n}{b}) + O(n^d)$.

(a) Notice that our recurrence relationship satisfies $a = b = 2$ and $d = 0.5$. As a result, $2 = a > b^d = \sqrt{2}$, and by the master theorem, $T(n) = O(n^{\log_b(a)}) = O(n^{\log_2(2)}) = O(n)$.

(b) We can repeatedly expand the definition of $T(n)$ and observe a pattern. We get that:

$$T(n) = T(n-2) + 1 = T(n-4) + 2 = T(n-6) + 3 = \ldots = k + T(n-2k) = k + 1,$$

where k satisfies $k = \lfloor n/2 \rfloor = O(n)$. Therefore, we can conclude that $T(n) = O(n)$.

(c) Notice that our recurrence relationship satisfies $a = 6$, $b = 4$, and $d = 2$. As a result, $6 = a < b^d = 16$, and by the master theorem, $T(n) = O(n^d) = O(n^2)$.

2. **(3 pt.)** Consider the following algorithm, which takes as input an array A:

```
def printStuff(A):
    n = len(A)
    if n <= 4:
        return
    for i in range(n):
        print(A[i])

    printStuff(A[:(n//3)]) # recurse on the first floor(n/3) elements of A
    printStuff(A[(n//3):2*(n//3)]) # recurse on the second floor(n/3) elements of A
    return
```

What is the asymptotic running time of `printStuff` on an array of length n?

[**We are expecting:** *The best answer you can give of the from "The running time of* ***printStuff****(n) is $O(\_\_\_\_)$, and a short explanation, including a clear statement of the relevant recurrence relation.*]

SOLUTION: The running time of printStuff(n) is $O(n)$. The running time satisfies a recurrence relation of the form:

$$T(n) = 2T(\lfloor n/3 \rfloor) + O(n),$$

since we have two recursive calls on $n/3$ elements each, and then there's $O(n)$ work to print out all n elements of A. We apply the master theorem with $a = 2, b = 3, d = 1$, and get that the running time is $O(n)$.

3. **(4 pt.)** [**Fishing.**] Plucky the Pedantic Penguin is fishing. Plucky catches n fish, but plans to keep only the k largest fish and to throw the rest back. Plucky has already named all of the fish, and has measured their length and entered it into an array F of length n. For example, F might look like this:

$$F = \begin{bmatrix} \text{(Frederick the Fish, 14.2in)} \\ \text{(Fabiola the Fish, 10in)} \\ \text{(Farid the Fish, 12.35in)} \\ \vdots \\ \text{(Felix the Fish, 6.234523in)} \\ \text{(Finlay the Fish, 6.234524in)} \end{bmatrix}$$

Give an $O(n)$-time deterministic algorithm that takes F and k as inputs and returns a list of the names of the k largest fish. You may assume that the lengths of the fish are distinct.

Note: Your algorithm should run in time $O(n)$ even if k is a function of n. For example, if Plucky wants to keep the largest $k = n/2$ fish, your algorithm should still run in time $O(n)$.

[**We are expecting:** *Pseudocode* ***AND*** *a short English description of your algorithm. You may (and, hint, may want to...) invoke algorithms we have seen in class. You do not need to justify why your algorithm is correct or its running time.*]

SOLUTION: The idea is to first run SELECT on F to find the $n-k+1$'st smallest fish, which is the same as the k'th biggest fish. Let's call this k'th biggest fish Flounder the Fish. Then we can run through all the fish, and compare them to Flounder the Fish. If a fish is longer than Flounder, we keep it; if not, we throw it back. (And we keep Flounder the Fish too).

SELECT runs in time $O(n)$, and it also takes time $O(n)$ to loop through all the fish, so the whole algorithm runs in time $O(n)$.

The pseudocode is:

```
def findBigFish(k,F):
     keep = []
     (kName, kLength) = SELECT(F,n-k+1), using the lengths as the keys
     for (fishName, fishLength) in F:
           if fishLength >= kLength:
                keep.append(fishName)
     return keep
```

Problems

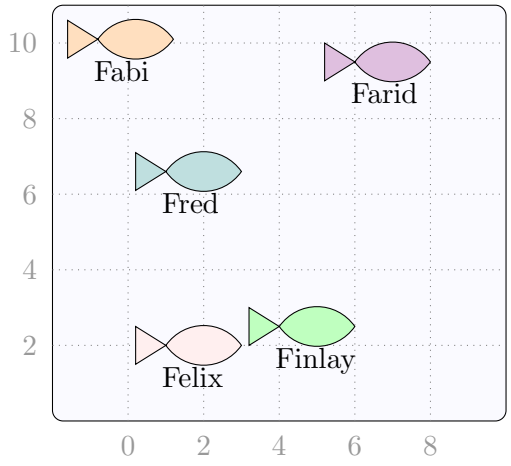
---

4. **(17 pt.) (Fish Friends)**

   You are given a list F of fish and (two-dimensional) coordinate locations in a pond, in feet. For example, the input might look like this:

$$F = \begin{bmatrix} \text{(Frederick the Fish, 2, 6.6)} \\ \text{(Fabiola the Fish, 0.2, 10.1)} \\ \text{(Farid the Fish, 7, 9.5)} \\ \vdots \\ \text{(Felix the Fish, 2, 2)} \\ \text{(Finlay the Fish, 5, 2.5)} \end{bmatrix}$$

   which corresponds to the two-dimensional layout:



   Your goal is to return the distance between the closest pair of fish. In the example above, Felix and Finlay are closest, at distance $\sqrt{(5-2)^2 + (2.5-2)^2} = 3.04$ feet, so on input F, your algorithm should return 3.04.

   Note: The fact that the fish have names is not important for solving this problem, we just included them to make the example more clear. If you like, you can assume that F is an array of coordinate pairs (x, y) with no names, rather than an array of triples (name, x, y). If you drop the names, state so clearly in your solution.

   (a) **(5 pt.)** Design a divide-and-conquer algorithm that determines whether there exists a pair of fish within 1 foot of each other, on an input list F with n fish. Your algorithm should run in time $O(n \log n)$. You may assume that n is a power of two if you like.

   [**Hint:** *Depending on how you do the problem, the following may eventually be helpful: Suppose that you have a bunch of fish, all of distance at least d from each other. Then at most 8 of these fish can be contained in any $2d \times d$ rectangle. (You can take this as a fact, but it might be fun to convince yourself of this).*]

[**We are expecting:** *Pseudocode and a clear English description of what it is doing. You do not need to prove correctness or justify the running time (yet). You may use any algorithm we have seen in class as a black box.*]

(b) **(4 pt.)** Design a divide-and-conquer algorithm that returns the distance between the closest pair of fish, on an input list F with n fish. Your algorithm should run in time $O(n \log n)$. You may assume that n is a power of two if you like.

[**Hint:** *We do not expect that it will be useful to use your solution from part (a) as a black box, but we do expect that similar ideas will be helpful for this part.*]

[**We are expecting:** *Pseudocode and a clear English description of what it is doing. Your response should be self-contained, meaning that it should not only state what is different than your part (a) solution, the whole pseudocode should be in this part. You do not need to prove correctness or justify the running time (yet). You may use any algorithm we have seen in class as a black box.*]

(c) **(4 pt.)** Explain why your algorithm in part (b) is correct. You don't need to give a formal proof, but your explanation should be convincing to the grader.

[**We are expecting:** *A clear explanation. If you don't think that your algorithm is correct, you can say what's wrong for partial credit.*]

(d) **(4 pt.)** Explain why the running time of your algorithm in part (b) is $O(n \log n)$.

[**We are expecting:** *A short explanation. Writing down a recurrence relation, explaining why it is relevant, and explaining why it has the desired solution is sufficient. If you don't think that the running time of your algorithm is $O(n \log n)$, say what you think the running time is and justify that instead for partial credit on this part.*]

SOLUTION:

(a) Our plan is to use a divide-and-conquer approach where we repeatedly split fish depending on the median x-coordinate, and then consider: the pairs of fish contained entirely in the left half (via a recursive call), the pairs of fish contained entirely in the right half (via a recursive calls), and the pairs of fish with one in the left half and one in the right half. Handling this third case will be the most sophisticated.

We will begin our algorithm with preprocessing where we create two lists in $O(n \log n)$ time: one with the fish sorted by x-coordinates, and one with the fish sorted by y-coordinates. Then, at each level of recursion, we divide the list in half according to the median x-coordinate, and recurse on either side. One recursive call will handle all pairs of fish where both were to the left of the median, and the other recursive call will handle all pairs of fish where both were to the right of the median. We still need to consider fish friends that cross the two halves. To do that, consider all the fish within 1 foot of the dividing line. Call these the "Frontier Fish". Then we need to find any pair of the frontier fish that are within 1 foot. To do this, we consider the frontier fish by y-coordinate (we use the fact that we've already sorted the fish by y-coordinate to make this list in time $O(n)$). For reasons we explain later in part (c), we compare each frontier fish to others within 15 of it in the ordering. (7 is also correct, but we find that 15 is clearer to explain, and the difference between 7 and 15 only affects the constant

in the big-Oh, not the asymptotic running time). So the algorithm goes through this list of sorted frontier fish, comparing every fish to every other fish within 15 of it in this ordering. If we ever found a pair of fish within 1 feet throughout this process, we return True. Otherwise, we return False.

```
# Euclidean distance between triples
# fish1 and fish2 are triples (name, x,y)
def dist( fish1, fish2 ):
     return sqrt( (fish1[1] - fish2[1])^2 + (fish1[2]-fish2[2])^2 )


def finalAlg(F):
    Use MergeSort to sort the fish in F by x-coordinate
    Let F_y be a copy of F that is sorted by y-coordinate
    return findDist1FishFriends(F, F_y)

def findDist1FishFriends(F, F_y):
     n = len(F)  # assume that n is a power of 2
     if n == 2:
          return dist( F[0], F[1] ) <= 1
          # There are only two fish, return the Euclidean dist between them.

     # split F in half
     L = F[:n/2]
     R = F[n/2:]
     # find the x-coordinate the separates L and R
     midline = (L[n/2-1][1] + R[0][1])/2

     # create corresponding halves for F_y
     L_y = []
     R_y = []
     for fish in F_y:
         if fish[1] < midline:
             L_y.append(fish)
         else:
             R_y.append(fish)

     # recurse on each half
     if findDist1FishFriends(L, L_y) or findDist1FishFriends(R, R_y):
          return True

     # Merge the two halves by looking at a +/- 1 strip around the midline.
     M = []
     for fish in F_y:
```

```
            if |fish[1] - midline| <= 1:
                 M.append( fish )

       for i = 0,1,...., len(M)-1:
            for j = 1, ...,  min(15,len(M)-1-i):
                 if dist( M[i] , M[i + j] ) <= 1:
                     return True
       return False
```

(b) This algorithm is mostly similar to part (a), except the distance with which we define "Frontier Fish" is influenced by the output of the recursive calls. We provide a self-contained solution as was requested in the problem set.

Our plan is to use a divide-and-conquer approach where we repeatedly split fish depending on the median x-coordinate, and then consider: the pairs of fish contained entirely in the left half (via a recursive call), the pairs of fish contained entirely in the right half (via a recursive calls), and the pairs of fish with one in the left half and one in the right half. Handling this third case will be the most sophisticated.

We will begin our algorithm with preprocessing where we create two lists in $O(n \log n)$ time: one with the fish sorted by x-coordinates, and one with the fish sorted by y-coordinates. Then, at each level of recursion, we divide the list in half according to the median x-coordinate, and recurse on either side. One recursive call will handle all pairs of fish where both were to the left of the median, and the other recursive call will handle all pairs of fish where both were to the right of the median. We still need to consider fish friends that cross the two halves. To do that, let d be the minimum of the two recursive calls, and consider all the fish within d of the dividing line. Call these the "Frontier Fish". Then we need to find any pair of the frontier fish that are closer than d to each other. To do this, we consider the frontier fish by y-coordinate (we use the fact that we've already sorted the fish by y-coordinate to make this list in time $O(n)$). For reasons we explain in the next part, we compare each frontier fish to others within 15 of it in the ordering. (7 is also correct, but we find that 15 is clearer to explain, and the difference between 7 and 15 only affects the constant in the big-Oh, not the asymptotic running time). So the algorithm goes through this list of sorted frontier fish, comparing every fish to every other fish within 15 of it in this ordering. Finally, we take the minimum of d and any frontier fish friends we found.

```
# Euclidean distance between triples
# fish1 and fish2 are triples (name, x,y)
def dist( fish1, fish2 ):
     return sqrt( (fish1[1] - fish2[1])^2 + (fish1[2]-fish2[2])^2 )


def finalAlg(F):
    Use MergeSort to sort the fish in F by x-coordinate
    Let F_y be a copy of F that is sorted by y-coordinate
    return findFineFishFriends(F, F_y)
```

```
def findFineFishFriends(F, F_y):
     n = len(F)  # assume that n is a power of 2
     if n == 2:
          return dist( F[0], F[1] )
          # There are only two fish, return the Euclidean dist between them.

     # split F in half
     L = F[:n/2]
     R = F[n/2:]
     # find the x-coordinate the separates L and R
     midline = (L[n/2-1][1] + R[0][1])/2

     # create corresponding halves for F_y
     L_y = []
     R_y = []
     for fish in F_y:
         if fish[1] < midline:
             L_y.append(fish)
         else:
             R_y.append(fish)

     # recurse on each half
     d = min( findFineFishFriends(L, L_y), findFineFishFriends(R, R_y) )

     # Merge the two halves by looking at a +/- d strip around the midline.
     M = []
     for fish in F_y:
          if |fish[1] - midline| <= d:
               M.append( fish )
     smallest = d
     for i = 0,1,...., len(M)-1:
          for j = 1, ...,  min(15,len(M)-1-i):
               if dist( M[i] , M[i + j] ) < smallest:
                   smallest = dist( M[i], M[i+j] )
     return smallest
```

(c) We split the fish into two halves, L and R. We recursively find d, which is the smallest distance between any fish friends both in L, and any fish friends both in R. The thing we should return is the minimum of d and the smallest distance between any fish in L and any fish in R. The only way that a fish in L and a fish in R can be within d of each other is if they are within d of the midline. So we put all such fish into the array M, which we call the Frontier Fish. We claim that we only need to consider frontier fish within ± 7 of each other in M, in order of y-coordinate. To see why, consider a fixed frontier fish Frankie, and suppose without loss of generality that Frankie was in L. Say

that Frankie has coordinates (x^*, y^*).

We want to consider all of the frontier fish whose y-coordinates are within d of y^*; that will make sure that we catch any fish within d of Frankie. We might worry that there could be $\Omega(n)$ such fish, which would be too many to look at...but! We claim that there are at actually most 15 such fish (other than Frankie).

To see why, consider the $2d \times d$ rectangle $R^* = [\text{midline}, \text{midline} + d] \times [y^* - d, y^* + d]$. Since this rectangle is contained on the right hand side of the midline, any two fish in R^* were also both in R, and so they are at least d apart from each other. By the hint, there are at most 8 fish in this rectangle R^*. (The reason why the hint is true is the following: imagine dividing up this rectangle into 8 squares of size $d/2 \times d/2$ in the natural way. Each of these squares has diameter $d/\sqrt{2} < d$, any so there can be at most one fish in each square, so 8 fish in the whole rectangle).

By the same logic, there are at most 7 fish other than Frankie in the rectangle $L^* = [\text{midline} - d, \text{midline}] \times [y^* - d, y^* + d]$.

Putting it all together, there are at most 15 frontier fish (other than Frankie) whose y-coordinates are within d of y^*, which is what we claimed. Thus, it suffices for us to go through M in sorted order and to compare only those fish within 15 of each other in that order.

(d) After we first sort the fish on x-coordinate and y-coordinate (which takes time $O(n \log n)$), the running time $T(n)$ satisfies

$$T(n) = 2T(n/2) + O(n).$$

The reason is that we break up each problem of size n into two problems of size $n/2$. To combine them, we have to (1) scan through all the fish to create M (time $O(n)$); and (2) scan through the sorted version of M to compare all pairs within 15 of each other (also time $O(n)$).

The solution to this recurrence relation is $O(n \log n)$, by the Master Theorem.