Finding min and max concurrently

Suppose you have an array of n distinct numbers. Early in your time learning programming, you learned how to find the min or max of such an array. Suppose you wanted to find *both* – the min and max.

One way to do this would be to find the min; this takes $n-1$ comparisons. You could then output and delete the min element and find the max of what remains, taking $n-2$ comparisons, for a total of $2n-3$ comparisons.

Can you find a way to find both using *strictly fewer than* $2n-3$ *comparisons*? Note that we are measuring the actual number of comparisons, not the growth rate of your function.

If you are having trouble starting, you may assume n to be odd or even (your choice).

Follow-Up: Could any algorithm solve the warm-up problem in fewer comparisons than your solution uses?

Counting Inversions: Our First Divide and Conquer Algorithm

Related reading: G/T §8.1

Recall the definition of an *inversion* in an array: a pair of indices i, j are an *inverted pair* if $i < j$ and $A[i] > A[j]$. That is, an inverted pair is when the larger element of the pair appears earlier in the array.

The following is an $\Theta(n^2)$ time way to count the inversions in an array:

```
count = 0
for i = 1...n do
  for j = i+1...n do
    if A[i] > A[j] then
      count++
return count
```

The paradigm we will now cover is **Divide and Conquer** algorithms, whose associated problems can often be solved in polynomial time by brute force, but the technique can give us a more efficient solution.

Question 1. Now suppose you want to count the number of inverted pairs in an array A, but we also know that $A[1 \ldots \frac{n}{2}]$ is sorted, as is $A[\frac{n}{2} + 1 \ldots n]$. Can we use this information to count inverted pairs faster? *Hint:* Note that, in this case, sometimes finding one inverted pair reveals that other inverted pairs exist. You don't have to list every inverted pair, merely count how many exist.

Question 2. Can we use the algorithm from the previous question to count the number of inverted pairs in an *unsorted* array faster than $\Theta(n^2)$? Give your algorithm and demonstrate its running time.

Master Theorem

Reading: Goodrich/Tamassia §11.1.1

It is common for a divide-and-conquer algorithm's running time to have a recurrence relation of the following form:

$T(n) = aT(n/b) + f(n)$, for some $a \geq 1$, $b > 1$, and $f(n)$ is asymptotically positive.

1. If there is a small constant $\varepsilon > 0$ such that $f(n)$ is $\mathcal{O}(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$

2. If there is a constant $k \geq 0$, such that $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$

3. If there is a small constant $\varepsilon > 0$ such that $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is[1] $\Theta(f(n))$.

Using the Master Theorem

Use the Master Theorem to solve the following:

1. $T(n) = 4T(n/2) + n$
2. $T(n) = 2T(n/2) + n \log n$
3. $T(n) = T(n/3) + n$
4. $T(n) = 9T(n/3) + n^{2.5}$

Using the Master Method

After we cover the Master Method, consider doing these as extra practice.

5. $T(n) = 2T(n/2) + 1$
6. $T(n) = 2T(n/2) + n$
7. $T(n) = 2T(n/2) + n^2$
8. $T(n) = 2T(n/4) + 1$
9. $T(n) = 2T(n/4) + \sqrt{n}$
10. $T(n) = 2T(n/4) + n$
11. $T(n) = 9T(n/3) + n$
12. $T(n) = T(2n/3) + 1$
13. $T(n) = 3T(n/4) + n \log n$
14. $T(n) = 2T(n/4) + n^2$
15. $T(n) = 2T(n/4) + n^4$
16. $T(n) = T(7n/10) + n$
17. $T(n) = 16T(n/4) + n^2$
18. $T(n) = 7T(n/3) + n^2$
19. $T(n) = 7T(n/2) + n^2$

[1] Technically, it must also be the case that $af(n/b) \leq \delta f(n)$ for some constant $\delta < 1$ and for all sufficiently large n. I will not give you any recurrence relations in CompSci 161 that fail to meet this condition.

Selection Algorithms

Let's take a look at *selection* algorithms: the goal is to find the kth smallest element in an unsorted list. That is, the element that would be S_k when sorted.

`Select(`S`, `k`)`

If n is small, brute force and return.
Pick a random $x \in S$ and put rest into:
    L, elements smaller than x
    G, elements greater than x
if $k \leq |L|$ **then**

else if $k == |L| + 1$ **then**

else

Randomized QuickSelect

Randomized QuickSelect chooses x uniformly at random.

Question 3. How long does Randomized QuickSelect take in expectation? In the worst case?

Deterministic Selection

Deterministic Quick Select instead chooses its pivot value in this manner:

- Divide S into $g = \lceil n/5 \rceil$ groups
- Each group has 5 elements (except maybe g^{th})
- Find median of each group of 5
- Find median of those medians
- Use that median as **pivot value** x.

Question 4. What will the value x be if the following vector of size 45 is the input? The first ten elements are in the top row, the second ten are in the second row, and so on.

870	647	845	742	372	882	691	341	461	596
989	151	100	729	101	397	825	587	363	283
595	524	930	259	133	955	620	970	430	280
839	139	735	590	782	913	378	474	255	739
875	150	791	779	792					

Question 5. What fraction of the input is guaranteed to be less than the pivot value? What fraction will be larger? How many elements could be in one or the other? Why?

Question 6. Write a recurrence for the running time of Deterministic QuickSelect.

Question 7. How long does it take to find the median of any particular group of size five?

Question 8. How long does it take to find the median of those medians?

Question 9. Suppose the resulting pivot *is not* the element we desire. What is the largest size the vector upon which we make a recursive call can be?

100	283	255	133	341				
101	363	378	259	461				
151	397	474	524	596	620	735	742	791
				691	955	782	845	792
				882	970	839	870	875

Integer Multiplication

Reading: G/T §11.2

Given two n-bit integers X and Y, compute $X \times Y$. The algorithm you learned for this in grade school takes time $\mathcal{O}(n^2)$.

For our divide-and-conquer algorithm, we are going to divide X and Y each into their "higher order" and "lower order" bits first; X_H is the $n/2$ higher-order bits, and X_L is the lower-order bits.

Example If $X = 156 =$ `10011100` and $Y = 225 =$ `11100001`, then:

X_H	X_L	Y_H	Y_L
`1001`	`1100`	`1110`	`0001`

Note that $X = X_H \times 2^{n/2} + X_L$ and $Y = Y_H \times 2^{n/2} + Y_L$

Initial Algorithm Using algebra, we can see that

$$\begin{aligned} X \times Y &= (X_H \times 2^{n/2} + X_L) \times (Y_H \times 2^{n/2} + Y_L) \\ &= X_H \cdot Y_H \times 2^n + (X_H Y_L + X_L Y_H) \times 2^{n/2} + X_L Y_L \end{aligned}$$

Finish the Algorithm:

`Algorithm Mult(`X, Y`)`
  Create X_H, X_L, Y_H, Y_L
  A = Mult(X_H, Y_H)

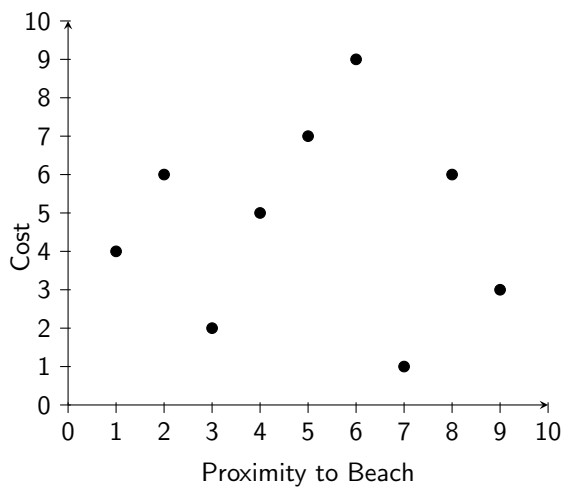
Question 10. That's four recursive calls, each of size $n/2$, plus some addition, which takes an additional $\mathcal{O}(n)$ time. Why isn't this a good algorithm for computing $X \times Y$? Can we do better?

Minima-Set Problem

Reading: Goodrich/Tamassia §11.4. We are given a set S of n points in the plane, we want to find the set of *minima* points. That is, if we include (x, y) in our output, we want to ensure that there is no point (x', y') in the output such that $x \geq x'$ and $y \geq y'$.

One way to think about it: suppose we have a database of hotels in which we can book rooms for our customers. A customer has, as their top two priorities, a hotel that is close to the beach and is inexpensive in cost. We can think of x as "proximity to the beach" and y as the cost for a room. We need to present a menu to choose from, since we don't know how the customer weighs these two objectives, but we know that when choosing between A and B, if A is further from the beach *and* is more expensive than B, the customer won't pick A.

Let's start the algorithm; this will look like many other Divide and Conquer algorithms you have seen. The algorithm, as printed in this handout, is *incomplete* – it is a good starting point, and we will finish the algorithm during the lecture.




```
MinimaSet(S)
  if n ≤ 1 then
    return S
  p ← median point in S by x-coordinate
  L ← points less than p
  G ← points greater than or equal to p
  M1 ← MinimaSet(L)
  M2 ← MinimaSet(G)
```

- Is $M_1 \cup M_2$ the correct return set?
  - If not, what could be incorrectly in there?
  - Are there any points that certainly belong in the output?
- How can we efficiently finish the divide-and-conquer?
- What is the resulting running time for the algorithm?