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 $i = 1 \ldots n$ **do**
  for $j = i + 1 \ldots n$ **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. $T(n) = 9T(n/3) + n^{2.5}$
5. $T(n) = 2T(\sqrt{n}) + \log n$

Using the Master Method

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

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

[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.

QuickSort

Related reading: G/T §8.2

The key subroutine in QuickSort is `partition`.

85	24	63	45	17	31	96	50

```
QuickSort(A, start, end)
```

if start $<$ end **then**
  q = partition(A, start, end)
  QuickSort(A, start, $q-1$)
  QuickSort(A, $q+1$, end)

Average Case Analysis of QuickSort. Suppose:

- All permutations are equally likely
- All n values are distinct (for simplicity)
- Define $S_1, S_2, \ldots S_n$ as sorted order.

Question 3. What is the probability we compare S_i and S_j?

Question 4. What is the expected number of comparisons in a run of QuickSort, under the assumptions above? Why?

Reinforcement: Sort the following array by QuickSort.

87	31	30	22	20	85	86	15	38	60	57	72	41	52	50	67	69	3	65	42

Selection Algorithms

Let's take a look at *selection* algorithms: the goal is to find the k^{th} 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

Two Algorithms

Randomized QuickSelect chooses x uniformly at random.

Question 5. How long does Randomized QuickSelect take in expectation? In the worst case?

Deterministic Quick Select instead does this:

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

Question 7. Write a recurrence for the running time of Deterministic QuickSelect.

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 8. 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?

Strassen's Algorithm (Time Permitting)

Reading: G/T §11.3. In algebra, you saw an algorithm to multiply two $n \times n$ matrices,

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} I & J \\ K & L \end{bmatrix}$$

- $I = AE + BG$
- $J = AF + BH$
- $K = CE + DG$
- $L = CF + DH$

If the matrices are 2×2, then $A \dots L$ are elements directly and we can compute each as per the right-hand side. If they are not, we can form the product matrix anyway. Each entry in the product matrix is the result of a dot product of the appropriate row from one matrix and column from the other. Because each dot product takes $\Theta(n)$ to compute, the end result is a $\Theta(n^3)$ time brute force algorithm.

Alternately, we can use a divide and conquer algorithm by treating each $n/2 \times n/2$ quadrant as a matrix and performing matrix multiplication instead of scalar multiplication.
Question 9. What is the running time of the second approach?

Question 10. Adding two matrices takes $\Theta(n^2)$ time. There's also another reason we should not expect to find an algorithm that isn't $\Omega(n^2)$ to solve this problem. What is it?

Strassen's Algorithm computes the resulting matrix in a different way than the straight-forward approach described above.

First, compute $S_1 \dots S_7$:

- $S_1 = A(F - H)$
- $S_2 = (A + B)H$
- $S_3 = (C + D)E$
- $S_4 = D(G - E)$
- $S_5 = (A + D)(E + H)$
- $S_6 = (B - D)(G + H)$
- $S_7 = (A - C)(E + F)$

Second, compute I, J, K, L:

$$\begin{aligned} I &= S_5 + S_6 + S_4 - S_2 \\ &= (A + D)(E + H) + (B - D)(G + H) \\ &\quad + D(G - E) - (A + B)H \\ &= AE + BG \end{aligned}$$

$$\begin{aligned} J &= S_1 + S_2 \\ &= A(F - H) + (A + B)H \\ &= AF - AH + AH + BH \\ &= AF + BH \end{aligned}$$

Similarly :

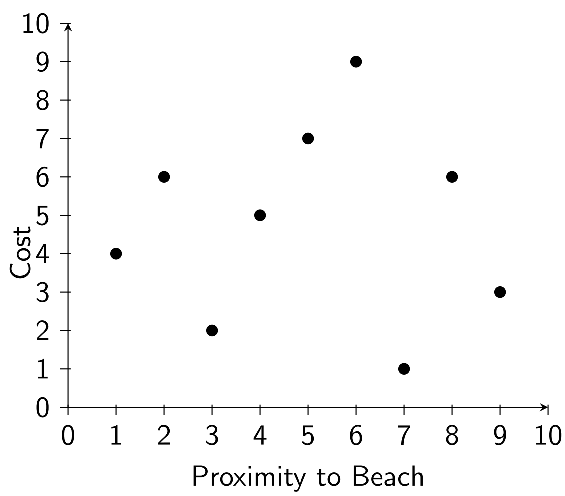
$$\begin{aligned} K &= S_3 + S_4 \\ L &= S_1 - S_7 - S_3 + S_5 \end{aligned}$$

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.



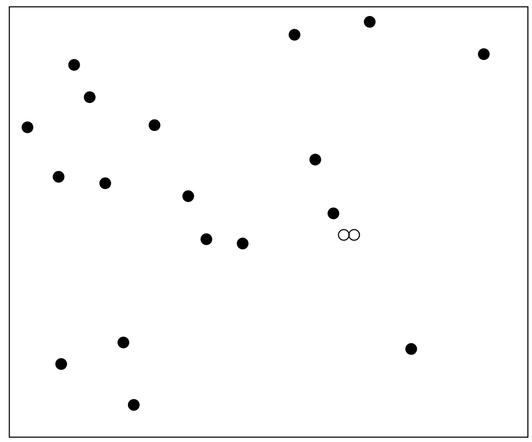
`MinimaSet(`S`)`
  if $n \leq 1$ **then**
    return S
  $p \leftarrow$ median point in S by x-coordinate
  $L \leftarrow$ points less than p
  $G \leftarrow$ points greater than **or equal to** p
  $M_1 \leftarrow$ `MinimaSet(`L`)`
  $M_2 \leftarrow$ `MinimaSet(`G`)`

- 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?

Closest Pair of Points

Reading: Goodrich/Tamassia §22.4. Suppose we have n points, each of which has an x-coordinate x_i and a y-coordinate y_i. Our goal is to find the pair of points p_i and p_j that are closest together. The distance between two points is $d(p_i, p_j)$.

Here is a Brute-Force approach to this problem:



`Closest-Pair`
Input: n points in $2D$-space
Output: The closest pair of points.

min $= \infty$
for $i = 2 \to n$ **do**
  for $j = 1 \to i-1$ **do**
    if $(x_j - x_i)^2 + (y_j - y_i)^2 <$ min **then**
      min $= (x_j - x_i)^2 + (y_j - y_i)^2$
      closestPair $= ((x_i, y_i), (x_j, y_j))$
return closestPair

What is the running time of this algorithm?

To improve on the running time of the brute-force algorithm, we can try to set up our usual start for divide and conquer. For convenience, let's assume the points are sorted by y-coordinate before we first call this algorithm. We can do this in $\mathcal{O}(n \log n)$ time first; if the eventual running time is $\Omega(n \log n)$, this won't matter, and if we achieve $o(n \log n)$ for the rest of the algorithm, this will dominate the running time.

`Closest-Pair`
Input: n points in $2D$-space
Output: The closest pair of points.

If P is sufficiently small, use brute force. // $\mathcal{O}(1)$
$x_m \leftarrow$ median x-value from P
$L \leftarrow$ any points from P with x-coordinate $\leq x_m$
$R \leftarrow$ any points from P with x-coordinate x_m
Let l_1 and l_2 be the closest pair of points in L, found recursively.
Let r_1 and r_2 be the closest pair of points in R, found recursively.
return whichever pair is closer together // Incorrect but good starting point.

The above algorithm is clearly incorrect; why?

How do we fix it?

How do we fix it while having a better running time than the brute force algorithm?

Finding min and max concurrently

:

Suppose you have an array of n distinct numbers. In the ICS 30-series, 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?