CpSc 8400: Design and Analysis of Algorithms

Instructor: Dr. Brian Dean — Spring 2016
Webpage: `http://www.cs.clemson.edu/~bcdean/` — TTh 12:30-1:45
Handout 5: Homework #2 Solutions — McAdams 119

2-1. Practice Solving Recurrences. The solutions to these recurrences are as follows:

(a) $T(n) = \Theta(n^2)$

(b) $T(n) = \Theta(n^{\log_4 5})$

(c) $T(n) = \Theta(n^{\log_2 3})$

(d) $T(n) = \Theta(n^2 \log n)$

(e) $T(n) = \Theta(n \log^3 n)$

(f) $T(n) = \Theta(n^{62} \log^{37} n)$

(g) $T(n) = \Theta(\log \log n)$

For the last part, the easiest way to arrive at the solution is to realize that taking the square root of n halves the number of bits required to represent n in binary (initially $\log_2 n$), so we can repeat this process at most $O(\log \log n)$ times before reducing n down to a constant. We could formalize this idea mathematically by letting $S(n) = T(2^n)$, so $S(n) = S(n/2) + 1$, which solves to $S(n) = \Theta(\log n)$. Since $T(n) = S(\log n)$, we therefore have $T(n) = \Theta(\log \log n)$.

2-2. Sorting Fractions. Since we know how to sort small integers efficiently, we attempt to transform this problem into an equivalent problem involving small integers. Note that for any two different fractions $\frac{a_i}{b_i} \neq \frac{a_j}{b_j}$, their difference $\left|\frac{a_i}{b_i} - \frac{a_j}{b_j}\right| = \frac{|a_i b_j - a_j b_i|}{b_i b_j}$ is at least $1/n^{2c}$ (since the numerator is at least 1, and the denominator is at most $n^c \cdot n^c$). Hence, if we scale up each fraction by a factor of n^{2c}, any pair of different fractions will differ by at least 1. By setting $x_i = \lfloor n^{2c} a_i / b_i \rfloor$, we obtain a set of integers $x_1 \ldots x_n$ whose sorted ordering is therefore equivalent to that of the original set of fractions. Since the x_i's are all in the range $0 \ldots n^{3c}$, we can sort them using radix sort in $O(n)$ time.

2-3. In-Place Merge Sort. There are several valid solutions for this problem, all of which use the key observation that we can merge sort any subarray X in place as long as there still exists a "scratch" subarray Y of length equal to that of X containing yet-unsorted elements. This is done by swapping the contents X and Y (which is easy to do in place), then by recursively sorting the first and second halves of Y in place (which we can do by induction, owing to the fact that X provides sufficient scratch space), and finally by merging back into X (swapping the elements of X

back into Y as X is filled up). The following is now probably the simplest solution to the in-place merge sort problem: (1) apply merge sort to the second half of our main array, using the first half as scratch space ($\Theta(n \log n)$ time), (2) *recursively* sort the first half in-place ($T(n/2)$ time), then (3) apply an $\Theta(n \log n)$ stable in-place merge. The running time of this approach is described by the recurrence $T(n) = T(n/2) + \Theta(n \log n)$, which solves to $T(n) = \Theta(n \log n)$.

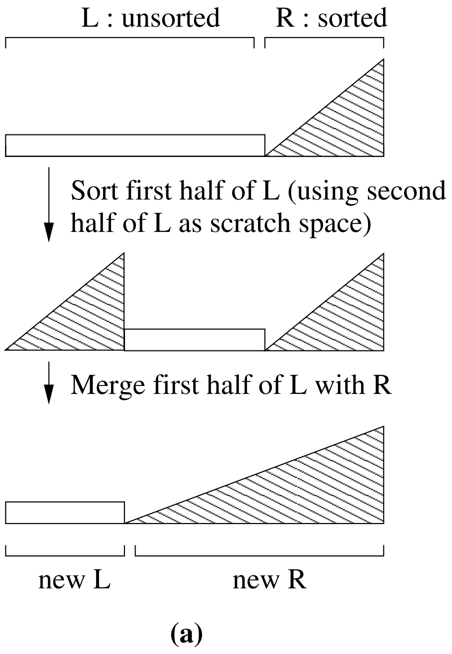
An alternative solution follows a series of phases of the form shown in Figure 1(a). We mentally partition our array into two pieces L (unsorted) and R (sorted), with initially L being the entire array, and R being empty. Each phase increases the size of R by halving the size of L. We do this by merge sorting the first half of L, using the second half of L as scratch space. We then merge the first half of L with R, merging into the space currently occupied by the second half of L plus R. As we merge into this space, we swap the unsorted elements from the second half of R out as before, and these will ultimately end up situated (in haphazard order) in the first half of L. Observe that this process merges properly despite the fact that one of the input arrays (R) starts out in the same memory space into which we are merging — this follows from the fact that it is impossible for the pointer at which we are writing the merged contents of our two input arrays to ever overtake the pointer to the first effective element of R (i.e., R is consumed fast enough that the output of the merge will never "collide" with R). Since L decreases in size by a factor of two in each phase, there are clearly only $O(\log n)$ phases. The running time of this approach consists of the time spent merging, which is linear in each phase ($O(n \log n)$ total) plus the time spent on calls to merge sort. Over the entire algorithm, we are calling merge sort on problems of sizes $x_1, \ldots, x_{k=O(\log n)}$ where $\sum_i x_i = n$. The running time required for all of these calls to merge sort is therefore

$$\Theta(\sum_i x_i \log x_i) = O(\sum_i x_i \log n) = O(n \log n).$$

2-4. Counting Distant Pairs of Points. We use divide and conquer. We divide the $n \times n$ 2D space under consideration into four vertical strips of size $n/4 \times n$; let us label these A, B, C, and D, as shown in Figure 1(b). Observe that points in A can only by distant with points in C or D, and points in B can only be distant with points in D. This gives us three distinct subproblems, which we call AC (counting pairs of distant points where one comes from A and one comes from C), BD, and AD.

We solve subproblems AC and BD recursively. Taking AC as an example, we loop over all the points in our matrix and extract just the points in A and C. Then we build an $n/2 \times n$ problem instance by placing strips A and C next to each-other, and we recursively compute distant points, where now the "x" criteria for being distant has halved from n to $n/2$ (since all pairs of points that are distant under this condition are distant in the original problem).

We solve subproblem AD in $O(p)$ time, where p denotes the number of points in our instance. All pairs of points where one comes from A and the other from D satisfy the "x" condition for being distant, leaving us with essentially a one-dimensional problem involving just the y dimension. Assuming we have sorted all the points by their y coordinates as a preprocessing step, we build an array A containing the y-sorted points from strip A, and an array D containing the y-sorted points from strip D. We then scan two pointers i and j simultaneously down both arrays, so that point $A[i]$ and point $D[j]$ are always separated by at least $n/2$ units of distance in the y dimension (i.e., every time we advance i, we move j forward until this condition is again satisfied). For each value of i during the scan, the points ahead of the pointer j are exactly those that are distant from $A[i]$ with larger y coordinates, so it is easy to count all these during the scan. We then repeat the process in reverse, to count distant points with smaller y coordinates.


L : unsorted
R : sorted
Sort first half of L (using second half of L as scratch space)
Merge first half of L with R
new L
new R
(a)

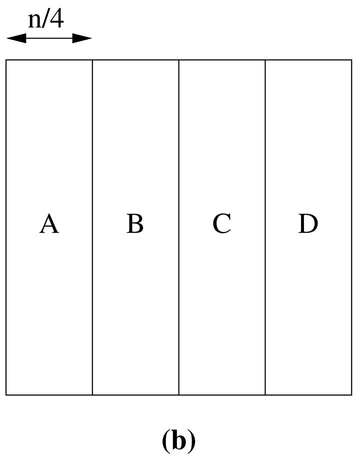


Figure 1: (a) One phase of in-place merge sort, and (b) the divide-and-conquer structure of the solution for counting distant pairs of points.

The "interleaved" structure of the subproblem decomposition was perhaps the most challenging aspect of this problem. A nice way to visualize the structure of this divide and conquer approach is shown in Figure 2. In (a), we have drawn an $n \times n$ matrix in the (i, j) entry indicates the subproblem of checking all pairs of distant points where one has x coordinate i and the other has x coordinate j. All pairs (i, j) of subproblems that need to be considered are shown as the shaded triangle. To solve these as quickly as possible, our algorithm first solves the "AD" subproblem in $O(n)$ time, which encompasses all subproblems in the large dark square shown in (b). The next level of recursion requires $O(n)$ total time and corresponds to the two slightly smaller squares shown in (c). Again, the next level of recursion takes $O(n)$ total time and consists of the four smaller squares shown in (d), and so on. The total recursion depth is $O(\log n)$, and we spend $O(n)$ time per level of recursion, for $O(n \log n)$ total time.

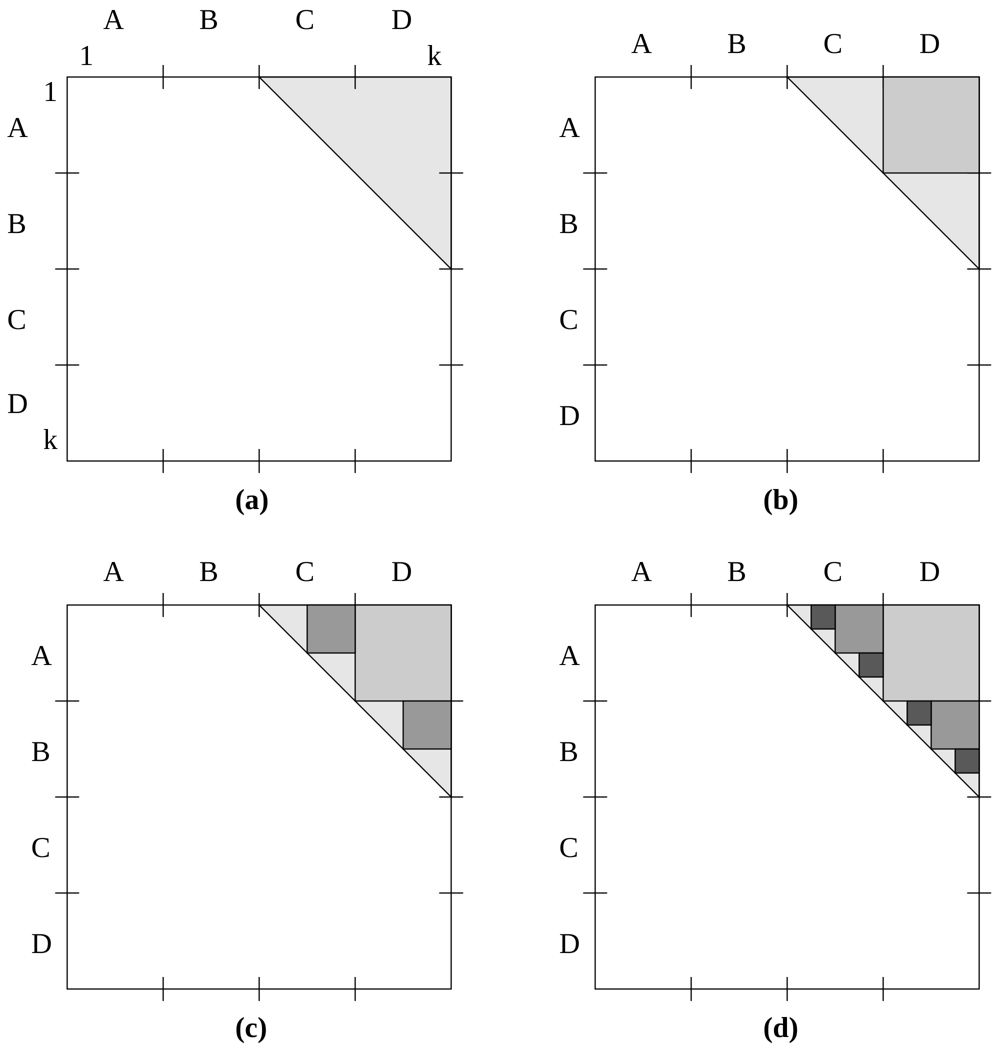


Figure 2: Visualization of the divide and conquer approach for the distant pairs of points problem.