Supplment to *A Practical Guide to Data Structures and Algorithms Using Java*

Divide-and-Conquer Algorithms

Sally A. Goldman and Kenneth J. Goldman *Handout 1*

Divide-and-conquer algorithms use the following three phases:

1. **Divide** the problem into smaller *subproblems*. A subproblem of a problem is a smaller input for the same problem. For example, for the problem of computing a closest pair of points, in a subproblem there will be fewer points but the task is still to find a closest pair of points. Generally, the input for the subproblem is a subset of the original input, but that need not be the case.

2. **Conquer** by recursively solving the subproblems (unless the problem size has reached a specified termination condition).

3. **Combine** the solutions to the subproblems to obtain the solution for the original problem.

To design a diode-and-conquer algorithm, you need to think about what subproblem solution would be helpful in computing the answer to the original problem. While many students initially feel like the "magic" is in the recursion, that step is one that can always be done. The real magic is in the combining, and finding the right subproblems to recursively solve. We now describe two sample divide-and-conquer algorithms.

Mergesort

The problem considered here is to sort an array a of n comparable elements $(a[0], \ldots, a[n-1])$ to be non-decreasing. Mergesort, covered in Section 11.4.2 of the text, is one of many algorithms that solve this problem. Let's look at how the divide-and-conquer steps are performed.

Divide - Split the elements to sort into two equal halves. To avoid copying portions of the array, typically the portion of the array being sorted is marked using a start index p and an end index r. The portion of the array being considered is $a[p], \ldots, a[r]$. So for the initial problem $p = 0$ and $r = n - 1$. Let $q = \lfloor (p + r)/2 \rfloor$. Then the problem of sorting $a[p], \ldots, a[r]$ is divided into two subproblems: sort $a[p], \ldots, a[q]$ and sort $a[q + 1], \ldots, a[r]$.

Conquer - If either subarray to sort has size 1 then just return. Otherwise, recursively sort the subarray.

Combine - Merge the two sorted subarrays. An important component of this algorithm is that merging two sorted subarrays is asymptotically more efficient than sorting the original array. In particular, there is a simple linear time algorithm to merge two sorted arrays. The details can be found on page 147 in the text within the `mergesortImpl` method. We just cover the intuition here. A supplemental array is used to hold the sorted array, which can then be moved back into $a[p], \ldots, a[r]$. An index of the current location (initially p) for the left half is maintained, an index of the current location (initially $q + 1$) of the right half is maintained, and an index of the current location into the supplemental array is maintained. Then the element at the current locations of the left and right halves are compared. The smaller of these elements (or either if equal) are moved into the next open location in the supplemental array. Then the index of the array from which the element was moved is incremented and the index into the supplemental array is implemented. Since constant time is used at each step, and there is one step for each of the elements in the portion of a being sorted, this merge algorithm has linear worst-case asymptotic time complexity.

For any given problem there can be more than one divide-and-conquer algorithm. Later in this course, we will study the quicksort algorithm which is an alternate divide-and-conquer sorting algorithm that spends linear time dividing the array into subarrays so that no computation is needed to combine.

Divide-and-Conquer Closest Pair of Points Algorithm

We assume that we have a point class, and that `points` is an array of n references to points. We assume that the point class includes the methods:

- `dist(q)` which returns the Euclidean distance between this point and point q.
- `leftOf(q)` which returns `true` when this point is left of point q. This method is guaranteed to order any points that share the same x-coordinate so as to define a unique total ordering among the points with respect to the x-coordinate. By convention, a point is not left of itself.
- `below(q)` which returns `true` when this point is below point q. As with `leftOf` below defines a unique total ordering among the points with respect to the y-coordinate. By convention, a point is not below itself.

The first step of the divide-and-conquer algorithm is a pre-processing step that both sorts `points` according the the x coordinate, and also sorts `points` according to the y coordinate. More specifically, let `ptsByX` be an array of references to the points that is sorted using `leftOf` as the comparator, and let `ptsByY` be an array of references to the points when sorted using `below` as the comparator. **An important invariant that must be maintained is that for any subproblem that `ptsByX` and `ptsByY` are permutations of the same set of points.**

We now describe the divide-and-conquer algorithm that takes `ptsByX` and `ptsByY`, and returns the distance between a closest pair of points. Let n be the number of points in `ptsByX` (which is required by the invariant to also be the number of points in `ptsByY`).

Divide - Split the set of points into a left half and a right half based on `ptsByX`. Let x_L be the x-coordinate of the rightmost point from the left half, and let x_R be the x-coordinate of the leftmost point from the right half. As part of the divide phase, the array of references `ptsByXLeft`, `ptsByXRight`, `ptsByYLeft`, and `ptsByYRight` must be created. This portion of the algorithm can be performed in linear time. Think about how that can be done.

Conquer - The simplest termination condition is to return ∞ when $n = 1$, and return the distance between the two points when $n = 2$. When $n > 2$, the two subproblems are recursively solved. Let d_L be the distance between the closest pair with input `ptsByXLeft`, `ptsByYLeft`, and let d_R be the distance between the closest pair with input `ptsByXRight`, `ptsByYRight`.

Combine - First compute $d = \min(d_L, d_R)$. Next create and construct an array `yStrip` so that it contains all points ordered as in `ptsByY` such that their x-coordinate is greater than $x_R - d$ and less than $x_L + d$. (See Figure 1.) Let `numInStrip` be the number of points in `yStrip`.

Finally, for each point p in `yStrip`, the distance between p and the next point q in `yStrip` is computed. If the distance between p and q is less than d, then d is updated to this distance. Once the difference in y-coordinates between p and q reaches d, then the distance between p and all remaining points in `yStrip` must be at least d, so p need not be considered any further. The following loop shows this final step of the combining more explicitly where `p.y` is the y coordinate of point p.

```
bfor (i = 0; i < numInStrip - 1; i++) {
    j = i+1;
    while (j < numInStrip && yStrip[j].y - yStrip[i].y < d) {
       d = min(d, yStrip[j].dist(yStrip[i]));
       j++;
    }
}
```

Finally d can be returned as the distance between the closest pair of points to the given input.

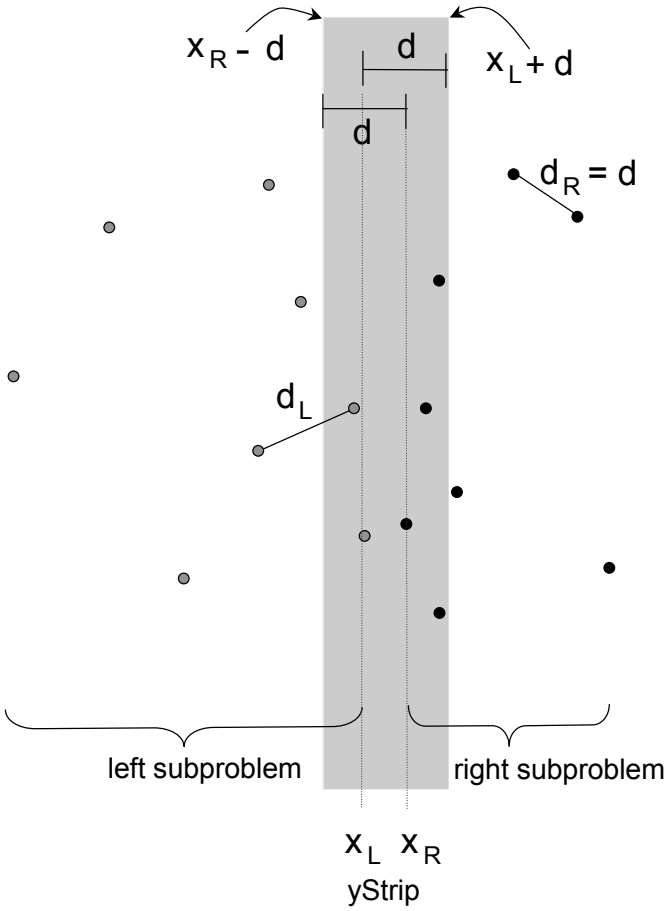

Figure 1: An illustration of the set up for the divide-and-conquer closest pairs algorithm.

Correctness

We now argue that the divide-and-conquer closest pair algorithm described in the last section always yields the correct answer. Let d_{opt} be the distance between the closest pair of points in the original input P. Let P_L be the points in the left subproblem, and let P_R be the points in the right subproblem. We use the following facts:

1. $d_{\text{opt}} \leq \min(d_L, d_R)$
2. If $d_{\text{opt}} < \min(d_L, d_R)$ then d_{opt} is the distance between a pair of points $p_1 \in P_L$ and $p_2 \in P_R$.
3. For all points $p_1 \in P_L$, any point $p_2 \in P_R$ that is not in `yStrip` has a distance from p_1 of at least d. The reason for this is that the difference in x-coordinates is at least d and so the distance must be at least d.
4. For all points $p_2 \in P_R$, any point $p_1 \in P_L$ that is not in `yStrip` has a distance from p_2 of at least d. Again, since the difference in x-coordinates is at least d and so the distance must be at least d.

Formally, the correctness argument uses induction on the number of recursive calls made. The base case is when $n = 1$ (for which ∞ is a correct answer) or when $n = 2$ (for which case the distance between the two points is the correct answer). We now focus on the inductive step. By the inductive hypothesis d_L is the distance between a closest pair of points in P_L and d_R is the distance between a closest pair of points in P_R. If $d_{\text{opt}} = d_L$ then the correct answer is returned since d is set to $\min(d_L, d_R)$ and will only be changed if a closer pair is found. Likewise, if $d_{\text{opt}} = d_R$ then the correct answer is returned.

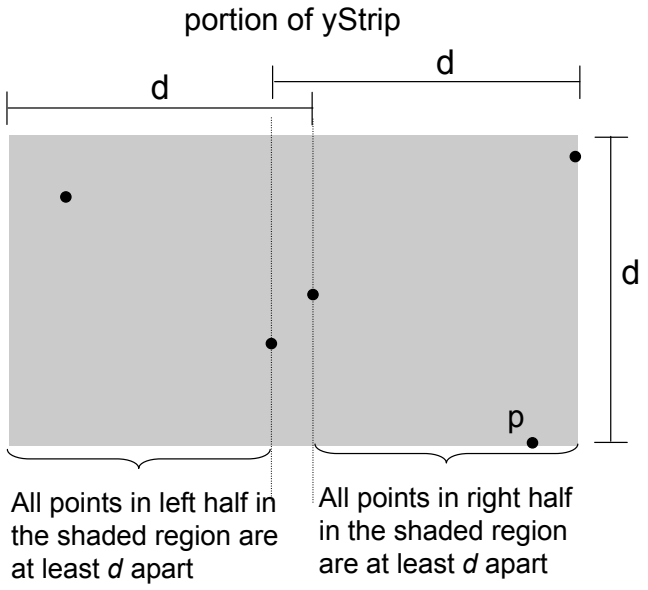

Figure 2: The region that holds all points for which a distance computation is performed when p is being considered.

We now consider when $d_{\text{opt}} < \min(d_L, d_R)$. From facts 2, 3, and 4 above it follows that this can only occur if d_{opt} is the distance between points $p_1 \in P_L \cap \texttt{yStrip}$ and $p_2 \in P_R \cap \texttt{yStrip}$. So all we have left to prove is that the distance between points p_1 and p_2 is computed in the combine step. Observe that since the points in `yStrip` are sorted by y-coordinate, once the difference in y-coordinates is $\geq d$ it will remain $\geq d$ for the remaining points in `yStrip`. Since the difference in y-coordinates is $\geq d$ so is the distance. Thus each point in `yStrip` is compared to all points above it with a difference of y-coordinate less than d. Since p_1 and p_2 have a distance less than d, their difference in y-coordinate must be less than d. Thus distance between p_1 and p_2 will be computed and after that update to $d = d_{\text{opt}}$ and cannot be changed further since there is no closer pair.

Analysis

Let n be the size of the original problem, and let $T(s)$ be the worst-case time complexity to solve a problem of size s. Let $f_1(s)$ be the worst-case time for the divide step for a problem of size s, and let $f_2(s)$ be the worst-case time for the combine step for a problem of size s.

For the mergesort algorithm $f_1(s)$ is a constant (i.e., independent of s) and $f_2(s)$ is linear (that is $f_2(s) \leq c \cdot s$ for some constant c).

When properly implemented, for the divide-and-conquer closest pair algorithm, $f_1(s)$ is linear. We now argue that for each point p considered in the combining step, the number of distance computations made for it is constant. Figure 2 illustrates all points considered in the while loop when $p =$ `yStrip[i]`. It is easily shown that at most 7 such points can fit in the shaded region. Since constant computation is used for each point, $f_2(s)$ is also linear.

Thus for both mergesort and the divide-and-conquer closest pair algorithm, $f_1(s) + f_2(s) \leq c \cdot s$ for some constant c. Our goal is to compute $T(n)$, the asymptotic time complexity for each of these algorithms for an input of size n. Unlike the brute-force closest pair algorithm, it is less clear how to compute the overall time complexity because it is not known how long each recursive call takes. However, observe that for the original problem of size n, $T(n/2)$ is the asymptotic time complexity for solving the left subproblem and $T(n/2)$ is the asymptotic time complexity for solving the right subproblem. So we can add together both the cost of dividing and combining ($f_1(s) + f_2(s)$) with the cost for the

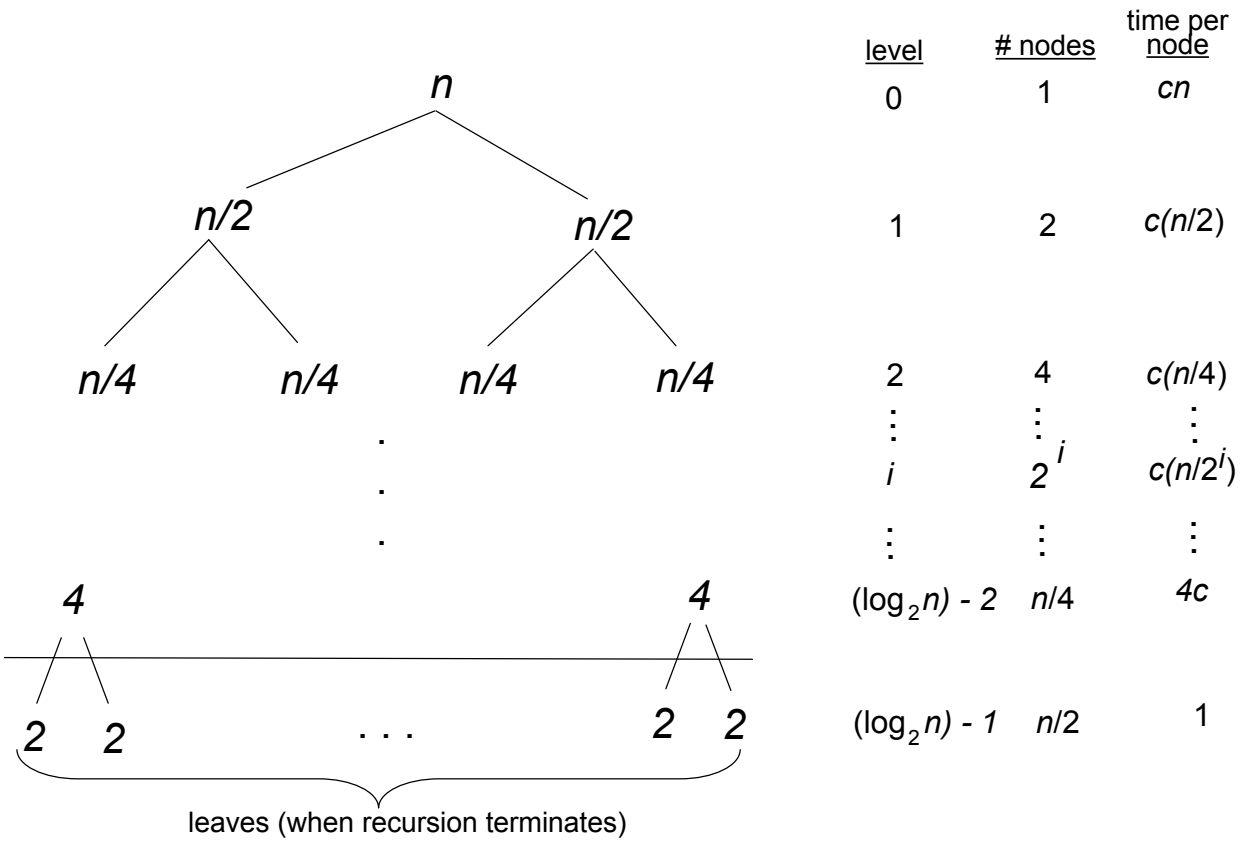

Figure 3: A recursion tree for the recurrence equation $T(n) = 2T(n/2) + cn$ for $n > 2$ and $T(2) = 1$.

recursive calls $(T(n/2) + T(n/2)) = 2T(n/2)$) to create the following **recurrence equation**

$$\begin{aligned} T(n) &= 1 \text{ for all } n \leq 2 \\ T(n) &= 2T(n/2) + cn \text{ for all } n > 2 \end{aligned}$$

since it takes one statement (or if you'd prefer constant time) when the termination condition is reached, and $2T(n/2)+cn$ time otherwise.

Appendix B.6 of the text gives a "cook-book method" to solve this Crecurrence. Here, we will solve it from scratch using a **recursion tree** under the assumption that n is a power of 2. In a recursion tree we unfold the recursion to show (abstractly) every recursive call made. In computing the time complexity, the important thing about each recursive call is the problem size, so that is all we show in Figure 3.

The total time spent for all recursive calls (except the termination steps) is the sum of the divide/combine times for each subproblem with size greater than 2. For the original problem of size n, cn time is spent doing the divide and combine steps. There are 2 subproblems of size $n/2$ and each spend $cn/2$ time doing the divide and combine steps. Thus the total divide/combine time for the problems of size $n/2$ is cn. Likewise, there are 4 problems of size $n/4$ each with a divide/combine time $cn/4$ for a total time of cn, and so on. Observe that the sequence $4, 8, \ldots, n$ has $(\log_2 n)-1$ elements in it. So the total time spent performing the divide and combine steps across the entire execution is $cn(\log_2 n - 1)$. Finally there are $n/2$ pairs considered in the termination step, for which $n/2$ statements are executed. So the total number of statements executed is at most $cn \log_2 n - cn + n/2 = cn \log_2 n - n(c - 1/2)$ which grows asymptotically as $n \log_2 n$. Finally, for the closest-pair algorithm by using mergesort for the pre-processing the total time for the entire algorithm grows asymptotically as $n \log_2 n$.