the divide-and conquer

Merge Sort, Quick Sort, Closest Pair, Convex Hull, Knapsack

Main Idea

- best-known general algorithm design technique.
- quite a few very efficient algorithms are specific implementations of this general strategy.
- Divide-and-conquer algorithms work according to the following general plan:
  1. A problem is divided into several subproblems of the same type, ideally of about equal size.
  2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
  3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

Running Time

- most typical case of divide-and-conquer a problem's instance of size n is divided into two instances of size n/2
- An instance of size n can be divided into b instances of size n/b, with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$.)
- Assuming that size n is a power of b to simplify our analysis, we get the following recurrence for the running time T (n):
- $T(n) = aT(n/b) + f(n)$,
- where f (n) is a function that accounts for the time spent on dividing an instance of size n into instances of size n/b and combining their solutions.
- Obviously, the order of growth of its solution T (n) depends on the values of the constants a and b and the order of growth of the function f (n).

Master Theorem

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Merge-Sort

- Mergesort is a perfect example of a successful application of the divide-and conquer technique.
- It sorts a given array A[0..n − 1] by dividing it into two halves
- A[0..n/2 − 1] and A[n/2..n − 1], sorting each of them recursivly, and then merging the two smaller sorted arrays into a single sorted one.

Merge Sort

ALGORITHM *Mergesort*$(A[0..n-1])$

```
//Sorts array A[0..n − 1] by recursive mergesort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
if n > 1
    copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
    copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
    Mergesort(B[0..⌊n/2⌋ − 1])
    Mergesort(C[0..⌈n/2⌉ − 1])
    Merge(B, C, A)    //see below
```

Merge Sort

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

```
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B and C
i ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ← C[j]; j ← j + 1
    k ← k + 1
if i = p
    copy C[j..q − 1] to A[k..p + q − 1]
else copy B[i..p − 1] to A[k..p + q − 1]
```

Example and Running time

Quick Sort

- quicksort divides them according to their value
- array partition
- A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it

Quick Sort

ALGORITHM *Quicksort*$(A[l..r])$

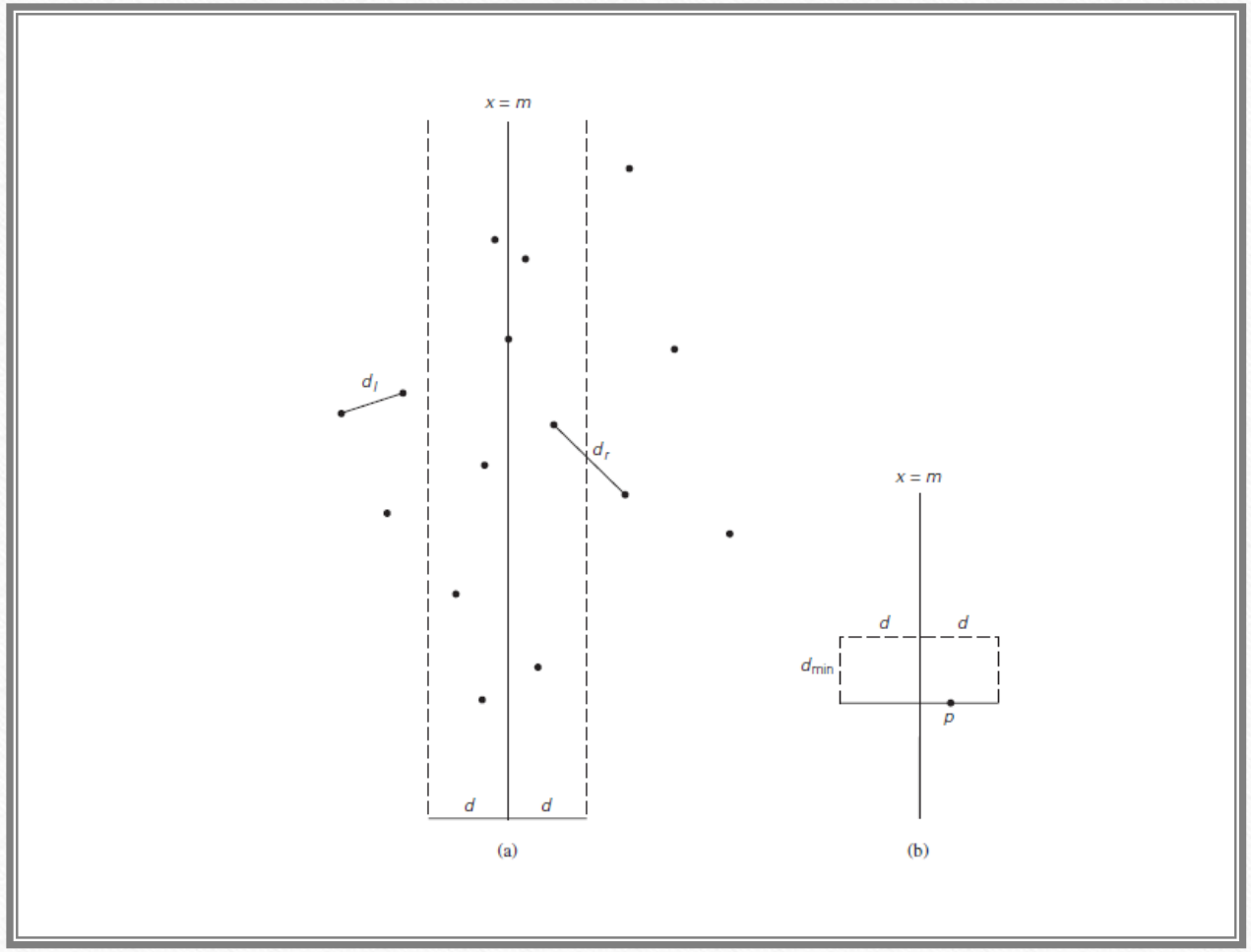
```
//Sorts a subarray by quicksort
//Input: Subarray of array A[0..n − 1], defined by its left and right
//         indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order
if l < r
    s ← Partition(A[l..r])  //s is a split position
    Quicksort(A[l..s − 1])
    Quicksort(A[s + 1..r])
```

Example

Closest Pair

- Let P be a set of $n > 1$ points in the Cartesian plane.
- Assume that the points are ordered in nondecreasing order of their x coordinate.
- It will also be convenient to have the points sorted in a separate list in nondecreasing order of the y coordinate; we will denote such a list Q.
- If $2 \leq n \leq 3$, the problem can be solved by the obvious brute-force algorithm.
- If $n > 3$, we can divide the points into two subsets Pl and Pr of $n/2$ and $n/2$ points, respectively, by drawing a vertical line through the median m of their x coordinates so that $n/2$ points lie to the left of or on the line itself, and $n/2$ points lie to the right of or on the line.
- Let dl and dr be the smallest distances between pairs of points in Pl and Pr , respectively, and let

  $d = \min\{dl, dr\}$



Closest Pair

- Note that d is not necessarily the smallest distance between all the point pairs because points of a closer pair can lie on the opposite sides of the separating line.
- Therefore, as a step combining the solutions to the smaller subproblems, we need to examine such points.
- Let *S* be the list of points inside the strip of width 2*d* around the separating line
- encounter a closer pair of points. Initially, *d*min = *d,* and subsequently *d*min ≤ *d.*

Closest Pair

ALGORITHM *EfficientClosestPair*(P, Q)

```
//Solves the closest-pair problem by divide-and-conquer
//Input: An array P of n ≥ 2 points in the Cartesian plane sorted in
//       nondecreasing order of their x coordinates and an array Q of the
//       same points sorted in nondecreasing order of the y coordinates
//Output: Euclidean distance between the closest pair of points
if n ≤ 3
    return the minimal distance found by the brute-force algorithm
else
    copy the first ⌈n/2⌉ points of P to array P_l
    copy the same ⌈n/2⌉ points from Q to array Q_l
    copy the remaining ⌊n/2⌋ points of P to array P_r
    copy the same ⌊n/2⌋ points from Q to array Q_r
    d_l ← EfficientClosestPair(P_l, Q_l)
    d_r ← EfficientClosestPair(P_r, Q_r)
    d ← min{d_l, d_r}
    m ← P[⌈n/2⌉ − 1].x
    copy all the points of Q for which |x − m| < d into array S[0..num − 1]
    dminsq ← d^2
    for i ← 0 to num − 2 do
        k ← i + 1
        while k ≤ num − 1 and (S[k].y − S[i].y)^2 < dminsq
            dminsq ← min((S[k].x − S[i].x)^2 + (S[k].y − S[i].y)^2, dminsq)
            k ← k + 1
return sqrt(dminsq)
```

Programming Assignment

- Implement closest pair algorithm using divide and conquer.
- 1) your code should take care of special case when x-ordering cannot split the given set into 2 non-empty sets, like in(1,0) (1,1) (1,2)all 3 points are on one vertical line, so x-ordering (by vertical line) cannot split the set. Switch to y-ordering (by horizontal line).Notice that when neither of the 2 ordering can split, you have a special case:all points in the set are equal - thus distance is 0.
- To submit:closestpair.cpp
- Deadline: 11-13-2017

Convex Hull

- General approach
- Quick Hull

General Approach

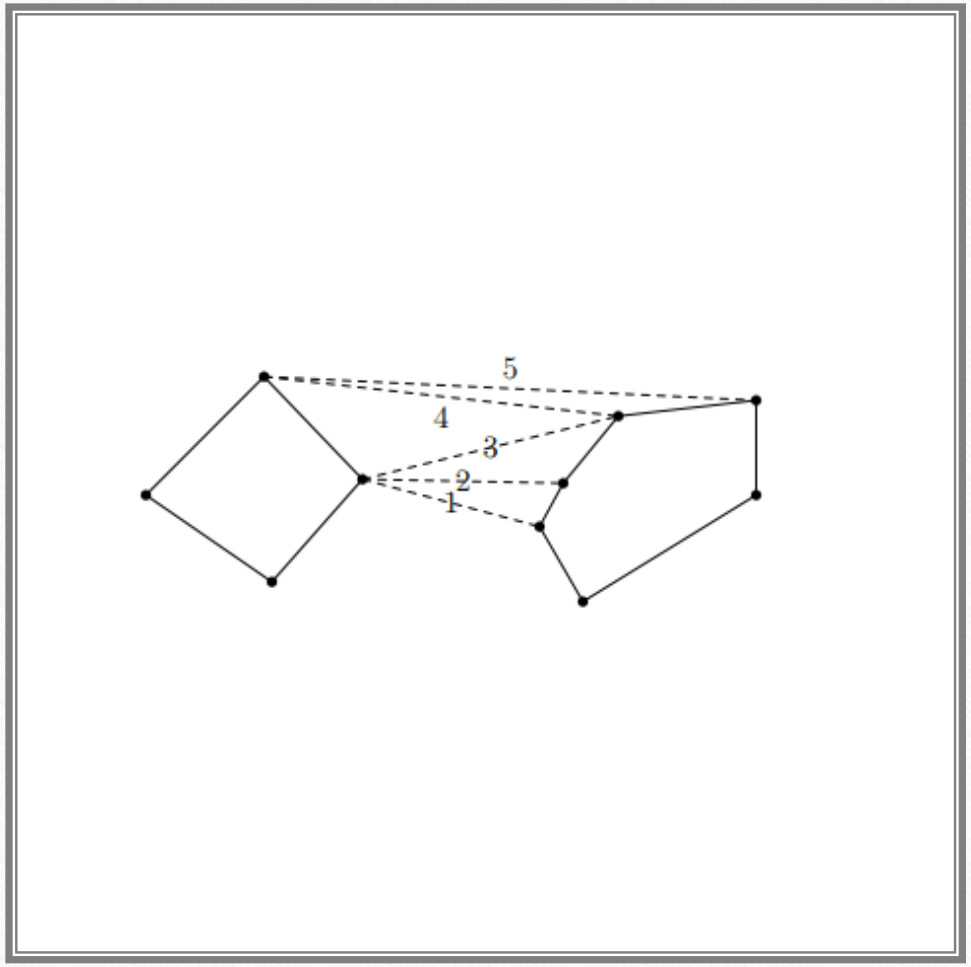
1. Divide the n points into two halves.
2. Find convex hull of each subset.
3. Combine the two hulls into overall convex hull.

Combining two hulls

- Convex hulls should not overlap. To ensure this, all the points are presorted from left to right. So we have a left and right half and hence a left and right convex hull.
- Define a bridge as any line segment joining a vertex on the left and a vertex on the right that does not cross the side of either polygon. What we need are the upper and lower bridges.
- The following produces the upper bridge:

Combining two hulls

1. Start with any bridge. For example, a bridge is guaranteed if you join the rightmost vertex on the left to the leftmost vertex on the right.
2. Keeping the left end of the bridge fixed, see if the right end can be raised. That is, look at the next vertex on the right polygon going clockwise, and see whether that would be a (better) bridge. Otherwise, see if the left end can be raised while the right end remains fixed.
3. If made no progress in (2) (cannot raise either side), then stop else repeat (2).

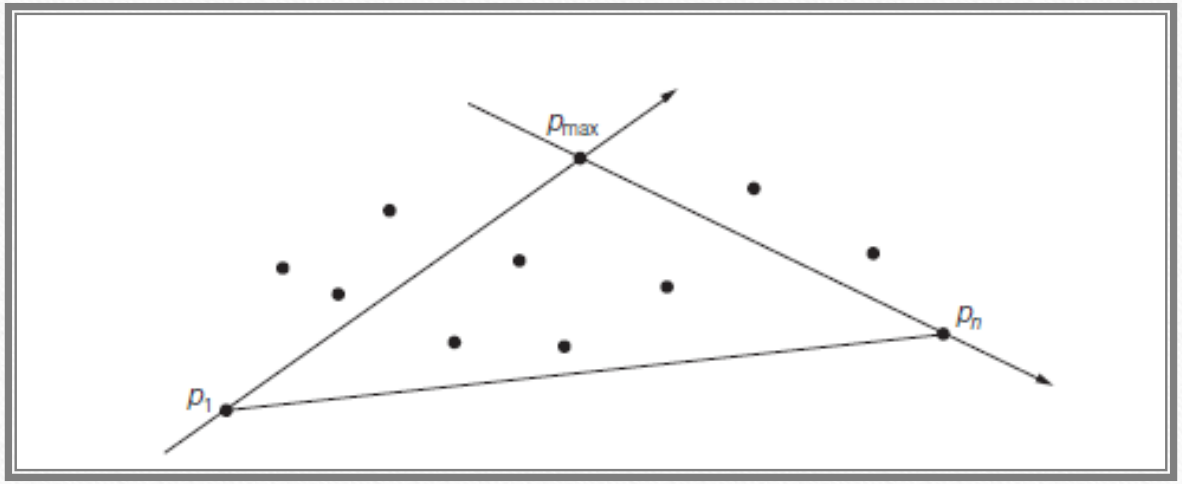


Combining two hulls

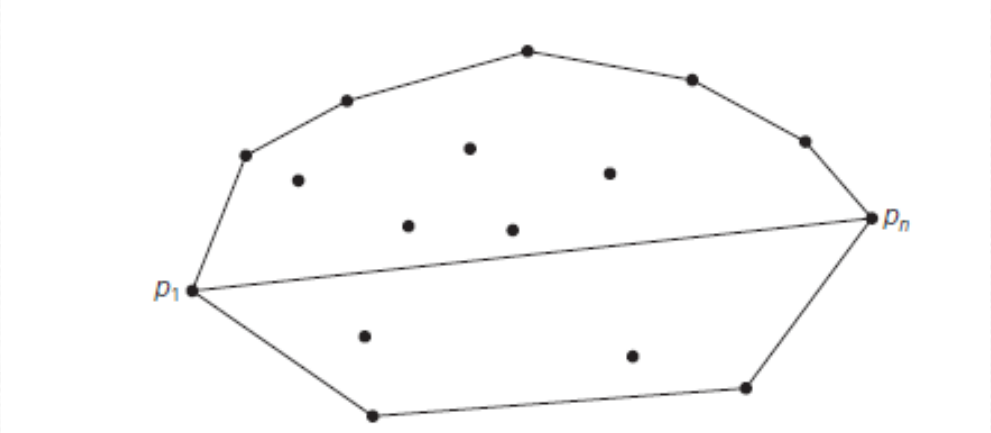
- Define a bridge as any line segment joining a vertex on the left and a vertex on the right that does not cross the side of either polygon. What we need are the upper and lower bridges.

Quickhull

- The leftmost point p1 and the rightmost point pn are two distinct extreme points of the set's convex hull
- Let p1pn be the straight line through points p1 and pn directed from p1 to pn.
- This line separates the points of S into two sets: S1 is the set of points to the left of this line, and S2 is the set of points to the right of this line.
- The points of S on the line p1pn, other than p1 and pn, cannot be extreme points of the convex hull and hence are excluded from further consideration.
- The boundary of the convex hull of S is made up of two polygonal chains: an "upper" boundary and a "lower" boundary.
- Upper hull is simply the line segment with the endpoints at p1 and pn. If S1 is not empty, the algorithm identifies point pmax in S1, which is the farthest from the line p1pn.

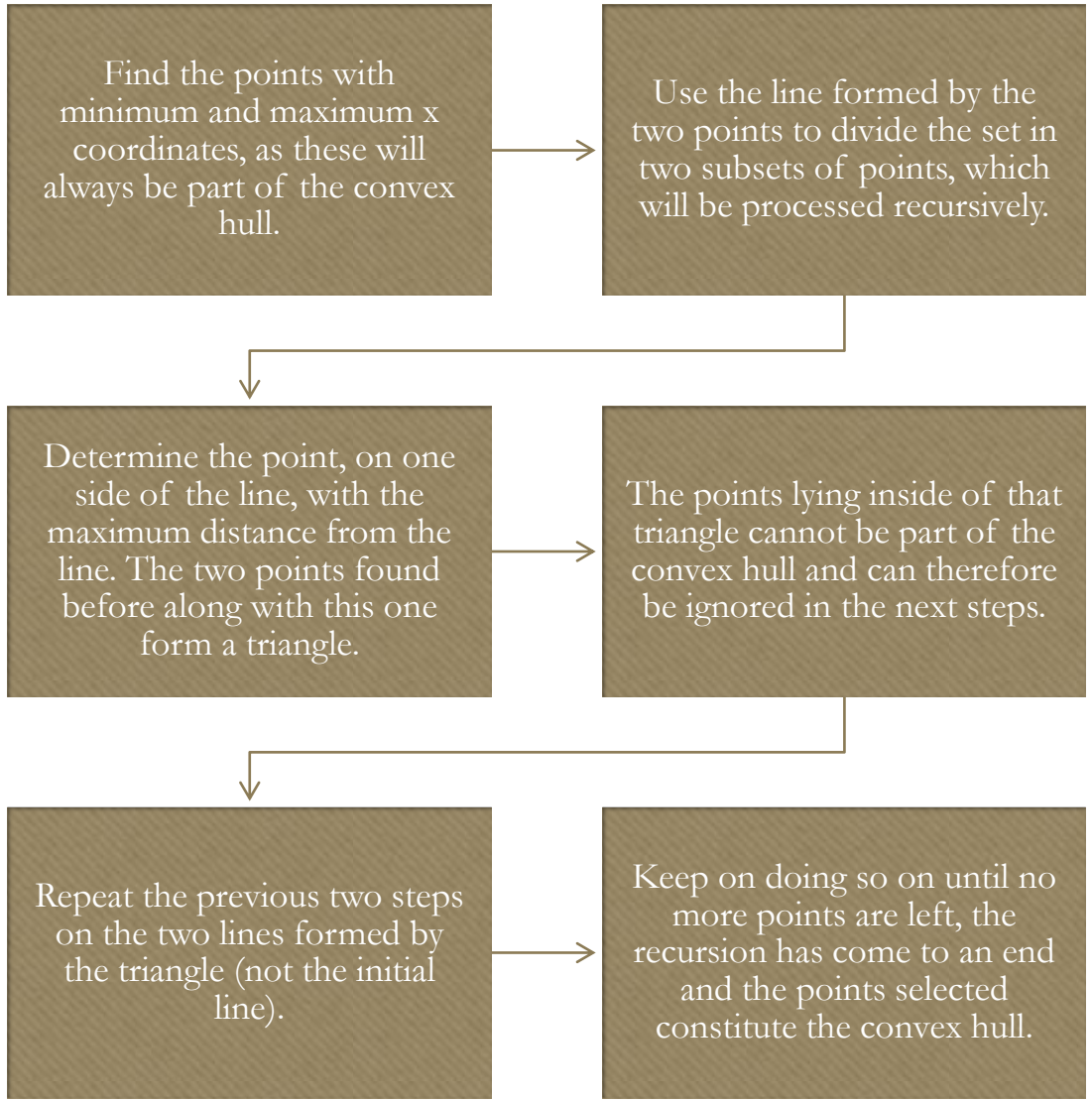


Quickhull



- Then the algorithm identifies all the points of set S1 that are to the left of the line p1pmax; these are the points that will make up the set S1,1.
- The points of S1 to the left of the line pmaxpn will make up the set S1,2.
- Therefore, the algorithm can continue constructing the upper hulls of
  - p1 ∪ S1,1 ∪ pmax and
  - pmax ∪ S1,2 ∪ pn
- recursively and then simply concatenate them to get the upper hull of the entire set p1 ∪ S1 ∪ pn.

Quickhull


Find the points with minimum and maximum x coordinates, as these will always be part of the convex hull.
Use the line formed by the two points to divide the set in two subsets of points, which will be processed recursively.
Determine the point, on one side of the line, with the maximum distance from the line. The two points found before along with this one form a triangle.
The points lying inside of that triangle cannot be part of the convex hull and can therefore be ignored in the next steps.
Repeat the previous two steps on the two lines formed by the triangle (not the initial line).
Keep on doing so on until no more points are left, the recursion has come to an end and the points selected constitute the convex hull.

Runtime

- Quickhull has the same complexity class (n2) as quicksort
- In the average case, however, we should expect a much better performance.
- First, the algorithm should benefit from the quicksortlike savings from the on-average balanced split of the problem into two smaller subproblems.
- Second, a significant fraction of the points—namely, those inside p1pmaxpn—are eliminated from further processing.
- Under a natural assumption that points given are chosen randomly from a uniform distribution over some convex region (e.g., a circle or a rectangle), the average-case efficiency of quickhull turns out to be linear.