Closest Pair

and

Linear Time Selection

The Kiss
by Brancusi

Closest Pair

- Problem definition:
  - We are given n points $p_i = (x_i, y_i)$, $i = 1,\ldots, n$.
  - How far apart are the closest pair of points?
    - We need the i and j values that minimize the Euclidean distance:

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

    - Then return this distance.
  - Brute force: compute distances for all $n(n-1)/2$ pairs and find the minimum.
    - This algorithm runs in time $\Theta(n^2)$.

Closest Pair by Divide and Conquer

- Sort by x-coordinate, then:
- Divide:
  - Find vertical line splitting points in half.
- Conquer:
  - Recursively find closest pairs in each half.
- Combine:
  - Check vertices near the border to see if any pair straddling the border is closer together than the minimum seen so far.
- Our goal:
  - $\Theta(n)$ overhead so that the total run time is $\Theta(n \log n)$.

Implementation Details

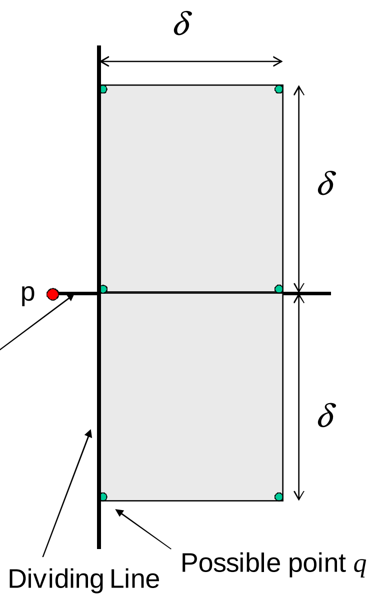
- Input: a set of points P sorted with respect to the x coordinate.
- Initially, P = all points, and we pay $\Theta(n \log n)$ to sort them before making the first call to the recursive subroutine.
- Given the sorted points, it is easy to find the dividing line.
  - Let P_L be points to the left of the dividing line.
  - Let P_R be points to the right of the dividing line.

Closest Pair Implementation Details

- Recursively:
  - Find closest pair in P_L: Let δ_L be their separation distance
  - Similarly find closest pair in P_R, with separation distance δ_R.
  - Clever observation: If the closest pair straddles the dividing line, then each point of the pair must be within $\delta = \min\{\delta_L, \delta_R\}$ of the dividing line.
    - This will usually specify a fairly narrow band for our "straddling" search.

Implementation Details

- Suppose p and q, $d(p,q) = \delta$ are candidate closest points, with p on the left and q on the right of the dividing line.
  - q **not** to the right of δ band.
  - if $p = (x, y)$ then with y coords interval $[y - \delta, y + \delta]$ can be successfully paired with p.
    - So, we need only look at points within δ above and below a horizontal line through p.
    - Since the points in this rectangle must be separated by at least δ we have **at most** 6 points to investigate. (Diagram shows this "worst case" situation).



Closest Pair Pseudocode

- Let P be a global array storing all the points with P_R and P_L defined as described earlier.
- Let **QL** be the subset of points in P_L that are at most δ (`delta` in the code) to the left of the dividing line.
- Let **QR** be the subset of points in P_R that are at most δ to the right of the dividing line.

```
//-------- main ---------
// P contains all the points
sort P by x-coordinate;
return closest_pair(1, n);
```

Closest Pair Pseudocode

```
function closest_pair(l,r)
  // Find the closest pair in P[l..r] (sorted by x-coordinate)
  if size(P) < 2 then return infinity;
  mid := (l + r)/2; midx := P[mid].x;
  dl := closest_pair(l, mid);
  dr := closest_pair(mid + 1, r);
  // Side effect: P[l..mid] and P[mid+1..r] are now sorted
  // wrt the y-coordinate
  delta := min(dl, dr);
  QL := select_candidates(l, mid, delta, midx);
  QR := select_candidates(mid + 1, r, delta, midx);
  dm := delta_m(QL, QR, delta);
  // Use merge to make P[l..r] sorted by y-coordinate
  Merge(l, mid, r);           // Merge as in MergeSort
  return min(dm, dl, dr);
```

Closest Pair Pseudocode

```
function select_candidates(l,r,delta,midx)
  // From P[l..r] select all points which are
  // at most delta from midx line
  create empty array Q;
  for i := l to r do
      if (abs(P[i].x - midx) <= delta)
            add P[i] to Q;
  return Q;
```

Closest Pair Pseudocode

```
function delta_m(QL,QR,delta)
  // Are there two points p in QL, q in QR such that
  // d(p,q)<=delta? Return distance of closest pair.
  // Assume QL and QR are sorted by the y coordinate.
  j := 1;   dm := delta;
  for i := 1 to size(QL) do
      p := QL[i];
      // find the bottom-most candidate from QR
      while (j <= size(QR) and QR[j].y < p.y-delta) do
            j := j+1;
      // check all candidates from QR starting with j
      k := j;
      while (k <= size(QR) and QR[k].y <= p.y + delta) do
            dm := min(dm, distance(p, QR[k]));
            k := k+1;
  return dm;
```

Closest Pair Analysis

– Let $T(n)$ be the time required to solve the problem for n points:

- Divide: $\Theta(1)$
- Conquer: $2T(n/2)$
- Combine: $\Theta(n)$

– The precise form of the recurrence is:

$$T(n) = T\left(\lceil n/2 \rceil\right) + T\left(\lfloor n/2 \rfloor\right) + \Theta(n)$$

which we can approximate by:

$$T(n) = 2T(n/2) + \Theta(n)$$

– Solution: $\Theta(n \log n)$.

Linear-Time Selection

- Problem statement:
  – Given an array A of n numbers $A[1..n]$ and a number i $(1 \le i \le n)$, find i^{th} smallest number in A.
  – Definition: Median of A is the $\lceil n/2 \rceil^{th}$ element in A.
    - Example: If $A = (7, 4, 8, 2, 4)$; then $|A| = 5$ and the 3^{rd} smallest element (and median) is 4.
  – Trivial solutions for our problem:
    - Sort the array and find the i^{th} element.
      – Execution time: $\Theta(n \log n)$
    - If i is small we can find i using a linear scan similar to finding a minimum or maximum in the array.
      Idea: keep current top i rather than current max only.

Linear-Time Selection (page 2)

- Strategy: Partition-based (divide and conquer) selection
  - Choose one element p from array A (pivot element)
  - Split input into three sets:
    - *LESS*: elements from A that are smaller than p
    - *EQUAL*: elements from A that are equal to p
    - *MORE*: elements from A that are greater than p
  - We then have three cases:
    - $i \leq |LESS|$: implies the element we are looking for is also the i^{th} smallest number in *LESS*,
    - $|LESS| < i \leq |LESS| + |EQUAL|$: implies the element we are looking for is p,
    - $|LESS| + |EQUAL| < i$: implies the element we are looking for is also the $(i - |LESS| - |EQUAL|)^{th}$ smallest element in *MORE*.

Linear-Time Selection

```
function SELECT(A, i)
  // find i-th element in array A
  p := choose_pivot(A);
  // partition A into LESS, EQUAL, MORE
  create new arrays LESS, EQUAL, MORE;
  for i := 1 to size(A) do
     if A[i] < p then add A[i] to LESS;
     if A[i] = p then add A[i] to EQUAL;
     if A[i] > p then add A[i] to MORE;
  // decide which case to pursue
  if(size(LESS) >= i) then
     return SELECT(LESS, i);
  else if(size(LESS) + size(EQUAL) >= i) then
     return p;    // No recursive call
  else
     return SELECT(MORE, i - size(LESS) - size(EQUAL));
```

Linear-Time Selection

- Choice of pivot:
  - Option 1: Choose an arbitrary element (for example, the first element).
    - If array is already sorted array and we are looking for the n^{th} smallest element, then execution time = $\Omega(n^2)$.
  - Option 2: Choose a random element.
    - This is better and gives us a "randomized" algorithm.
    - In this case the *expected* running time is $\Theta(n)$, while worst-case running time is still $\Theta(n^2)$.
      - See [CLRS, 9.2], if interested.

Linear-Time Selection

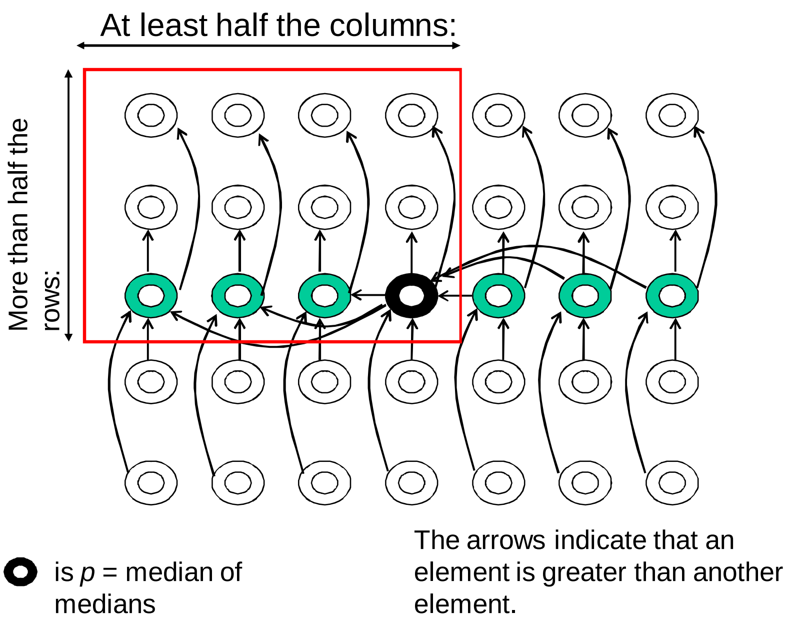
- Another choice of pivot:
  - Option 3: Use "grouping by fives" to select the pivot:

1. Split the array $A[1..n]$ into $n/5$ groups each with 5 elements.
2. From each group select the third smallest element (i.e., take median of each of the groups).
   - Denote the set of these elements as *MEDIANS*.
3. Recursively call SELECT to obtain the median of *MEDIANS*.
   - (i.e.: the $\lceil n/2 \rceil^{th}$ smallest element of *MEDIANS*).
4. Take the resulting element as pivot p.

Running Time

- Lemma:
  - At least $1/4$ of the elements in A are smaller than or equal to p (so $|MORE| \le 3n/4$).
- Proof:
  - Sort elements in each of the groups from smallest to largest and ordering groups by their median.
  - Represent the whole set A by a table where each group is depicted as a single column and the columns are ordered by their medians.
  - Then the following figure demonstrates the claim (for the case where n is a multiple of 5):

Running Time



Running Time

- Similar lemma:
  - At least $1/4$ of the elements in A are greater than or equal to p (so $|LESS| \leq 3n/4$).

- In summary:
  - If we use *p* as pivot in SELECT, then arrays $LESS$ and $MORE$ **each** have at most $3n/4$ elements.

Running Time

- Run time analysis:
  - Running time $T(n)$ of SELECT using "group by fives":
  - Divide phase: $\Theta(n)$
  - Conquer phase:
    - To select "median of medians" we need time: $T(\lceil n/5 \rceil)$
    - To run selection on one of the arrays: $LESS$ or $MORE$, we need time: $\leq T(\lfloor 3n/4 \rfloor)$.
  - Combine phase: There is no combine work.
  - Thus we have: $T(n) = T(\lceil n/5 \rceil) + T(\lfloor 3n/4 \rfloor) + \Theta(n)$ with $T(1) \in \Theta(1)$.

Running Time

- Claim Running time of the SELECT algorithm is $T(n) \le cn = O(n)$ (constant c to be determined later).
  - Proof by induction on n (substitution method):
    - Base case:
      - For $n < 40$, claim holds as long as c is large enough.
    - Induction step:
      - Assume that $n \geq 40$ and that for all $n_0 < n$, $T(n_0) \le cn_0$. Then:

$$T(n) \le T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\left\lfloor \frac{3n}{4} \right\rfloor\right) + kn$$

$$\le c\left(\frac{n}{5}+1\right) + \frac{3cn}{4} + kn \;\le\; \frac{cn}{5} + \frac{cn}{40} + \frac{3cn}{4} + kn \qquad \text{Note: } c \le \frac{cn}{40} \text{ if } n \ge 40.$$

$$\le \frac{39cn}{40} + kn \le cn$$

as long as $k \le c/40$, i.e. $c \geq 40k$.

Quick Sort Revisited

- Recall QuickSort:
  1. Select pivot element p.
  2. Split the array into two parts: elements smaller than p and elements larger than p.
  3. These can be sorted separately.
  - If lucky we get sub-arrays of approximately the same size to achieve $\Theta(n \log n)$ running time.
    - However, if we are unlucky, we get $\Omega(n^2)$ running time.
  - Idea: What if we use our SELECT algorithm to select pivot p to be the median?
    - Then every time we split, we guarantee “almost equal" splits thus having $\Theta(n \log n)$ worst-case running time.
    - Quite slow in practice (constant in O(…) is too large).
    - Better to just select a random element (it can be proven that then we get $\Theta(n \log n)$ expected running time).

Making Divide and Conquer Faster

- In practice:
  - Divide and conquer algorithms often have too much overhead (for recursion, etc.) ➔ slower for small size data sets than naïve algorithm.
  - Running time of a divide and conquer algorithm can be reduced if we abort the recursion at some point and solve small subproblems using the naïve algorithms.
  - When to “divide" and when to use the trivial algorithm needs to be determined on a case by case basis (via experiment or analysis).

An Example

```
function SELECT(A,i)
* if size(A)<100 then
* sort elements of A;
* return A[i];
  else
     // find i-th element in array A:
     (The same code we studied earlier).
```