Carlos Mc Gregor c.mcgregormuro@mail.utoronto.ca

CSC373S: Algorithm Design, Analysis & Complexity

Lecture 04

Monday January 13, 2017

based on notes by Denis Pankratov

Divide & Conquer (continue)

Closest Pair of Points in Euclidean 2D Space (continue)

Let us remember that this question can be written the following way:

$$\delta = \min(d(p_1, p_2), d(p_3, p4))$$

Reworking last lecture's proof since it was inaccurate:
Claim: rectangle R has at most 8 points
Proof: by contradiction, if it had ≥ 9 points, either R_1 or R_2 has at least 5 points.
Claim: If $\delta \times \delta$ square has 5 points then there are 2 points of distance $< \delta$.

To accomplish this, we need to split the square into 4 squares of size $\delta/2 \times \delta/2$, each with a diagonal $= \sqrt{\delta^2/4 + \delta^2/4} = \delta/\sqrt{2} < d$.

Intuitively, if we have 5 points that we need to place in 4 squares, by the Pidgeonhole Principle, one square contains at least 2 points. Thus, we can construct an algorithm with the following X, Y arrays:

X - copy of all points sorted by x-coordinate (ties broken by increments in y-coordinate)

Y - copy of all points sorted by y-coordinate

Algorithm:

```
def ClosestPairHelper(X, Y, n):
  if n <= 3:
    return ClosestPairBruteForce(X, n)
  lx = X[floor(n/2)].x
  ly = X[floor(n/2)].y

  X_1 = X[1..floor(n/2)]
  X_2 = X[floor(n/2) + 1..n]

```

```
    initialize Y_1 of size floor(n/2)
    initialize Y_2 of size floor(n/2)

    i = j = 1

    for k = 1 to n:
      if Y[k].x < lx or (Y[k].x = lx and Y[k].y <= ly):
        Y_1[i] = Y[k]
        i += 1
      else:
        Y_2[j] = Y[k]
        j += 1

    (p_1, p_2) = ClosestPairHelper(X_1, Y_1, floor(n/2))
    (p_3, p_4) = ClosestPairHelper(X_2, Y_2, floor(n/2))

    d = float('inf')
    p' = p'' = None

    if d(p_1, p_2) < d(p_3, p_4):
      delta = d(p_1, p_2)
      (p', p'') = (p_1, p_2)
    else:
      delta = d(p_3, p_4)
      (p', p'') = (p_3, p_4)

    # We make ~Y the empty set
    ~Y = []

    # First, we keep the points in the interval
    for i = 1 to n:
      if abs(Y[i].x = lx) <= delta
        ~Y.append(Y[i])

    # Runs in O(n)
    for k = 1 to len(~Y):
      for j = k + 1 to min(k + 7, len(~Y)):
        if d(~Y[k], ~Y[j]) < delta:
          delta = d(~Y[k], ~Y[j])
          (p', p'') = (~Y[k], ~Y[j])
```

Let $T(n) = \#$ of real operations their function performs on the worst-case operation of length n. Then:

$$T(n) = \begin{cases} \mathcal{O}(1), & \text{if } n \leq 3 \\ 2T(n/2) + \mathcal{O}(n), & \text{if } n > 3 \end{cases}$$

Applying the *Master Theorem*, this evaluates to $\mathcal{O}(n \log n)$.

L-Tiling

Input: $n \times n$ chessboard, $n = 2^k, k \geq 1$, 1 square removed
Output: valid *L-Tiling* of the board

Algorithm:

```
def Tiling(A, n):
  if n = 2:
    # then A is an L-Tile
  else:
    /* split A into 4 quadrants Q_1, Q_2,.., Q_4
    / exactly 1 quadrant, say Q, contains a
    / missing square
    / place l-tile at the center so that the tile
    / does not overlap Q */
    # solve 4 quadrants recursively
```

Let $T(n) = \#$ of tile placements performed.

$$T(n) = \begin{cases} 1, & \text{if } n = 2 \\ 4T(n/2) + 1, & \text{if } n > 2 \end{cases}$$

Applying the *Master Theorem*, this evaluates to $\mathcal{O}(n^2)$.

List Inversion

Define: An array A of n integers (i, j) is an inversion if $i < j$ implies that $A[i] > A[j]$.

Input: A array of n integers
Output: # of inversions in A

Trivial Algorithm:
Examine all $\big(n \quad 2\big)$ pairs of indices, which gives us a $\mathcal{O}(n^2)$ runtime, measured by comparisons.

Idea: sort & count (similar to MergeSort approach)

Algorithm:

```
def ListInversion(A):
  # split A into two subarrays A_1 and A_2 of
  # size floor(n/2) and ceil(n/2)
```

```

/* # of inversions in which A[j] participates = 0 */
if A_1[i] <= A_2[j]:
  A[k] = A_1[i]
  i += 1

/* A_2[j] < A_1[i] <= A_1[i+1] <= .. A_1[floor(n/2)]
/  -> size floor(n/2)-i+1 */

else: /* A_2[j] < A_1[j] */
  A[k] = A_2[j]
  j += 1
```

This has a recurrence of $T(n) = 2T(^n/_2) + \mathcal{O}(n)$, which solves to $\mathcal{O}(n \log n)$ when using the *Master Theorem*.