CS 341: Algorithms
Module 4: Divide and Conquer

Armin Jamshidpey, Eugene Zima

Based on lecture notes by many previous CS 341 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2019

Divide-and-Conquer

A general algorithmic paradigm (strategy):

- Divide: Split a problem into several subproblems.
- Conquer: Solve the subproblems (recursively) applying the same algorithm.
- Combine: Use subproblem results to derive a final result for the original problem.

Examples: binary search, quick sort, merge sort, third solution to the Bentley's problem from lecture 1, etc..

When can we use Divide and Conquer?

- Original problem is easily decomposable into subproblems (we do not want to see "overlap" in the subproblems).
- Combining solutions is not too costly.
- Subproblems are not overly unbalanced.

Counting Inversions

Problem

Given a permutation $a_1, a_2, \ldots, a_n$ of the numbers $1, 2, \ldots, n$ find the number of pairs (a_i, a_k) such that $i < k$ and $a_i > a_k$.

Brute force solution: check all $\binom{n}{2}$ pair of $i, k : i < k$ and count...
Easy to implement, but worst case running time is in $\Theta(n^2)$.

Divide and Conquer

- Split array in two almost equal parts.
- Solve 2 subproblems (counting inversions in the left (r_L) and right (r_R) half separately).
- Count number of pairs (r) of (a_i, a_k) such that a_i is in the left half a_k is in the right half and $a_i > a_k$.
- Return $r_L + r_R + r$.

Note, that the value of r does not depend on the relative order of values in the left half, and does not depend on the relative order in the right side.
Are we allowed to change given array? (What if not?)

Divide and Conquer

- Split array in two almost equal parts.
- Solve 2 subproblems (counting inversions in the left (r_L) and right (r_R) half separately), sorting the halves at the same time.
- set r to 0.
- Count number of pairs (r) of (a_i, a_k) such that a_i is in the left half, a_k is in the right half and $a_i > a_k$ by merging: when an element from right half is moved into the merged list, add the number of remaining elements in the left list to r.
- Return $r_L + r_R + r$ and the result of merging (sorted list).

Algorithm 1: MergeAndCount(A, B, m, n)

```
1  i ← 1; j ← 1; r ← 0; C ← ∅
2  while i ≤ m  and j ≤ n
3      if (A[i] ≤ B[j]) then
4          append A[i] to C
5          i++
6      else
7          append B[j] to C
8          j++
9          r ← r + m − i + 1
10 append A[i..m] and B[j..n] to C
11 return ( C, r )
```

Algorithm 2: SortAndCount(A, n)

1 If $n \leq 3$ sort and count with a trivial algorithm and return.

2 $(S_L, r_L) \leftarrow$ SortAndCount$(A[1..n/2])$

3 $(S_R, r_R) \leftarrow$ SortAndCount$(A[n/2+1..n])$

4 $(S, r) \leftarrow$ MergeAndCount(S_L, S_R)

5 **return** $(\ S, r_L + r_R + r\ )$

Closest Pair

problem: Closest Pair

Instance: a set Q of n distinct points in the Euclidean plane,

$$Q = \{Q[1], \ldots, Q[n]\}.$$

Find: Two distinct points $Q[i] = (x, y), Q[j] = (x', y')$ such that the Euclidean distance

$$\sqrt{(x' - x)^2 + (y' - y)^2}$$

is minimized.

We actually describe how to find the smallest distance between pairs of points. This can be further rectified to find the points in question.

Closest Pair: Problem Decomposition

Suppose we pre-sort the points in Q with respect to their x-coordinates (this takes time $\Theta(n \log n)$).

Then we can easily find the vertical line that partitions the set of points Q into two sets of size $n/2$: this line has equation $x = Q[m].x$, where $m = n/2$.

The set Q is global with respect to the recursive procedure ClosestPair1.

At any given point in the recursion, we are examining a subarray $(Q[\ell], \ldots, Q[r])$, and $m = \lfloor (\ell + r)/2 \rfloor$.

We call ClosestPair1$(1, n)$ to solve the given problem instance.

Closest Pair: Solution 1

Algorithm 3: ClosestPair1(ℓ, r)

```
1  if ℓ = r  then
2      δ ← ∞
3    else
4      m ← ⌊(ℓ + r)/2⌋
5      δ_L ← ClosestPair1(ℓ, m)
6      δ_R ← ClosestPair1(m + 1, r)
7      δ ← min{δ_L, δ_R}
8      R ← Select(ℓ, r, δ, Q[m].x )
9      R ← Sort_y(R)
10     δ ← CheckStrip(R, δ)
11 return ( δ )
```

Selecting Candidates from the Vertical Strip

Algorithm 4: Select($\ell, r, \delta, xmid$)

```
1 j ← 0
2 for i ← ℓ to r do
3 if |Q[i].x − xmid| ≤ δ
4    then
5 j ← j + 1
6 R[j] ← Q[i]
7 return ( R )
```

Checking the Vertical Strip

Algorithm 5: CheckStrip(R, δ)

```
1  t ← size(R)
2  δ_m ← δ
3  for j ← 1 to t − 1 do
4      for k ← j + 1 to min{t, j + 7} do
5          x ← R[j].x
6          x′ ← R[k].x
7          y ← R[j].y
8          y′ ← R[k].y
9          δ_m ← min{δ_m, sqrt((x′ − x)^2 + (y′ − y)^2)}
10 return ( δ_m )
```

Closest Pair: Solution 2

Algorithm 6: ClosestPair2(ℓ, r)

```
1  if ℓ = r  then
2      δ ← ∞
3  else
4      m ← ⌊(ℓ + r)/2⌋
5      xmid ← Q[m].x
6      δ_L ← ClosestPair2(ℓ, m)
7      comment: Q[ℓ], ..., Q[m] is sorted WRT y-coordinates
8      δ_R ← ClosestPair2(m + 1, r)
9      comment: Q[m + 1], ..., Q[r] is sorted WRT y-coordinates
10     δ ← min{δ_L, δ_R}
11     Merge_y(ℓ, m, r)
12     R ← Select(ℓ, r, δ, xmid)
13     δ ← CheckStrip(R, δ)
14 return ( δ )
```