COMP20007 Design of Algorithms

Divide-and-Conquer Algorithms

Lars Kulik

Lecture 9

Semester 1, 2020

Divide and Conquer

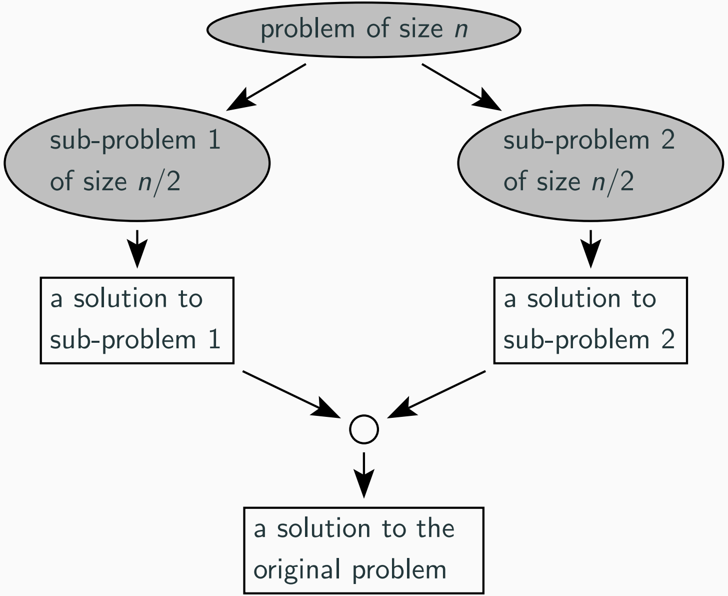
We earlier saw recursion as a powerful problem solving technique.

The divide-and-conquer strategy tries to make the most of this:

1. Divide the given problem instance into smaller instances.
2. Solve the smaller instances recursively.
3. Combine the smaller solutions to solve the original instance.

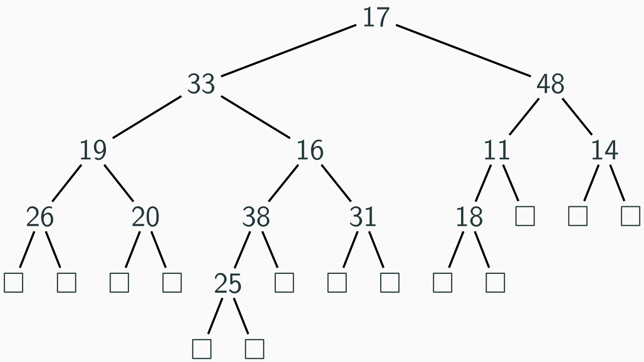
This works best when the smaller instances can be made to be of equal size.

Split-Solve-and-Join Approach



Binary Trees Again

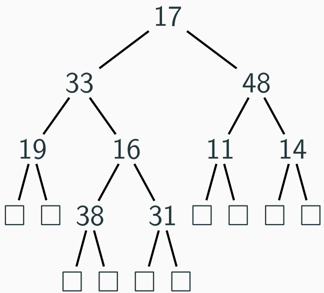
An example of a binary tree, with empty subtrees marked with □:



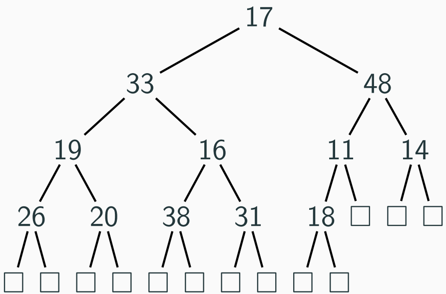
This tree has height 4, the empty tree having height -1.

Binary Tree Concepts

Special trees have their external nodes $\square$ only at level h and $h + 1$ for some h:



A full binary tree: Each node has 0 or 2 children.



A complete tree: Each level filled left to right.

Binary Tree Concepts

A non-empty tree T has a root T_{root}, a left subtree T_{left}, and a right subtree T_{right}.

Recursion is the natural way of calculating the height:

function $\textsc{Height}(T)$
  if T is empty **then**
    return -1
  else
    return $max(\textsc{Height}(T_{left}), \textsc{Height}(T_{right})) + 1$

Binary Tree Concepts

It is not hard to prove that the number x of external nodes $\square$ is always one greater than the number n of internal nodes.

The function $\textsc{Height}$ makes a tree comparison (empty or non-empty?) per node (internal and external), so altogether $2n + 1$ comparisons.

Binary Tree Traversal

Preorder traversal visits the root, then the left subtree, and finally the right subtree.

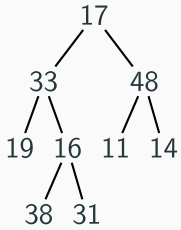
Inorder traversal visits the left subtree, then the root, and finally the right subtree.

Postorder traversal visits the left subtree, the right subtree, and finally the root.

Level-order traversal visits the nodes, level by level, starting from the root.

Binary Tree Traversal: Preorder

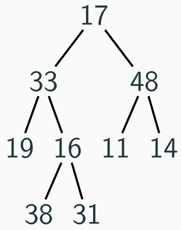
function PREORDERTRAVERSE(T)
    if T is non-empty **then**
        visit T_{root}
        PREORDERTRAVERSE(T_{left})
        PREORDERTRAVERSE(T_{right})



Visit order for the example: 17, 33, 19, 16, 38, 31, 48, 11, 14.

Binary Tree Traversal: Inorder

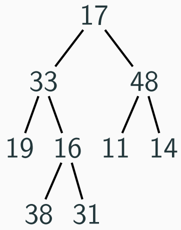
function InorderTraverse(T)
  if T is non-empty **then**
    InorderTraverse(T_{left})
    visit T_{root}
    InorderTraverse(T_{right})



Visit order for the example: 19, 33, 38, 16, 31, 17, 11, 48, 14.

Binary Tree Traversal: Postorder

function POSTORDERTRAVERSE(T)
  if T is non-empty **then**
    POSTORDERTRAVERSE(T_{left})
    POSTORDERTRAVERSE(T_{right})
    visit T_{root}



Visit order for the example: 19, 38, 31, 16, 33, 11, 14, 48, 17.

Preorder Traversal Using a Stack

We could also implement preorder traversal of T by maintaining a stack explicitly.

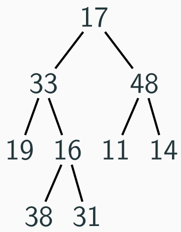
$push(T)$
while the stack is non-empty **do**
    $T \leftarrow pop$
    visit T_{root}
    if T_{right} is non-empty **then**
        $push(T_{right})$
    if T_{left} is non-empty **then**
        $push(T_{left})$

In an implementation, the elements placed on the stack would not be whole trees, but pointers to the corresponding internal nodes.

Tree Traversal Using a Queue: Level-Order

Level-order traversal results if we replace the stack with a queue.

inject(T)
while the queue is non-empty **do**
  $T \leftarrow eject$
  visit T_{root}
  if T_{left} is non-empty **then**
    inject(T_{left})
  if T_{right} is non-empty **then**
    inject(T_{right})



Visit order for the example: 17, 33, 48, 19, 16, 11, 14, 38, 31.

The Closest Pair Problem Revisited

In Lecture 5 we gave a brute-force algorithm for the closest pair problem: Given n points in the Cartesian plane, find a pair with minimal distance.

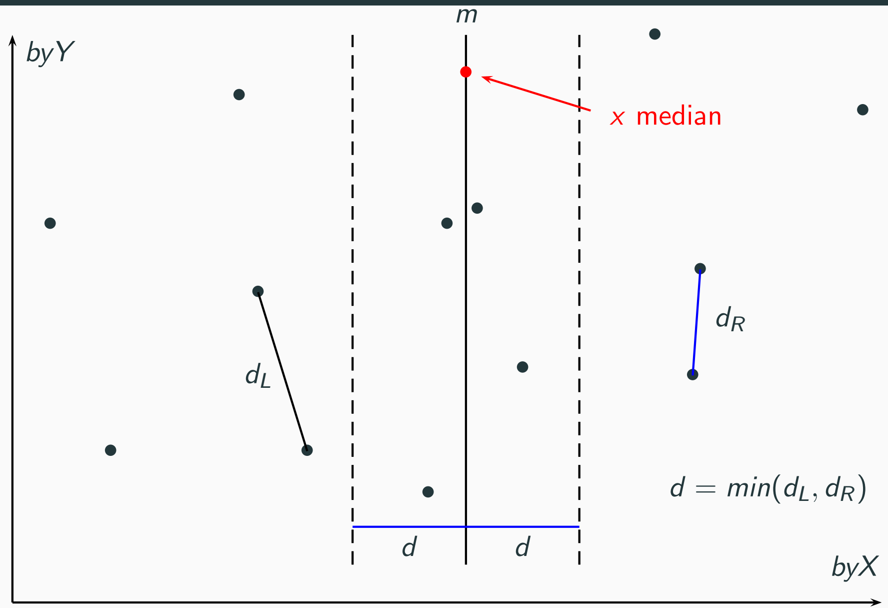
The brute-force method had complexity $\Theta(n^2)$. We can use divide-and-conquer to do better, namely $\Theta(n \log n)$.

First, sort the points by x value and store the result in array byX.

Also sort the points by y value and store the result in array byY.

Now we can identify the x median, and recursively process the set P_L of points with lower x values, as well as the set P_R with higher x values.

The Closest Pair Problem Revisited



The Closest Pair Problem Revisited

The recursive calls will identify d_L, the shortest distance for pairs in P_L, and d_R, the shortest distance for pairs in P_R.

Let m be the x median and let $d = min(d_L, d_R)$. This d is a candidate for the smallest distance.

But d may not be the global minimum—there could be some close pair whose points are on opposite sides of the median line $x = m$.

For candidates that may improve on d we only need to look at those in the band $m - d \leq x \leq m + d$.

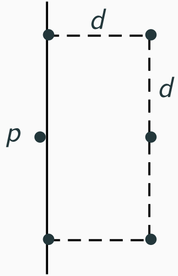
So pick out, from array byY, each point p with x-coordinate between $m - d$ and $m + d$, and keep these in array S.

For each point in S, consider just its “close” neighbours.

The Closest Pair Problem Revisited

The following calculates the smallest distance and leaves the (square of the) result in *minsq*.

It can be shown that the while loop can execute at most 5 times for each i value—see diagram.



$minsq \leftarrow d^2$
copy all points of byY with $|x - m| < d$ to array S
$k \leftarrow |S|$
for $i \leftarrow 0$ to $k - 2$ **do**
    $j \leftarrow i + 1$
    while $j \leq k - 1$ and $(S[j].y - S[i].y)^2 < minsq$ **do**
        $minsq \leftarrow min(minsq, (S[j].x - S[i].x)^2 + (S[j].y - S[i].y)^2)$
        $j \leftarrow j + 1$