CSC 421: Algorithm Design & Analysis

Spring 2019

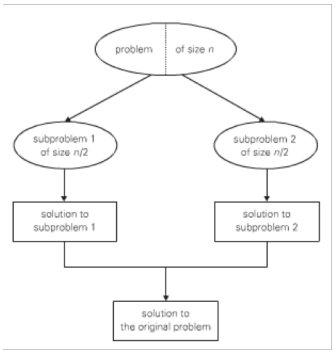
Divide & conquer

- divide-and-conquer approach
- familiar examples: merge sort, quick sort
- other examples: closest points, large integer multiplication
- tree operations: binary trees, BST

Divide & conquer

probably the best-known problem-solving paradigm

1. divide the problem into several subproblems of the same type (ideally of about equal size)
2. solve the subproblems, typically using recursion
3. combine the solutions to the subproblems to obtain a solution to the original problem



Not always a win

some problems can be thought of as divide & conquer, but are not efficient to implement that way

- e.g., counting the number of 0's in a list of numbers:
  1. recursively count the number of 0's in the 1st half of the list
  2. recursively count the number of 0's in the 2nd half of the list
  3. add the two counts

$$
\begin{aligned}
\text{Cost}(N) &= 2\,\text{Cost}(N/2) + C \\
&= 2\,[2\,\text{Cost}(N/4) + C] + C \\
&= 4\,\text{Cost}(N/4) + 3C \\
&= 4\,[2\,\text{Cost}(N/8) + C] + 3C \\
&= 8\,\text{Cost}(N/8) + 7C \\
&= \ldots \\
&= N\,\text{Cost}(N/N) + (N-1)C \\
&= NC' + (N-1)C \\
&= (C + C')N - C
\end{aligned}
$$

➔O(N)

the overhead of recursion makes this much slower than a simple iteration (i.e., decrease & conquer)

3

Master Theorem

Suppose $\text{Cost}(N) = a\,\text{Cost}(N/b) + C\,N^d$.

- that is, divide & conquer is performed with polynomial-time overhead

$$
\text{Cost}(N) = \begin{cases} O(N^d) & \text{if } a < b^d \\ O(N^d \log N) & \text{if } a = b^d \\ O(N^{\log_b a}) & \text{if } a > b^d \end{cases}
$$

- e.g., zero count algorithm has Cost(N) = 2Cost(N/2) + C

  a = 2, b = 2, d = 0 ➔ $2 > 2^0$ ➔ $O(N^{\log_2 2})$ ➔ O(N)

4

Merge sort

we have seen divide-and-conquer algorithms that are more efficient than brute force

e.g., merge sort list[0..N-1]

1. if list N <= 1, then DONE
2. otherwise,
   a) merge sort list[0..N/2]
   b) merge sort list[N/2+1..N-1]
   c) merge the two sorted halves

$$\text{Cost}(N) = \begin{cases} O(N^d) & \text{if } a < b^d \\ O(N^d \log N) & \text{if } a = b^d \\ O(N^{\log_b a}) & \text{if } a > b^d \end{cases}$$

recall: Cost(N) = 2Cost(N/2) +CN

- merging is O(N), but requires O(N) additional storage and copying
- we can show this is O(N log N) by unwinding, or
- a = 2, b = 2, d = 1 ➔ $2 = 2^1$ ➔ O(N log N) by Master Theorem

5

Why is halving most common?

consider a variation of merge sort that divides the list into 3 parts

1. if list N <= 1, then DONE
2. otherwise,
   a) merge sort list[0..N/3]
   b) merge sort list[N/3+1..2N/3]
   c) merge sort list[2N/3+1..N-1]
   d) merge the three sorted parts

Cost(N) = 3Cost(N/3) + CN

a = 3, b = 3, d = 1 ➔ $3 = 3^1$ ➔ O(N log N)

$$\text{Cost}(N) = \begin{cases} O(N^d) & \text{if } a < b^d \\ O(N^d \log N) & \text{if } a = b^d \\ O(N^{\log_b a}) & \text{if } a > b^d \end{cases}$$

in general, X Cost(N/X) + CN ➔ O(N log N)
X Cost(N/X) + C ➔ O(N)

- dividing into halves is simplest, and just as efficient as thirds, quarters, …

6

Quick sort

Collections.sort implements quick sort, another O(N log N) sort which is faster in practice

e.g., quick sort list[0..N-1]

1. if list N <= 1, then DONE
2. otherwise,
   a) select a pivot element (e.g., list[0], list[N/2], list[random], …)
   b) partition list into [items < pivot] + [items == pivot] + [items > pivot]
   c) quick sort the < and > partitions

best case: pivot is median

Cost(N) = 2Cost(N/2) +CN ➔ O(N log N)

worst case: pivot is smallest or largest value

Cost(N) = Cost(N-1) +CN ➔ $O(N^2)$

7

Quick sort (cont.)

average case: O(N log N)

there are variations that make the worst case even more unlikely

- switch to selection sort when small
- median-of-three partitioning

  instead of just taking the first item (or a random item) as pivot, take the median of the first, middle, and last items in the list

  ✓ if the list is partially sorted, the middle element will be close to the overall median

  ✓ if the list is random, then the odds of selecting an item near the median is improved

refinements like these can improve runtime by 20-25%

however, $O(N^2)$ degradation is still possible

8

Closest pair

given a set of N points, find the pair with minimum distance

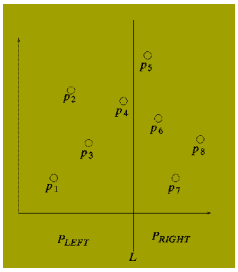
- brute force approach:
  consider every pair of points, compare distances & take minimum

  big Oh? $O(N^2)$

- there exists an O(N log N) divide-and-conquer solution

Assume the points are sorted by x-coordinate (can be done in O(N log N))

1. partition the points into equal parts using a vertical line in the plane
2. recursively determine the closest pair on left side (Ldist) and the closest pair on the right side (Rdist)
3. find closest pair that straddles the line (Cdist), each within min(Ldist,Rdist) of the line (can be done in O(N))
4. answer = min(Ldist, Rdist, Cdist)



Cost(N) = 2 Cost(N/2) + CN ➔ O(N log N)

9

Multiplying large integers

many CS applications, e.g., cryptography, involve manipulating large integers

long multiplication:

```
   123456
 × 213121
   123456
 + 2469120
 + 12345600
 + 370368000
 + 1234560000
+ 24691200000
  26311066176
```

long multiplication of two N-digit integers requires N^2 digit multiplications

10

Divide & conquer multiplication

can solve (a x b) by splitting the numbers in half & factoring

consider a_1 = 1st N/2 digits of a, a_0 = 2nd N/2 digits of a (similarly for b_1 and b_0)

$$(a \times b) = (a_1 10^{N/2} + a_0) \times (b_1 10^{N/2} + b_0)$$
$$= (a_1 \times b_1)10^N + (a_1 \times b_0 + a_0 \times b_1)10^{N/2} + (a_0 \times b_0)$$
$$= c_2 10^N + c_1 10^{N/2} + c_0$$

where

- $c_2 = a_1 \times b_1$
- $c_0 = a_0 \times b_0$
- $c_1 = (a_1 + a_0) \times (b_1 + b_0) - (c_2 + c_0)$

EXAMPLE: a = 123456 b = 213121

- $c_2 = a_1 \times b_1 = 123 \times 213 = 26199$
- $c_0 = a_0 \times b_0 = 456 \times 121 = 55176$
- $c_1 = (a_1 + a_0) \times (b_1 + b_0) - (c_2 + c_0) = (123 + 456) \times (213 + 121) - (26199 + 55176)$
  $= 579 \times 334 - 81375 = 193386 - 81375 = 112011$

$(a \times b) = c_2 10^N + c_1 10^{N/2} + c_0 = 26199000000 + 55176000 + 112011 = 26311066176$

11

Efficiency of divide & conquer multiplication

in order to multiply N-digit numbers

- calculate c_2, c_1, and c_0, each of which involves multiplying N/2-digit numbers

MultCost(N) = 3 MultCost(N/2) + C

from Master Theorem: a = 3, b = 2, d = 0 ➔ $3 > 2^0$ ➔ $O(N^{\log_2 3}) \approx O(N^{1.585})$

worry: in minimizing the number of multiplications, we have increased the number of additions & subtractions

- a similar analysis can show that AddCost(N) is also $O(N^{\log_2 3})$

does $O(N^{\log_2 3})$ really make a difference vs. $O(N^2)$?

- has been shown to improve performance for as small as N = 8
- for N = 300, can run more than twice as fast

12

Dividing & conquering trees

since trees are recursive structures, most tree traversal and manipulation operations can be classified as *divide & conquer algorithms*

- can divide a tree into root + left subtree + right subtree
- most tree operations handle the root as a special case, then recursively process the subtrees

- e.g., to display all the values in a (nonempty) binary tree, divide into
  1. *displaying the root*
  2. *(recursively) displaying all the values in the left subtree*
  3. *(recursively) displaying all the values in the right subtree*

- e.g., to count number of nodes in a (nonempty) binary tree, divide into
  1. *(recursively) counting the nodes in the left subtree*
  2. *(recursively) counting the nodes in the right subtree*
  3. *adding the two counts + 1 for the root*

13

BinaryTree class

```
public class BinaryTree<E> {
    protected TreeNode<E> root;

    public BinaryTree() {
        this.root = null;
    }

    public void add(E value) { … }

    public boolean remove(E value) { … }

    public boolean contains(E value) { … }

    public int size() { … }

    public String toString() { … }
}
```

to implement a binary tree, need to store the root node

- the root field is "protected" instead of "private" to allow for inheritance
- the empty tree has a null root
- then, must implement methods for basic operations on the collection

14

`size` method

divide-and-conquer approach:

BASE CASE: if the tree is empty, number of nodes is 0

RECURSIVE: otherwise, number of nodes is
(# nodes in left subtree) + (# nodes in right subtree) + 1 for the root

note: a recursive implementation requires passing the root as parameter

- will have a public "front" method, which calls the recursive "worker" method

```
public int size() {
    return this.size(this.root);
}

private int size(TreeNode<E> current) {
    if (current == null) {
        return 0;
    }
    else {
        return this.size(current.getLeft()) +
               this.size(current.getRight()) + 1;
    }
}
```

15

`contains` method

divide-and-conquer approach:

BASE CASE: if the tree is empty, the item is not found
BASE CASE: otherwise, if the item is at the root, then found

RECURSIVE: otherwise, search the left and then right subtrees

```
public boolean contains(E value) {
    return this.contains(this.root, value);
}

private boolean contains(TreeNode<E> current, E value) {
    if (current == null) {
        return false;
    }
    else {
        return value.equals(current.getData()) ||
               this.contains(current.getLeft(), value) ||
               this.contains(current.getRight(), value);
    }
}
```

16

`toString` method

must traverse the entire tree and build a string of the items

- there are numerous patterns that can be used, e.g., in-order traversal

BASE CASE: if the tree is empty, then nothing to traverse

RECURSIVE: recursively traverse the left subtree, then access the root, then recursively traverse the right subtree

```
public String toString() {
    if (this.root == null) {
        return "[]";
    }
    String recStr = this.toString(this.root);
    return "[" + recStr.substring(0,recStr.length()-1) + "]";
}

private String toString(TreeNode<E> current) {
    if (current ==  null) {
        return "";
    }
    return this.toString(current.getLeft()) +
           current.getData().toString() + "," +
           this.toString(current.getRight());
}
```

17

Other tree operations?

add?

remove?

numOccur?

height?

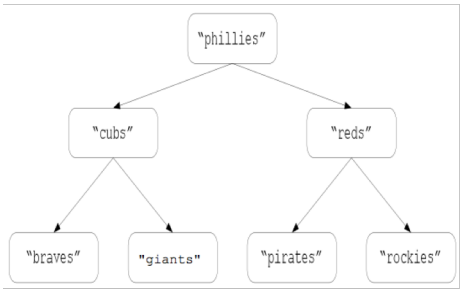
weight?

18

Binary search trees

a *binary search tree* is a binary tree in which, for every node:

- the item stored at the node is ≥ all items stored in its left subtree
- the item stored at the node is < all items stored in its right subtree



are BST operations divide & conquer?

- contains?
- add?
- remove?

what about balanced BSTs?
(e.g., red-black trees, AVL trees)

19