CS350 ABET Objective 10

Part 1 - Divide-and-Conquer

Design divide and conquer algorithms.

Textbook Section 5, ~15%.

Divide and conquer is an important algorithm design paradigm based on multi-branched recursion. This paradigm is the basis of efficient algorithms for problems such as sorting (e.g., quicksort, merge sort), multiplying large numbers and matrices, and several geometric problems.

The correctness of a divide and conquer algorithm is usually proved by mathematical induction (Objective 5), and its computational cost is often determined by solving recurrence relations (Objective 4).

Divide-and-Conquer

1. Divide a problem P into subproblems of same kind and size.
2. Solve each subproblem (typically recursively)
3. Combine the solutions of the subproblems into the solution for P.

Example:

$$a_0 + a_1 + \ldots a_{n-1} = \left(a_0 + a_1 + \ldots a_{\lfloor n/2 \rfloor - 1}\right) + \left(a_{\lfloor n/2 \rfloor} + \ldots a_{n-1}\right)$$

Compare with decrease-and-conquer:

$$a_0 + a_1 + \ldots a_{n-1} = a_0 + \left(a_1 + \ldots a_{n-1}\right)$$

Which is better? (round-off errors)

Efficiency of Div-and-Conq

Assume a ≥ 1, b > 1, n power of b

$$T(n) = aT(n/b) + f(n)$$

Master Theorem: if $f(n) \in \Theta(n^d)$, d ≥ 0, then:

$$T(n) = \begin{cases} \Theta(n^d) & if\ a < b^d, \\ \Theta(n^d \log n) & if\ a = b^d, \\ \Theta(n^{\log_b a}) & if\ a > b^{d.} \end{cases}$$

Same for *O* and *Ω*.

Example: addition above, a=2, b=2, and d=0:

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$$

Mergesort

Sort an array by divide-and-conquer.

1. Split it in the middle.
2. Sort the two halfs.
3. Merge them.

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A) //see below

Merge

Merge 2 sorted arrays:

Transfer the smaller of the first elements

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i];\ i \leftarrow i + 1$
    else $A[k] \leftarrow C[j];\ j \leftarrow j + 1$
    $k \leftarrow k + 1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Efficiency of Mergesort

Assume n is a power of 2.

$$C(n) = 2C(n/2) + C_{merge}(n) \quad for\ n > 1$$
$$C(1) = 0$$

Worst case of merge: $C_{merge}(n) = n - 1$

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad for\ n > 1$$
$$C_{worst}(1) = 0$$

From Master Th.: $C_{worst}(n) \in \Theta(n \log n)$

From recurrence: $C_{worst}(n) = n \log_2 n - n + 1$

Average: ~0.25n less than worst.

Quicksort

Divide-and-Conquer like Mergesort, but divides on value rather than position.

ALGORITHM *Quicksort*$(A[l..r])$

```
//Sorts a subarray by quicksort
//Input: Subarray of array A[0..n − 1], defined by its left and right
//         indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order
if l < r
    s ←Partition(A[l..r])  //s is a split position
    Quicksort(A[l..s − 1])
    Quicksort(A[s + 1..r])
```

Most of the work is done by *Partition*, the rest is mainly recursion.

Note that *l* and *r* in the formal argument *A[l..r]* must be considered as variables.

Partition

A partition is an arrangement of an array such that all the elements before A[s] come before A[s] and similarly all the elements after.

ALGORITHM *HoarePartition*$(A[l..r])$

```
//Partitions a subarray by Hoare's algorithm, using the first element
//        as a pivot
//Input: Subarray of array A[0..n − 1], defined by its left and right
//        indices l and r (l < r)
//Output: Partition of A[l..r], with the split position returned as
//        this function's value
p ← A[l]
i ← l; j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])   //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

Efficiency of Quicksort

Assume n is a power of 2.

$$C_{best}(n) = 2C_{best}(n/2) + n \qquad for\ n > 1$$
$$C_{best}(1) = 0$$

From Master Th.: $C_{best}(n) \in \Theta(n \log n)$

From recurrence: $C_{best}(n) = n \log_2 n$

Worst case (array sorted): $C_{worst}(n) \in \Theta(n^2)$

Aver case: $C_{aver}(n) \approx 2n \ln n \approx 1.39n \log_2 n$

Quicksort notes

"Usually" runs faster than mergesort and heapsort.

Weaknesses:

1. Worst case: $O(n^2)$
2. Stack for recursion: $O(\log n)$

Improvements:

1. Median-of-three pivot
2. Switch to insert sort for small arrays
3. Three-way partition

Binary Trees

A *binary tree* is either of two values: a *leaf* or a *branch* consisting of a *root* and two binary trees called *left* and *right*.

ALGORITHM $Height(T)$

//Computes recursively the height of a binary tree
//Input: A binary tree T
//Output: The height of T
if $T = \varnothing$ **return** -1
else return $\max\{Height(T_{left}), Height(T_{right})\} + 1$

Efficiency: size is number of branches (decorations), operation is addition (same as comparison of max).

$$A(n(T)) = A(n(T_l)) + A(n(T_r)) + 1 \quad for\ n(T) > 0$$
$$A(0) = 0$$

Height makes one addition for each branch.

$$A(n) = n$$

Traversals

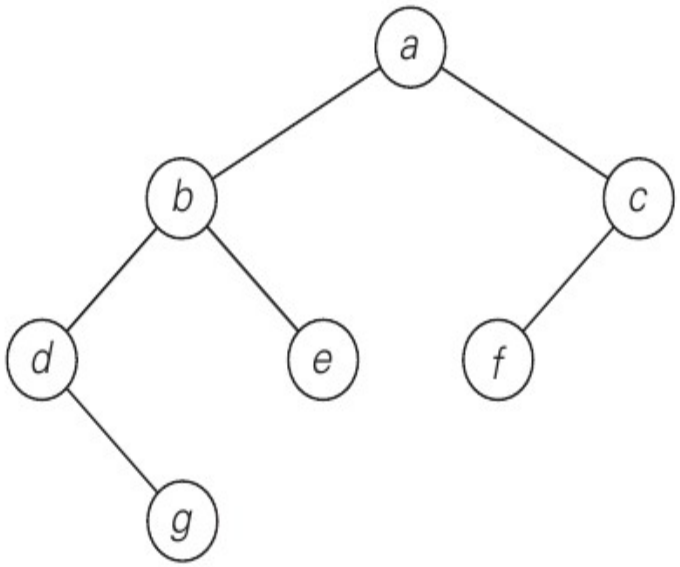
Recur on left and right, visit the root before or after or in between.

Pre-order: root, left right

In-order: left, root, right

Post-order: left, right, root

Example:



preorder: *a, b, d, g, e, c, f*
inorder: *d, g, b, e, a, f, c*
postorder: *g, d, e, b, f, c, a*

Multiplications

How many digit * digit for two n-digit numbers? By hand is n * n. It can be less.

Example:

$$23*14=\left(2\cdot10^1+3\cdot10^0\right)*\left(1\cdot10^1+4\cdot10^0\right)$$
$$\left(2*1\right)10^2+\left(2*4+3*1\right)10^1+\left(3*4\right)10^0$$

Then:

$$2*4+3*1=\left(2+3\right)*\left(1+4\right)-2*1-3*4$$

where the last 2 are already computed.

It generalizes to a number of 2n digits:

$$1234*5678=\left(12\cdot10^2+34\right)*\left(56\cdot10^2+78\right)$$

Efficiency

A multiplication of n-digit numbers requires 3 multiplications of n/2-digit numbers:

$$M(n) = 3\,M(n/2) \qquad for\ n > 1$$
$$M(1) = 1$$

Solve for $n = 2^k$:

$$M(2^k) = 3\mathrm{M}(2^{k-1}) = \ldots 3^k\, M(2^{k-k}) = 3^k$$

since $k = \log_2 n$:

$$M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$$

What about additions?

$$A(n) = 3\,A(n/2) + c\,n \qquad for\ n > 1$$
$$A(1) = 1$$

Master Th. gives:

$$A(n) \in \Theta\left(n^{\log_2 3}\right)$$

Strassen's MM

Multiply two 2*2 matrices with 7 * and 18 + instead of 8 * and 4 + by brute force.

Scales to larger sizes.

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

$$\begin{aligned} m_1 &= (a_{00} + a_{11}) * (b_{00} + b_{11}), \\ m_2 &= (a_{10} + a_{11}) * b_{00}, \\ m_3 &= a_{00} * (b_{01} - b_{11}), \\ m_4 &= a_{11} * (b_{10} - b_{00}), \\ m_5 &= (a_{00} + a_{01}) * b_{11}, \\ m_6 &= (a_{10} - a_{00}) * (b_{00} + b_{01}), \\ m_7 &= (a_{01} - a_{11}) * (b_{10} + b_{11}). \end{aligned}$$

Strassen's Efficiency

Number of * assuming $n=2^k$:

$$M\left(2^k\right)=7\mathrm{M}\left(2^{k-1}\right)=\ldots 7^k\, M\left(2^{k-k}\right)=7^k$$

$$M\left(n\right)=7^{\log_2 n}=n^{\log_2 7}\approx n^{2.807}$$

Recurrence relation:

$$A\left(n\right)=7\,A\left(n/2\right)+18\left(n/2\right)^2 \qquad for\, n>1$$
$$A\left(1\right)=1$$

Master Th. Gives:

$$A\left(n\right)\in\Theta\left(n^{\log_2 7}\right)$$

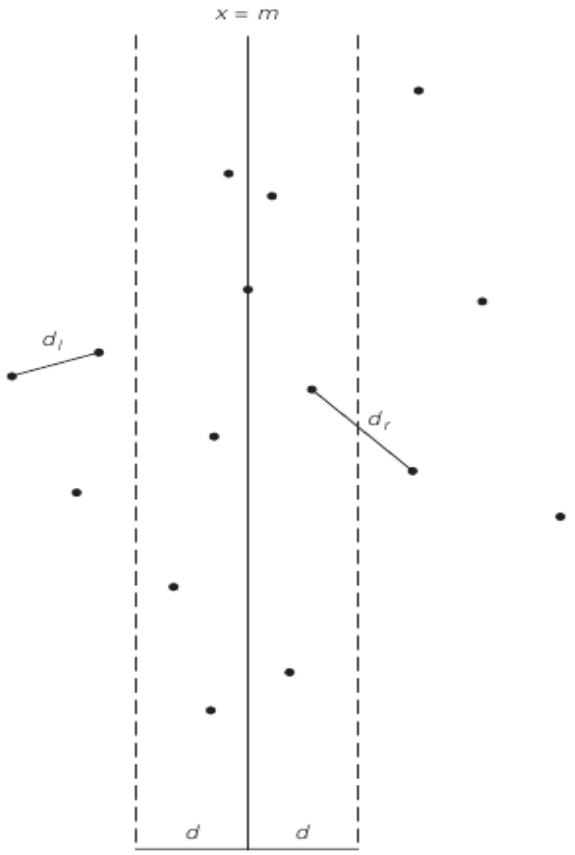
There are tighter bounds by algorithms that are not practical.

Closest-Pair

Find two closest points in a set of *n* points.

Brute-force algorithm is $O(n^2)$

Idea: divide (and conq) points into two subsets of n/2 points each by drawing a separating line "in the middle"; recur on each subset; check pairs of point that cross the separating line.



Closest-Pair (2)

Let d be the distance of closest pair so far. Have 2 sorted lists of points according to their x and y coordinates.

Only points with x within d of the separating line must be checked.

For any such point p with given y, only points with y within d of p must be checked.

The number of such points is small (6) no matter what. Thus recurrence is:

$$T(n)=2T(n/2)+f(n)$$

where $f(n)\in\Theta(n)$. Thus by Master Th:

$$T(n)\in\Theta(n\log_2 n)$$

References

Textbook Section 5

Web: http://en.wikipedia.org/wiki/
has pages for all the problems/algorithms and the Master Theorem.