ECS 122A Lecture 3

Jasper Lee

Divide + Conquer

2 different styles:

1. Divide
2. - Go into 1 piece
   vs
   - Work on every piece then combine

Play Mary Wootters' video

Examples of D+C?
- Binary search
- Mergesort
- Quicksort

Mergesort

function sort (Array A of length n)

if $n = 1$ or $n = 0$ > return A

return merge(sort($A[1..\lfloor \frac{n}{2} \rfloor]$), sort($A[\lfloor \frac{n}{2} \rfloor + 1..n]$))

What guarantees do we want from sort and merge?

$O(n \log n)$ time ← Sort: on input a numerical array A of size n, outputs sorted version of A.

$O(m+n)$ time ← merge: on input 2 sorted arrays of sizes m, n, outputs sorted version of union/concat.

Induction step for sort:

1. Observe output is a permutation of A: sort permutes two halves of array, merge preserves existence of elements

By induction

2. Both sort calls output sorted versions of array halves. Guarantee of merge

implies output is sorted ✓

(Won't write / analyze merge here)

More advanced example:

Count # inverted pairs ~ Has trivial $O(n^2)$ alg

Def: Indices i, j is an inversion if
$i < j$ — not necessarily adjacent
and $A[i] > A[j]$

How to divide and conquer?

Key observation:
Any pair $i < j$ must either be:
- Both in first half } count in recursive calls
- Both in second half
- i in first half } count in "merge" step
  j in second half

Count along w/ mergesort

```
function countInv(Array A[size m], Array B[size n]):
    a_ind ← 1; b_ind ← 1; Count ← 0;
    while a_ind ≤ m and b_ind ≤ n:
        if A[a_ind] ≤ B[b_ind]
            a_ind++;
        else
            count += m - a_ind + 1;
            b_ind++;
```

$O(m+n)$ time. Each loop iter increments a count

return count

Guarantee: on input sorted arrays A, B
outputs # inversions in concat(A, B)

How to analyze?

Induction hypothesis:

implicit induction variable = # loop iterations

At the end of every loop iteration,
count holds # inv in concat(A, B[1..b_ind-1])
and a_ind, b_ind increasing

Induction step:
Assume hypothesis holds for prev loop iter. (end of)

Case 1: a_ind increments in cur loop
By induction hypothesis. ✓

Case 2: b_ind increments
Denote val of b_ind before loop iter by old_b_ind
val of b_ind after ∨ (So new_b_ind = old_b_ind+1)
new_b_ind

Know: A[a_ind] > b[old_b_ind]
= b[new_b_ind - 1]
because loop incremented b_ind

Technically relevant only when a_ind > 1

Know: A[a_ind - 1] ≤ b[old_b_ind]
because when a_ind - 1 was incremented,
A[a_ind - 1] ≤ b[prev_b_ind]
≤ b[old_b_ind]

So #inv in concat(A, B[new_b_ind − 1])
− #inv in concat(A, B[old_b_ind − 1])

= m − a_ind + 1 //

Wrap up:

What if loop terminated when b_ind ≤ n?

Last increment must be a_ind.

So A[m] ≤ B[b_ind]

So A[m] ≤ everything in B[b_ind .. n]

So # inv in concat(A, B[1 .. (b_ind − 1)]
= # inv in concat(A, B) □

Longest monochromatic path in binary tree

Given full binary tree of size n
- Each vertex is coloured red or black.

Find longest all-red path. (length ≐ # nodes)

in O(n) time

Trivial $O(n^2)$ alg by trying all red vertices as DFS root

Tree traversal for longest path from root, on both sides.

Observation:
Path either goes through root
or in one of the 2 subtrees

recursion

```
function longestPath(Tree root)
    If root is red
        len1 ← DFS down left subtree for
                longest red path
             + DFS down right subtree for
                longest red path.
    else len1 ← 0
    len2 ← longestPath(left child)
    len3 ← longestPath(right child)

return max(len1, len2, len3)
```

$T(n) \leq 2T(\frac{n}{2}) + c \cdot n \Rightarrow \Theta(n \log n)$

Improve to $O(n)$ since recursion same as DFS!

Induction hypothesis on size of input tree:
Recursive call computes
- longest red path in input tree
- longest red path starting from root

```
function newLongestPath(Tree root)
if root = NULL return (0,0)
    (len_left, len_left_root) ← nLP(left child)

    (len_right, len_right_root) ← nLP(right child)

    If root is red
        len_through_root = len_left_root + 1
                         + len_right_root
        len_root = 1 + max(len_left_root, len_right_root)
    else len_through_root = 0 = len_root
```

return (max (len_left, len_right, len_throughroot),
len_root)

Induction step an exercise.
Be very clear what each quantity means.