Math 182 Lecture 7

Chapter 4 Divide-and-Conquer

Recall: Last time we did Merge-Sort, an example of divide-and-conquer.

Divide: Divide the problem into some number of similar subproblems of smaller size.

Conquer: Solve the subproblems, perhaps by recursively calling same algorithm. If subproblem small enough, then solve directly.

Combine: Combine sols of subproblems to get sol of original problem.

§4.1 The maximum-subarray problem

Maximum-Subarray Problem:

Given an array $A[1..n]$ of numbers, find indices $1 \le i \le j \le n$ s.t. the sum of the numbers in the subarray is maximal among subarrays.

Example:

$$A[1..16] = \langle 13, -3, -25, 20, -3, -16, -23, \underline{18, 20, -7, 12}, -5, -22, 15, -4, 7 \rangle$$

Max subarray is

$A[8..11]$ since

$18+20-7+12 = 43$, and no other subarray gives a bigger sum

```
NAIVE-MAX-~~CROSSING~~-SUBARRAY(A)
 1  max = −∞
 2  max-i = 0
 3  max-j = 0
 4  for i = 1 to A.length          ← all possible i's
 5      for j = i to A.length      ← all possible j's
 6          sum = sum of A[i] through A[j]
 7          if sum > max
 8              max = sum
 9              max-i = i
10              max-j = j
11  return max, max-i, max-j
```

Running time: $n = A.length$

note that (i,j) runs through all pairs s.t. $1 \le i \le j \le n$.

# of pairs $= \frac{n(n+1)}{2}$

= # of subsets of $\{1,\dots,n\}$ of size 2 + # subsets of size 1.

$= \binom{n}{2} + n$

$= \Omega(n^2)$

Q: Can we do better? Can we obtain $o(n^2)$?

A: Yes, w/ divide-and-conquer.

A divide-and-conquer solution

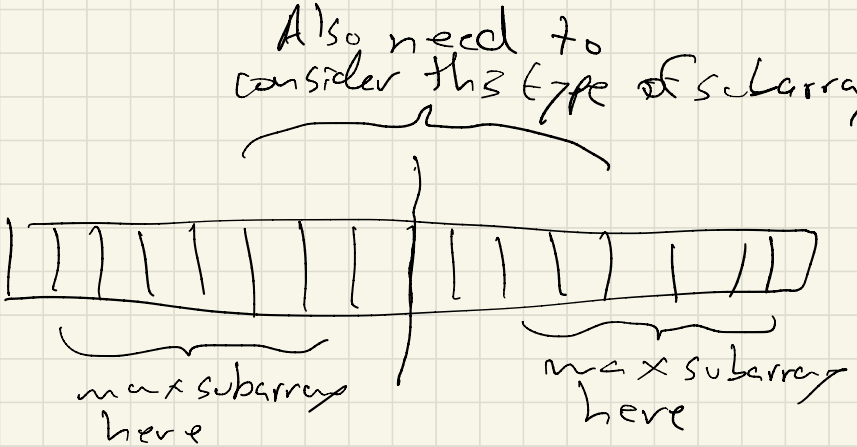
Idea:

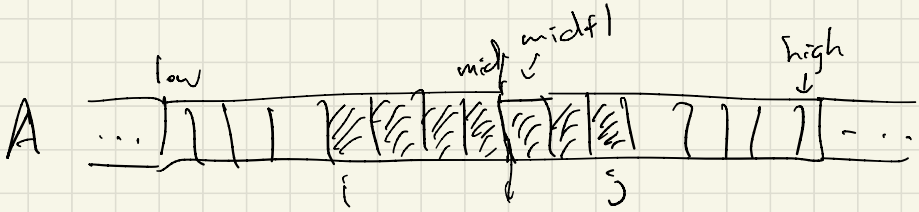
(1) Divide array into left half and right half

(2) Recursively find max subarray on each half

(3) Find max subarray which crosses midpoint

(4) Take the max of these three separate max subarrays.





Subroutine which does (3) above.

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4       key = key + A[i]
 5       if sum > left-sum
 6            left-sum = sum
 7            max-sum = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11       sum = sum + A[j]
12       if sum > right-sum
13            right-sum = sum
14            max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

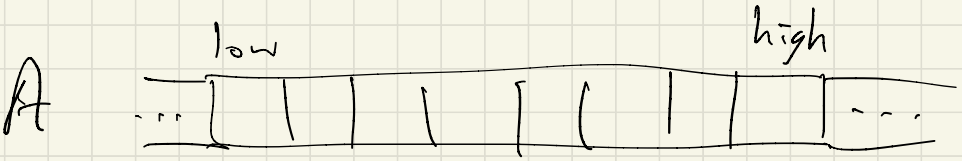
finds max subarray of form $A[i..mid]$ where $low \le i \le mid$

finds max subarray of form $A[mid+1..j]$ where $mid+1 \le j \le high$

Running Time $n = high - low + 1$

Lines 1, 2, 8, 9, 15 are $\Theta(1)$
each iteration of 4-7 and 11-14 runs in $\Theta(1)$
so running time determined by # iterations of both loops
loop 4-7: mid - low + 1 times
loop 11-14: high - mid - 1 + 1 times
high - low + 1 = n times

Thus running time is $\Theta(n)$ linear.



Main algorithm

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2       return (low, high, A[low])
 3       // base case: only one element
 4  else mid = ⌊(low + high)/2⌋
 5       (left-low, left-high, left-sum) =
              FIND-MAXIMUM-SUBARRAY(A, low, mid)
 6       (right-low, right-high, right-sum) =
              FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
 7       (cross-low, cross-high, cross-sum) =
              FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 8       if left-sum ≥ right-sum and left-sum ≥ cross-sum
 9            return (left-low, left-high, left-sum)
10       elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
11            return (right-low, right-high, right-sum)
12  else return (cross-low, cross-high, cross-sum)
```

base case

divide

divide

conquer

recursively call to find max subarrays on each half

finds max subarray which crosses midpoint.

combine

find true max subarray.

(1) line 1: if high == low, then have array of length 1 → base case.

(2) o/w, go to line 4 first compute mid which tells us how to divide array into subarrays

(3) lines 5-7: find 3 candidates for max subarray.

(4) lines 8-12: find true max subarray from 3 candidates.

Running time analysis ($n = high - low + 1$)

(Case 1) Sps $n = 1$. Then run lines 1-2, in constant time $\Theta(1)$.

(Case 2) Sps $n \geq 2$. high > low

Divide: lines 1, 4 $\rightsquigarrow \Theta(1)$

Conquer: line 5: $T(n/2)$
line 6: $T(n/2)$
line 7: $\Theta(n)$

Combine: lines 8-12: $\Theta(1)$

$$T(n) = \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1)$$
$$= 2T(n/2) + \Theta(n)$$

Summarize:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n \geq 2 \end{cases}$$

Since this is same recurrence as Merge-Sort, conclude

$$T(n) = \Theta(n \lg n) = o(n^2)$$

§4.2 The substitution method

Recurrences Def: a *recurrence* for $T(n)$ is a function or inequality which describes $T(n)$ in terms of smaller inputs:

eg $$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n \geq 2. \end{cases}$$

E.g.

- $T(n) = T(n/3) + T(2n/3) + \Theta(n)$
- $T(n) = T(n-1) + \Theta(1)$

Technically recurrence for Merge-Sort is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \quad \leftarrow \text{tells you nothing.} \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n \geq 2. \end{cases}$$

For things like this, we will ignore floors and ceilings if it helps.

For base cases can assume there is some n_0 s.t. for all $n \le n_0$

$$T(n) = \Theta(1)$$

thus can often ignore specifying base case in recurrence.

$$T(n) = 2T(n/2) + \Theta(n)$$

Substitution Method

Idea: Want to get Θ- or O- bound on $T(n)$, $T(n)$ is given by a recurrence.

Substitution method involves two steps:

(1) Guess the solution (somehow)

(2) Use induction to find value of constants which makes the guess work.

Example

The recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

satisfies $T(n) = O(n \lg n)$.

Discussion: Need to find $c > 0, n_0 > 0$ s.t. for all $n \ge n_0$

$$0 < T(n) \le cn \lg n. \quad (*)$$

(First goal): find conditions on c which

makes induction work)

Assume (*) true for $m < n$. Note that

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\le 2c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor + n \quad \text{by (*)} \\ &\le cn \lg n/2 + n \\ &= cn\lg n - cn\lg 2 + n \\ &= cn\lg n - cn + n \\ &= cn\lg n - (c-1)n \end{aligned}$$

want $\rightarrow \le cn\lg n$

require $(c-1)n \ge 0 \iff c-1 \ge 0$

$\iff \boxed{c \ge 1}$

↑ provided c satisfies this, inductive step will work.

What about base case(s)? We want to prove $T(n) = O(n\lg n) \iff \exists c>0, n_0 > 0$ $\forall n \ge n_0 \; 0 < T(n) \le cn\lg n$

Have 2 freedoms we can use:

(1) We are free to increase c.

(2) We are free to increase n_0 to a value we get to choose!

To deal w/ annoying base case(s) do the following:

(3) Choose n_0 large enough s.t. smaller values of n

referenced in recurrence/inequality make sense.

(4) Choose c large enough s.t. inequality holds for base cases.

For instance, sps $T(1) = 1$.

Is there $c \geq 1$ s.t. $1 = T(1) \leq c \cdot 1 \cdot \lg 1$?

no! $= 0$

Thus inductive argument must start after $n=2$ and $n=3$.

Base cases: $n=2, n=3$

$T(2) = 2 \cdot T(1) + 2 = 4$

$T(3) = 2 \cdot T(1) + 3 = 5$

want c big enough s.t.

$T(2) = 4 \leq c\, 2 \lg 2$

$T(3) = 5 \leq c\, 3 \lg 3$

so take $c \geq \max\left\{1, 2, 5/3\lg 3\right\}$

Thus for proof of $T(n) = O(n \lg n)$

$n_0 = 2$ and $c = 2$ will work.