Menu

- Problem: peak finding
  - 1 dimension
  - 2 dimensions

- Technique: *Divide and conquer*

- *details about the 1st pset in the end of the lecture*
Peak Finding: 1D

- Consider an array $A[1…n]$:

  \[
  \begin{array}{cccccccc}
  10 & 13 & 5 & 8 & 3 & 2 & 1 \\
  \end{array}
  \]

- An element $A[i]$ is a peak if it is not smaller than its neighbor(s). I.e.,
  - if $i \neq 1, n$: $A[i] \geq A[i-1]$ and $A[i] \geq A[i+1]$
  - If $i=n$: $A[n] \geq A[n-1]$

- Problem: find any peak.
Peak Finding: Ideas?

- **Algorithm I:**
  - Scan the array from left to right
  - Compare each $A[i]$ with its neighbors
  - Exit when found a peak

- **Complexity:**
  - Might need to scan all elements, so $T(n) = \Theta(n)$
Peak Finding: Ideas II?

- Algorithm II:
  - Consider the middle element of the array and compare with neighbors
    - Else \( A[n/2] \) is a peak!
      (since \( A[n/2-1] \leq A[n/2] \) and \( A[n/2] \geq A[n/2+1] \))
- Running time?
Algorithm II: Complexity
Algorithm II: Complexity

- We have

\[ T(n) = T(n/2) + \Theta(1) \]

- Unraveling the recursion,

\[ T(n) = \Theta(1) + \Theta(1) + \ldots + \Theta(1) = \Theta(\log n) \]

- \( \log n \) is much much much better than \( n \)!
Divide and Conquer

• Very powerful design tool:
  – *Divide* input into multiple disjoint parts
  – *Conquer* each of the parts separately (using recursive call)

• Occasionally, we need to *combine* results from different calls (not used here)
Peak Finding: 2D

- Consider a 2D array \( A[1\ldots n, 1\ldots m] \):

\[
\begin{array}{ccc}
10 & 8 & 5 \\
3 & 2 & 1 \\
7 & 13 & 4 \\
6 & 8 & 3
\end{array}
\]

- An element \( A[i] \) is a 2D peak if it is not smaller than its (at most 4) neighbors.

- Problem: find any 2D peak.
2D Peak Finding: Ideas?
Algorithm I: use the 1D algorithm

• Algorithm I:
  – For each column \( j \), find its *global* maximum \( B[j] \)
  – Apply 1D peak finder to find a peak (say \( B[j] \)) of \( B[1...m] \)
• Running time ?
  …is \( \Theta(n \cdot m) \)
• Correctness:
  – \( B[j] \) not smaller than \( B[j-1], B[j+1] \)
  – For any \( k \), \( B[k] \) not smaller than any element from the \( k \)-th column of \( A \)
  – Therefore, \( B[j] \) not smaller than any element from the columns \( j-1, j \) and \( j+1 \) of \( A \)
  – But this includes all neighbors of \( B[j] \) in \( A \), so \( B[j] \) is a peak in \( A \)
Algorithm I’: use the 1D algorithm

• Observation: 1D peak finder uses only $O(\log m)$ entries of $B$
• We can modify Algorithm I so that it only computes $B[j]$ when needed!
• Total time?
  …only $O(n \log m)$!
  – Need $O(\log m)$ entries $B[j]$
  – Each computed in $O(n)$ time
Algorithm II

- Pick middle column ( \( j = m/2 \) )
- Find global maximum \( a = A[i,m/2] \) in that column (and quit if \( m = 1 \))
- Compare \( a \) to \( b = A[i,m/2-1] \) and \( c = A[i,m/2+1] \)
  - If \( b > a \) then recurse on left columns
  - Else, if \( c > a \) then recurse on right columns
  - Else \( a \) is a 2D peak!
Algorithm II: Example

- Pick middle column ( \( j = m/2 \) )
- Find *global* maximum \( a = A[i,m/2] \) in that column (and quit if \( m=1 \))
- Compare \( a \) to \( b = A[i,m/2-1] \) and \( c = A[i,m/2+1] \)
- If \( b > a \)
  then recurse on left columns
- Else, if \( c > a \)
  then recurse on right columns
- Else \( a \) is a 2D peak!
Algorithm II: Correctness

- Claim: If \( b > a \), then there is a peak among the left columns
- Proof (by contradiction):
  - Assume no peak on the left
  - Then \( b \) must have a neighbor \( b_1 \) with higher value
  - And \( b_1 \) must have a neighbor \( b_2 \) with higher value
  - ...
  - We have to stay on the left side – why?
  - (because we cannot enter the middle column)
  - But at some point, we would run out the elements of the left columns
  - Hence, we have to find a peak at some point
Algorithm II: Complexity

• We have

\[ T(n,m) = T(n, m/2) + \Theta(n) \]

Recursion

• Hence:

\[ T(n,n) = \Theta(n) + \Theta(n) + \ldots + \Theta(n) = \Theta(n \log m) \]

Scanning middle column

\[ \log_2 m \]
Faster than $O(n \log n)$?

- Idea:
  
  Reading only $O(n + m)$ elements, reduce an array of $n \times m$ candidates to an array of $n/2 \times m/2$ candidates.

- Pictorially:

  [Diagram showing the reduction of an array of candidates from $n \times m$ to $n/2 \times m/2$]

read only $O(n + m)$ elements
Faster than $O(n \log n)$?

- Hypothetical algorithm has recursion:

$$T(n, m) = T \left( \frac{n}{2}, \frac{m}{2} \right) + \Theta(n + m)$$

- Hence:

$$T(n, m) = \Theta(n + m) + \Theta \left( \frac{n + m}{2} \right) + \Theta \left( \frac{n + m}{4} \right) + \ldots + \Theta(1) = \Theta(n + m)$$
Towards a linear-time algorithm

What elements are useful to check?

- suppose we find global max on the cross
Towards a linear-time algorithm

What elements are useful to check?

- suppose we find global max on the cross
- if middle element done!
Towards a linear-time algorithm

What elements are useful to check?

- find global max on the cross
- if middle element done!
- o.w. two candidate sub-squares
- determine which one to pick by looking at its neighbors not on the cross (as in Algorithm II)

Claim: The sub-square chosen by the above procedure (if any), always contains a peak of the large square.

BUT: Claim 2: Not every peak of the chosen sub-square is necessarily a peak of the large square. Hence, it is hard to recurse…
First Problem Set

• out tonight, by 9pm
  – part A: theory, due at 11.59pm, Sept 21st
  – part B: implementation, due at 11.59pm, Sept 23rd

• deadline policy:
  – 6 days of credit can be used for delayed homework submission
  – at most 2 days can be used for the same deadline (total of 12 deadlines: 6psets x 2parts)

• details on the class website