Recurrences, Closest Pair and Selection

Lecture 7
February 10, 2015

Solving Recurrences
Two general methods:

- Recursion tree method: need to do sums
  - elementary methods, geometric series
  - integration
- Guess and Verify
  - guessing involves intuition, experience and trial & error
  - verification is via induction

Recurrence: Example I

Consider \( T(n) = 2T(n/2) + n/\lg n \).

- Construct recursion tree, and observe pattern.
- \( i \)th level has \( n_i = 2^i \) nodes.
- problem size at node of level \( i \) is \( n/2^i \).
- work at node of level \( i \) is \( w_i = \frac{n}{2^i} / \lg \frac{n}{2^i} \).
- Total work at \( i \)th level is \( n_i \cdot w_i = 2^i \cdot \frac{n}{2^i} / \lg \frac{n}{2^i} = n / \lg \frac{n}{2^i} \).
- Summing over all levels \( T(n) = \sum_{i=0}^{\lg n-1} n_i \cdot w_i = \sum_{i=0}^{\lg n-1} \frac{n}{\lg \frac{n}{2^i}} = n \sum_{j=1}^{\lg n} \frac{1}{j} = nH_{\lg n} = \Theta(n \log \log n) \)
Recurrence: Example II

- Consider...
- What is the depth of recursion?
  \[\sqrt{n}, \sqrt[3]{n}, \sqrt[4]{n}, \ldots, O(1)\].
- Number of levels: \(n^{2^{-L}} = 2\) means \(L = \log \log n\).
- Number of children at each level is 1, work at each node is 1.
- Thus, \(T(n) = \sum_{i=0}^{L} 1 = \Theta(L) = \Theta(\log \log n)\).

Recurrence: Example III

- Consider \(T(n) = \sqrt{n}T(\sqrt{n}) + n\).
- Using recursion trees: number of levels \(L = \log \log n\).
- Work at each level? Root is \(n\), next level is \(\sqrt{n} \times \sqrt{n} = n\), so on. Can check that each level is \(n\).
- Thus, \(T(n) = \Theta(n \log \log n)\).

Recurrence: Example IV

- Consider \(T(n) = T(n/4) + T(3n/4) + n\).
- Using recursion tree, we observe the tree has leaves at different levels (a lop-sided tree).
- Total work in any level is at most \(n\). Total work in any level without leaves is exactly \(n\).
- Highest leaf is at level \(\log_4 n\) and lowest leaf is at level \(\log_{4/3} n\).
- Thus, \(n \log_4 n \leq T(n) \leq n \log_{4/3} n\), which means \(T(n) = \Theta(n \log n)\).

Part II

Closest Pair
Closest Pair - the problem

**Input**  Given a set $S$ of $n$ points on the plane
**Goal**  Find $p, q \in S$ such that $d(p, q)$ is minimum

Applications

- Basic primitive used in graphics, vision, molecular modelling
- Ideas used in solving nearest neighbor, Voronoi diagrams, Euclidean MST

Algorithm: Brute Force

1. Compute distance between every pair of points and find minimum.
2. Takes $O(n^2)$ time.
3. Can we do better?

Closest Pair: 1-d case

**Input**  Given a set $S$ of $n$ points on a line
**Goal**  Find $p, q \in S$ such that $d(p, q)$ is minimum

Algorithm

1. Sort points based on coordinate
2. Compute the distance between successive points, keeping track of the closest pair.

Running time $O(n \log n)$

1. Can we do this in better running time?
2. Can reduce Distinct Elements Problem (see lecture 1) to this problem in $O(n)$ time. Do you see how?
Generalizing 1-d case

- Can we generalize 1-d algorithm to 2-d?
- Sort according to \(x\) or \(y\)-coordinate??
- No easy generalization.

First Attempt

**Divide and Conquer I**

- Partition into 4 quadrants of roughly equal size.
- Find closest pair in each quadrant recursively.
- Combine solutions.
- But... How to partition the points in a balanced way?

New Algorithm

**Divide and Conquer II**

- Divide the set of points into two equal parts via vertical line.
- Find closest pair in each half recursively.
- Find closest pair with one point in each half
- Return the best pair among the above 3 solutions

New Algorithm

**Divide and Conquer II**

- Divide the set of points into two equal parts via vertical line
- Find closest pair in each half recursively
- Find closest pair with one point in each half
- Return the best pair among the above 3 solutions

- Sort points based on \(x\)-coordinate and pick the median. Time \(= O(n \log n)\)
- How to find closest pair with points in different halves? \(O(n^2)\) is trivial. Better?
Combining Partial Solutions

- Does it take $O(n^2)$ to combine solutions?
- Let $\delta$ be the distance between closest pairs, where both points belong to the same half.

Sparsity of Band

Divide the band into square boxes of size $\delta/2$

**Lemma**
Each box has at most one point

**Proof.**
If not, then there are a pair of points (both belonging to one half) that are at most $\sqrt{2}\delta/2 < \delta$ apart!

Searching within the Band

**Lemma**
Suppose $a, b$ are both in the band $d(a, b) < \delta$ then $a, b$ have at most two rows of boxes between them.

**Proof.**
Each row of boxes has height $\delta/2$. If more than two rows then $d(a, b) > 2 \cdot \delta/2!$
Corollary
Order points according to their y-coordinate. If \( p, q \) are such that \( d(p, q) < \delta \) then \( p \) and \( q \) are within 11 positions in the sorted list.

Proof.
1. \( \leq 2 \) points between them if \( p \) and \( q \) in same row.
2. \( \leq 6 \) points between them if \( p \) and \( q \) in two consecutive rows.
3. \( \leq 10 \) points between if \( p \) and \( q \) one row apart.
4. \( \Rightarrow \) More than ten points between them if \( d(p, q) > \delta \). A contradiction.

The Algorithm

\[
\text{ClosestPair}(P) :
\]
1. Find vertical line \( L \) splits \( P \) into equal halves: \( P_1 \) and \( P_2 \)
2. \( \delta_1 \leftarrow \text{ClosestPair}(P_1) \)
3. \( \delta_2 \leftarrow \text{ClosestPair}(P_2) \)
4. \( \delta = \min(\delta_1, \delta_2) \)
5. Delete points from \( P \) further than \( \delta \) from \( L \)
6. Sort \( P \) based on y-coordinate into an array \( A \)
7. for \( i = 1 \) to \( \lfloor |A|/2 \rfloor \)
   for \( j = i + 1 \) to \( \min\{i + 11, |A|\} \)
   if \( \text{dist}(A[i], A[j]) < \delta \) update \( \delta \) and closest pair

- Step 1, involves sorting and scanning. Takes \( O(n\log n) \) time.
- Step 5 takes \( O(n) \) time.
- Step 6 takes \( O(n\log n) \) time
- Step 7 takes \( O(n) \) time.

Running Time
The running time of the algorithm is given by

\[
T(n) \leq 2T(n/2) + O(n\log n)
\]

Thus, \( T(n) = O(n\log^2 n) \).

Improved Algorithm
Avoid repeated sorting of points in band: two options
1. Sort all points by y-coordinate and store the list. In conquer step use this to avoid sorting
2. Each recursive call returns a list of points sorted by their y-coordinates. Merge in conquer step in linear time.

Analysis: \( T(n) \leq 2T(n/2) + O(n) = O(n\log n) \)
Quick Sort

Quick Sort [Hoare]
- Pick a pivot element from array
- Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$
- Recursively sort the subarrays, and concatenate them.

Example:
- array: 16, 12, 14, 20, 5, 3, 18, 19, 1
- pivot: 16
- split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
- put them together with pivot in middle

Problem - Selection

Input Unsorted array $A$ of $n$ integers
Goal Find the $j$th smallest number in $A$ (rank $j$ number)

Example
$A = \{4, 6, 2, 1, 5, 8, 7\}$ and $j = 4$. The $j$th smallest element is 5.

Median: $j = \lfloor (n + 1)/2 \rfloor$

Time Analysis
- Let $k$ be the rank of the chosen pivot. Then,
  \[ T(n) = T(k - 1) + T(n - k) + O(n) \]
- If $k = \lceil n/2 \rceil$ then
  \[ T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n). \]
Then, $T(n) = O(n \log n)$.
  - Theoretically, median can be found in linear time.
- Typically, pivot is the first or last element of array. Then,
  \[ T(n) = \max_{1 \leq k \leq n} (T(k - 1) + T(n - k) + O(n)) \]
In the worst case $T(n) = T(n - 1) + O(n)$, which means $T(n) = O(n^2)$. Happens if array is already sorted and pivot is always first element.

Algorithm I
- Sort the elements in $A$
- Pick $j$th element in sorted order
Time taken = $O(n \log n)$

Do we need to sort? Is there an $O(n)$ time algorithm?
Algorithm II
If $j$ is small or $n-j$ is small then
1. Find $j$ smallest/largest elements in $A$ in $O(jn)$ time. (How?)
2. Time to find median is $O(n^2)$.

Divide and Conquer Approach
- Pick a pivot element $a$ from $A$
- Partition $A$ based on $a$.
  $A_{\text{less}} = \{x \in A \mid x \leq a\}$ and $A_{\text{greater}} = \{x \in A \mid x > a\}$
- $|A_{\text{less}}| = j$: return $a$
- $|A_{\text{less}}| > j$: recursively find $j$th smallest element in $A_{\text{less}}$
- $|A_{\text{less}}| < j$: recursively find $k$th smallest element in $A_{\text{greater}}$ where $k = j - |A_{\text{less}}|$. 

Time Analysis
- Steps:
  - Partitioning step: $O(n)$ time to scan $A$
  - How do we choose pivot? Recursive running time?
- Suppose we always choose pivot to be $A[1]$.
- Say $A$ is sorted in increasing order and $j = n$.
- Exercise: show that algorithm takes $\Omega(n^2)$ time.

A Better Pivot
- Suppose: pivot $\ell$th smallest element where $n/4 \leq \ell \leq 3n/4$.
- That is pivot is approximately in the middle of $A$.
- $\Rightarrow n/4 \leq |A_{\text{less}}| \leq 3n/4$ and $n/4 \leq |A_{\text{greater}}| \leq 3n/4$.
- If we apply recursion,
  \begin{equation}
  T(n) \leq T(3n/4) + O(n)
  \end{equation}
  Implies $T(n) = O(n)$!
- How do we find such a pivot?
- Randomly? This works!
  Analysis a little bit later.
- Can we choose pivot deterministically?
Divide and Conquer Approach

A game of medians

Idea

1. Break input $A$ into many subarrays: $L_1, \ldots, L_k$.
2. Find median $m_i$ in each subarray $L_i$.
3. Find the median $x$ of the medians $m_1, \ldots, m_k$.
4. Intuition: The median $x$ should be close to being a good median of all the numbers in $A$.
5. Use $x$ as pivot in previous algorithm.

But we have to be...

More specific...

1. Size of each group?
2. How to find median of medians?

Algorithm for Selection

A storm of medians

select($A$, $j$):

Form lists $L_1, L_2, \ldots, L_{\lceil n/5 \rceil}$, where $L_i = \{A[5i - 4], \ldots, A[5i]\}$

Find median $b_i$ of each $L_i$ using brute-force

Find median $b$ of $B = \{b_1, b_2, \ldots, b_{\lceil n/5 \rceil}\}$

Partition $A$ into $A_{\text{less}}$ and $A_{\text{greater}}$ using $b$ as pivot

if $|A_{\text{less}}| = j$ return $b$
else if $|A_{\text{less}}| > j$
    return select($A_{\text{less}}$, $j$)
else
    return select($A_{\text{greater}}$, $j - |A_{\text{less}}|$)

How do we find median of $B$? Recursively!

Choosing the pivot

A clash of medians

1. Partition array $A$ into $\lceil n/5 \rceil$ lists of 5 items each.

   $L_1 = \{A[1], A[2], \ldots, A[5]\}, L_2 = \{A[6], \ldots, A[10]\},$

   $\ldots, L_i = \{A[5i + 1], \ldots, A[5i - 4]\}, \ldots,$

   $L_{\lceil n/5 \rceil} = \{A[5\lceil n/5 \rceil - 4], \ldots, A[n]\}$.

   For $i = 1, \ldots, n/5$: compute median $b_i$ of $L_i$

   ...using brute-force in $O(1)$ time. Total $O(n)$ time

   Let $B = \{b_1, b_2, \ldots, b_{\lceil n/5 \rceil}\}$

   Find median $b$ of $B$

Lemma

Median of $B$ is an approximate median of $A$. That is, if $b$ is used a pivot to partition $A$, then $|A_{\text{less}}| \leq 7n/10 + 6$ and $|A_{\text{greater}}| \leq 7n/10 + 6$.

Running time of deterministic median selection

A dance with recurrences

$T(n) = T(\lceil n/5 \rceil) + \max\{T(|A_{\text{less}}|), T(|A_{\text{greater}}|)\} + O(n)$

From Lemma,

$T(n) \leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 + 6 \rfloor) + O(n)$

and

$T(1) = 1$

Exercise: show that $T(n) = O(n)$
Median of Medians: Proof of Lemma

**Proposition**

There are at least\( \frac{3n}{10} - 6 \) elements greater than the median of medians \( b \).

**Proof.**

At least half of the \( \lceil n/5 \rceil \) groups have at least 3 elements larger than \( b \), except for last group and the group containing \( b \). So \( b \) is less than

\[
3(\lceil 1/2 \lceil n/5 \rceil \rceil - 2) \geq \frac{3n}{10} - 6
\]

Figure: Shaded elements are all greater than \( b \)

Questions to ponder

- Why did we choose lists of size 5? Will lists of size 3 work?
- Write a recurrence to analyze the algorithm’s running time if we choose a list of size \( k \).
Median of Medians Algorithm

Due to:
"Time bounds for selection".

How many Turing Award winners in the author list?
All except Vaughn Pratt!

Takeaway Points

- Recursion tree method and guess and verify are the most reliable methods to analyze recursions in algorithms.
- Recursive algorithms naturally lead to recurrences.
- Some times one can look for certain type of recursive algorithms (reverse engineering) by understanding recurrences and their behavior.