CSE 101, Winter 2018

Design and Analysis of Algorithms

Lecture 5: Divide and Conquer (Part 2)

Class URL: http://vlsicad.ucsd.edu/courses/cse101-w18/
A Lower Bound on Convex Hull

• Task: sort the set of numbers \{3,1,4,5,2\}

• Task: find the convex hull of the set of points \{(3,9), (1,1), (4,16), (5,25), (2,4)\}
A Lower Bound on Convex Hull

- **Given:** Sorting LB is $\Omega(n \log n)$
- **Reduction from Sorting to Convex Hull**
  - **Input:** an arbitrary instance $I_{\text{SORT}}$ of SORT
  - **Transform** $I_{\text{SORT}} = \{x_1, x_2, \ldots, x_n\}$ into an instance $I_{\text{C-HULL}} = \{(x_1, x_1^2), (x_2, x_2^2), \ldots, (x_n, x_n^2)\}$ of C-HULL
  - **Solve** the C-HULL problem for instance $I_{\text{C-HULL}}$
  - **Transform** solution of $I_{\text{C-HULL}}$ to solution of $I_{\text{SORT}}$

At the end of last time:
1. An instance of the SORT problem, $I_{\text{SORT}}$, is a set of numbers $\{x_1, \ldots, x_n\}$
2. An instance of the C-HULL problem is a set of $(x, y)$ points in the plane.
3. Say that we are given an *arbitrary instance* of the sorting problem (which means “Given *any* instance of the sorting problem”), $I_{\text{SORT}}$
4. We can generate from $I_{\text{SORT}}$ an instance of C-HULL, $I_{\text{C-HULL}}$, $\{(x_1, x_1^2), \ldots, (x_n, x_n^2)\}$, in $O(n)$ time.
5. If we can come up with a solution of the C-HULL problem for instance $I_{\text{C-HULL}}$, then we can generate a solution to the given SORT instance $I_{\text{SORT}}$ in $O(n)$ time.
6. This reduction – *from* the SORT problem *to* the C-HULL problem – allows us to establish a lower bound on the complexity of C-HULL
7. Suppose there exists an algorithm ALG which solves the C-HULL problem in, say, $O(n)$ time. Then, for an arbitrary instance of the sorting problem, we could apply (4) = $O(n)$ time, then ALG = $O(n)$ time, then (5) = $O(n)$ time to achieve an $O(n)$ algorithm for sorting.
8. But this would violate the known lower bound on the time complexity of sorting.
9. Our reduction *from* the SORT problem *to* the C-HULL problem enables us to “transfer” the $\Omega(n \log n)$ LB for SORT to the C-HULL problem.
A Lower Bound on Convex Hull

- **Given:** Sorting LB is $\Omega(n \log n)$
- **Reduction from Sorting to Convex Hull**
  - **Input:** an arbitrary instance $I_{\text{SORT}}$ of SORT
  - **Transform** $I_{\text{SORT}} = \{x_1, x_2, \ldots, x_n\}$ into an instance $I_{\text{C-HULL}} = \{(x_1, x_1^2), (x_2, x_2^2), \ldots, (x_n, x_n^2)\}$ of C-HULL
  - **Solve** the C-HULL problem for instance $I_{\text{C-HULL}}$
  - **Transform** solution of $I_{\text{C-HULL}}$ to solution of $I_{\text{SORT}}$

- **Note:** Each **Transform** has $O(n)$ complexity
  $\Rightarrow$ C-HULL solver cannot be faster than $\Omega(n \log n)$
The Closest Pair Problem

- **Closest Pair:** Given \( n \) points in the plane, return the closest pair
- Naïve \( \Theta(n^2) \)
- D/Q Approach: \( \Theta(n \log n) \)
  - (1) Split into two pointsets \( S_1, S_2 \) by x-coord = natural order
  - (2) Find closest pair distances \( d_1, d_2 \) in \( S_1, S_2 \) respectively
    - without loss of generality, can assume \( d_1 < d_2 \)
  - (3) Merge the solutions
    - Do we have to check all \( s_1-s_2 \) pairs, \( s_1 \in S_1, s_2 \in S_2 \)?
    - What if there are lots of points in middle strip?
  - Key: Step (3)
    - Observation: There are at most \( O(1) \) points in middle strip with \( \Delta y \leq d_1 \)
DQ Closest Pair

• (1) sort by x- and y-coords tool: one-time sort $O(n \log n)$
• (2) solve subproblems of size $n/2$ $2T(n/2)$
• (3) eliminate points outside strips $O(n)$
• (4) “sort” by y-coord within strips $O(n)$
• (5) compare each point to $\leq 6$ neighbors $O(n)$

Complexity

- $O(n \log n) + T(n) = 2T(n/2) + O(n)$
- $T(2) = 1$
- $T(n) \in O(n \log n)$

Again, with one-time sort, can always output subproblems in y-sorted order.
DQ Closest Pair Comment

• Note that we **exploited** geometry
• Example: what is the maximum possible degree of a vertex in a minimum spanning tree?
  – If the problem is geometric? “planar pointset”
  – If the problem is non-geometric? “edge-weighted graph”
The Selection Problem

• The **SELECTION** problem
  – Given a list L and a number k
  – **Select** (L, k) returns the \( k^{th} \) smallest element of L
    Note: if \( k = |L|/2 \), then **Select**(L,k) returns the **median**

• What is an efficient algorithm?
  Note: Sorting + finding \( k^{th} \) smallest \( \rightarrow O(n \log n) \)

**Can do better using DQ**  (maybe, recalling idea of QSort “pivot”)
DQ Selection

- Pick a value $v$
- Split $S$ into three parts: $S_{<v}$, $S_{=v}$, $S_{>v}$

Select $(S,k) =$
- $\text{Select}(S_{<v}, k)$ if $k \leq |S_{<v}|$ (look in left)
- $v$ if $|S_{<v}| < k \leq |S_{<v}| + |S_{=v}|$
- $\text{Select}(S_{>v}, k - |S_{<v}| - |S_{=v}|)$ if $k > |S_{<v}| + |S_{=v}|$ (look in right)

- Single split operation effectively reduces size of search space
  - By how much? Depends on $v$ !!!
  - If we could magically pick $v$ so that $|S_{<v}|, |S_{>v}| \approx |S|/2$
    then we would have $T(n) \leq T(n/2) + O(n) \Rightarrow T(n) = O(n)$

"balanced"

Time on array of size $n$

Time for "Split"
DQ Selection

- Pick a “splitter” value $v$
- “Split” $S$ into three parts: $S_{<v}$, $S_{=v}$, $S_{>v}$

Select $(S,k) =$

- $\text{Select}(S_{<v},k)$ if $k \leq |S_{<v}|$ (look in left)
- $v$ if $|S_{<v}| < k \leq |S_{<v}| + |S_{=v}|$
- $\text{Select}(S_{>v},k - |S_{<v}| - |S_{=v}|)$ if $k > |S_{<v}| + |S_{=v}|$ (look in right)

- Each split operation effectively reduces size of search space
  - But, by how much? Depends on $v$!!!
  - If we could magically pick $v$ so that $|S_{<v}|$, $|S_{>v}| \approx |S|/2$ then we would have $T(n) \leq T(n/2) + O(n) \Rightarrow T(n) = O(n)$
Randomized Selection (1)

- We can’t “magically” find a “balanced” $v$, so let’s choose $v$ randomly from the array $S$
- Worst case is $\Omega(n^2)$ i.e., $n + (n-1) + \ldots + 1$ (like Qsort worst case)
  - But this is highly unlikely $\Rightarrow$ let’s try to analyze expected runtime:

$$T(n) = \text{expected time for Select on array of size } n$$

- Suppose we could guarantee a “good” $v$ as the splitter, e.g., such that $n/4 \leq |S_{<v}|, |S_{>v}| \leq 3n/4$
  $\Rightarrow$ subproblem size $\leq 3n/4$

???

$$T(n) \leq T(3n/4) + O(n)$$
Randomized Selection (2)

- Suppose we could guarantee a “good” \( v \) as the splitter, e.g., such that \( n/4 \leq |S_{<v}|, |S_{>v}| \leq 3n/4 \)
  \[ \Rightarrow \text{subproblem size} \leq 3n/4 \]

- Let \( R(n) \equiv \text{expected number of split operations needed before array is reduced to } \leq 3n/4 \text{ elements} \)

\[ T(n) \leq T(3n/4) + O(n \times R(n)) \]

- \( R(n) = \text{find a “balanced” splitter } v \)
- \( O(n) = \text{perform split operation, make subproblem} \)
- \( T(3n/4) = \text{solve subproblem} \)
Randomized Selection (3)

• Suppose we could guarantee a “good” \( v \) as the splitter, e.g., such that \( n/4 \leq |S_{<v}|, |S_{>v}| \leq 3n/4 \)
  \[ \Rightarrow \text{subproblem size} \leq 3n/4 \]

• Let \( R(n) \equiv \text{expected number of split operations needed before array is reduced to} \leq 3n/4 \text{ elements} \)

\[
T(n) \leq T(3n/4) + O(n \times R(n))
\]

\( R(n) = \text{find a “balanced” splitter} \ v \)
\( O(n) = \text{perform split operation, make subproblem} \)
\( T(3n/4) = \text{solve subproblem} \)

Fact: \( R(n) \leq 2 \implies T(n) \leq T(3n/4) + O(n) \)

*** Why is \( R(n) \leq 2 \) ?
Analysis: DQ Selection is $O(n)$

- **Suppose**: $t(n) \leq dn + t(3n/4)$
- **Claim**: $t(n) \leq kn$ for some $k$ would follow

- **Proof strategy**: “**Constructive Induction**” (“**Substitution**”)
  - **Strong Induction Hypothesis**: $t(m) \leq km$ for $m \leq n-1$
  - **Induction Step**: $t(n) \leq dn + k(3n/4)$
    $$= (d + 3k/4)n$$
    which we want to be equivalent to $t(n) \leq kn$
  - But this will be true as long as we pick a value of $k \geq 4d$
The MaxMin Problem

- **MaxMin**: Given list of n numbers, return largest and smallest
- Naïve: 2(n-1) comparisons (two passes)
- DQ approach
  - \( n = 1 \rightarrow 0 \) comparisons needed
  - \( n = 2 \rightarrow 1 \) comparison needed
  - else: bisect list
    - make recursive calls
    - return \( \max(\max_1, \max_2), \min(\min_1, \min_2) \)
- **#comparisons**: \( T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2, n > 2 \)
**Intuitive MaxMin Lower Bound**

- **“Information argument”**
  - Start: Nothing known about n elements
  - End: “Neither Max nor Min” known about all but 2 elements

- **Four “buckets”**
  - Know Nothing
  - Not Max
  - Not Min
  - Neither Max nor Min

![Diagram showing the relationships between the four buckets](attachment://diagram.png)
DQ for the MaxMin Problem

• $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2$, $n > 2$

• Transform with $S(k) = T(2^k)$

\[
S(k) = 2S(k-1) + 2
\]

\[
S(k) - 2S(k-1) = 2
\]

\[
= 1^n \cdot 2
\]

C.P. = $(x - 2)(x - 1)$ with roots 2, 1

\[
S(k) = c_12^k + c_21^k
\]
DQ for the MaxMin Problem

- $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2, \ n > 2$
- Transform with $S(k) = T(2^k)$

$$S(k) = c_12^k + c_21^k$$

Initial Conditions

- $S(1) = c_1 \cdot 2 + c_2 = 1$
- $S(2) = c_1 \cdot 4 + c_2 = 4$

$\Rightarrow c_1 = 3/2, \ c_2 = -2$

$$T(n) = T(2^k) = S(k) = \frac{3}{2} \cdot 2^{\log_2 n} - 2 = \frac{3n}{2} - 2$$
DQ for the MaxMin Problem

- \( T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2, \ n > 2 \)
- Transform with \( S(k) = T(2^k) \Rightarrow S(k) = 2S(k-1) + 2 \)

**Note:** The recurrence \( a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n \ p(n) \) has solution \( t_n = \sum_{i=1}^{k} c_i r_i^n \) where \( r_i \) are roots of the C.P.: 
\( (a_0 x^k + a_1 x^{k-1} + \ldots + a_k) (x - b)^{d+1} = 0 \)

\[ S(n) - 2S(n-1) = 1^n \cdot 2 \]

\( \Rightarrow a_0 = 1, a_1 = -2, \ b = 1, \ p(n) = 2, \ d = 0 \)
\( \Rightarrow \text{C.P.} = (x-2)(x-1)^1 \)
\( \Rightarrow S(k) = c_1 2^k + c_2 1^k \)

- Initial conditions: \( S(1) = c_1 \cdot 2 + c_2 = 1 \)
\( S(2) = c_1 \cdot 4 + c_2 = 4 \Rightarrow c_1 = 3/2, \ c_2 = -2 \)
- \( T(n) = T(2^k) = S(k) = 3/2 \cdot 2^{\log_2 n} - 2 = 3n/2 - 2 \)

i.e., \( n \) a power of 2
Sorting (With Comparisons)

• Input: sequence of numbers
  Output: a sorted sequence
• Observe: Sorting == Identifying a Permutation

KEY OBSERVATIONS
1. We need at least as many leaves in this tree as there are possible outcomes (= n! permutations of n elements)
2. The number of comparisons needed to get to the leaf at greatest depth from the root is the worst-case complexity of the algorithm that is embodied by this comparison tree
Searching an Ordered List

- Input: ordered list $L$ of $n$ numbers, and a target number $x$
- Output: Find $x$ if it exists in $L$
- Observe: Searching == Identifying the index of $x$ in $L$

**KEY OBSERVATIONS**

1. We generally use binary search, which takes $\log(n)$ time in the worst case. Why is binary search the best possible strategy? (*)
2. The number of comparisons needed to get to the leaf at greatest depth from the root is the worst-case complexity of the algorithm that is embodied by this comparison tree
3. Need at least $n$ leaves
4. Height of comparison tree is $\Omega(\log n)$

(*) Interpolation Search is better than binary search when we know something about the distribution of the elements we are searching. For example, when humans use Interpolation Search when looking up something in a dictionary or a phone book.
A Lower Bound on Sorting Complexity

• In the “comparison model of computation”, can we find a lower bound on the complexity of sorting?
• From last slide: Sorting \equiv Identifying Permutation
• Binary Tree: Root at level (height) 0
• **Theorem:**
  – There exists some c > 0 such that for all algorithms which use comparisons to sort, and for all input sizes n, at least one input requires cn \log n comparisons
• **Fact:**
  – Binary tree of height h has at most 2^h leaves
• **Observation from last slide:**
  – n! leaves needed \implies comparison tree must have h \geq \log(n!)
  – h is maximum (= worst-case) #comparisons needed to sort input of size n using the corresponding algorithm
Sorting Lower Bound (DETAIL)

- **Goal:** \( \log(n!) \in \Theta(n \log n) \)
- **Claim:** \( \log(n!) \in O(n \log n) \)
  
  \[
  n! \leq n^n \Rightarrow \log n! \leq n \log n
  \]

- **Claim:** \( \log(n!) \in \Omega(n \log n) \)
  
  \[
  n! \geq \left(\frac{n}{2}\right)^{n/2} \Rightarrow \log n! \geq \frac{n}{2} \log(n/2)
  \]
  
  \[
  \Rightarrow 2 \log n! \geq n \log \left(\frac{n}{2}\right)
  \]

  **Observe:** \( \log \left(\frac{n}{2}\right) + 1 = \log n \)

  \[
  2 \log \left(\frac{n}{2}\right) \geq \log n \quad \forall n > 2
  \]

  \[
  \Rightarrow 4 \log n! \geq n \cdot 2 \log \left(\frac{n}{2}\right)
  \]

  \[
  \Rightarrow 4 \log n! \geq n \log n
  \]
What About LB For the Average Case?

Of possible interest:
http://www.academia.edu/693793/A_simplified_derivation_of_timing_complexity_lower_bounds_for_sorting_by_comparisons

• Worst-case analysis can be uninformative
  – QuickSort: worst case is $n^2$
  – Simplex Method for linear programming: worst case is exponential

• Can we lower-bound “average case” complexity?

• First question: Is “average case” well-defined?
  – Want $\sum_i p_i d_i \equiv$ expected depth of a leaf in the comparison tree
  – $d_i \equiv$ depth of leaf $i$; $i = \text{input with probability } = p_i$
  – Assume all input permutations equally probable (“equiprobable”)
Average-Case Complexity of Sorting

• Q: Even though all sorting algs have some input which requires $n \log n$ time, is there an algorithm with better than $(n \log n)$ average-case performance?

• Theorem: If all $n!$ input permutations equiprobable, then any decision tree that sorts has expected depth $\Omega(n \log n)$.
  – Let $D(m)$ be smallest sum of leaf depths over all binary trees with $m$ leaves
  – Claim: $D(m) \geq m \log m$.
  – If Claim true, use $m = n!$ and fact that $\log n! \in \Theta(n \log n)$
    \[ \rightarrow D(n!) \geq n! \log n! \rightarrow \text{average leaf depth is } \Omega(n \log n) \]
Average-Case Complexity of Sorting (cont.)

- **Claim**: $D(m) \geq m \log m$

  Proof by induction on $m$.
  
  $(D(T) \equiv$ sum of leaf depths of tree $T$, where unambiguous.)
  
  Claim trivial for $m = 1$; assume Claim $\forall m < k$ *(strong I.H.)*.
  
  Any tree $T$ with $k$ leaves can be viewed as a root and two subtrees $T_i$ and $T_{k-i}$ (with $i$ and $k-i$ leaves respectively)
  
  $\Rightarrow D(T) = i + D(T_i) + (k-i) + D(T_{k-i})$
  
  $D(k) = \min_{1 \leq i \leq k} \left[ k + D(i) + D(k-i) \right]$
  
  $\geq k + \min_i [D(i) + D(k-i)]$
  
  $\geq k + \min_i [i \log i + (k-i) \log (k-i)] \quad$ *(by I.H.)*
  
  which is minimized for $i = k/2$.
  
  $\Rightarrow D(k) \geq k + k \log (k/2) = k + k(\log k - 1) = k \log k$