Big Data Algorithms Oct 18, 2017

Week 6: Algorithmic Lower Bound for Data of Fixed Distribution

Prof. Xiaotie Deng *Scribed by Yinghong Fang*

Contents

0 Notation

- T(n)
  The time complexity for input size n.
- big O
  Intuitive understanding: If a = $O(b)$, a $\leq$ b
  More specific, see `https://en.wikipedia.org/wiki/Big_O_notation`
- Ω
  Intuitive understanding: If a = $\Omega(b)$, a $\geq$ b
  More specific, see `https://en.wikipedia.org/wiki/Big_Omega_function`

1 Warm-up

We know the average time complexity for *quick sort*(`https://en.wikipedia.org/wiki/Quicksort`) is $O(nlogn)$. And merge sort can achieve O($nlogn$) in the worst case. O($nlogn$) is the upper bound, but what is the lower bound?

From the following discussion, we can find that time complexity's lower bound for sort algorithms (based on comparison) in the worst case is also $\Omega(nlogn)$! For each of these algorithm based on comparison (quick sort, merge sort...), **we can produce a sequence of n input numbers that causes the algorithm to run in** $\Omega(nlogn)$ **time.**

However, it doesn't say lower bound for sorting algorithm is $\Omega(nlogn)$. See a counter example: counting sort `https://en.wikipedia.org/wiki/Counting_sort` for more if interested.

2 Compute Lower Bound for Comparison Sorting

Proof. For each of these algorithms based on comparison (quick sort, merge sort...), we can produce a sequence of n input numbers that causes the algorithm to run in $\Omega(nlogn)$ time.

1. There are n! permutations for n numbers (assume n numbers are different).

   Example:
   Input:
   a_1 a_2 a_3
   Output:
   a_1 a_2 a_3
   a_1 a_3 a_2
   a_2 a_1 a_3
   a_2 a_3 a_1
   a_3 a_1 a_2
   a_3 a_2 a_1
   n = 3, and result have 3! = 6 permutations

   That is, to sort n numbers, the sorted results will have n! possibilities. In other words, **the possible solution set's size is n!**.
   For the above situation, the possible solution set is $\{(a_1, a_2, a_3), (a_1, a_3, a_2), (a_2, a_1, a_3), (a_2, a_3, a_1), (a_3, a_1, a_2), (a_3, a_2, a_1)\}$. And $|\{(a_1, a_2, a_3), (a_1, a_3, a_2), (a_2, a_1, a_3), (a_2, a_3, a_1), (a_3, a_1, a_2), (a_3, a_2, a_1)\}| = n!$

2. Each comparison(a_i, a_j) splits the solution set into two part. part1: $a_i > a_j$, part2: $a_i \leq a_j$.

   Example:
   Input:
   Using the above example. We pick two number(a_1, a_2) to compare them.

   - If $a_1 > a_2$, then our updated solution set will be $\{(a_1, a_2, a_3), (a_1, a_3, a_2), (a_3, a_1, a_2)\}$.
   - Otherwise $(a_1 \leq a_2)$, then our updated solution set will be $\{(a_2, a_1, a_3), (a_2, a_3, a_1), (a_3, a_2, a_1)\}$.

3. We continue choosing two number, comparing it and updating solution set until the solution set has only one item. Then the only item is the sorted result. And we count how many comparisons/cuts are needed.

We want to prove that "For each of these algorithms based on comparison (quick sort, merge sort...), we can produce **a sequence of n input numbers** that causes the algorithm to run in $\Omega(nlogn)$ time."

So, we let the **adversary** be the worst sequence of n input numbers in the above context. That is, an **adversary** is the input data which makes the worst case. For *insert sort*, the **adversary** is a decreasing sequence-(9 8 7 5 4). In this case, insert sort will cost n^2 time. For an increasing sequence-(4 5 7 8 9), insert sort will cost only n time.

Idea:
For comparison based sorting:

- What our designed smart sorting algorithm does is:
  Choose two number for comparison, and split the whole solution set into two smaller solution sets part1, part2. (part1 + part2 = the whole solution set)
- What the evil adversary does is:
  Let the updated solution set be the large one from part1 and part2.

Example:
Suppose the current solution set is $\{(a_1, a_2, a_3), (a_1, a_3, a_2), (a_3, a_1, a_2)\}$.

- What our designed smart sorting algorithm does is:
  choose to compare a_2, a_3.
- What the evil adversary does is:
  The evil adversary can control the input data because it is the input data itself. The evil adversary thinks that if $a_2 > a_3$, the updated solution set will have only item $\{(a_1, a_2, a_3)\}$, oh, no!. But if $a_2 \leq a_3$, then the updated solution set will have two item $\{(a_1, a_3, a_2), (a_3, a_1, a_2)\}$ left.
  Of course, the adversary will let $a_2 \leq a_3$, because it makes the worst case.

Now, count how many comparison/cuts needed:

1. The solution set's size is n!.
2. Carefully choose two numbers(a_i, a_j), so that they will give equal split.
   Because the adversary will choose the large one, so an equal split is the best choice.

3. Since every comparison will cut the solution at most half, then needed cuts at least are $log_2(|\text{The whole solution set}|) = log_2(n!) \geq nlogn$.
   Let k be the needed cuts, then $\frac{|n!|}{2^k} <= 1$, then $k \geq log_2(n!)$
   Using Stirling's approximation $n! \approx \sqrt{2\pi n}\ \left(\frac{n}{e}\right)^n$, $log_2(n!) \geq log_2(\left(\frac{n}{e}\right)^n) = nlog_2 n - nlog_2 e = \Omega(nlogn)$

3 More: Game Tree Model

Let's play a game, see Figure 1. The rules are followings:

- The current node is the root node.
- Two players (player1, player2) play in turn. Each player chooses a children node from the current node. And the current node updates to the children node.
- If the current node is a leaf node, then the game finishes. The player1' score is the leaf node's number. And the player2' score is (10 - leaf node's number).
- Player1 plays first.

So, the player1 wants to max the result value, but the player2 wants to the min the result value, and the player1 wants min the result value...

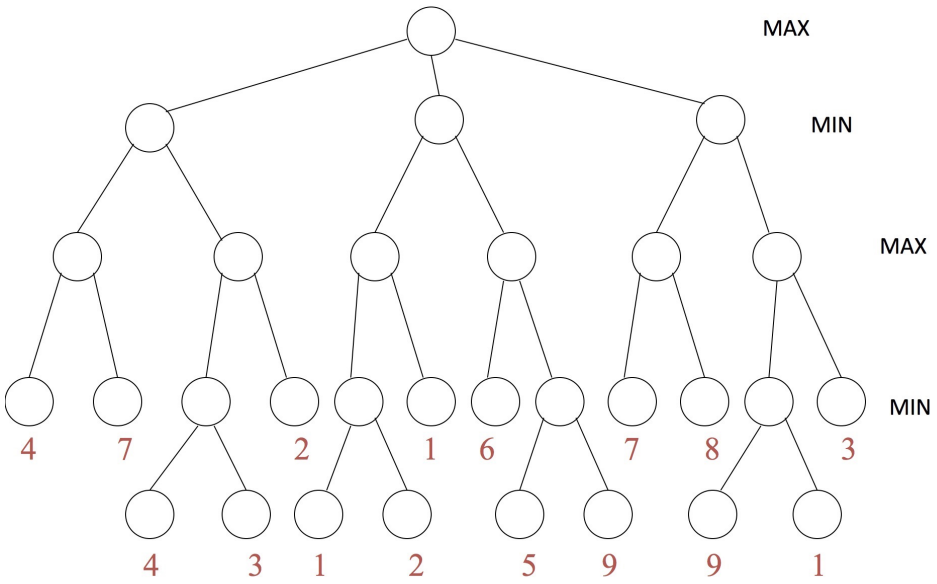

Figure 1: Game tree

So, one question comes out. How to do a choice for player1 and player2.

1. First number all nodes as $node_1$,$node_2$,..., $node_m$

2. Define two functions C(node), V(node).
   C(node) is designed for player1. It means the maximum value which can get from the node.

V(node) is designed for player2. It means the minimum value which can get from the node.

3. C($node_i$) = max(V($node_i$-child1), V($node_i$-child2), ... V($node_i$-childk_1)); V($node_i$) = min(C($node_i$-child1), C($node_i$-child2), ... C($node_i$-childk_2))
4. Using dynamic programming, we can calculate answer bottom-up.

Let's go back to sorting based on comparison.

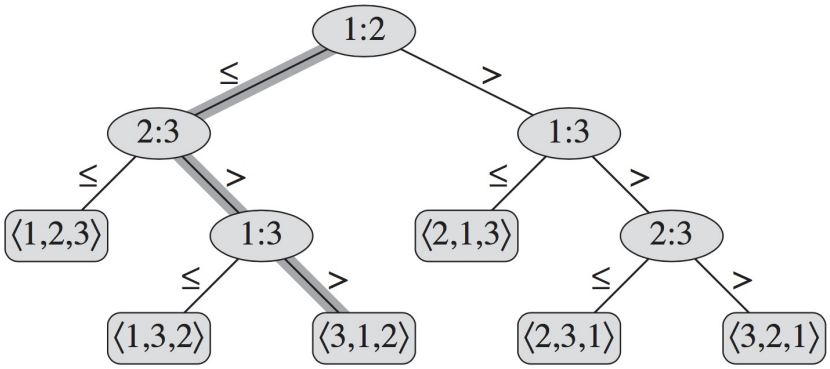

Figure 2: Game tree for sorting based on comparison

Every inner node(all node except leaf node) is a comparison between two number. Every leaf node is a possible sorting result.

Example

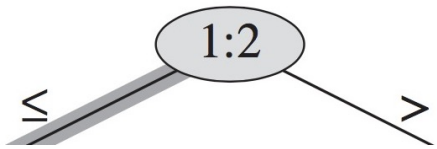

Figure 3: An inner node

At this node, compare a_1 and a_2. If $a_1 \leq a_2$, then choose the left branch, else choose the right branch.

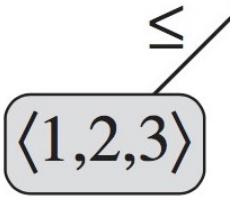

Figure 4: A leaf node

This is a leaf node, at this node the solution set have only one item (1,2,3).

Some conclusion:

- There are n! leaf nodes.
  Since every leaf node is a possible result for sorting problem. Besides, at every inner node, the solution set is divided into two disjoint smaller solution sets. One part is $a_i > a_j$, the other part is $a_i \leq a_j$. So the number of leaf nodes equals to the size of solution set. Since the size of solution set is n!. Thus, the number of leaf nodes are n!.

- The worst case time complexity equals to the height of the tree.
  Since inside inner node, there is a comparison. Then for an input number sequence, the number of comparisons equals to the length of the path it passes.

  Let the longest path as $path_1$, the length of the $path_1$ is $length_1$,and the leaf node along the path is $node_1$. Then we choose data corresponding to the leaf node $node_1$ as the input data. Then we can let the algorithm need $length_1$ comparisons. And the length of longest path equals to the height of the tree. So in the worst case, the time complexity equals the height of the tree.

- The height of the tree = $\Omega(nlogn)$, where n is the size of the input numbers.
  Since a binary tree of height h has no more than 2^h leaves, we have

  $$n! \leq 2^h$$

  Then,

  $$h \geq log(n!) = \Omega(nlogn)$$

  So, the worst case time complexity is $\Omega(nlogn)$

Since, we proved that the worst case time complexity for deterministic comparison sorting is $\Omega(nlogn)$. Here, we want show that the average time complexity for deterministic

comparison sorting is also $\Omega(nlogn)$.
If the tree is completely balanced, then each leaf is at depth $log_2(n!)$ and we are done. So, what we want to do is **to show unbalanced tree has no less average depth**. This is not too hard to see: given some unbalanced tree, we take two sibling leaves at largest depth and move them to be children of the leaf of smallest depth. Since any unbalanced tree can be modified to have a smaller average depth, such a tree cannot be one that minimizes average depth, and therefore the tree of smallest average depth must in fact be balanced.

Note

- It's up to the algorithm to do its split.
  In other words, what our designed smart sorting algorithm does is:
  choosing two number for comparison, split the whole solution set into two smaller solution sets part1, part2.

- It's up to the adversary to do the choose which part after splitting.
  In other words, what the evil adversary does is:
  let the updated solution set be the large one between part1 and part2.

4 Yao's Principle

Yao's principle: the best randomized algorithm on the worst time complexity is equivalent to the worst distribution for the best average algorithmic solution.

Thus, to establish a lower bound on the performance of randomized algorithms, it suffices to find an appropriate distribution of difficult inputs and to prove that no deterministic algorithm can perform well against that distribution.

Thus, to find a lower bound for a randomized sorting algorithm(based on comparison), we only need to find an appropriate distribution. In this distribution, the minimum of every deterministic algorithm's average time complexity is the lower bound. Since, we proved that for any deterministic comparison-based sorting algorithm A, the average-case number of comparisons is at least $log_2(n!) = nlogn$. So, we can know that lower bound for randomized sorting algorithm is also $log_2(n!) = nlogn$. That is, for any randomized sorting algorithm, there is an input for which the randomized algorithm take time at least $nlogn$.

Remark.
Please let me know if there are any mistakes.

References

[1] https://my.vanderbilt.edu/cs260/files/2012/07/GameTreeSearch.pdf

[2] https://www.cs.purdue.edu/homes/egrigore/Fall12/lect20.pdf

[3] https://www.cs.cmu.edu/~avrim/451f11/lectures/lect0913.pdf