Lower Bounds

CS240 Spring 2020

Rui Fan

Upper and lower bounds

- What is the minimum resources (time, space, etc.) needed to solve a problem?
- Consider sorting n numbers.
  - Insertion sort takes $O(n^2)$ time.
  - This puts an upper bound of $O(n^2)$ on the time to sort n numbers.
  - Merge sort takes $O(n \log n)$ time.
  - This puts an upper bound of $O(n \log n)$ on the time to sort n numbers.
- We want to make the upper bound as low as possible, i.e. solve the problem faster.
- Suppose an algorithm A solves problem X in f(n) time when input size is x.
  - Then f(n) is an upper bound on the complexity of X.

Upper and lower bounds

- What about the least amount of time to solve X?
- Suppose we know that any algorithm that solves X takes at least g(n) time, when X has size n.
  - Then g(n) is a lower bound on the complexity of X.
- If the lower bound g(n) is large, it means problem X is hard to solve.
  - Ex NP-Hard problems are hard because they (probably) have super-exponential lower bounds.
- To show a lower bound, we need to give a proof.
  - Usually we show if an algorithm takes too little time, it must sometimes produce the wrong answer.
- The lower bound for a problem depends on the computational model.
  - If a model has very powerful primitive operations, then algorithms can run faster, and the lower bound is smaller.
- If the complexity of an algorithm for problem X matches the lower bound for problem X, the algorithm is optimal, and the lower bound is tight.

A warm-up

- Say we want to find the larger of two numbers x and y.
  - We can do this with 1 comparison, so this is an upper bound.
  - What's the lower bound? Do we need at least 1 comparison? Can we do 0 comparisons?
  - No. Suppose an algorithm doesn't compare x and y.
    - So basically, the algorithm declares either x or y to be bigger, without looking at them.
  - Say the algorithm declares x bigger. Then let's set y > x.
    - Algorithm won't notice this, cause it doesn't compare x and y.
    - So algorithm still declares x is bigger, which is wrong.
    - This type of argument is called indistinguishability, and is frequently used when proving lower bounds.
    - Same argument if algorithm always declares y bigger without comparing.
  - Hence, any algorithm must do at least 1 comparison, so 1 is a lower bound.

Outline

- We'll prove lower bounds for the following problems.
  - Merging two lists.
  - Finding the max.
  - Finding the max and min.
  - Sorting n numbers.

Merging two lists

- How many comparisons needed to merge two lists of size n into sorted order?
  - During the execution, the algorithm can compare some input elements a and b, and get back response “a<b”, “a=b” or “a>b”.
- If the lists are sorted, 2n-1 comparisons is an upper bound.
- Let’s prove this is also a lower bound.
- Let the input lists be $a_1,a_2,...,a_n$ and $b_1,b_2,...,b_n$, and suppose $a_1<b_1<a_2<b_2<...<a_n<b_n$.
  - So the algorithm must output $a_1,b_1,a_2,b_2,...,a_n,b_n$.
- When comparing some a_i and b_j, it gets back the following response:
  - $a_i < b_j$ if $i \leq j$.
  - $a_i > b_j$ if $i > j$.
- We show any algorithm has to perform $\geq 2n - 1$ comparisons to merge the two lists.
  - This gives a 2n-1 time lower bound on merging, since a merging algorithm must correctly merge any two input lists, including the two lists above.

Merging two lists

- Claim Any correct algorithm must compare a_i to b_i, for every i.
  - Suppose not; say the algorithm doesn't compare a_1 to b_1.
  - Now, if the input was actually $b_1<a_1<a_2<b_2<...<a_n<b_n$, then the algorithm still outputs $a_1,b_1,...,a_n,b_n$, which is wrong.
    - Because the algorithm doesn't compare a_1 and b_1, it can't distinguish the new input from the original.
  - Same argument if algorithm doesn't compare a_i to b_i, for any i.
  - So algorithm does n comparisons of this type.
- Claim Any correct algorithm must compare b_i to a_{i+1}, for every $i<n$.
  - If not, then say it doesn't compare b_1 to a_2. Then it can't distinguish original input from input $a_1<a_2<b_1<b_2<...<a_n<b_n$, and will give wrong answer.
  - Thus, n-1 comparisons of this type.
- So, any algorithm must do at least 2n-1 comparisons.
- So 2n-1 is a lower bound on the complexity to merge into sorted order.

Finding the max

- How many comparisons to find the largest number in an unsorted array of n distinct numbers.
- Upper bound: n-1.
- Lower bound: also n-1.
- To prove this, we'll keep track of what information the algorithm learns at it executes.
  - Say algorithm never compared some element to any other element.
    - Then the algorithm doesn't know anything about this element. It could be the max, or not the max.
    - Thus, the algorithm can't correctly output the max without comparing this element to some others.
  - Say there are two elements, and both are larger than every element they've been compared to.
    - Then either one of them could be the max.
    - So algorithm can't output the max without comparing these two elts.
- Let's formalize this intuition.

Finding the max

- At any stage of the alg, give every array element one of 3 colors, white, blue or red.
  - White means this element has never been compared to any other element.
  - Blue means this element is bigger than all the elements it's been compared to.
  - Red means this element was smaller than some element it was compared to.
  - Let w_k, b_k, r_k be number of white, blue and red elements after A has done k comparisons.
    - So initially, $w_0=n$ and $b_0=r_0=0$.
- We'll show that for any k, $w_k+b_k \geq n-k$.
- We'll show that as long as $w_k+b_k > 1$, A can't terminate.
- Hence, when A terminates, we have $w_k+b_k = 1$, and A must have done $k \geq n-1$ comparisons.

Finding the max

- Claim For any k, $w_k+b_k \geq n-k$.
- Proof By induction on k. Claim holds for k=0.
  - For larger k, consider the k'th comparison. It must either be between: 2 white elements (WW case), a white and blue element (WB case), a white and red (WR), 2 reds (RR), 2 blues (BB), red and blue (RB).
  - Do a case by case analysis.
  - WW: Make the first element > second element.
    - This is possible, because both elements are white, so neither have been in any comparisons, so they can be in either order.
    - After comparison, first element becomes blue, second element red.
    - Number of whites decreases by 2, blues increases by 1.
    - By induction, $w_{k-1}+b_{k-1} \geq n-k+1$. Also, $w_k=w_{k-1}-2$, and $b_k=b_{k-1}+1$. So $w_k+b_k \geq n-k$.
  - WB: Make the first element < second element.
    - This is possible, since first element hasn't been in any comparisons.
    - So first element becomes red, second remains blue.
    - So $w_k=w_{k-1}-1$, $b_k=b_{k-1}$, so $w_k+b_k \geq n-k$.

Finding the max

- WR: Make the first element > second element. First element becomes blue, second stays red.
  - So $w_k = w_{k-1} - 1$, $b_k = b_{k-1} + 1$, so $w_k + b_k \geq n-k+1 > n-k$.
- RR: Make first element > second element. Both elements stay red.
  - $w_k + b_k = w_{k-1} + b_{k-1} \geq n-k+1 > n-k$.
- BB: Make first element > second element. First one stays blue, second becomes red.
  - $w_k + b_k = w_{k-1} + b_{k-1} - 1 \geq n-k$.
- RB: Make first element < second element. Both elements stay same color.
  - $w_k + b_k = w_{k-1} + b_{k-1} \geq n-k+1 > n-k$.
- Hence $w_k + b_k \geq n-k$ by induction.

Finding the max

- Claim Suppose after making k comparisons, we have $w_k+b_k>1$. Then A cannot terminate.
- Proof Say A terminates, and outputs a value x as the max.
  - Since $w_k+b_k>1$, either $w_k \geq 1$, or $b_k>1$.
  - If $w_k \geq 1$, then there's a white element y that's never been compared to x (or any other elt).
    - Make y > x. Then the algorithm is wrong.
  - If $b_k>1$, then there are at least 2 blue elements.
    - x must be a blue element.
      - If x is red, it's not max.
      - x is not white, by above.
    - Take another blue element z. x and z were never compared.
      - If they had been, either x or z would have turned red.
    - Make z > x. Now A is wrong.
- Since A can't terminate as long as $w_k+b_k>1$, then $k \geq n-1$ when A terminates.
- So A does $\geq n-1$ comparisons.

Finding the max and min

- How many comparisons does it take to find the max and min elements in an unsorted array A of n distinct numbers.
- Upper bound 1: 2n-2 comparisons.
- Upper bound 2: 3n/2-2 comparisons.
  - Pair up the elements, A[1] and A[2], A[3] and A[4], etc.
  - Compare the elements in each pair (n/2 comps total).
  - Put all the bigger elements in a temp array Big, put all the smaller elements in temp array Small.
    - Big and Small each have size n/2.
  - Find the max element in Big and output it as max of A (n/2-1 comparisons).
  - Find the min element in Small and output it as the min of A (n/2-1 comparisons).
- Upper bound 3: 3n/2-2 comparisons via divide and conquer.
  - Exercise.
- Lower bound: 3n/2-2 comparisons!

Finding the max and min

- Intuition for proof is similar to one for max.
- At any stage of alg, give each array element one of 4 colors, white, blue, red and purple, representing what the algorithm knows about the element.
  - White means this element has never been compared against any other element.
  - Blue means this element is bigger than all the elements it’s been compared against.
  - Red means this element was smaller than every element it was compared against.
  - Purple means this element was bigger than some elt(s) it was compared to, and smaller than some other(s).

Finding the max and min

- To terminate, algorithm must eliminate all white elements, since these could be the min or max.
- Also, algorithm can only leave one blue and one red.
  - Else either of two blues can be max, either of two reds can be min.
- As comparisons happen, algorithm gets more info, and elements change color, e.g. from white to blue, red to purple, etc.
- Too few comparisons means the algorithm doesn’t have time to eliminate all whites, and all but 1 blue and red.
- Proof keeps track of number of whites, blues and reds after some number of comparisons.

Finding the max and min

- Label each comparison by its type.
  - E.g. WW is comparison between two white elts.
  - There are 10 types, WW, WB, WR, WP, BB, BR, BP, RR, RP, PP.
- Denote the number of comparisons of type WW by ww, number of WB comps by wb, etc. 10 numbers total.
- Let w, b, r denote number of whites, blues and reds, resp., at some stage of the algorithm.

Finding the max and min

- Claim 1 When A terminates, w=0 and b=r=1.
- Proof Say A outputs x as max, y as min.
  - Neither x nor y can be white, since we can make a white element be neither max nor min.
  - If there is a white element z when A terminates, we can make z > x, and A is wrong. So w=0.
  - x must be a blue element, as in the finding max proof.
  - If there's another blue element z, then x and z weren't compared, so we can make z > x, and A is wrong. So b=1.
  - y must be a red element.
  - If there's another red element z, then we can make z < y, and A is wrong. So r=1.

Finding the max and min

- The table states what happens when each type of comparison occurs. Similar to the case analysis in finding max proof.
  - Ex If WW occurs, make the first element > second element (denoted $E_1>E_2$), so these elements become blue and red (BR).
  - Ex If WB occurs, we make the first element < second element (denoted $E_1<E_2$), so the elements become red and blue (RB).

Comparison type	Result	Comparison type	Result
WW	$E_1>E_2$, BR	BR	$E_1>E_2$, BR
WB	$E_1<E_2$, RB	BP	$E_1>E_2$, BP
WR	$E_1>E_2$, BR	RR	$E_1<E_2$, RP
WP	$E_1>E_2$, BP	RP	$E_1<E_2$, RP
BB	$E_1>E_2$, BP	PP	$E_1<E_2$, PP

Finding the max and min

- Claim 2 At any stage of the alg, we have
  - $w = n - 2ww - rw - bw - pw$.
  - $b = ww + rw + pw - bb$.
  - $r = ww + bw - rr$.
- Proof These follow just by counting w,b,r using the table on the previous page.
  - For w, there are initially n whites. Each WW comparison removes 2 whites. Each RW, BW or PW comp removes 1 white.
  - For b, each WW, RW or PW comparison creates 1 blue element. Each BB comparison removes 1 blue.
  - For r, each WW, BW comparison creates 1 red element. Each RR removes 1 red.

Finding the max and min

- Theorem Any algorithm performs at least 3n/2-2 comparisons.
- Proof The total number of comparisons is $C = ww + wb + wr + wp + bb + br + bp + rr + rp + pp$.
  - By claims 1 and 2, when A terminates we have $2ww + rw + bw + pw = n, bb = ww + rw + pw - 1, rr = ww + bw - 1$.
  - So $bb + rr \ = \ 2ww + rw + bw + pw - 2 \ = \ n - 2$.
  - $C \geq \ ww + wb + wr + wp + bb + rr$
    $= \ ww + wb + wr + wp + n - 2$
    $= \ n - ww + n - 2$
    $= 2n - 2 - ww$.
  - $ww \leq \ n/2$, because each WW comp decreases number of whites by 2, and there are only n whites.
  - So $C \geq 3n/2 - 2$.

Sorting

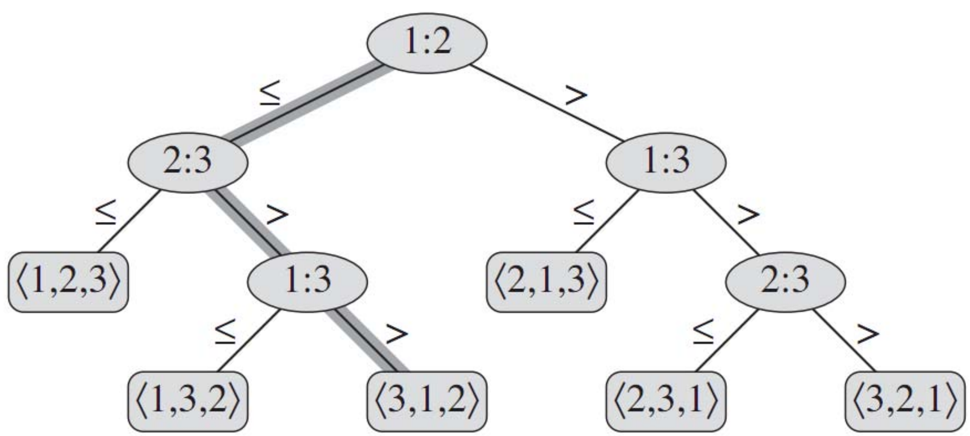
- How many comparisons are needed to sort n numbers?
- Upper bound: O(n log n) using merge sort.
- Lower bound: $\Omega(n \log n)$.
- To prove the lower bound, we first need a model for how a comparison-based sorting algorithm works.
  - This is called the decision tree model.
- The lower bound is not valid in other models.
  - If an algorithm can do things besides comparing two numbers, e.g. look at the digits of a number, it can sort faster than $\Omega(n \log n)$ time.
  - Lower bounds can be very sensitive to the computational model.

Decision trees

- In this model, in each step, algorithm can only compare a pair of numbers x, y.
- Based on result of the comparison, it decides next pair of numbers to compare.
  - So an execution of the algorithm is a sequence of comparisons, each comparison determined by result of previous comparison.
- When the algorithm terminates, it outputs a permutation representing the sorted order of the input.
- The complexity of the algorithm is the most number of comparisons it does before terminating.

Decision trees

- Model behavior of the algorithm by a binary tree.
  - Each internal node is a pair of number x,y to compare.
  - If $x \leq y$, go to left child. If $x > y$, go to right child.
  - Each leaf represents an output, and is labeled with a permutation representing the sorted order of the inputs.
- An execution is simply a path from root to a leaf.
  - At any node, the algorithm has obtained some info from the comparisons it's done.
  - It uses this info to decide the next comparison to do.
  - Eventually, it obtains enough info to generate an output.
- Complexity of algorithm is the length of the longest root-leaf path.



Lower bound for sorting

- Given n numbers as input, they can be in n! different orders.
- Given an input order, algorithm must output that order.
  - So decision tree of algorithm must have a leaf labeled with that order.
  - So the decision tree has $\geq$ n! leafs.
- Say height of decision tree is h.
  - The complexity of the algorithm is h.
  - Since decision tree is binary, it has $\leq 2^h$ leaves.
- So $2^h \geq$ (# leaves of dec tree) $\geq$ n!, and so $h \geq \log_2(n!)$.
  - $\log_2(n!) = \log_2 n + \log_2(n-1) + \cdots + \log_2 1 \geq$ $\log_2 n + \log_2(n-1) + \cdots + \log_2(n/2) \geq$ $\frac{n}{2}(\log_2 n - 1) = \Omega(n \log n)$.
  - Can also use Stirling's approximation.
- So we proved the algorithm does $\Omega(n \log n)$ comparisons.