7 The Lower Bound Theory

7.1 What is a lower bound?

- Algorithmics/upper bounds:

  Prove that a problem can be solved in time $O(f(n))$ by designing and analyzing a specific algorithm for the problem, for some $f(n)$ that we aim to reduce as much as possible. We say that $O(f(n))$ is an upper bound to the problem. An upper bound is always paired with an algorithm. It is the amount of time sufficient to solve the problem.

- Complexity/lower bounds:

  Prove that any algorithm capable of solving the problem correctly on all of its instances must take time $\Omega(g(n))$, for some $g(n)$ that we try to push as large as possible. We say that $g(n)$ is a lower bound to the problem. A lower bound shows the difficult nature of the problem, so it has nothing to do with any specific algorithm. It is the amount of time necessary to solve the problem by any algorithm within the model.

- Optimality of an algorithm:

  If $f(n) = \Theta(g(n))$, then we say that we have found the most efficient algorithm possible, except perhaps for changes in the hidden multiplicative constant.

- A graphical explanation.

- Example: Matrix multiplication.

  Lower bound: n^2

  Upper bounds: $n^3 \to n^{2.81} \to n^{2.61} \to n^{2.376} \to \cdots$

7.2 Trivial lower bounds by input/output

- We can also use the amount of input data that any correct algorithm needs to process or the amount of data that any correct algorithm needs to produce to establish trivial lower bounds.

- For example in the matrix multiplication problem, since the multiplication of two $n \times n$ matrices is another $n \times n$ matrix and there are n^2 entries to compute, n^2 is obviously a lower bound. (Not yet proved to be tight.)

- For another example, consider the problem of generating all permutations of n objects. Since there are $n!$ permutations to generate, $\Omega(n \cdot n!)$ is an obvious lower bound. (Tight.)

- Evaluation of an n-degree polynomial requires that all $n+1$ coefficients be processed. Thus the trivial lower bound is $\Omega(n)$. (Tight.)

- Some bounds established by input/output are not correct lower bounds. For example, searching target in a sorted array of n elements does not require to check all n elements.

- Some lower bounds established by input/output are not trivial to prove. For example, determining connectivity of an undirected graph by its adjacency matrix requires to check the existence of each of the $n(n-1)/2$ potential edges.

7.3 The decision tree method

Reading: CLRS 8.1

- A comparison-based model for proving worst-case lower bound:

  Construct a decision tree for sorting 3 items by comparisons. Each interior node represents a comparison. Each tree edge is an outcome ($<$ or $>$). Each leaf is a possible output (a sorted list). Each path from the root to a leaf represents the steps of sorting a certain type of input. There are $n!$ leaves in the tree, corresponding to $n!$ permutations. The height of the tree gives the maximum number of comparisons in the algorithm for all inputs, thus the worst case time complexity of the algorithm the tree represents.

  How to use a decision tree to establish a lower bound: For each comparison-based sorting algorithm, A_i, there is a decision tree, T_i. Let $height(T_i)$ be the height of T_i, thus the worst-case time of A_i. Obviously, $\min_i\{height(T_i)\}$ is a lower bound to the problem.

- Lemma: A binary tree of height h has at most 2^h leaves.

  Proof We assume that the height of a tree is the number of edges on the longest path between the root and a leaf. We induct on h. When $h = 0$, the tree has just one node. There is 1 leaf, which is no larger than 2^0. Assume for $i \leq h-1$, a tree of height i has no more than 2^i leaves. Now consider a tree with height h. Assume the heights of the left and right subtrees are h_L and h_R, respectively. so $h = \max\{h_L, h_R\} + 1$. By the inductive hypothesis, the numbers of leaves in the left and right subtrees are at most 2^{h_L} and 2^{h_R}, respectively. So the number of leaves in the tree with height h is at most $2^{h_L} + 2^{h_R} \leq 2^h$.

- Theorem: Any comparison-based sorting algorithm requires $\Omega(n \log n)$ time in the worst case.

  Proof Let T be a decision tree for any algorithm that sorts n elements by comparisons. T has $n!$ leaves. So the height of the tree, h, is at least $\log n!$. (Assume not. Then $h < logn!$. By the lemma, the number of leaves in the tree is at most $2^h < 2^{logn!} = n!$, which is impossible.) So for any sorting algorithm, the worst-case time complexity is the height of the decision tree, thus at least $\log n! = \Omega(n \log n)$.

- Both Merge Sort and Heap Sort achieve the optimality of sorting since their worst-case times match the established lower bound, $n \log n$. Radix Sort has a worst-case time of $\Theta(d(n+k))$, where d is the number of digits and k is the base. It is $\Theta(n)$ when $d = O(1)$ and $k = O(n)$. This is not a contradiction to the $n \log n$ lower bound, since Radix Sort is not comparison based.

- A comparison-based model for proving average-case lower bound:

  Again consider the example of sorting 3 elements. As in any average-case analysis, assume that all elements in the list are distinct and that all permutations are equally likely. For any sorting algorithm, there is a corresponding decision tree with $n!$ leaves. Let d_i be the depth, or the number of edges on the path between the root and a leaf i. Let p_i be the probability that permutation (leaf) i occurs. Then the average-case time of the algorithm that the tree represents is $\sum_i p_i d_i = \frac{1}{n!} \sum_i d_i$.

leaf	abc	acb	cab	bac	bca	cba
probability	p_1	p_2	p_3	p_4	p_5	p_6
depth	d_1	d_2	d_3	d_4	d_5	d_6

- Lemma: Let T_m be any binary tree with m leaves such that each nonleaf has exactly 2 children. Define $D(T_m) = \sum_i d_i$ (the sum of the depths of all leaves in T). Define $D_m = \min_{\forall T_m}\{D(T_m)\}$. Prove that $D_m \geq m \log m$.

  Proof We induct on m. When $m = 1$, the tree has only one node. So $D_1 = 0 \geq 1 \log 1$. Assume that for any $i \leq m-1$, $D_i \geq i \log i$. Now consider any tree with m leaves. Assume there are i leaves in the left subtree and $m-i$ leaves in the right subtree. ($i \neq 0$ and $i \neq m$ since each nonleaf has two children.) So

$$\begin{aligned} D_m &= \min_{1 \leq i \leq m-1}\{D_i + D_{m-i} + m\} \\ &\geq m + \min_{1 \leq i \leq m-1}\{i \log i + (m-i)\log(m-i)\} \end{aligned}$$

  Define function $f(x) = x \log x + (m-x)\log(m-x)$. Then the derivative $f'(x) = x\frac{1}{\ln 2}\frac{1}{x} + \log x + (m-x)\frac{1}{\ln 2}\frac{-1}{m-x} - \log(m-x) = \log x - \log(m-x)$. Let $f'(x) = 0$. Then $\log x = \log(m-x)$, which implies $x = \frac{m}{2}$. So $f(x)$ achieves the minimum when $x = \frac{m}{2}$. To continue,

$$\begin{aligned} D_m &\geq m + \min_{1 \leq i \leq m-1}\{i \log i + (m-i)\log(m-i)\} \\ &\geq m + \frac{m}{2}\log\frac{m}{2} + \frac{m}{2}\log\frac{m}{2} \\ &= m + m\log\frac{m}{2} \\ &= m \log m \end{aligned}$$

- Theorem: Any comparison-based sorting algorithm requires $\Omega(n \log n)$ time in the average case.

  Proof Let T be a decision tree for any algorithm that sorts n elements by comparisons. T has $n!$ leaves. So the average depth of the leaves is $\frac{1}{n!}\sum_i d_i \geq \frac{1}{n!} D_{n!} \geq \frac{1}{n!} n! \log n! = \log n!$. So for any sorting algorithm, the average-case time complexity is the average depth of the decision tree, thus at least $\log n! = \Omega(n \log n)$.

- Bucket Sort: not comparison-based, with $O(n)$ average time.

  Let $A = (a_1, \ldots, a_n)$, where $a_i \in [0,1)$ is generated randomly such that $a_1, \ldots, a_n$ are distributed uniformly in $[0,1)$. For example, $A = (0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.61)$. Let B be an array of intervals in $[0,1)$, where the length of each interval is $\frac{1}{n}$. In particular, $B[i]$ represents interval $[\frac{i}{n}, \frac{i+1}{n})$ for $0 \le i \le n-1$. Each $B[i]$ will be a pointer to a list of numbers falling into the corresponding interval.

  for $i \leftarrow 1$ to n

  insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$

  for $i \leftarrow 0$ to $n-1$

  sort list $B[i]$ using any sorting algorithm

  concatenate $B[0], \ldots, B[n-1]$ into a list for output

  Why should $A[i]$ go to list $B[\lfloor nA[i] \rfloor]$? Assume $A[i]$ goes to $B[x]$. Then $\frac{x}{n} \le A[i] < \frac{x+1}{n}$. So $x \le nA[i] < x+1$. So $nA[i] - 1 < x \le nA[i]$. Since x is an integer, $x = \lfloor nA[i] \rfloor$.

  Time complexity: $\Theta(n^2)$ or $\Theta(n \log n)$ in the worst case; $\Theta(n)$ in the average case since each list in B has 1 element on average.

7.4 The adversary method

- The adversary strategy: An adversary provides worst-case input that forces all algorithms solving the problem to work as hard as possible.

  Example: Alice (1) has a date (month/day) in mind (this is what she tells Bob); (2) answers Bob's questions honestly; and (3) wants to force Bob to ask as many questions as possible. Bob (1) asks yes/no questions and (2) wants to guess the date as quickly as possible.

  Bob: Is it in the winter? (December, January and February)

  Alice: No (since there are more dates in the other three seasons).

  Bob: Is the first letter of the month's name in the first half of the alphabet?

  Alice: Yes (since among the remaining 9 months, 6 are in the first half of the alphabet, March, April, May, June, July, August).

  .....

  Alice didn't pick a date at all. Is this cheating? Not to Bob. In fact, Alice will not do so until the need for consistency in her answers pins her down. This is how a good adversary should be.

  For a comparison-based model, the adversary works as follows. When the algorithm asks to compare two variables, a and b, it does not know the values of the variable and is given only the outcome of the comparison–a is larger or b is larger. On the other hand, the adversary who is responsible for deciding the outcome gives out the outcome of the comparison in any way he likes as long as hie choice is consistent with all of his previous answers. So in the eyes of the algorithm, the adversary is honest.

- Finding the max and min at the same time:

  Let $C_{1,n}(n)$ be the number of comparisons necessary and sufficient to find the maximum and minimum of n integers. Prove that $C_{1,n}(n) = \lceil \frac{3n}{2} \rceil - 2$.

  Upper bound: $C_{1,n}(n) \le \lceil \frac{3n}{2} \rceil - 2$. Wish to design an algorithm which takes at most $\lceil \frac{3n}{2} \rceil - 2$ comparisons. The algorithm first uses $n-1$ comparisons to find the maximum among the n numbers and then $\lceil \frac{n}{2} \rceil - 1$ more comparisons to find the minimum among the $\lceil \frac{n}{2} \rceil$ non-winners in the first round.

  Show an example of finding the max and min in five integers by using two tournament trees.

  Lower bound: $C_{1,n}(n) \ge \lceil \frac{3n}{2} \rceil - 2$. Assume all keys are distinct. At any time in any algorithm, there are four kinds of elements.

  Novice: not compared yet;

  Winner: always wins so far;

  Loser: always loses so far;

  Moderate: wins some and loses some.

Define a state (i,j,k,l), where i, j, k, and l are the numbers of novices, winners, losers, and moderates, respectively. The initial state is $(n,0,0,0)$ and the final state is $(0,1,1,n-2)$. Question: What is the minimum number of comparisons from the initial state to the final state? A state transition table is given below:

	(i,j,k,l)
NN	$(i-2,j+1,k+1,l)$
NW	$(i-1,j,k,l+1)$ or $(i-1,j,k+1,l)$
NL	$(i-1,j+1,k,l)$ or $(i-1,j,k,l+1)$
NM	$---$
WW	$(i,j-1,k,l+1)$
WL	(i,j,k,l) or $(i,j-1,k-1,l+2)$
WM	$---$
LL	$(i,j,k-1,l+1)$
LM	$---$
MM	$---$

Assume that an adversary provides the input and uses the following strategy:

algorithm	NN	NW	NL	WW	WL	LL
adversary	arbitrary	W wins	N wins	arbitrary	W wins	arbitrary

Why is the adversary able to make the choices it does in the table above? For an NW comparison, it can make N to be arbitrarily small so that W is the winner. For the similar reason in an NL comparison, it can make N to be arbitrarily large so that L is the loser. In a WL comparison, the adversary can increase W (or decrease L) so that W beats L without changing the outcomes of the previous comparisons. Since the algorithm cannot see the numbers but only the comparison results, this action by the adversary is totally legal.

In the initial state, $i+j+k=n$, while in the final state $i+j+k=2$. Since only a WW or LL comparison decreases $i+j+k$ (by 1), we need to make at least $n-2$ WW or LL comparisons. Next consider the change of i in the initial and final states. It decreases from $i=n$ to $i=0$. Since WW and LL comparisons never change the value of i, some NN, NW, NL comparisons are needed. Since such a comparison decreases i by at most 2, a minimum of $\lceil \frac{n}{2} \rceil$ comparisons are needed. So the total number of comparisons necessary is $n-2+\lceil \frac{n}{2} \rceil = \lceil \frac{3n}{2} \rceil - 2$.

7.5 Reduction

- Proving lower bounds by reduction:

  Let P be the problem whose lower bound, $f(n)$, we wish to prove. Let Q be a problem whose lower bound, $g(n)$, is known. For any algorithm A for P, we try to define an algorithm B for Q which calls A. See a flowchart of such an algorithm.

  We assume that there is an algorithm, say A, for P that takes time $o(f(n))$. Then from the chart above, there is an algorithm, B, for Q, with time $t_1(n)+o(f(n))+t_2(n)$. If $t_1(n)+o(f(n))+t_2(n)=o(g(n))$, then we just found an algorithm which beats the proven lower bound, $g(n)$, for Q. This is of course impossible. So there is no algorithm for P with time faster than $f(n)$. Therefore, $f(n)$ is a lower bound for P.

- The convex hull problem: Given a set P of n points on the 2-D plane. The *convex hull* of the point set, denoted by $CH(P)$, is a convex polygon, represented by its vertices in counterclockwise order, say, $p_1,p_2,\ldots,p_k$, where $p_i \in P$ for $i=1,2,\ldots,k$, such that the polygon contains all the other points in P.

  Next we will prove that any algorithm that finds $CH(P)$ for a given P of n points takes at least $n\log n$ time in the worst case.

  Assume there is an algorithm A for the convex hull problem which takes $o(n\log n)$ worst-case time. Now we will design a sorting algorithm B that calls A. For any input list of n numbers $x_1,x_2,\ldots,x_n$, algorithm B first converts the list in $O(n)$ to a point set $(x_1,x_1^2),(x_2,x_2^2),\ldots,(x_n,x_n^2)$ and feed the point set as input to algorithm A. Algorithm A then computes the convex hull of the point set in $o(n\log n)$ time. Because of the special locations of the points, the convex hull will contain all points as the vertices of the polygon, and the points are output in counterclockwise order. Algorithm B then spends $O(n)$ to search the point in the output list with the smallest x-coordinate, and from there to produce a list of n x-coordinates, which is clearly the sorted version of the original input list to B. So algorithm B sorts a list of n numbers in time $O(n)+o(n\log n)+O(n)=o(n\log n)$. This is a contradiction to the $\Omega(n\log n)$ lower bound for sorting.