Lecture 2. Low complexity bounds and Sorting algorithms.

Dmitri Shmelkin

Math Modeling Competence Center at Moscow Research Center of Huawei
Shmelkin.Dmitri@huawei.com

February 6, 2023

Overview

Low bounds need the algorithm class to be fixed

• In the first lecture we considered the Maximum problem and gave **Brute Force** quadratic and **Natural** linear algorithm to solve it.
• May a faster, e.g., a logarithm compexity algorithm exist?
• This is the question of **low bound** for the problem itself.
• To prove a bound, we have to make the class of algorthms explicit.

Definition 1

Algorithm is called **deterministic** if the sequence of steps is fully determined by the input and moreover, if for two different inputs first k steps are the same and the auxillary data stored after k steps are equal, then $k+1$-th step is the same.

In every specific problem, to prove a low bound for deterministic algorithms we have to define the possible steps of an algorithm in a more or less specific terms.

Reducing Maximum to Weighing

- First of all, let's assume that the input is read to the array D of size n (why?).
- Let's narrow down the class of algorithms in the Maximum problem to those, which may only ask to compare $D[i] \leq D[j]$ and get the bit of answer.
- This is the classical statement of finding the heaviest coin by weighing.

Theorem 2

There is no any deterministic algorithm of weighing n coins that always finds the heaviest coins in less than $n-1$ steps.

Corollary 3

Maximum is a problem of $\Theta(n)$ complexity class.

Proof of the Theorem 2

• Assume a correct algorithm exists with $< n-1$ comparison for every input
• Every input determines the sequence of comparisons made by the algorithm
• Assign to the input a graph with nodes related to coins and an edge $i-j$ if coins $D[i] \geq D[j]$ be compared by the algorithm.

Lemma 4

A connected graph on n nodes has at least $n-1$ edges.

• Consider an input and let i_0 be the index of the maximum found
• Take a connected component, C, that does not contain i_0 and add some positive number, M, to $D[j]$ for each $j \in C$. Conisider this new input.
• By Definition 1, the output for this new input is the same, $D[i_0]$.
• For M large enough get a contradiction. $\square$

Element Query low bound

- Consider the element query in a sorted array D of integers (or another set with linear order) that returns the index, $i(q)$, of the input number q, in the set.
- Consider a stronger form used in the range query, with an additional Boolean variable *ShiftDown* that applies when the number, q, is not in the set. In that case the query returns the nearest element smaller/larger than q, if $ShiftDown = 1/0$.
- Consider the algorithms, which only may check inequalities $i(q) \leq j$ or $i(q) \geq j$; this ability is due to the array is sorted so comparison of q with elements od D is done by the data structure, but no direct access to the elements of D is allowed.
- As an important presolve, the algorithm may and will ensure $q \in [D[1], D[n]]$.
- Binary Element Query comply with the above rules

Theorem 5

For a set of n indexes there is no a deterministic algorithm that might do query for any q and both $0/1$ values of ShiftDown in less than $\lceil \log_2(n) \rceil$ inequality checks.

Corollary 6

Element query with inequality checks is a problem of $\Theta(\log_2(n))$ complexity class.

Proof of the Theorem 5

- Assume that a correct algorithm with $k < \lceil \log_2(n) \rceil$ checks exists
- Let $S(q)$ be the inequality checks results, a sequence of at most k of $0/1$
- Notice that if $S(q_1) = S(q_2)$ for $q_1 < q_2$, then $i(q_1) = i(q_2)$ and q_1, q_2 belong to the interval $[D[i(q_1)], D[i(q_1) + 1])$ closed from the left side, if $ShiftDown = 1$ and to $(D[i(q_1) - 1], D[i(q_1)]]$, if $ShiftDown = 0$
- In particular, the map $q \to S(q)$ is injective on the the subset considered
- However, there are at most $2^k < n$ different sequences.
- The contradiction proves the statement $\square$
- **Remark** The proof is based on a so-called *cardinality* argument.

Sorting (or Ordering) Problem

• We gave more than one example of usability of sorting as a method to build efficient data structures to solve query problems. But how to solve sorting itself?
• Besides so many different algorithms availble, also the problem statement may have important additional requirements:
• **On-line** (or incremental) problem requires to be able to work with the set being delivered element-by-element or batch-by-batch.
• **Off-line** algorithms may start when the whole set is stored in an array. Then, **in-place** problem requires to use only the memory occupied initially, plus some $O(1)$ memory for computations.
• Usually the set to be sorted consists of numbers. However, a-priori knowledge about the possible numbers may become crucial, as we will see today.
• The algorithms of course apply to any set with linear order, C++ programmers may say that sorting applies to a **template** enabled with the comparison method.

Insertion Sorting

• The idea of insertion sorting is quite natural: to sort first 2, then first 3, etc. elements of the array. Every step j is incremental and consists in insertion of $D[j]$ into the previously ordered $D[1], \cdots, D[j-1]$.

• Those who played cards, may remember doing so when getting her card set

Insertion Sorting 1. For $j = 2, \ldots, n$ do sort the segment $[1, j]$:

2. Set $v := D[j]$

2. For $i = j-1, j-2, \ldots, 1$ do:

3. If $D[i] > v$, $D[i+1] := D[i]$

4. else $D[i+1] := v$; break the i-loop

Theorem 7

Insertion sorting is of $\Theta(n^2)$ *complexity*

Proof.

• It is obvious that in the worst case the number of steps **3** is $j-1$ so totally $1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2}$.

• For the input array $D[j] = n + 1 - j, j = 1, \cdots, n$, all these step be done. □

Online Sorting Low Bound

- The Insertion Sorting is an on-line algorithm: it takes a new element of the array and incrementally sorts the larger array at every read-on step.
- Although the Insertion Sorting looks naïve it is an optimal on-line Sorting:

Theorem 8

The on-line sorting problem is of $\Theta(n^2)$ complexity

Proof.

- Since Insertion sorting is of $\Theta(n^2)$ complexity, we just need to prove that a smaller compexity algorithm may not solve on-line sorting
- Every on-line algorithm has n incremental step and it is enough to prove that that the incremental problem of adding an element to a sorted array with m elements belongs to $\Omega(m)$ class
- The latter statement holds true for the worst case of $D[j] = n + 1 - j$ because any algorithm at least needs to copy m elements to new places. □

Divide and Conquer. Recursion

• We need to return to the principle introduced and illlustrated by Binary search
• Unlike Binary search, in most cases application needs recursive call of the same function for the subproblems.
• Assume that a recursive function divides the input of size n into $p > 0$ problems of size $\lceil n/q \rceil, q > 1$ and uses $O(n^d), d \geq 0$ complexity to both split into subproblems and then, combine their solutions to the one for the whole problem. Then the recursive eqiality holds for the complexity $T(n)$

$$T(n) = pT(\lceil n/q \rceil) + f(n), f \in O(n^d) \tag{1}$$

Theorem 9

If $T(n)$ fulfills equation (1), then holds

$$T(n) = \begin{cases} O(n^d), & \text{if } d > \log_q(p) \\ O(n^d \log_2(n)), & \text{if } d = \log_q(p) \\ O(n^{\log_q(p)}), & \text{if } d < \log_q(p) \end{cases}$$

Proof of Theorem 9

- We first give the proof for the case when $n = q^r, r \in \mathbb{N}, f(n) = Cn^d$
- inserting (1) to itself, we get:

$$T(n) = Cn^d + pT(n/q) = Cn^d + Cpn^d/q^d + p^2 T(n/q^2) =$$

$$= Cn^d + Cpn^d/q^d + Cp^2 n^d/q^{2d} + p^3 T(n/q^3) = Cn^d \sum_{j=0}^{r} (p/q^d)^j \qquad (2)$$

- 3 cases for the geometric progression with $r = \log_q(n)$ terms in (2):
- if $p > q^d \Leftrightarrow d < \log_q(p)$, then the last term $Cp^{\log_q(n)} = Cn^{\log_q(p)}$ is the largest
- if $p = q^d \Leftrightarrow d = \log_q(p)$, then all $\log_q(n)$ terms equal to Cn^d yield $Cn^d \log_q(n)$
- if $p < q^d \Leftrightarrow d > \log_q(p)$, then the first term Cn^d is the largest.
- The general case is reduced to the above one: $T(n)$ for $f < Cn^d$ is smaller than or equal to $T(q^{\lceil log_q(n) \rceil})$ for $f = Cn^d$. □

Merging Sorted Arrays

• Merge Sort algorithm could also be called Divide-and-Sort. It is a recursive algorithm that splits array into two halves, calls itself for the halves and then merges two sorted arrays into one.

• To merge two sorted arrrays is the central idea. It is convenient to think the input as a single segment in an array composed of two sorted sub-segments.

• So as input, Merge takes a "left" array, L and 3 indices, $iBegin \leq iMiddle \leq iEnd$ within limits of L. Output goes to "right" array, R to the same indices:

Merge: **1.** $i = iBegin, j = iMiddle, k = iBegin$

//throughout, the least of $L[i]$ and $L[j]$ is written to $R[k]$

2. while$(k \leq iEnd)$

3. if $(i < iMiddle$ and $(j \geq iEnd$ or $L[i] \leq L[j]))$: $R[k] := L[i]; i = i + 1;$

4. else: $R[k] := L[j]; j = j + 1;$

5. $k = k + 1$

Lemma 10

Complexity of Merge is $\Theta(iEnd - iBegin)$.

Merge Sort

• For simplicity, consider a version that makes copies while computations:

MergeSort: Input is an array A and limits, $iBegin, iEnd$ to order the elements of the segment $A[iBegin : iEnd]$ without changing other elements.

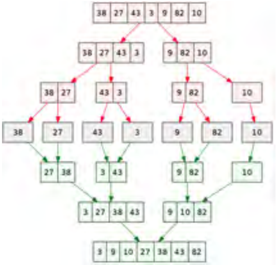
1. if $(iEnd - iBegin \leq 1)$: return // 1-element array is sorted
2. $iMiddle = \lfloor (iBegin + iEnd)/2 \rfloor$
3. Make a copy, $B := A[iBegin : iEnd]$
4. $MergeSort(B, iBegin, iMiddle)$
5. $MergeSort(B, iMiddle + 1, iEnd)$
6. $Merge(B, iBegin, iMiddle, iEnd, A)$// merge sorted subsegments in B to A

Remark: Merge Sort is NOT an in-place algorithm but there is an elegant version that only uses the initial array memory and one copy of it.

Lemma 11

The total complexity of the lines **1,2,3,6** *is* $O(iEnd - iBegin)$.

Merge Sort Complexity



Theorem 12

MergeSort is of $O(n\log(n))$ complexity.

Proof: MergeSort is a recursion that splits a problem of size n into $p = 2$ problems of size $\lceil n/q \rceil, q = 2$. According to the Lemma 11, it uses $O(n^d), d = 1$ complexity to split the problem an combine their solutions into the one for the whole problem. So the equality $d = \log_q(p)$ holds, hence, by the Theorem 9, the complexity of Merge sort is $O(n\log(n))$. □

Decision Tree of Algorithms (very informal)

- Consider a problem with input vector $\overrightarrow{in} \in \mathbb{R}^b$ and output, $\overrightarrow{out} \in \mathbb{R}^c$.
- In general, every algorithm is a finite sequence of operators $L_1, L_2, \cdots, L_M$, where each operator is either an arithmetic function $L_i = f_i(\overrightarrow{in})$ or an if-operator with operands computed by previous operators and stored in memory.
- We may assume that every if-operator is binary, so has two cases
- So we may think of any algorithm as an oriented graph with the root and the nodes (except the root) having incoming degree 1 and either outgoing degree 1 (for arithmetic) or 2 (for if-operators).
- Every oriented path of arithmetic nodes either leads to a leaf or to a if-operator
-in the former case, we combine all operators to a single *leaf function*
-in the latter case, we collapse the sequence with the subsequent if-operator to a comparison of aggregate functions values.
- Such a graph may have oriented cycles and we exclude these cases by requiring that the oriented graph has no oriented cycles, hence, is a finite binary tree.

Definition 13

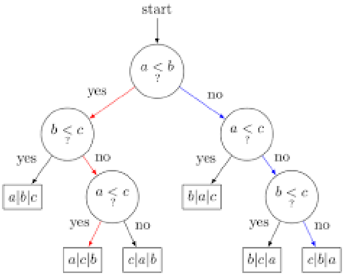
The binary tree above is called *Decision tree* of the algorithm.

Decision Tree and complexity

Theorem 14

The algorithm complexity is equal to the depth of its Decision tree and is not less than $\lceil \log_2(n_{leaf}) \rceil$, where n_{leaf} is the number of leaves.

Watch the example of decision tree: for some algorithm that sorts 3 numbers



Main Theorem

Definition 15

Decision tree is called *linear* if all if-operators look like $l(\overrightarrow{in}) \vee 0$, where $\vee$ stands for either of $<, \leq, >, \geq$ operators and $l : \mathbb{R}^b \to \mathbb{R}$ is a linear function. Also leaf functions must be continuous.

Theorem 16

Let W be the image of $\mathbb{R}^b$ in $\mathbb{R}^c$, that is, all possible outputs of the algorithm. Then the height of the tree is greater than or equal to $\lceil \log_2(n_W) \rceil$, where n_W is the number of connected components of W.

Proof: For every leaf y of the tree, let M_y be the pre-image, that is, all vectors in_v taken to y by the tree. Because of linearity assumption, M_y is convex, and continuos leaf function takes it to just one connected component of W. So we have: $n_{leaf} \geq n_W$ and we are done by Theorem 14 $\square$

Remark: Theorem 16 is similar in statement and proof with the one of [Dobkin-Lipton, 1979], where the problem of recognition is considered.

Low bound for sorting problem

Theorem 17

The low bound of Sorting problem in the class of algorithms with linear decision tree is $\Omega(n\log(n))$.

Proof: Note that every solution of the sorting problem for an array A may provide, as a by product, the permutation of the indices of input elements that makes the numbers ordered. More precisely, the additional output is $\sigma : \{1, \cdots, n\} \to \{1, \cdots, n\}$ such that $A_{\sigma(1)} \leq A_{\sigma(2)} \leq \cdots \leq A_{\sigma(n)}$. Let us think of this permutation as the output $out_v \in \mathbb{R}^n$. There are precisely $n!$ orders and each of them is the output for a suitable input. Theorems 14,16 imply that every sorting algorithm with linear decision tree belongs to $\Omega(\log(n!))$. We may apply an obvious inequality: $n! > n^{\lfloor n/2 \rfloor}$, hence, $\log(n!) > \lfloor n/2 \rfloor \log(n) \in \Omega(n\log(n)$ to get the result. Or we may apply Stierling formula:

$$n! \sim \sqrt{\pi(2n + \frac{1}{3})}n^n e^{-n} \tag{3}$$

to get a more precise (in some sense) lower bound.

There exist $O(n)$ complexity sorting algorithms

- Linear decision tree algorithms are interesting but there are algorithms efficiently solving problems with complexity below the low bound from Theorem 17
- In particular, in contrast to Theorem 17, there are sorting algorithms with linear complexity in n, the size of array, but in each case, subject to additional assumptions about the numbers to sort.
- For example, assume that A contains n natural numbers in the range $[0, K]$.

Counting Sort: Input: array A, bound K. Output: sorted array B.

1. Allocate an additional array C of length K and initiate with 0.
2. for $i := 1 : n$
3. $C[A[i]] = C[A[i]] + 1$ //$C[k]$ is the number of array elements equal to k
4. for $j := 2 : K$
5. $C[j] = C[j] + C[j-1]$ //now $C[k]$ is the number of array elements $\leq k$
6. for $i := n : 1$
7. $B[C[A[i]]] = A[i]$; $C[A[i]] = C[A[i]] - 1$;

Theorem 18

Counting sort is of complexity $O(n + K)$.

References

D.Dobkin and R.Lipton, *On the complexity of computations under varyuing set of primitives*, *Journal of Computer and System Sciences* **18** (1979), 86 -91.