Lecture 1. Algorithms and their complexity.

Dmitri Shmelkin

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

January 20, 2023

Math
Modeling
Lab

Overview

Algorithms as solutions for problems

• The word **Algorithm** is quite popular and sometimes people do not really pay attention to the sense of this concept.

• It is very likely that if an interviewer asks a person on a, say, Euclidean Algorithm, the answer will start with "I have two non-negative integers, a, b and I assume that $a > b$ and divide a by b and get remainder. Then...".

• The answer does not explain, for which reason the action is taken, what are the preconditions, assumptions, the result.

• Algorithm makes only sense if solves (at least, *approximately*) some well-posed problem such as to find an Object that fulfills some Constraints and fits the Objective feature (such as extremal value of objective function).

• Euclidean algorithm finds the greatest common divisor of a, b. Here Object means Natural Number, Constraints are to divide both a, b, and Objective function is the value.

• The *input* and *output* of the algorithm are to be clarified

• The very first questions are: whether a solution exists and is unique, and what the expected Algorithm should do in case of non-uniqueness?

Algorithm problem statement may need a theory

Classical situation that occurs typically in the Algorithm domain is well illustrated by the Systems of linear equations,

$$Ax = b,\ A \textit{ is } m \times n \textit{ matrix},\ x \textit{ is } n \times 1 \textit{ column},\ b \textit{ is } m \times 1 \textit{ column}. \tag{1}$$

1. Algorithms: Gauss method, Jacobi method, or LU-decomposition.
2. What does a *solution* means? Does it always exist or is unique?
3. *Determined, Indetermined*, or *Overdetermined system*
4. *Homogeneous. Inhomogeneous system, Fundamental solutions*

$$x = x_0 + \sum_{j=1}^{r} \alpha_j x_j, \alpha_j \in \mathbb{R}. \tag{2}$$

The input assumptions may change much

1. Linear algebra solves (1) for $x \in \mathbb{R}^n$. How about $x \in \mathbb{Z}^n$?
2. Linear algebra algorithms do not work, because these use division.
3. Assume A and b are integer: system of linear *Diofantine equations*.
4. Similar to $\mathbb{R}^n$ case, inhomogeneous is reduced to homogeneous.
5. Solutions $S = \{x \in \mathbb{Z}^n | Ax = 0\}$ are a subgroup in $\mathbb{Z}^n$.
6. Fundamental Abel group Theorem: S is a free Abelian group generated by linear independent (over $\mathbb{Q}^n$) fundamental solutions.
7. Algorithm: *Smith normal form*: integer matrices, U, $m \times m$ and V, $n \times n$ such that $\det U = 1, \det V = 1$, and $C = UAV$ is a diagonal.
8. Then, (1), $Ax = b$ is equivalent to

$$Cy = Ub; y = V^{-1}x. \tag{3}$$

9. Possible entries of y obtained from per-row scalar Diophantine equations from $Cy = Ub$. Then recover x as $x = Vy$.
10. Remains to find the Smith Normal Form see [Smith, 1861].

Complexity of algorithms

Example Maximum of a list of numbers. The input is a sequence of integer numbers $x_1, \cdots, x_n$. We need to find a sequence member with the maximum value and with the minimal index, to get the answer fully determined.
Brute Force Algorithm: Write all numbers in a 1-dimensional array D, then check every pair $D[i], D[j]$ and write 1 to the entry $C[i,j]$ of a 2-dimensional array, if $D[i] \geq D[j]$ and 0, otherwise. Then find the upper row consisting of all 1 (e.g. checking the sum of the row entries to be equal to n).

Obviously (?) Brute Force solves the problem correctly, how about efficiency?
In general, algorithm efficiency is measured by the working time estimated as the **number of operations** (logical, arithmetical) and the **size of memory** used.
Brute Force operations: n times writing to the array D; n^2 initializations of the array C; $n(n-1)/2$ comparison and for each of them, 2 writings to the array C; $n(n-1)$ in summation over the rows; and finally, at most n comparisons of the sums with n. Totally, the number of operations is $3n^2 + \alpha n + \beta$.
Brute Force Memory: n integers for D, n^2 integers for C, so $n^2 + n$ totally.

Complexity of algorithms. Θ Formalism.

We showed that the number of operations of Brute Force algorithm for Maximum problem is equal to $3n^2 + \alpha n + \beta$. In most cases only the asymptotic of the complexity growth with n tending to infinity matters and even the leading coefficients do not matter. So we write that Brute Force has $\Theta(n^2)$ complexity.

Definition

Let $n \in \mathbb{N}$ denote the size of problem's input, let $f : \mathbb{N} \to \mathbb{R}$ be some function. We say that some algorithm has complexity $\Theta(f)$ if there are positive numbers $c_1 \leq c_2 \in \mathbb{R}$ such that for any $n \in \mathbb{N}$ and any input of size n the number of algorithm's operations is between $c_1 f(n)$ and $c_2 f(n)$.

In particular, if algorithm's complexity is a polynomial of degree k, then the algorithm is of $\Theta(n^k)$ complexity.

Upper and Lower bounds. O and Ω Formalism.

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}_{>0}$ be some functions. We say $f \in O(g)$ and $g \in \Omega(f)$ if there is $C \in \mathbb{R}_{>0}$ such that $\frac{f(n)}{g(n)} \leq C$ for any $n \in \mathbb{N}$.

In particular, $n \log_2(n) \in O(n^k)$ for $k > 1$. This motivates:

Definition

An algorithm is called polynomial if in $O(n^k)$ for some $k \in \mathbb{N}$.

Polynomal algorithms constitute the main focus of this course: linear, quadratic, cubic, and those of complexity $\Theta(\log_2(n), \Theta(n \log_2(n))$ etc.
Problems can typically be solved by algorithms of different complexities, so we may and will talk about Upper and Lower bounds of problem complexity:

Definition

Upper bound: a problem is of class $O(f)$, if there is an algorithm in $O(f)$ solving the problem. *Lower bound*: the problem is of class $\Omega(f)$ if it can be proven than none algorithm from $O(f) \setminus \Theta(f)$ solves the problem.

Complexity of algorithms and problems. Improvement.

Brute Force implies Maximum problem has $O(n^2)$ upper bound. Does it belong to $\Omega(n^2)$? Consider another algorithm for Maximum problem.
Natural Algorithm Write all numbers in a 1-dimensional array D, set integer variable $MaxIndex := 1$. Run integer index j from 2 to n, if $D[j] > D[MaxIndex]$ then assign $MaxIndex := j$.

Natural operations: 1 assignments before the loop and $n-1$ loop steps with 1 comparison and at most 1 assignment each step, totally $2n-1$.
Natural memory: n integers for D and besides, 2 integesr for $j, MaxVal$.
Thus, Natural is $\Theta(n)$ complexity and Maximum is in $O(n)$ class, not $\Omega(n^2)$.
Is low bound also $\Omega(n)$? Right but will be proven later on.
Remark: We may apply Natural to the input list of numbers so reducing necessary memory from $O(n)$ to $O(1)$.

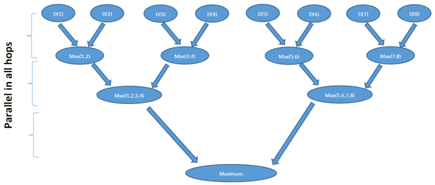
Parallel computations may change complexity.

If considering parallel computing and having parallel processing, with k workers, then totally different algorithm appear with a smaller compexity.

Theorem

Parallel Binary Tree algorithm *is in* $\Theta(\sum_{i=1}^{\lceil \log_2(n) \rceil} \lceil \frac{n}{k2^i} \rceil) = \Theta(\frac{n}{k})$.

$$\sum_{i=1}^{\lceil \log_2(n) \rceil} \frac{1}{n} \lceil \frac{n}{k2^i} \rceil < \frac{1}{n} \sum_{i=1}^{\lceil \log_2(n) \rceil} (\frac{n}{k2^i} + 1) = \sum_{i=1}^{\lceil \log_2(n) \rceil} \frac{1}{k2^i} + \frac{\lceil \log_2(n) \rceil}{n} < \sum_{i=1}^{\infty} \frac{1}{k2^i} + \frac{\lceil \log_2(n) \rceil}{n} \to_{n \to \infty} \frac{1}{k} \quad (4)$$



Do 7 comparison, 4 + 2 + 1. If having 4 workers, time delay is 3 < 7

Not all problems have a polynomial upper bound.

Example Prime number test. Input is a natural number, N. Need to verify whether N is prime or not.
Algorithm For each integer $i = 2, \cdots, \lfloor\sqrt{N}\rfloor$ divide N by i with remainder, if remainder is 0, return 0.
Analysis Algorithm compexity is $\Theta(\sqrt{N})$, so is this polynomial?!
The input is N represented as the string of (binary) digits. Then the size of input is $n = \lfloor\log_2(N)\rfloor + 1$ and the algorithm complexity is $\Theta(2^{n/2})$.
Challenge For a long time it was not known whether a polynomial algorithm may solve Prime number test. In 2002 Agrawal–Kayal–Saxena published a tricky algorithm and proved $O(\log(N)^{12})$ complexity bound. Under some conjectures its compexity reduces to $O(\log(N)^6)$.

• A related problem is to factor n into product of primes. An obvious enhancement of the above exponential algorithm solves this and it is not known whether a polynomial algorithm may solve.

• The expected exponential complexity of factorization problem is the main idea behind several cryptographic codes using the fact that the above problem is *intractable*.

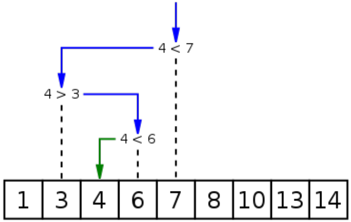
Divide-and-conquer principle.

Divide-and-Conquer priciple goes back to Julius Caesar and beyond, to Philip II of Macedon, father of Great Alexander.
The idea is to somehow decompose the problem into several similar simpler ones
Solve the subproblems and combine their solutions into a one for the whole one.
As the basis, problems of small size are solved explicitly.
In many cases, Divide-and-conquer implementation uses *recursion*
We start with probably the simplest example of this principle, Binary Search.

Binary search for element query: algorithm.

Problem Element query. Assume we have a large set of n different integers or any other objects with an integer indexing. How to organise the data in a structure allowing to answer the standard query: for some integer q return the position in the structure, where the number is stored, or 0, if it is not present?
Preprocess and Data structure: Sort numbers and put to a linear array, D.
Binary Element Query Algorithm: Having a queried number q as input, do:

```
1. if $q > D[n]$ or $q < D[1]$ return 0;
2. Set integers $l := 1, u := n$. while $l < u$ do:
3. $i = \lfloor (l+u)/2 \rfloor$. if $D[i] < q, l := i$ else if $D[i] > q, u := i$ else return $i$;
4. return 0
```



Binary search for element query: analysis.

Definition

Loop Invariant is a function taking loop status data or any statement about this.

Invariant: before and after every loop holds $i \leq q \leq u$.
Invariant: Let $w := u - i$ and wn, wp be the values of w in the next and previous loops, respectively. Then either $wn = \frac{wp}{2}$ or $wn = \frac{wp-1}{2}$ or else $wn = \frac{wp+1}{2}$.
Using these invariants, one proves:

Theorem

1. *Algorithm returns the correct position of q in D if exists, otherwise returns 0.*
2. *The number of loops and the algorithm complexity are of class $\Theta(\log_2(n))$.*

In the next step we will use an enhanced version of binary search, a method $BinaryElementQuery(q, low)$, where an additional Boolean variable low applies to the case, when q belongs to $(D[1], D[n])$ but is not equal to any of $D[i]$. In this case q is in between of some $D[j]$ and $D[j+1]$ and the enhanced method returns $D[j]$, if $low = 1$ and $D[j+1]$, otherwise. Note that j is equal to the value of l, in the moment, when the code reachs line **4**.

Binary search for range query.

Problem Range query. Assume we have a large set of n different integers. How to organise the data in a structure allowing to answer the standard query: for some integers $a \leq b$ return all the elements of the set which belong to the range $[a, b]$?
Preprocess and Data structure: Sort numbers and put to a linear array, D.
Binary Range Query Algorithm Having a query $[a, b]$, proceed as follows:

```
1. if $(a > D[n])$ or $(b < D[1])$ return $\emptyset$;
2. if $(a < D[1])$ set $a := D[1]$; if $(b > D[n])$, set $b := D[n]$
3. integer $s := BinaryElementQuery(a, 0)$, $f := BinaryElementQuery(b, 1)$
4. return $D(s), \cdots, D(f)$.
```

Theorem

Binary Range Query Algorithm complexity is $\Theta(\log_2(n)) + k)$, where k is the size of returned subset.

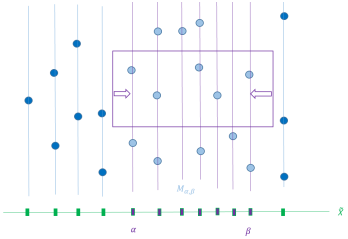
Range search for higher dimensions. Issues.

In 2d, 3d, etc, it is quite natural to consider range search problem as follows:
Problem Range query. Assume we have a large set of n different point $M \in \mathbb{R}^2$. How to organise the data in a structure allowing to answer the standard query: for some integers $a \le b, c \le d$ return the elements of M, which belong to the range $[a, b] \times [c, d]$?
Structure preparation: If reusing the idea of ordered array from 1d, we do:
1. Sort the points according to Y coordinate and split into an ordered arrays of horizontal subsets, $M = \sqcup_\beta M_\beta$ of points with the same Y coordinate equal to β.
2. Sort the points in each horizontal subset M_β according to X coordinate.
Fake Binary Range Query Algorithm with $[a, b] \times [c, d]$:
3. Applying 1d Binary Range Query, find all horizontal subsets M_β with $\beta \in [c, d]$
4. Applying 1d Binary Range Query to every horizontal subset found in **3**, find a smaller subset with $X \in [a, b]$.
Good news: Because of 1d Binary Range Query, both steps **3,4** have $\Theta(\log_2(n) + k)$ complexity, where k for step **4** is the size of the returned set...
Bad news .... but that for step **3** k may happen to fall in $\Theta(n)$:-(

Binary Range Query in 2d: data structure and algorithm

Data structure: Let $\widetilde{X}$ be the ordered array of X cooridnates in M.
For $\alpha, \beta \in \tilde{X}$, store the points $M_{\alpha,\beta} \subseteq M$ with $\alpha \leq X \leq \beta$ ordered by Y.
Let $minX, maxX, minY, maxY$ be the minimal/maximal values of X, Y over M.
Binary Range Query: with with $[a, b] \times [c, d]$:

```
1. if $(a > maxX)$ or $(b < minX)$ or $(c > maxY)$ or $(d < minY)$ return $\emptyset$
2. Set $a := max(a, minX), b := min(b, maxX)$
3. Set $c := max(c, minY), d := min(d, maxY)$
4. $\alpha := BinaryElementQuery(\widetilde{X}, a, 0)$, $\beta := BinaryElementQuery(\widetilde{X}, b, 1)$
5. return $BinaryRangeQuery(M_{\alpha,\beta}, [c, d])$.
```



Binary Range Query in 2d: analysis

Remark: The set $M_{\alpha,\beta}$ consists of points ordered by Y coordinate and 1d BinaryRangeQuery in line **5** finds the points by their Y coordinate.

Theorem

A: *Binary Range Query is correct.* **B:** *It uses the precomputed data of size $O(n^3)$ and belongs to the class $\Theta(\log_2(n) + k)$, where k is the size of returned subset.*

Proof: A: If $m \in M$ has $X(m) \in [a, b]$, then $X(m) \in [\alpha, \beta]$ by definition of $\widetilde{X}$.
B: The structure $M_{\alpha,\beta}, \alpha, \beta \in \widetilde{X}, \alpha < \beta$ consists of $O(n^2)$ containers of size $O(n)$ each. The query complexity follows from the Theorems about 1d binary element and range queries.
Remark: The size $O(n^3)$ may seem unnecessary: $M_{\alpha,\gamma} = M_{\alpha,\beta} \cup M_{\beta,\gamma}$. But the Y order of the union needs $O(n)$ complexity to recover from the parts orders.

Other sublinear range queries.

• The above 2d range query of $O(\log_2(n))$ complexity has a price of $O(n^3)$ size data structure. Is there a logarithm complexity query with smaller memory?
• The answer is very likely to be NO at least if the memory size is $O(n)$.
• Even $O(n^2)$ memory is prohibitive in computational geometry, so the above query with $O(n^3)$ memory is not used in practice.
• However, sublinear set range queries with linear sized memory exist. So-called *KD-tree* is a binary tree data structure that occupies linear memory and the query complexity is of class $\Theta(\sqrt{n} + k)$, where k is the size of returned set. See definitions and details in [BCKO]: section 5.2 and Lemma 5.4.
• This fact is in correlation with the calculus situation:

$\mathbb{R}$ has a linear order consistent with topology but $\mathbb{C}$ has no.

References

Smith, Henry J. Stephen, "On systems of linear indeterminate equations and congruences". *Phil. Trans. R. Soc. Lond.* **151** (1), 1861, 293–326.

de Berg, M., Cheong, O., van Krefeld, M., Overmars, M., "Computational Geometry". Springer.