1 Basic Algorithmic Tools

In lecture 1, we introduced you to some of the major hypotheses in Fine-Grained Complexity, and told you about many of the results that were known: both algorithmic results for various bottleneck problems, and many reductions between problems. In this lecture, we will give you a crash course in some of the basic algorithmic tools that are most commonly used over and over in fine-grained complexity, in order to design new algorithms. It's our hope that, by introducing you to some of these tools early on in the course, you'll get a good overview of what is known and what you might want to try in the open problem sessions.

We won't be able to *prove* many of the theorems stated in the lecture (some of the proofs are beyond the scope of the course!). We encourage you to look into them yourselves, as you like.

2 Sorting

Let's start with something simple. As we all know:

Theorem 2.1 *Given a list of n numbers, each described in b bits, we can sort the list in $O(n \log n \cdot b)$ time.*

Here, we assume that a comparison between two numbers takes $O(b)$ time, but the time can be faster for some models of computation. For example, on the so-called Word RAM with wordsize w, the memory is divided up into "registers", each of which contains an w-bit string (a word). Each simple operation runs in constant time and is performed on a constant number of registers. (Simple operations include: add/multiply two registers and put the result in two other registers, determine the minimum among two registers and put the outcome in another register, etc.)

2.1 Application: Subset Sum

Sorting alone can be useful for finding ways to improve over the obvious algorithms. Let's give an unexpected application of sorting: to solve an NP-hard problem!

SUBSET SUM: Given n integers $x_1, \ldots, x_n$, is there a subset $S \subseteq [n]$ such that $\sum_{i \in S} x_i = 0$?

You've probably seen in other classes that this problem is NP-complete.[1]

By trying all possible 2^n subsets, we can check whether a given set of n numbers has a subset sum solution in $2^n \cdot \text{poly}(n) \leq O^\star(2^n)$ time. And now I'm telling you that sorting can be used to speed-up this running time. This may sound a bit crazy: Sorting takes nearly-linear time. How are we going to solve an NP-hard problem with it?

Theorem 2.2 (Horowitz and Sahni [HS74]) SUBSET SUM *can be solved in $O^\star(2^{n/2})$ time.*

[1] If you haven't, I have a neat proof of a hardness reduction from 1-in-3SAT which is simpler than what I've seen in textbooks... if you bug me (Ryan), I'll probably write it up.

The algorithm for SUBSET SUM will start very similarly to the reduction from CNF-SAT to OV in the previous lecture, but it will have a much nicer conclusion: we'll just get a faster algorithm for SUBSET SUM. We will give a reduction from SUBSET SUM to Sorting.

1. Split $S = \{x_1, \ldots, x_n\}$ into two sets L and R, where $|L| = |R| \leq n/2 + 1$
2. List all $t = O(2^{n/2})$ subsets $L_1, \ldots, L_t$ of L, and all $t = O(2^{n/2})$ subsets $R_1, \ldots, R_t$ of R.
3. For all $i = 1, \ldots, t$, let $\ell_i = \sum_{a \in L_i} a$; the value of the subset sum defined by L_i. Similarly define $r_i = \sum_{b \in R_i} b$.
4. Sort the list:
$$\ell_1, \ldots, \ell_t, -r_1, \ldots, -r_t$$
in $O^\star(2^{n/2})$ time. (Here we assume the integers are poly(n)-bits long. It's OK if they aren't; it's still a polynomial factor in the input length.)
5. Scan through the sorted list, and look for an i and j such that $\ell_i = -r_j$. (Note that there will be an adjacent pair on the list, if such a pair exists!)

Now observe:

$$\text{There exists an } i,j \text{ such that } \ell_i + r_j = 0 \Leftrightarrow \text{there's an } i,j \text{ such that } \sum_{a \in L_i} a + \sum_{b \in R_j} b = 0$$
$$\Leftrightarrow \text{the instance has a subset sum solution.}$$

Therefore, by scanning the sorted list in $O(2^{n/2})$ time, we determine whether or not there's a subset sum. We have square-rooted the running time of the 2^n brute-force algorithm! And in fact, there's no faster algorithm for SUBSET SUM known.

This sorting-and-searching is a common technique, often called "meet-in-the-middle'. The reduction from CNF-SAT to OV from the last lecture came from an attempt to reduce CNF-SAT to a simpler problem like sorting.

3 Matrix Multiplication

Another tool we can often use to improve over brute-force algorithms is matrix multiplication:

> MATRIX MULTIPLICATION: Given two $n \times n$ matrices A and B, compute the $n \times n$ matrix $(A \cdot B)$ defined as:
> $$(A \cdot B)[i,j] = \sum_{k=1}^{n} A[i,k] \cdot B[k,j].$$

The obvious method for multiplying $n \times n$ matrices takes $O(n^3)$ additions and multiplications. Starting with Strassen's 1969 algorithm using $O(n^{\log_2(7)}) \leq O(n^{2.81})$ additions and multiplications [Str69], there have been many improvements on the constant ω, the matrix multiplication exponent. The latest word on matrix multiplication is:

Theorem 3.1 ([ADV+24]) MATRIX MULTIPLICATION *can be computed in $O(n^\omega)$ additions and multiplications, where $\omega < 2.371339$.*

(Comment: although we say "additions and multiplications", this translates into "time" directly, if the entries in the matrix are small. This comment applies to all other theorems in this lecture with "additions and multiplications as well.)

3.1 Example: 3-Clique / Triangle

MATRIX MULTIPLICATION is extremely versatile. A canonical application is to the following problem:

> 3-CLIQUE / TRIANGLE: Given an n-node graph, are there nodes a, b, c which form a triangle, i.e., are there edges $\{a, b\}$,$\{b, c\}$,$\{c, a\}$?

The obvious algorithm tries all possible a, b, c, and runs in $O(n^3)$ time. Matrix multiplication allows us to beat this brute-force search bound.

Theorem 3.2 ([IR78]) TRIANGLE *can be decided in* $O(n^\omega)$ *time.*

Given a "hard" problem Π that seems to require brute-force, if we can get a fine-grained reduction from Π to TRIANGLE, then we can solve Π faster.

TRIANGLE ALGORITHM: Let A be the $n \times n$ adjacency matrix of an n-node graph. Compute $B = A \cdot A$. If there is an i, j such that $B[i, j] \neq 0$ and $A[i, j] = 1$, then output *triangle* else output *no triangle*.

The algorithm is correct because:

$$\begin{aligned}\text{there are } i,j,k \text{ which form a triangle} &\Leftrightarrow \exists i,j,k \text{ such that} \{i,j\}, \{j,k\}, \{k,i\} \text{ are edges} \\ &\Leftrightarrow \exists i,j,k \text{ such that } A[i,j] = 1 \text{ and } A[i,k] \cdot A[k,j] = 1 \\ &\Leftrightarrow \exists i,j \text{ such that } A[i,j] = 1 \text{ and } B[i,j] \neq 0.\end{aligned}$$

4 Rectangular Matrix Multiplication

Is the matrix multiplication exponent $\omega = 2$? We don't know, and there is controversy over whether we should expect that it is 2. But sometimes for fine-grained applications, we need a matrix multiplication algorithm which is "optimal" in some sense. What can we do?

Well, if the "middle dimension" of the matrices is small (say we are multiplying $n \times \sqrt{n}$ and $\sqrt{n} \times n$ matrices), then we still need $\Omega(n^2)$ time to output the answer. But because we made the input smaller, we can hope for a faster algorithm than the $n \times n$ case of matrix multiplication. The formal problem is:

> (n, m, n)-RECTANGULAR MATRIX MULTIPLICATION: Given an $n \times m$ matrix A and $m \times n$ B, compute the $n \times n$ matrix $A \times B$.

There are two kinds of "optimal" rectangular matrix multiplication algorithms that are used often in fine-grained complexity. One kind has only poly$(\log n)$ extra running time overhead, and the other has $n^{o(1)}$ overhead. Depending on your intended application, you might choose one or the other.

Theorem 4.1 ([Cop82]) $(n, n^{0.1}, n)$-RECTANGULAR MATRIX MULTIPLICATION *can be computed in* $O(n^2 \cdot poly(\log n))$ *additions and multiplications.*[2]

Theorem 4.2 ([VXXZ24]) *For all* $\varepsilon > 0$, $(n, n^{0.32}, n)$-RECTANGULAR MATRIX MULTIPLICATION *can be done in* $O(n^{2+\varepsilon})$ *additions and multiplications.*

[2] We also need to work over a finite field, but let's not worry about that here.

4.1 Example: k-Dominating Set

Another graph problem that comes up often in fine-grained and fixed-parameter complexity is the k-Dominating Set problem:

> k-DOMINATING SET: Given an n-node graph, is there a set of k nodes S such that every node $v \notin S$ has an edge to some node in S?

Put another way, S is a dominating set if we can get to any other node in the graph by taking some edge out of S. When k is given as part of the input, k-Dominating Set is NP-complete. But for fixed k, there is a polynomial time algorithm:

For all $\binom{n}{k}$ ways to choose the subset S, check in $O(k \cdot n)$ time if every other vertex is adjacent to a node in S.

This trivial algorithm runs in $O(n^{k+1})$ time. It can be improved for large enough k, using rectangular matrix multiplication:

Theorem 4.3 *For all $k \geq 4$ and $\varepsilon > 0$, $2k$-DOMINATING SET can be solved in $O(n^{2k+\varepsilon})$ time.*

Proof. Given a graph over the nodes $\{1, \ldots, n\}$, let $\mathcal{C}$ be the collection of all possible subsets of the nodes of size k; note $|\mathcal{C}| = \binom{n}{k}$. We prepare an $\binom{n}{k} \times n$ matrix A and an $n \times \binom{n}{k}$ matrix B, where the rows of A and columns of B are indexed by S. For a k-set $S \in \mathcal{C}$, and $i = 1, \ldots, n$, we define

$$A[S, i] = \begin{cases} 0 & \text{if node } i \text{ has an edge to a node in } S, \text{ or } i \in S \\ 1 & \text{otherwise} \end{cases}$$

and

$$B[i, S] = \begin{cases} 0 & \text{if node } i \text{ has an edge to a node in } S, \text{ or } i \in S \\ 1 & \text{otherwise} \end{cases}$$

Let $C = A \times B$. By Theorem 4.2 and the fact that $k \geq 4$, C can be computed in $O(n^{2k+\varepsilon})$ time for every $\varepsilon > 0$. We claim that there is a $2k$-dominating set in the graph if and only if C contains a 0-entry. This follows because:

$$\begin{aligned} C[S, T] = 0 &\Leftrightarrow \text{for all } i, A[S, i] \cdot B[i, T] = 0 \\ &\Leftrightarrow \text{every node has an edge to } S \cup T \text{ or is in } S \cup T \\ &\Leftrightarrow S \cup T \text{ is a dominating set.} \end{aligned}$$

□

So we can find a $2k$-dominating set in about n^{2k} time. Can the algorithm for k-DOMINATING SET be improved further? It turns out that this would refute SETH! See [PW10].

5 Useful Theorems About Polynomials

Polynomials are extremely useful in algorithms: when you can convert your input into a reasonable problem about polynomials, this often translates to more efficient algorithms. Here we consider polynomials over the integers, but our discussion will also apply to polynomials over any sufficiently large field (at least as large as the degree of the polynomial).

The first magic trick we can do with polynomials is that we can evaluate them very quickly on many points of our choosing. These results can be found in the nice book [vzGG13].

Theorem 5.1 (Multipoint Evaluation) *Given $p(x) = \sum_{i=0}^{n} c_i x^i$ listed as its coefficients, we can evaluate $p(x)$ on any k points, in $O((n+k) \cdot poly(\log(n+k)))$ additions and multiplications.*

Sometimes, when the points are nicely chosen to make evaluation easy, Theorem 5.1 theorem referred to as the Fast Fourier Transform.

Theorem 5.1 is a huge improvement over the obvious algorithm, which would take at least $k \cdot n$ operations (n operations for each point).

Letting $k = n + 1$, and letting the points be $0, 1, \ldots, n$, we can think of this theorem as a *transformation* of a polynomial, from one representation into another representation. We start with the *coefficient representation* of $p(x)$:

$$[c_0, \ldots, c_n],$$

and we transform it into a *point representation*:

$$[p(0), \ldots, p(n)].$$

Recall that any list of $n+1$ function values $(x_0, f(x_0)), \ldots, (x_n, f(x_n))$ defines a degree-n polynomial. We can also efficiently invert this transformation, going from the point representation back to the coefficient representation:

Theorem 5.2 (Interpolation) *Given $(x_0, y_0), \ldots, (x_n, y_n)$, we can compute a degree-n polynomial $p(x) = \sum_{i=0}^{n} c_i x^i$ such that $p(x_i) = y_i$ for all i, in $n \cdot poly(\log n)$ additions and multiplications.*

Given these two theorems, we can *multiply* two polynomials efficiently as well:

Theorem 5.3 (Multiplication) *Given $p(x), q(x)$ of degree n, we can compute the polynomial $p(x) \cdot q(x)$ in $n \cdot poly(\log n)$ additions and multiplications.*

Again, this is a big improvement over the obvious algorithm, which would take at least $\Omega(n^2)$ time to compute all the terms of the product.

Proof. Let's describe how one can efficiently multiply, given the Evaluation and Interpolation theorems. The idea is that we first evaluate $p(x)$ and $q(x)$ at $2n+1$ points (Theorem 5.1), getting the point representations $[p(0), \ldots, p(2n)]$ and $[q(0), \ldots, q(2n)]$. Then we multiply the point representations together, getting $[p(0) \cdot q(0), \ldots, p(2n) \cdot q(2n)]$. Now this is a point representation of a polynomial that agrees with $p(x) \cdot q(x)$ on $2n+1$ points. But the polynomial $p(x) \cdot q(x)$ has degree at most $2n$, and is therefore uniquely defined by its behavior on $2n+1$ points. Therefore this is a point representation of the polynomial $p(x) \cdot q(x)$. By interpolation (Theorem 5.2), we can recover the coefficients of $p(x) \cdot q(x)$ efficiently. □

5.1 Example: 3-SUM on small magnitude numbers

Last time, we introduced you to the 3-SUM problem: given a set of n integers, are there three which sum to zero? There is an $O(n^2)$ time algorithm, and the 3-SUM Hypothesis is that there is no $n^{2-\varepsilon}$ time algorithm, for any $\varepsilon > 0$. Here, we show that one can improve the running time significantly, if the numbers have small magnitude.

Theorem 5.4 *Suppose $S \subseteq [-M, M] \cap \mathbb{Z}$ and $|S| = n$. Then 3-SUM on S can be solved in $O(M \cdot poly(\log M) + n)$ time.*

Proof. We'll solve a variant on 3-SUM which turns out to be equivalent (future pset problem?). In particular, given a set S_1, S_2, S_3 of n integers in $[-M, M]$, we want to determine if there are numbers $x_1 \in S_1$, $x_2 \in S_2$, $x_3 \in S_3$ such that $x_1 + x_2 + x_3 = 0$.

First, add M to every integer in S_1, S_2, S_3: our interval of integers is now $[0, 2M]$, and our goal now is to find three integers that sum to $3M$.

We will encode this problem using polynomials. For $i = 1, 2, 3$, define

$$p_i(x) = \sum_{k \in S_i} x^k.$$

This is a polynomial of degree at most $2M$, which has a 0-coefficient for x^k if $k \notin S$, and a 1-coefficient for x^k if $k \in S$. Consider the product

$$q(x) = p_1(x) \cdot p_2(x) \cdot p_3(x).$$

What is encoded in the coefficient for x^{3M} in $q(x)$? It counts the number of ways to choose an x^{k_1} in the sum for $p_1(x)$, an x^{k_2} in the sum for $p_2(x)$, and an x^{k_3} in the sum for $p_3(x)$, such that $k_1 + k_2 + k_3 = 3M$. That is, the x^{3M} coefficient is nonzero if and only if there is a 3-SUM solution with one number from each S_i.

Finally, observe it takes $O(n + M)$ time to form the polynomials, and $M \cdot \text{poly}(\log M)$ time to multiply them. □

6 "Four Russians", i.e., The Art of Tables (Optional Material)

This is really a technique more than a particular theorem, but it's often very useful for finding algorithms that beat the obvious algorithm by polylog factors. (This technique is usually involved when people talk about "shaving logarithms" from the running times of problems.) It originates from a paper by four scientists working in the former Soviet Union, some of whom were Russian [ADKF70].

The basic idea is to find a computation that is repeated often in an algorithm, and to try to replace the computation with a lookup table in a clever way.

We'll demonstrate the idea on matrix-vector multiplication. Suppose we have an $n \times n$ 0-1 matrix A, and we wish to compute $A \cdot v$ for a given vector $v \in \{0, 1\}^n$, modulo 2. (That is, all the inner products are computed mod 2.) It seems that in general, this computation should require $\Omega(n^2)$ time, because we should have to read the entire matrix A.

By preprocessing A, and make $A \cdot v$ computations faster, as follows. We choose a block-size b, and partition A into $(n/b)^2$ blocks which are $b \times b$. (In particular, the n rows are partitioned into n/b groups of b rows each, and the n columns are partitioned similarly.) Index the $b \times b$ blocks $A_{i,j}$ for $i, j = 1, \ldots, n/b$, so that A looks like:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n/b} \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ A_{n/b,1} & A_{n/b,2} & \cdots & A_{n/b,n/b} \end{bmatrix}$$

For every $i, j = 1, \ldots, n/b$ and $w \in \{0, 1\}^b$, we compute the result $A_{i,j} \cdot w$ and store the result in a lookup table $T_{i,j}$. (Since we are working modulo 2, note that $A_{i,j} \cdot w$ is just a b-bit vector.) This takes time $O((n/b)^2 \cdot 2^b \cdot b^2) \leq O(n^2 \cdot 2^b)$.

Now suppose we want to compute $A \cdot v$ for a given $v \in \{0, 1\}^n$. We prepare b-bit blocks $w_1, \ldots, w_{n/b}$ which will hold the answer, and are initially all-zero. We partition v into b-bit blocks

$$v = [v_1 \cdots v_{n/b}].$$

For all i, j, we look up $x_{i,j} = A_{i,j} \cdot v_j$ in table $T_{i,j}$, and add the vector $x_{i,j}$ to w_i.

> **Exercise:** Verify that the final value of w is indeed $A \cdot v$.

Assuming the word-size of our machine is at most b, and that lookups take $O(1)$ time, the operation $A \cdot v$ takes $O((n/b)^2)$ time on a Word RAM. For $b = \varepsilon \log n$, we can conclude the following:

After at most $O(n^{2+\varepsilon})$ steps of preprocessing on A, we can prepare a data structure that can compute $A \cdot v$ for any given $v \in \{0, 1\}^n$ in $O(n^2/\log^2 n)$ time [Wil07].

References

[ADKF70] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradžev. On economical construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk SSSR*, 194(11), 1970.

[ADV+24] Josh Alman, Ran Duan, Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. More asymmetry yields faster matrix multiplication. *CoRR*, abs/2404.16349, 2024.

[Cop82] Don Coppersmith. Rapid multiplication of rectangular matrices. *SIAM J. Comput.*, 11(3):467–471, 1982.

[HS74] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *J. ACM*, 21(2):277–292, 1974.

[IR78] Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. *SIAM J. Comput.*, 7(4):413–423, 1978.

[PW10] Mihai Pătraşcu and Ryan Williams. On the possibility of faster SAT algorithms. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1065–1075. SIAM, 2010.

[Str69] Volker Strassen. Gaussian elimination is not optimal, 1969.

[VXXZ24] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*, pages 3792–3835. SIAM, 2024.

[vzGG13] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra (3. ed.)*. Cambridge University Press, 2013.

[Wil07] Ryan Williams. Matrix-vector multiplication in sub-quadratic time: (some preprocessing required). In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 995–1001. SIAM, 2007.