**15-750: Graduate Algorithms** January 18, 2016

# Lecture 1: Introduction and Strassen's Algorithm

*Lecturer: Gary Miller* *Scribe: David Witmer*

## 1 Introduction

### 1.1 Machine models

In this class, we will primarily use the Random Access Machine (RAM) model. In this model, we have a memory and a finite control. The finite control can read or write to any location in the memory in unit time. In addition, it can perform arithmetic operations $+$, $-$, $\times$, and $\div$ in unit time. We will not require a more formal definition.

Later in the semester, we will consider the Parallel Random Access Machine (PRAM) model and circuit models. We will *not* discuss caching models (though they may appear on a homework assignment), memory hierarchies, or pipelining.

### 1.2 Asymptotic complexity

We will only consider asymptotic complexity of algorithms: The input size (typically $n$) grows to infinity. We now define the asymptotic notation we will use to describe time and space used by algorithms in this setting.

**Definition 1.1.** Let $f, g : \mathbb{N} \to \mathbb{N}$.

1. $f(n) \in O(g(n))$ if there exist $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$.
2. $f(n) \in o(g(n))$ if for all $c > 0$, there exists $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$.

For intuition, $f \in O(g)$ means that $f$ grows no faster than $g$, and $f \in o(g)$ means that $f$ grows more slowly than $g$. We often write $f = O(g)$ to indicate $f \in O(g)$ and $f = o(g)$ for $f \in o(g)$.

There are two alternative definitions for $f \in \Omega(g)$.

**Definition 1.2.** $f \in \Omega(g)$ if $g \in O(f)$.

**Definition 1.3.** $f \in \Omega(g)$ if there exists $c > 0$ such that for all $n_0 \in \mathbb{N}$, there exists $n_1 \geq n_0$ such that $f(n_1) \geq c \cdot g(n)$.

The phrase "for all $n_0 \in \mathbb{N}$, there exists $n_1 \geq n_0$" simply means "infinitely often". Intuitively, $f \in \Omega(g)$ means that $f$ grows at least as quickly as $g$. The two definitions usually agree, but there are some cases in which they don't. For instance, $|n \sin n| \in \Omega(n)$ under Definition 1.3, but this is not true under Definition 1.2. Such examples will not come up in this class, so for us these definitions are essentially equivalent.

**Example 1.4.** Show that $2n^2 + n + 1 \in O(n^2)$.

*Proof.* We try $c = 3$. We need to find $n_0$ such that $2n^2 + n + 1 \leq 3n^2$ when $n \geq n_0$. This is equivalent to $n^2 - n - 1 \geq 0$, and it is easy to see that this holds for $n \geq 2$. We can therefore choose $c = 3$ and $n_0 = 2$.

Alternatively, we can use L'Hôpital's Rule. Observe that

$$\lim_{n\to\infty} \frac{2n^2+n+1}{n^2} = \lim_{n\to\infty} \frac{4n+1}{2n} = 2,$$

This means that for any $\epsilon > 0$, there exists an $n_0$ for which we can choose $c = 2+\epsilon$. □

# 2 Strassen's Algorithm

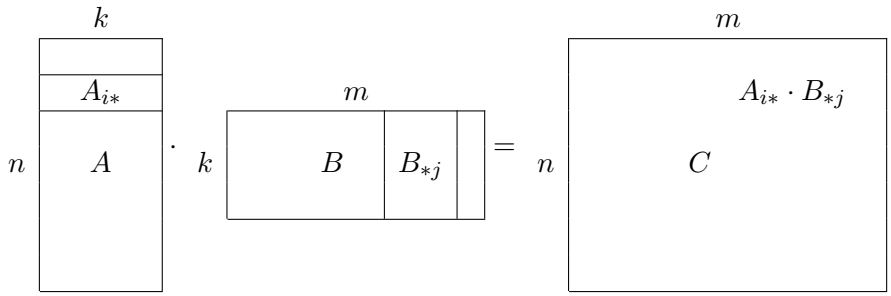
For an $n \times m$ matrix $M$, we will denote the $(i,j)$ entry by $M_{ij}$, the $i$th row by $M_{i*}$, and the $j$th column by $M_{*j}$.

## 2.1 Matrix multiplication

**Definition 2.1.** Given matrices $A \in \mathbb{R}^{n\times k}$ and $B \in \mathbb{R}^{k\times m}$, $AB$ is defined to be a matrix in $\mathbb{R}^{n\times m}$ such that

$$(AB)_{ij} = \sum_{t=1}^{k} A_{it}B_{tj}.$$

Here is a picture:




The product of two matrices can also be represented as the sum of outer products of their row and column vectors.

**Claim 2.2.** *Let $A \in \mathbb{R}^{n\times k}$ and $B \in \mathbb{R}^{k\times m}$. Then*

$$AB = \sum_{t=1}^{k} A_{*t}B_{t*}^{\top}.$$

This follows immediately from Definition 2.1, since $(A_{*t}B_{t*}^{\top})_{ij} = A_{it}B_{tj}$.

We recall some basic facts about matrix multiplication.

**Fact 2.3.**

1. $(AB)C = A(BC)$
2. *In general,* $AB \neq BA$.
3. *For any scalar* $\lambda$,

$$\lambda A = \begin{pmatrix} \lambda & & \\ & \ddots & \\ & & \lambda \end{pmatrix} A.$$

4. $A(B+C) = AB + AC$

Our goal in this lecture is to give an algorithm for matrix multiplication that is as efficient as possible with respect to both time and space. We will start by giving a naive algorithm that runs in time $O(n^3)$ and then show how we can do better using Strassen's Algorithm. We will only consider *dense* matrix multiplication, in which most of the entries of the input matrices are nonzero. For sparse matrices, in which most of the entries are 0, there are algorithms for matrix multiplication that leverage this sparsity to get a better runtime. We will not discuss algorithms for sparse matrices in this class.

## 2.2 Naive algorithm

We begin with a naive algorithm that loops through all entries of the output and computes each one.

**Algorithm 1** Naive matrix multiplication

**Input:** $A, B \in \mathbb{R}^{n \times n}$
**Output:** $AB$
  **for** $i = 1$ to $n$ **do**
    **for** $j = 1$ to $n$ **do**
      Set $C_{ij} = \sum_{t=1}^{n} A_{it} B_{tj}$
    **end for**
  **end for**
  **return** $C$

This requires $n^3$ multiplications and $(n-1)n^2$ additions, so the total runtime is $O(n^3)$.

## 2.3 Recursive algorithm

Next, we will give a recursive algorithm that also runs in time $O(n^3)$. Strassen's Algorithm will use a similar recursive framework to achieve subcubic runtime. We assume that $n = 2^k$ for some $k$. For $A \in \mathbb{R}^{n \times n}$, we write

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},$$

where each of the $A_{ij}$'s is $n/2 \times n/2$.

**Algorithm 2** Recursive matrix multiplication

**Input:** $A, B \in \mathbb{R}^{n \times n}$
**Output:** $AB$

```
function M(A,B)
    if A is 1 × 1 then
        return a_11 b_11
    end if
    for i = 1 to 2 do
        for j = 1 to 2 do
            Set C_ij = M(A_i1, B_1j) + M(A_i2, B_2j)
        end for
    end for
    return [C_11 C_12; C_21 C_22]
end function
```

**Correctness** First, we need to prove that this algorithm is correct. That is, we need to prove that $M(A, B) = AB$. We will do this by induction on $n$. In the base case, $n = 1$ and the algorithm correctly returns $a_{11}b_{11}$.

In the inductive case, we assume that $M(A, B) = AB$ for all $n \times n$ matrices with $n < n_0$. From the definition of matrix multiplication, it is clear that

$$(AB)_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j}.$$

By induction, $M(A_{i1}, B_{1j}) = A_{i1}B_{1j}$ and $M(A_{i2}, B_{2j}) = A_{i2}B_{2j}$, so we see that

$$(AB)_{ij} = M(A_{i1}, B_{1j}) + M(A_{i2}, B_{2j}) = M(A, B).$$

**Runtime** Define $T(n)$ to be the number of operations required for the algorithm to multiply two $n \times n$ matrices. The algorithm makes eight recursive calls. It also adds two $n \times n$ matrices, which requires $cn^2$ time for some constant $c$. We therefore obtain the following recurrence:

$$T(n) \leq 8T(n/2) + cn^2 \qquad T(1) = 1.$$

**Claim 2.4.** $T(n) = O(n^3)$.

We will prove this claim in Section 3.

If the number of recursive calls is smaller, the runtime will be faster. Specifically, we will also prove the following claim in Section 3.

**Claim 2.5.** *Assume*

$$T(n) \leq 7T(n/2) + cn^2 \qquad \textit{and} \qquad T(1) = 1.$$

*Then* $T(n) = O(n^{\log_2 7})$.

Strassen's Algorithm makes only *seven* recursive calls, so it runs in time $O(n^{\log_2 7}) = O(n^{2.807\cdots})$, faster than $O(n^3)$.

## 2.4 Strassen's Algorithm

We again consider multiplying $n \times n$ matrices broken into $n/2 \times n/2$ blocks as follows:

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}, \qquad N = \begin{pmatrix} E & F \\ G & H \end{pmatrix}.$$

Consider the following matrices:

$$\begin{aligned} S_1 &= (B - D)(G + H) \\ S_2 &= (A + D)(E + H) \\ S_3 &= (A - C)(E + F) \\ S_4 &= (A + B)H \\ S_5 &= A(F - H) \\ S_6 &= D(G - E) \\ S_7 &= (C + D)E. \end{aligned}$$

The product $MN$ can be computed using these seven matrices.

**Claim 2.6.**

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} S_1 + S_2 - S_4 + S_6 & S_4 - S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{pmatrix}.$$

*Proof.* We will prove the claim for only the lower left submatrix. The proofs for the other three are similar. In particular, we want to show that $S_6 + S_7 = CE + DG$. Plugging in definitions, we obtain

$$S_6 + S_7 = D(G - E) + (C + D)E = DG - DE + CE + DE = CE + DG. \qquad \square$$

We can now write down Strassen's Algorithm.

**Algorithm 3** Strassen's Algorithm

**function** STRASSEN($M$,$N$)
  **if** $M$ is $1 \times 1$ **then**
    **return** $M_{11}N_{11}$
  **end if**
  Let $M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$ and $N = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$
  Set $S_1 = \text{STRASSEN}(B - D, G + H)$
  Set $S_2 = \text{STRASSEN}(A + D, E + H)$
  Set $S_3 = \text{STRASSEN}(A - C, E + F)$
  Set $S_4 = \text{STRASSEN}(A + B, H)$
  Set $S_5 = \text{STRASSEN}(A, F - H)$
  Set $S_6 = \text{STRASSEN}(D, G - E)$
  Set $S_7 = \text{STRASSEN}(C + D, E)$
  **return** $\begin{pmatrix} S_1 + S_2 - S_4 + S_6 & S_4 - S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{pmatrix}$
**end function**

**Correctness** The correctness of this algorithm follows immediately from Claim 2.6.

**Runtime** There are seven recursive calls and the additions and subtractions take time $cn^2$ for some constant $c$. The runtime $T(n)$ therefore satisfies

$$T(n) \leq 7T(n/2) + cn^2 \qquad T(1) = 1. \tag{1}$$

By Claim 2.5, the runtime is $O(n^{\log_2 7})$.

As a side note, we point out that we are doing 18 additions of $n/2 \times n/2$ matrices, so we can take the constant $c$ to be $9/2$.

We make two remarks about this algorithm. First, even though it is asymptotically faster than the naive algorithm, Strassen's Algorithm does not become faster than the naive algorithm until the dimension of the matrices is on the order of 10,000. Instead of recursing all the way down to $1 \times 1$ matrices, we can just run the naive algorithm once the matrices become small enough to get a faster algorithm.

Second, we have not paid much attention to the space the algorithm is using. Computing $S_1, \ldots, S_7$ and the matrix that we return in the last step requires $O(n^2)$ space. We can therefore write a recursion that is identical to (1) for space to see that the total space used is $O(n^{\log_2 7})$. The naive algorithm uses only $O(n^2)$ space: We need $2n^2$ space for the inputs $A$ and $B$, allocate $n^2$ space for the output, and then construct the output by computing each $A_{it}A_{tj}$ and adding it to location $(i, j)$ in the output matrix.

## 2.5 Running Strassen's Algorithm using only $O(n^2)$ space

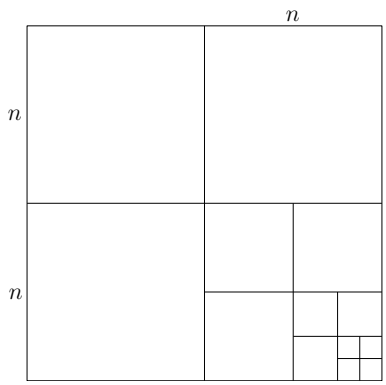
By being a bit more careful about how we use space, we can also run Strassen's Algorithm in $O(n^2)$ space. Rather than computing each of the $S_i$'s in its own space, we can use the same space for each of these recursive calls. We start with $3n^2$ space for the two input matrices and the output matrix. We then allocate $3(n/2)^2$ space for the recursive call to compute $S_1$. Once $S_1$ is returned, we add it to the upper left quadrant of the output matrix. In this same $3(n/2)^2$ space, we compute $S_2$, and then we add it to the upper left and lower right quadrants of the output matrix. We continue in the same manner for the rest of the $S_i$'s.

Let $W(n)$ be the amount of memory needed to multiply two $n \times n$ matrixes. From the above discussion, we see that

$$W(n) = 3n^2 + W(n/2).$$

**Claim 2.7.** $W(n) \leq 4n^2$.

The picture below shows that the claim holds.



The three $n \times n$ squares are the space needed for the top-level call, the three $n/2 \times n/2$ squares are the space needed for the first level of recursion, etc. We will also give a more formal proof in Section 3

## 3 Solving recurrences

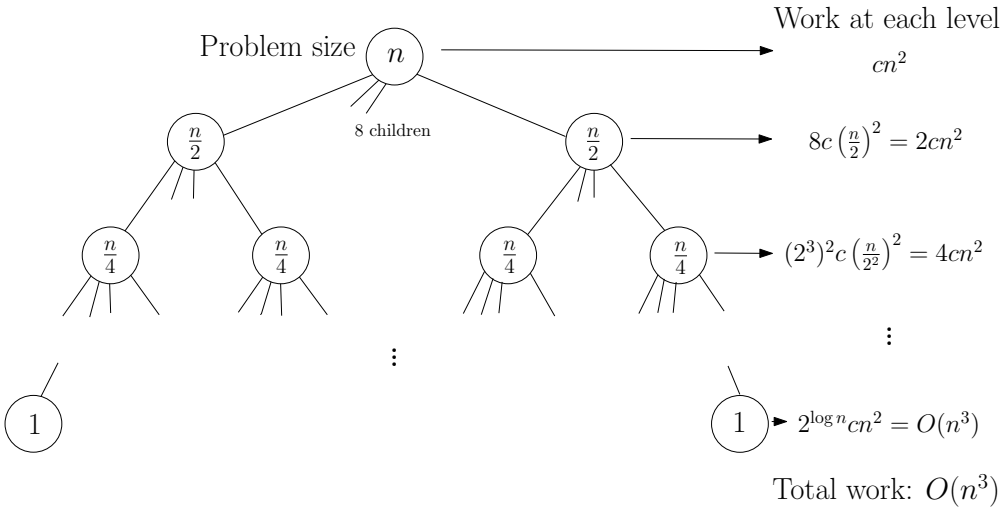
In this section, we will show how to solve the three recurrences mentioned in the last section using the "tree of recursive calls" method.

**Claim 2.4.** If it holds that

$$T(n) \leq 8T(n/2) + cn^2 \qquad T(1) = 1,$$

then $T(n) = O(n^3)$.

To solve this recurrence, we recursively evaluate $T$ and write resulting function calls in a tree.



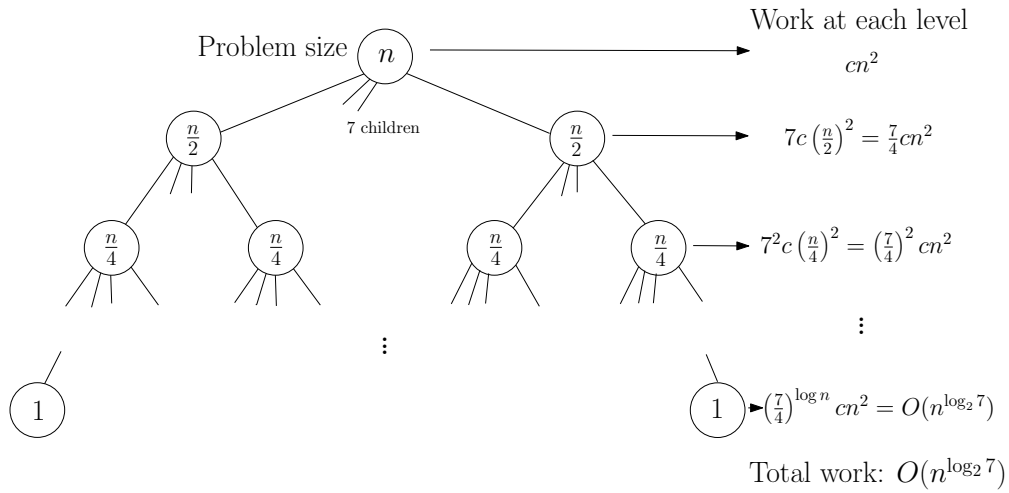

We then calculate the total amount of work at each level and sum over all levels of the tree.

**Claim 2.5.** Assume

$$T(n) \leq 7T(n/2) + cn^2 \qquad \text{and} \qquad T(1) = 1.$$

Then $T(n) = O(n^{\log_2 7})$.

To show this, we consider a similar tree of recursive calls.




**Claim 2.7.** Assume

$$W(n) = 3n^2 + W(n/2).$$

Then $W(n) \leq 4n^2$.

Here, there is only one subproblem, so the recursive tree is just a path.

$$\begin{aligned}
W(n) &= 3n^2 + W(n/2) \\
&= 3n^2 + 3\left(\frac{n}{2}\right)^2 + W(n/4) \\
&= 3n^2 + \frac{3}{4}n^2 + 3\left(\frac{n}{4}\right)^2 + W(n/8) \\
&\vdots \\
&= 3n^2\left(\frac{1}{4} + \frac{1}{4^2} + \cdots\right) \\
&\leq 3n^2\left(\frac{1/4}{1-1/4}\right) \\
&= 4n^2.
\end{aligned}$$