Students should be familiar with inductive proofs, recursion, data structures, and programming at the CS3114 level.

**Algebraic and Numeric Algorithms**

- Measuring cost of arithmetic and numerical operations:
  - Measure size of input in terms of *bits*.
- Algebraic operations:
  - Measure size of input in terms of *numbers*.
- In both cases, measure complexity in terms of basic arithmetic operations: $+, -, *, /$.
- Sometimes, measure complexity in terms of bit operations to account for large numbers.
- Size of numbers may be related to problem size:
  - Pointers, counters to objects.
  - Resolution in geometry/graphics (to distinguish between object positions).

**Exponentiation**

Given positive integers $n$ and $k$, compute $n^k$.

**Algorithm:**

```plaintext
p = 1;
for (i=1 to k)
    p = p * n;
```

**Analysis:**

- Input size: $\Theta(\log n + \log k)$.
- Time complexity: $\Theta(k)$ multiplications.
- This is exponential in input size.

**Faster Exponentiation**

Write $k$ as:

$$k = b_12^{t} + b_{t-1}2^{t-1} + \cdots + b_12 + b_0, \ b \in \{0, 1\}.$$ 

Rewrite as:

$$k = (\cdots (b_2 + b_{t-1})2 + \cdots + b_2)2 + b_12 + b_0.$$ 

New algorithm:

```plaintext
p = n;
for (i = t-1 downto 0)
    p = p * p * exp(n, b[i])
```

**Analysis:**

- Time complexity: $\Theta(t) = \Theta(\log k)$ multiplications.
- This is exponentially better than before.
Greatest Common Divisor

- The Greatest Common Divisor (GCD) of two integers is the greatest integer that divides both evenly.
- Observation: If \( k \) divides \( n \) and \( m \), then \( k \) divides \( n - m \).
- So,
  \[
  f(n, m) = f(n - m, n) = f(m, n - m) = f(m, n).
  \]
- Observation: There exists \( k \) and \( l \) such that
  \[
  n = km + l \quad \text{where} \quad m > l \geq 0. \\
  n = \lfloor n/m \rfloor m + n \mod m.
  \]
- So,
  \[
  f(n, m) = f(m, l) = f(m, n \mod m).
  \]

GCD Algorithm

\[
 f(n, m) = \begin{cases} 
 n & m = 0 \\
 f(m, n \mod m) & m > 0 
\end{cases}
\]

```c
int LCF(int n, int m) {
  if (m == 0) return n;
  return LCF(m, n % m);
}
```

Analysis of GCD

- How big is \( n \mod m \) relative to \( n \)?
  \[
  n \geq m \Rightarrow n/m \geq 1 \\
  \Rightarrow 2\lfloor n/m \rfloor > n/m \\
  \Rightarrow m\lfloor n/m \rfloor > n/2 \\
  \Rightarrow n - n/2 > n - m\lfloor n/m \rfloor = n \mod m \\
  \Rightarrow n/2 > n \mod m
  \]
- The first argument must be halved in no more than 2 iterations.
- Total cost:

Multiplying Polynomials (1)

\[
 P = \sum_{i=0}^{n-1} p_ix^i, \quad Q = \sum_{i=0}^{n-1} q_ix^i.
\]
- Our normal algorithm for computing \( PQ \) requires \( \Theta(n^2) \) multiplications and additions.
Multiplying Polynomials (2)

- Divide and Conquer:
  \[
  P_1 = \sum_{i=0}^{n/2-1} p_i x^i \\
  P_2 = \sum_{i=n/2}^{n-1} p_i x^{-i/2}
  \]
  \[
  Q_1 = \sum_{i=0}^{n/2-1} q_i x^i \\
  Q_2 = \sum_{i=n/2}^{n-1} q_i x^{-i/2}
  \]
  \[
  PQ = (P_1 + x^{n/2}P_2)(Q_1 + x^{n/2}Q_2) \\
  = P_1Q_1 + x^{n/2}(P_2Q_1 + P_1Q_2) + x^nP_2Q_2.
  \]

- Recurrence:
  \[
  T(n) = 4T(n/2) + O(n).
  \]
  \[
  T(n) = \Theta(n^\log_25).
  \]

Multiplying Polynomials (3)

Observation:
\[
(P_1 + P_2)(Q_1 + Q_2) = P_1Q_1 + (Q_1P_2 + P_1Q_2) + P_2Q_2
\]
\[
(Q_1P_2 + P_1Q_2) = (P_1 + P_2)(Q_1 + Q_2) - P_1Q_1 - P_2Q_2
\]
Therefore, PQ can be calculated with only 3 recursive calls to a polynomial multiplication procedure.

Recurrence:
\[
T(n) = 3T(n/2) + O(n)
\]
\[
\log_b a = \log_23 \approx 1.59.
\]
\[
T(n) = \Theta(n^{1.59}).
\]

Matrix Multiplication

Given: \(n \times n\) matrices A and B.

Compute: \(C = A \times B\).

\[
c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}
\]

Straightforward algorithm:
- \(\Theta(n^3)\) multiplications and additions.

Lower bound for any matrix multiplication algorithm: \(\Omega(n^2)\).

Strassen's Algorithm

1. Trade more additions/subtractions for fewer multiplications in \(2 \times 2\) case.
2. Divide and conquer.

In the straightforward implementation, \(2 \times 2\) case is:
\[
c_{11} = a_{11}b_{11} + a_{12}b_{21}
\]
\[
c_{12} = a_{11}b_{12} + a_{12}b_{22}
\]
\[
c_{21} = a_{21}b_{11} + a_{22}b_{21}
\]
\[
c_{22} = a_{21}b_{12} + a_{22}b_{22}
\]
Requires 8 multiplications and 4 additions.
Another Approach (1)

Compute:

\[
\begin{align*}
m_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\
m_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
m_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) \\
m_4 &= (a_{11} + a_{12})b_{22} \\
m_5 &= a_{11}(b_{12} - b_{22}) \\
m_6 &= a_{22}(b_{21} - b_{11}) \\
m_7 &= (a_{21} + a_{22})b_{11}
\end{align*}
\]

7 multiplications and 18 additions/subtractions.

Another Approach (2)

Then:

\[
\begin{align*}
c_{11} &= m_1 + m_2 - m_4 + m_6 \\
c_{12} &= m_4 + m_5 \\
c_{21} &= m_6 + m_7 \\
c_{22} &= m_2 - m_3 + m_5 - m_7
\end{align*}
\]

Strassen’s Algorithm (cont)

Divide and conquer step:

Assume \( n \) is a power of 2.

Express \( C = A \times B \) in terms of \( \frac{n}{2} \times \frac{n}{2} \) matrices.

\[
\begin{bmatrix}
c_{11} & c_{12} \\
c_{21} & c_{22}
\end{bmatrix} =
\begin{bmatrix}
a_{11} & a_{12} \\
a_{21} & a_{22}
\end{bmatrix}
\begin{bmatrix}
b_{11} & b_{12} \\
b_{21} & b_{22}
\end{bmatrix}
\]

Strassen’s Algorithm (cont)

By Strassen’s algorithm, this can be computed with 7 multiplications and 18 additions/subtractions of \( \frac{n}{2} \times \frac{n}{2} \) matrices.

Recurrence:

\[
\begin{align*}
T(n) &= 7T(n/2) + 18(n/2)^2 \\
T(n) &= \Theta(n^{\log_27}) = \Theta(n^{2.81}).
\end{align*}
\]

Current “fastest” algorithm is \( \Theta(n^{2.376}) \).

Open question: Can matrix multiplication be done in \( O(n^2) \) time?

But, this has a high constant due to the additions. This makes it rather impractical in real applications.

But this “fastest” algorithm is even more impractical due to overhead.
Introduction to the Sliderule

Compared to addition, multiplication is hard.

In the physical world, addition is merely concatenating two lengths.

Observation: \[ \log nm = \log n + \log m. \]

Therefore, \[ nm = \text{antilog}(\log n + \log m). \]

What if taking logs and antilogs were easy?

Introduction to the Sliderule (2)

The sliderule does exactly this!
- It is essentially two rulers in log scale.
- Slide the scales to add the lengths of the two numbers (in log form).
- The third scale shows the value for the total length.

Representing Polynomials

A vector \( a \) of \( n \) values can uniquely represent a polynomial of degree \( n - 1 \)
\[ P_a(x) = \sum_{i=0}^{n-1} a_i x^i. \]

Alternatively, a degree \( n - 1 \) polynomial can be uniquely represented by a list of its values at \( n \) distinct points.
- Finding the value for a polynomial at a given point is called evaluation.
- Finding the coefficients for the polynomial given the values at \( n \) points is called interpolation.

Multiplication of Polynomials

To multiply two \( n - 1 \)-degree polynomials \( A \) and \( B \) normally takes \( \Theta(n^2) \) coefficient multiplications.

However, if we evaluate both polynomials, we can simply multiply the corresponding pairs of values to get the values of polynomial \( AB \).

Process:
- Evaluate polynomials \( A \) and \( B \) at enough points.
- Pairwise multiplications of resulting values.
- Interpolation of resulting values.
Multiplication of Polynomials (2)

This can be faster than $\Theta(n^2)$ if a fast way can be found to do evaluation/interpolation of $2n - 1$ points (normally this takes $\Theta(n^2)$ time).

Note that evaluating a polynomial at 0 is easy, and that if we evaluate at 1 and -1, we can share a lot of the work between the two evaluations.

Can we find enough such points to make the process cheap?

An Example

Polynomial A: $x^2 + 1$.
Polynomial B: $2x^2 - x + 1$.
Polynomial AB: $2x^4 - x^3 + 3x^2 - x + 1$.

Notice:

$$AB(-1) = (2)(4) = 8$$
$$AB(0) = (1)(1) = 1$$
$$AB(1) = (2)(2) = 4$$

But: We need 5 points to nail down Polynomial AB. And, we also need to interpolate the 5 values to get the coefficients back.

Nth Root of Unity

The key to fast polynomial multiplication is finding the right points to use for evaluation/interpolation to make the process efficient.

Complex number $\omega$ is a primitive nth root of unity if

1. $\omega^n = 1$ and
2. $\omega^k \neq 1$ for $0 < k < n$.

$\omega^0, \omega^1, \ldots, \omega^{n-1}$ are the nth roots of unity.

Example:

1. For $n = 4$, $\omega = i$ or $\omega = -i$.
2. $n = 4, \omega = i$.
3. $n = 8, \omega = \sqrt[8]{i}$.
Discrete Fourier Transform

Define an \( n \times n \) matrix \( V(\omega) \) with row \( i \) and column \( j \) as:
\[
V(\omega) = \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & i & -1 & -i \\
1 & -1 & 1 & -1 \\
1 & -i & 1 & i
\end{bmatrix}
\]

Example: \( n = 4, \omega = i \):
\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & i & -1 & -i \\
1 & -1 & 1 & -1 \\
1 & -i & 1 & i
\end{bmatrix}
\]

Let \( \mathbf{a} = [a_0, a_1, ..., a_{n-1}]^T \) be a vector.
The Discrete Fourier Transform (DFT) of \( \mathbf{a} \) is:
\[
F_\omega = V(\omega) \mathbf{a} = \mathbf{v}.
\]
This is equivalent to interpolating the polynomial at the \( n \)th roots of unity.

Array example

For \( n = 8, \omega = \sqrt{i} \), \( V(\omega) = \)
\[
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & \sqrt{7} & i & \sqrt{7} & -1 & -\sqrt{7} & -i & -\sqrt{7} \\
1 & i & -1 & 1 & -i & 1 & -i & 1 \\
1 & i & \sqrt{7} & -i & \sqrt{7} & 1 & -\sqrt{7} & i \\
1 & -1 & 1 & 1 & -1 & 1 & 1 & -1 \\
1 & -\sqrt{7} & i & -i & \sqrt{7} & -1 & i & \sqrt{7} \\
1 & -i & -1 & -i & 1 & 1 & i & 1 \\
1 & -i & \sqrt{7} & i & -\sqrt{7} & -1 & i & \sqrt{7}
\end{bmatrix}
\]

Inverse Fourier Transform

The inverse Fourier Transform to recover \( \mathbf{a} \) from \( \mathbf{v} \) is:
\[
F^{-1}_\omega = \mathbf{a} = (V(\omega))^{-1} \cdot \mathbf{v}.
\]
\[
(V(\omega))^{-1} = \frac{1}{n} V(1/\omega).
\]
This is equivalent to interpolating the polynomial at the \( n \)th roots of unity.

An efficient divide and conquer algorithm can perform both the DFT and its inverse in \( \Theta(n \log n) \) time.

Fast Polynomial Multiplication

Polynomial multiplication of \( A \) and \( B \):
- Represent an \( n - 1 \) degree polynomial as \( 2n - 1 \) coefficients:
  \[
  [a_0, a_1, ..., a_{n-1}, 0, ..., 0]
  \]
- Perform DFT on representations for \( A \) and \( B \).
- Pairwise multiply results to get \( 2n - 1 \) values.
- Perform inverse DFT on result to get \( 2n - 1 \) degree polynomial \( AB \).
Parallel Algorithms

- **Running time**: $T(n, p)$ where $n$ is the problem size, $p$ is number of processors.
- **Speedup**: $S(p) = T(n, 1)/T(n, p)$.
  - A comparison of the time for a (good) sequential algorithm vs. the parallel algorithm in question.
- **Problem**: Best sequential algorithm might not be the same as the best algorithm for $p$ processors, which might not be the best for $\infty$ processors.
- **Efficiency**: $E(n, p) = S(p)/p = T(n, 1)/(pT(n, p))$.
- **Ratio of the time taken for 1 processor vs. the total time required for $p$ processors**:
  - Measure of how much the $p$ processors are used (not wasted).
  - Optimal efficiency = 1 = speedup by factor of $p$.

Parallel Algorithm Design

- Would need a new algorithm for every $p$!

Approach (2): Pick best algorithm for $p = \infty$, then convert to run on $p$ processors.

Hopefully, if $T(n, p) = X$, then $T(n, p/k) \approx kX$ for $k > 1$.

Using one processor to emulate $k$ processors is called the parallelism folding principle.

Parallel Algorithm Design (2)

Some algorithms are only good for a large number of processors.

$$T(n, 1) = n$$

$$T(n, n) = \log n$$

$$\frac{T(1)}{T(n, n)} = n/\log n$$

$$\frac{S(n)}{T(n, n)} = 1/\log n$$

For $p = 256$, $n = 1024$.

$$T(1024, 256) = 4 \log 1024 = 40$$

For $p = 16$, running time $= (1024/16) + \log 1024 = 640$.

Speedup $< 2$, efficiency $= 1024/(16 \times 640) = 1/10$.

Good in terms of speedup.

1024/256, assuming one processor emulates 4 in 4 times the time.

$$E(1024, 256) = 1024/(256 \times 4) = 1/10.$$  

But note that efficiency goes down as the problem size grows.
Amdahl’s Law

Think of an algorithm as having a parallelizable section and a serial section.

Example: 100 operations.
- 80 can be done in parallel, 20 must be done in sequence.

Then, the best speedup possible leaves the 20 in sequence, or a speedup of $100/20 = 5$.

Amdahl’s law:

$$\text{Speedup} = \frac{(S + P) / (S + P / N)}{S + P} \leq \frac{1}{S},$$

for $S =$ serial fraction, $P =$ parallel fraction, $S + P = 1$.

Amdahl’s Law Revisited

However, this version of Amdahl’s law applies to a fixed problem size.

What happens as the problem size grows?

Hopefully, $S = f(n)$ with $S$ shrinking as $n$ grows.

Instead of fixing problem size, fix execution time for increasing number $N$ processors (and thus, increasing problem size).

Scaled Speedup:

$$\text{Scaled Speedup} = \frac{(S + P \times N) / (S + P)}{S + P \times N} = S + (1 - S) \times N = N + (1 - N) \times S.$$