Part 4

Induction, Correctness Proofs, Recurrence Relation, Multiplication of Matrices, Introduction to Number Theory
Table of Contents

7.1 Mathematical Induction (Reading Assignment)
  • 7.1.1 Principle of Mathematical Induction
  • 7.1.2 Variations of the Principle of Mathematical Induction

7.2 Correctness Proofs (Reading Assignment)
  • 7.2.1 Loop Invariants
  • 7.2.2 Induction on Input Size
  • 7.2.3 Correctness Proofs for Parallel Algorithms: The 0/1 Sorting Lemma

7.3 Mathematical Properties of Binary Trees (Reading Assignment)
  • 7.3.1 Lower Bounds for the Depth of Binary Trees
  • 7.3.2 Internal and Leaf Path Lengths of Binary Trees

7.4 Solving Recurrence Relations for Complexity
  • 7.4.1 Solving Linear Recurrence Relations Relating the \( n \)th Term to \( (n-1) \)st Term
  • 7.4.2 Solving Linear Recurrence Relations Relating the \( n \)th Term to \( (n/b) \)th Term
  • 7.4.3 Interpolating Asymptotic Behavior
Often we would like to establish the validity of a proposition or formula concerning the set of positive integers such as:

\[ 1^2 + 2^2 + \ldots + n^2 = \frac{n (n + 1) (2n + 1)}{6} \]

This formula can be easily verified for any particular small \( n \) by direct computing. The question is how do we verify that the formula is true for all positive integers \( n \)?
7.1.1 Principle of Mathematical Induction

- **Basis step**: P(1) is true
- **Induction step**: if P(k) is true for any given k then P(k+1) must also be true

Now let's prove previous equation:

**Basis step**: \(1^2 = \frac{1(1 + 1)(2 + 1)}{6} \) ... is true

**Induction step**: \(1^2 + 2^2 + \ldots + k^2 + (k + 1)^2 = (1^2 + 2^2 + \ldots + k^2) + (k + 1)^2\)

\[= \frac{k(k + 1)(2k + 1)}{6} + (k + 1)^2\]

\[= \frac{(k + 1)(k + 2)(2k + 3)}{6} \] ... and therefore P(k+1) is true
7.1.2 Variations of Principle of Mathematical Induction

3 variations of mathematical induction are frequently encountered in the analysis of algorithms

1. The sequence of propositions starts with an index different from 1, such as 0. Then the basis step starts with this initial index. The induction step remains the same, and the two steps together establish the truth of the proposition.

2. The propositions are only finite in number, P(1), ..., P(l). Then the induction step is modified to require that k < l. The conclusion then drawn is that P(1), ..., P(l) are all true if the basis and induction steps are valid.

3. Induction Step (strong form): For any positive integer k, if p(j) is true for all positive integers j <= k, then P(k+1) must also be true.
A complete analysis of an algorithm should not only describe its complexity, but should also include a verification of its correctness.

A technique for verifying correctness is to identify certain loop invariants associated with loops in the pseudocode for an algorithm.

A loop invariant is a statement about the value of a variable or condition after each iteration of the loop.
7.2.1 Loop Invariants

To illustrate the idea of loop invariants, consider the algorithm EgyptianPowers given below;

```
Function EgyptianPowers(x,n)
Input : x (a real number), n (a power of two, n = 2^m)
Output : x^n
j := 1
Product := x
while j<n do
    Product := Product * Product
    j := j + j
end while
Return (Product)
End EgyptianPowers
```

- **Basis step**: After k=0 passes, j has the value $1 = 2^0$ and Product has the value $x = x^{2^0}$. Thus, the two loop invariants have the stated values.

- **Induction step**: Assume that after k passes, j has the value $2^k$, and Product has the value $x^j$, for $k < m$. Then $j = 2^k < 2^m = n$, and another pass through the while loop is performed. During each pass of the while loop in Egyptian Powers, j is doubled, and Product is squared. Thus after k+1 passes, j has the value $2^k + 2^k = 2^{k+1}$ and Product has the value $x^{2^k} x^{2^k} = x^{2^{k+1}}$, completing the induction step.
7.2.2 Induction on Input Size

To illustrate a correctness proof that uses induction on the input size, we consider a variation of the algorithm BinarySearch.

```plaintext
Function BinarySearch2(L[low,high],X) recursive
Input : L[low,high], X (search item)
Output : Returns the index of an occurrence of X in the list, or 0 if X is not in the list.
        if low = high then
            if X = L[low] then
                return(low)
            else
                return(0)
        endif
        else
            mid := (low+high+1) / 2
            if X < L[mid] then
                return(BinarySearch2(L[low:mid-1], X))
            else
                return(BinarySearch2(L[mid:high], X))
            endif
        endif
End BinarySearch2
```
7.2.2 Induction on Input Size

Correctness proof of BinarySearch2

- **Basis step**: If \( k = 1 \), then \( low = high \), so that the sublist \( L[low:high] \) consists of the single element \( L[low] \), and the algorithm correctly compares \( X \) to \( L[low] \)

- **Induction step**: Assume BinarySearch2 works correctly on all sublists \( L[low:high] \) of size \( k = high - low + 1 \), for \( 1 \leq k < n \). Now consider a sublist \( L[low:high] \) of size \( k+1 \).

Since \( k+1 > 1 \), \( low \neq high \), so that \( mid \) is the assigned the value \( (low+high+1) / 2 \) and \( X \) is compared to \( L[mid] \). If \( X < L[mid] \), then BinarySearch2 invokes itself recursively. We consider the case where \( X < L[mid] \). Since \( L \) is ordered, this means that \( X \) does not occur in the sublist \( L[mid:high] \). Thus, if \( X \) occurs at all in \( L[low:high] \), then it must occur in the sublist \( L[low:mid-1] \). Since \( mid - 1 < high \), the size of the latter sublist is at most \( k \).

Thus, BinarySearch2 works correctly on \( L[low:high] \)
Many simply described parallel sorting algorithms, such as OddEvenMergeSort1DMesh are surprisingly difficult to prove for a general input list. However, the powerful 0/1 Sorting Lemma allows us to restrict attention to 0/1 lists when establishing correctness.

**0/1 Sorting Lemma**: Suppose an oblivious comparison-exchange sorting algorithm works correctly on all lists of size \( n \) consisting of only 0s and 1s. Then it works correctly on all lists of size \( n \).
7.3 Mathematical Properties of Binary Trees

The analysis of the complexity of many problems and algorithms discussed in this text depends on various mathematical properties of the binary trees. In this section we establish a number of these properties relating to depth, internal path length and leaf path length.
7.3.1 Lower Bounds for the Depth of Binary Trees

Given any binary tree $T$, we will use the notation:

- $N = N(T) \rightarrow$ number of nodes
- $I = I(T) \rightarrow$ number of internal nodes
- $L = L(T) \rightarrow$ number of leaf nodes
- $D = D(T) \rightarrow$ depth of tree
7.3.1 Lower Bounds for the Depth of Binary Trees - Propositions

**Proposition 1:** Suppose $T$ is any binary tree having $N$ nodes. Then $T$ has depth at least:

$$\lfloor \log_2 N \rfloor$$

**Proposition 2:** Suppose $T$ is any binary tree. Then the number of leaf nodes is one greater than the number of internal nodes of $T$; that is,

$$I(T) = L(T) - 1 \quad \text{(equivalently we have } N(T) = 2L(T) - 1)$$

**Proposition 3:** Suppose $T$ is any binary tree. Then the depth of $T$ satisfies,

$$D(T) \geq \lceil \log_2 L(T) \rceil$$

**Proposition 4:** Suppose $T$ is any binary tree. Then $T$ is full at the second-deepest level if, and only if, all the leaf nodes are contained in two levels ($D-1$ and $D$).

**Proposition 5:** If a binary tree is full at the second-deepest level, then the depth $D(T)$ and the number of leaf nodes $L(T)$ are related by

$$D(T) = \lceil \log_2 L(T) \rceil$$
7.3.2 Internal and Leaf Path Lengths of Binary Trees

- The *internal path length* $\text{IPL}(T)$ of a binary tree $T$ is defined as the sum of the lengths of the paths from the root to the internal nodes as the internal nodes vary over the entire tree.

- The *leaf path length* $\text{LPL}(T)$ of a binary tree $T$ is defined as the sum of the lengths of the paths from the root to the leaf nodes.
7.3.2 Internal and Leaf Path Lengths of Binary Trees - Propositions

- **Proposition 6**: Given any 2-tree $T$ having $I$ internal nodes.
  \[ \text{IPL}(T) = \text{LPL}(T) - 2I \]

- **Proposition 7**: Given any 2-tree $T$ with $L$ leaf nodes,
  \[ \text{LPL}(T) \geq L \lfloor \log_2 L \rfloor + 2(L - 2 \lfloor \log_2 L \rfloor) \]

- **Corollary 1**: If $T$ is any binary tree having $L$ leaf nodes, then
  \[ \text{LPL}(T) \geq \lceil L \log_2 L \rceil \]
  Further, if $T$ is a full binary tree, then inequality is an equality.
A linear recurrence relation is one of the form:

\[ t(n) = c_1 t(n-1) + c_2 t(n-2) + \ldots + c_k t(n-k) + f(n), \]

initial condition; \( t(0) = d_0, \ldots, t(k-1) = d_{k-1} \),

for constants \( c_1, \ldots, c_k, d_0, \ldots, d_{k-1} \) and

some fixed function \( f(n) \)
7.4.1 Solving Linear Recurrence Relations
Relating the $n$th Term to (n-1)st Term

**General Recurrence Relation ;**

\[ t(n) = a t(n-1) + f(n) \]
\[ t(n) = a(at(n-2) + f(n-1)) + f(n) = a^2 t(n-2) + af(n-1) + f(n) \]
\[ t(n) = a^2 (at(n-3) + f(n-2)) + af(n-1) + f(n) \]
\[ t(n) = a^3 t(n-3) + a^2 f(n-2) + af(n-1) + f(n) \]

\[ \vdots \]

\[ t(n) = a^{n-1} b + \sum_{i=2}^{n} a^{n-i} f(i) \]
7.4.2 Solving Linear Recurrence Relations
Relating the \( n \)th Term to \( (n/b) \)th Term

- **General Recurrence Relation**;
  \[
t(n) = at(n/b) + f(n)
  \]
  \[
t(n) = a(at(n/b^2) + f(n/b)) + f(n) = a^2t(n/b^2) + af(n/b) + f(n)
  \]
  \[
  .
  \]
  \[
t(n) = a^kc_1 + a^{k-1}f(n/b^{k-1}) + a^{k-2}f(n/b^{k-2}) + \ldots + af(n/b) + f(n)
  \]
  \[
t(n) = a^kc_1 + \sum_{i=0}^{k-1} a^i f(n/b^i), \text{ where } k = \log_b n
  \]

- **Special case** \( f(n) = cn; \)
  \[
t(n) = \begin{cases} 
  c_1n + cn \log_b n & a=b \\
  c_1n^{\log_b a} + c \left( n^{\log_b a} - n \right) / (a/b) - 1 & a \neq b
  \end{cases}
  \]
7.4.3 Interpolating Asymptotic Behavior

Proposition 1: If \( g(n) \in \mathcal{F} \) has the property that \( g(cn) \in \Theta(g(n)) \), then \( f(cn) \in \Theta(g(n)) = \Theta(f(n)) \) for all \( f(n) \in \Theta(g(n)) \).

Proposition 2: Suppose \( g(n) \in \mathcal{F} \) is eventually nondecreasing and \( \Theta \)-invariant under scaling. Further, suppose \( f(n) \in \mathcal{F} \) is eventually nondecreasing and \( f(b^n) \in \mathcal{X}(g(b^n)) \), \( n \in \mathcal{N} \), where \( \mathcal{X} \) is one of the classes \( \Theta, \Omega, O \), and \( b > 1 \). Then \( f(n) \in \mathcal{X}(g(n)) \).

Proposition 3: If \( t(n) \) is eventually nondecreasing and satisfies the recurrence relation \( t(n) = at(n/b) + c \), with initial condition \( t(1) = c_1 \), where \( a, b, c, c_1 \) are positive constants, \( b > 1 \), then

\[
t(n) = \begin{cases} 
\Theta(n^{\log_b a}), & a \neq 1 \\
\Theta(\log n), & a = 1 
\end{cases}
\]

Proposition 4: If \( t(n) \) is eventually nondecreasing and satisfies the recurrence relation \( t(n) = at(n/b) + cn \), with initial condition \( t(1) = c_1 \), where \( a, b, c, c_1 \) are positive constants, \( b > 1 \), then

\[
t(n) = \begin{cases} 
\Theta(n), & a < b \\
\Theta(n \log n), & a = b \\
\Theta(n^{\log_b a}), & a > b 
\end{cases}
\]
Consider the recurrence relation in the form of

\[ C_0 = 1 \quad \text{and} \quad C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i} \quad \text{for} \quad n \geq 0; \]

\[ C_n = \text{???} \]
7.5 Generating Functions

7.5.1 The Generating Function Paradigm for Solving Recurrence Relations

7.5.2 Obtaining a Formula for the Number of Binary Trees on N Nodes

7.6 Polynomial Interpolation

7.6.1 Lagrange Interpolation
7.6.2 Newtonian Interpolation

\[ C_0 = 1 \text{ and } C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i} \text{ for } n \geq 0; \quad C_n = ??? \]
Generating Functions

Multiple reference recurrence relations like \( \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \), \( \text{fib}(0) = 0 \), \( \text{fib}(1) = 1 \), determining fibonacci numbers 0, 1, 1, 2, 3, 5, 8, 13, ..., or \( t(n) = 2t(n-1) + 1 \), \( t(0) = 0 \) for Towers of Hanoi problem, can be solved by using Generating functions.

For a sequence of numbers \( a_0, a_1, a_2, \ldots a_n, \ldots \) the Generating Function is the formal power series:

\[
g(x) = \sum a_n x^n = a_0 + a_1 x + a_2 x^2 + \ldots a_n x^n + \ldots
\]
Generating Functions

\[ g(x) = \sum_{n=0}^{\infty} a_n x^n = a_0 + a_1 x + a_2 x^2 + \ldots a_n x^n \]

e.g.: Generating function for fibonacci sequence is the power series

\[ x + x^2 + 2x^3 + 3x^4 + 5x^5 + 8x^6 + 13x^7 + 21x^8 + \ldots \]

The generating function for a sequence \((a_n)\) is simply used to “encode” the sequence in a form that is amenable to algebraic manipulation. For fibonacci sequence we will get the explicit formula:

\[ \text{fib}(n) = \left( \frac{1}{\sqrt{5}} \right) \left[ \left( \frac{1 + \sqrt{5}}{2} \right)^n - \left( \frac{1 - \sqrt{5}}{2} \right)^n \right] \]
How to Solve a Given Recurrence Relation

Suppose we have \( t(n) = c_1 t(n-1) + c_2 t(n-2) \), where \( t(0) = b_0, \ t(1) = b_1 \)

i ) Form \( g(x) = \sum_{n=0}^{\infty} t(n)x^n \)

ii ) Convert \( g(x) \) into a closed form formula by doing the following procedure

\[
g(x) = t(0) + t(1)x + t(2)x^2 + t(3)x^3 + \ldots \ldots
\]

\[
- c_1xg(x) = -c_1t(0)x - c_1t(1)x^2 - c_1t(2)x^3 + \ldots \ldots
\]

\[
+ \quad - c_2x^2g(x) = -c_2t(0)x^2 - c_2t(1)x^3 + \ldots \ldots
\]

\[
g(x)[1-c_1x+c_2x^2] = t(0) + x[t(1)-c_1t(0)] + x^2[t(2)-c_1t(1)-c_2t(0)] + \ldots \ldots
\]

By definition \( t(2) = c_1t(1) + c_2t(0) \) and so on.

\[
g(x) = (b_0 + x[b_1 - c_1 b_0]) / (1-c_1x+c_2x^2)
\]

Then solve by finding roots (by method of partial fractions)

iii ) Expand \( g(x) \) into a power series with explicit coefficients

\[
g(x) = \sum_{n=0}^{\infty} d(n)x^n
\]

Here \( d(n) \) corresponds to \( t(n) \) in step (i)
Example: Solve Towers of Hanoi

\[ t(n) = 2t(n-1) + 1, \quad t(0) = 0 \]

\[ g(x) = \sum_{n=0}^{\infty} t(n)x^n = 0 + x + 3x^2 + 7x^3 + 15x^4 + \ldots + a_n x^n + \ldots \]

\[ -2xg(x) = -2x^2 - 6x^3 - 14x^4 + \ldots \]

\[ g(x)[1-2x] = x[1 + x + x^2 + x^3 + x^4 + \ldots + x^{n-1} + \ldots] \]

Since \( 1 + x + x^2 + x^3 + x^4 + \ldots + x^{n-1} + \ldots = 1/(1-x) \)

\[ g(x) = \frac{x}{(1-2x)(1-x)} = \frac{1}{1-2x} - \frac{1}{1-x} \]

\[ g(x) = \sum_{n=0}^{\infty} [(2x)^n - x^n] = \sum_{n=0}^{\infty} (2^n - 1) x^n \]

And \( 2^n - 1 \) corresponds to \( t(n) \)

For \( t(n) = 2t(n-1) + 1, \quad t(0) = 0 \)
we have \( t(n) = 2^n - 1, \quad n \geq 0 \)
Formula for General Linear Recurrences

For any linear recurrence relation of type
\[ t(n) = c_1 t(n-1) + c_2 t(n-2) + \ldots + c_k t(n-k), \]
initial conditions \( t(0) = b_0, \ldots, t(k) = b_k \)

We can derive a formula using the method described above.

\[ g(x) = b_0 + (b_1 - c_1 b_0) x + \ldots + (b_{k-1} - c_1 b_{k-1} - \ldots - c_k b_0) x^{k-1} \]
\[ \frac{1 - c_1 x - \ldots - c_k x^k}{1 - c_1 - \ldots - c_k x} \]
Obtaining a Formula for the Number of Binary Trees on N Nodes

The number $b_n$ of binary trees on $n$ nodes equals the $n$th Catalan number:

$$b_n = \frac{1}{(n+1)} \binom{2n}{n}$$

$b_n$ is given by the following quadratic recurrence relation:

$$b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1} = b_0 b_{n-1} + b_1 b_{n-2} + b_2 b_{n-3} + \ldots + b_{n-1} b_0$$

Let $g(x)$ be the generating function for the number $b_n$ of binary trees on $n$ nodes, that is:

$$g(x) = \sum_{n=0}^{\infty} b_n x^n = b_0 + b_1 x + b_2 x^2 + \ldots + b_n x^n + \ldots$$

$$xg(x) = \sum_{n=0}^{\infty} b_n x^n = b_0 x + b_1 x^2 + \ldots + b_{n-1} x^n + \ldots$$

$$g(x)(xg(x)) = (g(x))^2 x = \sum_{n=1}^{\infty} \left( \sum_{i=0}^{n-1} b_i b_{n-i-1} \right) x^n =$$

$$= (b_0 b_0) x + (b_0 b_1 + b_1 b_0) x^2 + \ldots + (b_0 b_{n-1} + \ldots + b_{n-1} b_0) x^n + \ldots$$

$$(g(x))^2 x = b_1 x + b_2 x^2 + b_3 x^3 + \ldots + b_n x^n + \ldots = g(x) - 1$$

Hence

$$(g(x))^2 x - g(x) - 1 = 0$$
Using the quadratic formula we have:

\[ g(x) = \frac{1}{2x} \left( 1 - \frac{(1-4x)^{1/2}}{2} \right) \]

Since \( b_0 = 1 \), it follows that:

\[ g(x) = \frac{1}{(1-4x)^{1/2}} \]

Using Binomial theorem \( (1+x)^n = \sum_{i=0}^{\infty} \binom{n}{i} x^i \) we have:

\[ g(x) = -\sum_{n=1}^{\infty} \binom{n}{1/2} (-4x)^n 2x \]

\[ = \sum_{n=0}^{\infty} \binom{1/2}{n+1} (-2)^n (2)^{n+1} x^n \]

\[ = \sum_{n=0}^{\infty} \left[ \frac{(1/2)(-1/2) (-3/2) (-5/2) \ldots (-2n-1/2)(-2)^{n+1}}{(n+1)!} \right] x^n \]

\[ = \sum_{n=0}^{\infty} \left[ \frac{(1)(3)(5)\ldots(2n-1)2^n}{(n+1)!} \right] x^n \]

\[ b_n = \frac{(1)(3)(5)\ldots(2n-1)2^n}{(n+1)!} \]

and \( (1)(3)(5)\ldots(2n-1)2^n = (n+1)(n+2)\ldots(2n-1)(2n) \)

so, \( b_n = \frac{(n+1)(n+2)\ldots(2n-1)(2n)}{(n+1)!} \) \[ = \frac{1}{(n+1)} \left( \begin{array}{c} 2n \end{array} \right) \]
Polynomial Interpolation

We are asking for a unique (interpolating) polynomial of degree at most \( n-1 \) whose graph passes through \( n \) points \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\), e.g. For points \((-1,1), (0,0), (1,0), (3,2)\), we need to find a polynomial of degree at most 3 that interpolates these points.

Simply we need to solve

\[
P(x) = a_3x^3 + a_2x^2 + a_1x + a_0
\]

for coefficients \(a_0, a_1, a_2, a_3\), by using \(y_i = P(x_i), i=1,2,3,4\)

\[
\begin{align*}
-a_3 + a_2 - a_1 + a_0 &= 1 \\
 a_0 &= 0 \\
a_3 + a_2 + a_1 + a_0 &= 0 \\
27a_3 + 9a_2 + 3a_1 + a_0 &= 2
\end{align*}
\]

Since \(a_3 = -1/24, a_2 = \frac{1}{2}, a_1 = -11/24, a_0 = 0\)

the graph of \(P(x)\) looks like:

\[
y = P(x) = -\frac{1}{24}x^3 + \frac{1}{2}x^2 - \frac{11}{24}x
\]
In general \( P(x) = a_{n-1}x^{n-1} + \ldots + a_1x + a_0 \) for \( n \) points \( \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \) can be obtained by solving the following linear system of \( n \) equations in \( n \) unknowns \( a_{n-1}, \ldots, a_1, a_0 \), by using \( y_i = P(x_i), i=1,2,3,\ldots,n \):

\[
\begin{align*}
a_{n-1}x_1^{n-1} + \ldots + a_1x_1 + a_0 &= y_1 \\
a_{n-1}x_2^{n-1} + \ldots + a_1x_2 + a_0 &= y_2 \\
\vdots \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \vdots \\
a_{n-1}x_n^{n-1} + \ldots + a_1x_n + a_0 &= y_n
\end{align*}
\]

The determinant of this system is known as Vandermonde determinant and it is nonzero, and this fact proves that there is a unique solution for this system.
Lagrange Interpolation

The following explicit formula was given by Lagrange for interpolating $P(x)$

$$P(x) = a_{n-1}x^{n-1} + ... + a_1x + a_0$$

for $n$ points $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$

$$P(x) = \sum_{1 \leq i \leq n} \left( \prod_{1 \leq j \leq n \atop j \neq i} \frac{x-x_j}{x_i-x_j} \right)y_i$$

For $n=3$

$$P(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)}y_1 + \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)}y_2 + \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)}y_3$$

A naive algorithm that simply translates this formula into pseudocode has $\Theta(n^3)$ complexity. Moreover this algorithm can be modified to obtain $\Theta(n^2)$ complexity. HWLA????
Newtonian Interpolation

This method is based on a recursive formula, which expresses the interpolating polynomial \( P_k(x) \) for the \( k \) points \((x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\) in terms of the interpolating polynomial \( P_{k-1}(x) \) for the \( k-1 \) points \((x_1, y_1), (x_2, y_2), \ldots, (x_{k-1}, y_{k-1})\). Let \( Q_k(x) = (x-x_1)(x-x_2)\ldots(x-x_k) \).

The interpolating polynomial \( P_k(x) \) satisfies the recurrence relation:

\[
P_k(x) = (y_k - P_{k-1}(x_k)) \frac{Q_k(x)}{Q_{k-1}(x_k)} + P_{k-1}(x), \quad 2 \leq k \leq n, \quad P_1(x) = y_1
\]
Example for Newtonian Interpolation

Using our previous set of points

\[((-1,1),(0,0),(1,0),(3,2)) = ((x_1,y_1), (x_2,y_2), (x_3,y_3), (x_4,y_4))\],

a sample calculation is as follows;

**Step 1:**

\[P_1(x) = y_1 = 1\]

**Step 2:**

\[P_2(x) = (y_2 - P_1(x_2)) + P_1(x) = -x \frac{Q_1(x)}{Q_1(x_2)}\]

**Step 3:**

\[Q_2(x) = (x-x_2)Q_1(x) = (x-0)(x-1) = x^2+x\]

\[P_3(x) = (y_3 - P_2(x_3)) Q_2(x) + P_2(x) = \frac{(x^2-x)/2}{Q_2(x_3)} + P_2(x)\]

**Step 4:**

\[Q_3(x) = (x-x_3)Q_2(x) = (x-1)(x^2+x) = x^3-x\]

\[P_4(x) = (y_4 - P_3(x_4)) Q_3(x) + P_3(x) = \frac{-1/24x^3+(1/2)}{Q_3(x_4)}\]

And this result agrees with our previous result found by linear algebra.
Algorithm for Newtonian Interpolation

procedure \textit{NewtonInterp}(X[1:n], Y[1:n], n, P(x))

\textbf{Input:} $X[1:n], Y[1:n]$ (arrays of real numbers), $n$ (a positive integer)

\textbf{Output:} $P(x) = a_{n-1}x^{n-1} + \ldots + a_1x + a_0$ (interpolating polynomial)

\begin{align*}
P(x) &:= Y[1] \{ P(x) \text{ is initialized to be the constant polynomial } y_1 \} \\
Q(x) &:= x - X[1] \{ Q(x) \text{ is initialized to be the linear polynomial } x - x_1 \},
\end{align*}

for $i := 2$ to $n$ do

PEval := HornerEval$(P(x), X[i])$

QEval := HornerEval$(Q(x), X[i])$

if $i < n$ then

$Q(x) := \text{PolyMult}(Q(x), x - X[i])$

endif

endfor

end NewtonInterp.

\textit{HornerEval} function evaluates a polynomial of the form

$a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$, in following manner

\[ (((a_4x + a_3)x + a_2)x + a_1)x + a_0 \]
Chapter 13

Multiplication of Matrices
Solving n unknown equations,
matrix inversion,
Gaussian elimination requires $2/3 \, n^3$

$AX=b$, $A;b$, reduced raw echelon form, $X;c$

\[
\begin{bmatrix}
2 & 1 & -1 & 8 \\
-3 & -1 & 2 & -11 \\
-2 & 1 & 2 & -3
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 2 \\
0 & 1 & 0 & 3 \\
0 & 0 & 1 & -1
\end{bmatrix}
\]
Multiplication of Matrices

\[ A = (a_{ij}) \text{ and } B = (b_{ij}) \quad 1 \leq i,j \leq n \]

where \( A \) and \( B \) are \( n \times n \) matrices

\[ AB = C \quad \text{where} \quad C = (c_{ij}) \quad \text{and} \]

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

Classic method of multiplying 2x2 matrices: 8 multiplication, 4 additions

\[
\begin{pmatrix}
a_{11} & a_{12} \\
a_{21} & a_{22}
\end{pmatrix}
\begin{pmatrix}
b_{11} & b_{12} \\
b_{21} & b_{22}
\end{pmatrix}
= 
\begin{pmatrix}
a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\
a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22}
\end{pmatrix}
\]
In 1969, Strassen discovered a way to do same product \( AB \) using only 7 multiplications and 18 additions:

- \( m_1 = (a_{11} + a_{22})(b_{11} + b_{22}) \)
- \( m_2 = (a_{21} + a_{22})b_{11} \)
- \( m_3 = a_{11}(b_{12} - b_{22}) \)
- \( m_4 = a_{22}(b_{21} - b_{11}) \)
- \( m_5 = (a_{11} + a_{12})b_{22} \)
- \( m_6 = (a_{21} - a_{11})(b_{11} + b_{12}) \)
- \( m_7 = (a_{12} - a_{22})(b_{21} + b_{22}) \)
Multiplication of Matrices

AB is given by

\[ AB = \begin{bmatrix}
  m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\
  m_2 + m_4 & m_1 + m_3 - m_2 + m_6
\end{bmatrix} \]
Multiplication of Matrices

Consider 2 nxn matrices, where n=2^k

Partition A and B into 4 (n/2)x(n/2) submatrices;

\[
\begin{align*}
A &= \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix} \\
B &= \begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\end{align*}
\]

\[
AB = \begin{bmatrix}
A_{11}B_{11} + A_{12}B_{21} \\
A_{21}B_{11} + A_{22}B_{21}
\end{bmatrix} \begin{bmatrix}
A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{12} + A_{22}B_{22}
\end{bmatrix}
\]
Multiplication of Matrices

- \( M_1 = (A_{11} + A_{22})(B_{11} + B_{22}) \)
- \( M_2 = (A_{21} + A_{22})B_{11} \)
- \( M_3 = A_{11}(B_{12} - B_{22}) \)
- \( M_4 = A_{22}(B_{21} - B_{11}) \)
- \( M_5 = (A_{11} + A_{12})B_{22} \)
- \( M_6 = (A_{21} - A_{11})(B_{11} + B_{12}) \)
- \( M_7 = (A_{12} - A_{22})(B_{21} + B_{22}) \)

\[
AB = \begin{bmatrix}
M_1 + M_4 & M_3 + M_5 \\
M_1 + M_4 & M_1 + M_3 - M_2 + M_6
\end{bmatrix}
\]
Multiplication of Matrices

The complexity of Strassen’s algorithm
Satisfies the recurrence relation,
\[ T(n) = 7 \ T\left(\frac{n}{2}\right), \quad n > 1, \ T(1) = 1 \]
\[ T(n) \in \theta \left( n^{\log_2 7} \right) \quad \ast n^{2.807} \]
Multiplication of Matrices

In 1980, Winograd discovered an algorithm with 15 additions and still 7 multiplications—check!

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

\[ AB = \begin{bmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{bmatrix} \]
Multiplication of Matrices

- As of 2008, Coppersmith-Winograd algorithm for nxn matrix multiplication is the fastest known algorithm with $n^{2.376...}$ (1987)
- Restrictions due to huge constants factors hidden in the big O notation...
  Search for algorithm details in the net....
- $n^{(2+\text{epsilon})}$ is possible... But how???
Numbers

Mathematics is the queen of the sciences and number theory is the queen of mathematics." — Gauss

It is also impossible to cover all aspects of number theory in this class...

Many public-key cryptography schemes use number theory e.g. RSA Cryptosystem, Elliptic curve cryptography.
The prime number theorem (PNT) describes the asymptotic distribution of the prime numbers.

The prime number theorem gives a rough description of how the primes are distributed.
Roughly speaking, the prime number theorem states that if a random number nearby some large number \( N \) is selected, the chance of it being prime is about \( 1 / \ln(N) \), where \( \ln(N) \) denotes the natural logarithm of \( N \).

For example, near \( N = 10,000 \), about one in nine numbers is prime, whereas near \( N = 1,000,000,000 \), only one in every 21 numbers is prime. In other words, the average gap between prime numbers near \( N \) is roughly \( \ln(N) \).
As a consequence of the prime number theorem, one gets an asymptotic expression for the $n$th prime number, denoted by $p_n = N(\ln N)$.
An even better approximation is
\[ p_n = N(\ln N) + N\ln(\ln N) + \ldots \]
Yet we quest for newer approximations.
In mathematics, a square-free integer is one divisible by no perfect square, except 1.

For example, 10 is square-free but 18 is not, as it is divisible by $9 = 3^2$.

The first few square-free numbers are 1, 2, 3, 5, 6, 7, 10, 11, 13, 14, 15, 17, 19, 21, 22, 23, 26, 29, 30, 31, 33, 34, 35, 37, 38, 39,...
A **composite number** is a positive integer which has a positive divisor other than one or itself. In other words, if \( n > 0 \) is an integer and there are integers \( 1 < a, b < n \) such that \( n = a \times b \) then \( n \) is composite.

By definition, every integer greater than one is either a prime number or a composite number.
The number one is a unit - it is neither prime nor composite. For example, the integer 14 is a composite number because it can be factored as $2 \times 7$. Likewise, the integers 2 and 3 are not composite numbers because each of them can only be divided by one and itself. The first few composite numbers (sequence A002808 in OEIS) are

$4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25, 26, 27, 28, 30, 32, 33, 34, 35, 36, 38,$
Carmichael numbers are very important because they pass the Fermat primality test but are not actually prime.

**Carmichael number** is a composite number $n$ ($n \geq 2$) and for any $c$ with $\text{GCD}(c, n) = 1$, we have $c^{n-1} \equiv 1 \mod n$ where $\text{GCD}$ denotes the Greatest Common Denominator.
Korselt Theorem: A positive composite integer \( n \) is a Carmichael number if and only if \( n \) is square-free, and for all prime divisors \( p \) of \( n \), it is true that \( \frac{n - 1}{p - 1} \) is a constant integer.
Quiz

Determine the three square-free integers that provide the smallest Carmichael number?
Determine the three square-free integers that provide the smallest Carmichael number?

Answer:

Smallest Carmichael number with three primes:

\[ 561 = 3 \times 11 \times 17 \]

\[ n = 561, \quad p_1 = 3, \quad p_2 = 11, \quad p_3 = 17, \text{ therefore,} \]

\[ \frac{n-1}{p_1-1} = \frac{561-1}{3-1} = 280 \text{ constant number} \]

\[ \frac{n-1}{p_2-1} = \frac{561-1}{11-1} = 55 \text{ constant number} \]

\[ \frac{n-1}{p_3-1} = \frac{561-1}{17-1} = 4 \text{ constant number} \]
Chapter 18

18.3 Monte Carlo Algorithms

- 18.3.1 Biased MC Algorithms
- 18.3.2 A MC algorithm for Testing Polynomial Equality
- 18.3.3 Primality Testing

Eg. http://www.brighton-webs.co.uk/montecarlo/concept.asp
Monte Carlo Algorithms

A MC algorithm is a probabilistic algorithm that has a certain probability of returning the correct answer whatever the input is considered.

A p-correct MC algorithm is a probabilistic alg. that returns the correct answer with the probability not less than $p$, independent of the input.
Biased MC Algorithms

A MC algorithm for a decision problem is \textit{false-biased} if it always correct when it returns the value \textit{false} and only has some (small) probability of making a mistake when returning the value \textit{true}. 
Biased MC Algorithms(2)

Idea is that the probability of returning correct output increases with repeated trials...

Example:
function MCRewpeat(k)
    for i:=1 to k do
        if MC returns .false. then
            return (.false.)
        endif
    endfor
    return(.true.) .........>>>> if all k execution returns true, then we return true!!!
end MCRewpeat

Proposition:
Suppose that we have a p-correct false-biased Monte Carlo algorithm. Then the algorithm MCRewpeat(k) is a
(1-power((1-p),k)) correct false biased MC algorithm.
Proof: For any given input $I$,

Let $q$: probability that MCR\text{epeat}(k)$ outputs \texttt{false}.

$p$: probability that MCR\text{epeat}(k)$ outputs the correct answer for the given input $I$.

$$p = q(I) \times \text{Prob}(\text{MCR\text{epeat}(k)} \text{ is correct } | \text{MCR\text{epeat}(k)} \text{ outputs } \texttt{false.}) +$$

$$(1-q(I)) \times \text{Prob}(\text{MCR\text{epeat}(k)} \text{ is correct } | \text{MCR\text{epeat}(k)} \text{ outputs } \texttt{true.})$$

$$\geq q(I) \times (1 - (1-p)^k) + (1-q(I)) \times (1 - (1-p)^k)$$

$$= (1 - (1-p)^k) \times (q(I) + 1 - q(I))$$

$$= (1 - (1-p)^k)$$

Assume $\frac{3}{4}$- correct MC algorithm for primality testing. If MC $\frac{3}{4}$ correct, then for $k=8$, MCR\text{epeat}(k)$ is $0.9999847$-correct. That is, MCR\text{epeat}(8) returns \texttt{true.} with a strong confidence.
A MC algorithm for testing Polynomial Equality

Given \( f(x) \), \( g(x) \), and \( h(x) \) with degree \( 2n \), \( n \) and \( n \) respectively, we want to test if \( f(x) = g(x) \cdot h(x) \).

Assume a false biased 1/3-correct MC algorithm, test \( f(x) = g(x) \cdot h(x) \).

Input: \( f(x) \), \( g(x) \), \( h(x) \)

Output: \texttt{true.} 100\% of the time if \( f(x) \) is equal to \( g(x) \cdot h(x) \) and at most 2/3 of the time \( f(x) \) is not equal to \( g(x) \cdot h(x) \).

\[
\text{function TestPolyEqual } (f(x), g(x), h(x)) \\
\text{call Random}\{1,2 \ldots, 3n\}, j) \\
\text{if } f(j) = g(j) \cdot h(j) \text{ then} \\
\text{ return (.true.)} \\
\text{else} \\
\text{ return (.false.)} \\
\text{endif} \\
\text{end TestPolyEqual}
\]

This algorithm has \( O(n) \) complexity since the evaluations of \( f(j), g(j), h(j) \) can be done with \( 2n, n \) and \( n \) multiplications, respectively, using the algorithm HornerEval.

We can have \( f(j) = g(j) \cdot h(j) \) at most 2n points for \( j: \{1,2 \ldots, 3n\} \). Hence TestPolyEqual is 1/3 correct. Also, Prob. that TestPolyEqualRepeat(20) makes an error is less than \( \text{Power}((2/3),20) < 0.0003 \ldots \)

Observe: \( \text{Power}((2/3),20) \) close to unity!
The primality testing problem and the related factoring problem of finding the prime factors of a composite number, play important roles in cryptography.

Assume $n$ is a $m$-digit number. Is this a prime number or not?

There are deterministic and probabilistic approaches for the answer...

The basic structure of randomized primality tests is as follows:

1. Randomly pick a number $a$.
2. Check some equality (corresponding to the chosen test) involving $a$ and the given number $n$. If the equality fails to hold true, then $n$ is a composite number, $a$ is known as a witness for the compositeness, and the test stops.
3. Repeat from step 1 until the required certainty is achieved.

After several iterations, if $n$ is not found to be a composite number, then it can be declared probably prime.
Primality Testing- earlier approach

Assume $n$ is a $m$-digit number. Is this a prime number or not?

A simple approach:
Divide $n$ to each integer between 2 and $\sqrt{n}$ (assume lower limit $\sqrt{n}$)

Input: $n$ positive integer
Output: true. If $n$ is a prime false. otherwise

function TestPrimality($n$)
    for $i := 2$ to $\sqrt{n}$ do (assume lower limit $\sqrt{n}$)
        if $n \mod i = 0$ then
            return false.
        endif
    endfor
    return true.
end TestPrimality
If $n$ has $m$ digits in base 10, then the number of digits in the decimal representation of $n$ is given by the upper limit of $m = \log_{10}(n+1)$.

Thus, the worse case occurs when $n$ is prime, at which the TestPrimality executes the `for` loop $\sqrt{n}$ times.

$W(n) = \sqrt{n} = \text{power}(10, m/2)$, where $n$ is the number of digits. Thus, TestPrimality is exponential with respect to input size, $m$. !!!!

eg. for an input having more than 40 digits, it would take millions of years to execute in worst case.!!!!
Primality Testing (another approach)

Another approach would be to check whether \( n = p_1 \cdot p_2 \) where \( p_1 \) and \( p_2 \) are two primes.

```plaintext
function MCTestPrimality(n)
call Random({2,...,\sqrt{n}},j)
if \( n \mod j = 0 \) then
    return(.false.)
else
    return(.true.)
endif
end MCTestPrimality
```

- If \( n = p_1 \cdot p_2 \), where \( p_1 < p_2 \) are primes, then the probability to return correct answer \( \text{false.} \) is less than \( 1/p_1 \).
- We need an algorithm with a \textit{fixed} positive probability for any size of input.
Primality Testing (some rules!!!)

- A simple, but very inefficient primality test uses Wilson's theorem, which states that $n$ is prime if and only if:
  
  $$(n-1)! = -1 \mod n$$

- Although this method requires about $n$ modular multiplications, rendering it impractical!!!.
Primality Testing (Fermats primality tests, a well known probabilistic approach)

If $n$ is prime, then for all positive integers $a < n$,
$$a^{n-1} \equiv 1 \pmod{n}$$

If the result is different from 1, then $n$ is composite. If it is 1, then $n$ may or may not be prime.....

The Fermat primality test is only a heuristic test; some composite numbers (Carmichael numbers) will be declared "probably prime".

Nevertheless, it is sometimes used if a rapid screening of numbers is needed, for instance in the key generation phase of the RSA public key cryptographic algorithm.
Primality Testing (5)

By utilizing Fermats theorem, *RSSTestPrimality* algorithm is designed, with a 0.75 correct false-biased property (see your book for details).

Many other probabilistic primality tests includes:
The *Miller–Rabin primality test* and *Solovay–Strassen primality test*

Also, deterministic primality tests includes
Manindra Agrawal, Neeraj Kayal and Nitin Saxena invented a new deterministic primality test, the *AKS primality test*, which they proved runs in \( \tilde{O}(\log n)^{12} \), later improved to \( \tilde{O}(\log n)^6 \) in 2004.

The *elliptic curve primality test*, which actually proves that the given number is prime, can be proven to run in \( O((\log n)^6) \)
the *cyclotomy test*; its runtime can be proven to be \( O((\log n)^c \log \log \log n) \), where \( n \) is the number to test for primality and \( c \) is a constant independent of \( n \).
Another Prob. Algorithm: Las Vegas

An algorithm that always returns a correct answer, but it may run arbitrarily long without producing an answer, or it might make a random decision from which there is no recovery (a dead end)

HWLA