Question 2.13

How much time is used to compute \( f(x) = \sum_{i=0}^{N} a_i x^i \):

a) using the “dumb” exponentiation algorithm?

b) using the “fast” exponentiation algorithm?

Solution

Let \( c \) be any integer between 1 and \( N \).

a) Using “dumb” exponentiation

\( c \) multiplications are used to compute \( a_c x^c \). Therefore \( \sum_{i=0}^{N} i \) multiplications are used to compute \( \sum_{i=0}^{N} a_i x^i \) (as well as \( N \) additions). Therefore the number of operations used to compute \( f(x) \) is:

\[
\frac{N(N+1)}{2} + N
\]

\[= O(N^2)\]

b) Using “fast” exponentiation

Approximately \( \log_2 c \) multiplications are used to compute \( a_c x^c \). Therefore \( \sum_{i=1}^{N} \log_2 i \) multiplications are used to compute \( \sum_{i=1}^{N} a_i x^i \) (as well as \( N \) additions). Therefore the number of operations used to compute \( f(x) \) equals:

\[
\frac{N}{i=1} \log_2 i + N
\]

\[= \log_2 1 + \log_2 2 + \log_2 3 + ... + \log_2 N + N
\]

\[= \log_2 1 \times 2 \times 3 \times ... \times N + N
\]

\[= \log_2 N! + N
\]

\[= O(\log N!)\]
**Question 2.14** Consider the following algorithm (known as *Horner’s rule*) to evaluate $f(x) = \sum_{i=0}^{N} a_i x^i$:

```
poly = 0;
for( i = n; i >= 0; i -- )
    poly = x × poly + a[i];
```

a) Show how the steps are performed by this algorithm for $x = 3$, $f(x) = 4x^4 + 8x^3 + x + 2$.
b) Explain why this algorithm works
c) What is the running time of this algorithm?

**Solution**
a) The values of the multiplicative constants are, $a_0 = 2$, $a_1 = 1$, $a_2 = 0$, $a_3 = 8$, $a_4 = 4$ and $x = 3$. Now using these just step through the loop one iteration at a time.

b) $f(x)$ can be rewritten as follows:

\[
f(x) = (a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + ... + a_2 x^2 + a_1 x + a_0) \\
= (x(a_n x^{n-1} + a_{n-1} x^{n-2} + a_{n-2} x^{n-3} + ... a_2 x + a_1) + a_0) \\
= (x(x(a_n x^{n-2} + a_{n-1} x^{n-3} + a_{n-2} x^{n-4} + ... a_2) + a_1) + a_0) \\
= . \\
= . \\
= . \\
= (x(x(...x(x(a_n) + a_{n-1}) + a_{n-2})...) + a_2) + a_1) + a_0)
\]

Observe that the sum of the value of the expression inside the $ith$ innermost brackets, equals the value of poly after $i$ iterations. Therefore the value of the expression inside the $(n+1)th$ innermost brackets (i.e. the outer brackets) equals the value of poly after $n + 1$ steps. But this also equals the value of $f(x)$.

c) $O(N)$ - because the for loop iterates $N + 1$ times, with a constant amount of work at each iteration.
Question 2.15
Give an efficient algorithm to determine if there exists an integer \( i \) such that \( A_i = i \) in an array of integers \( A_1 < A_2 < A_3 < \ldots < A_N \). What is the running time of your algorithm?

Solution
Let \( c \) be an integer between 1 and \( N \). Observe that,

- if \( A_c < c \), then \( A_i < i \), for \( i = 1 \) to \( (c - 1) \).
- if \( A_c > c \), then \( A_i > i \), for \( i = (c + 1) \) to \( N \)

Algorithm:
Check if the middle element satisfies \( A_i = i \), and if it does the answer is yes.
If \( A_i < i \) we can apply the same strategy to the subarray to the right of the middle element.
If \( A_i > i \) we can apply the same strategy to the subarray to the left of the middle element.

Runtime Analysis:
The problem is halved in size at each step, for a constant amount of work. Therefore \( O(\log N) \).
Question 2.20
a) Write a program to determine if a positive integer, \( N \), is prime.
b) In terms of \( N \), what is the worst case running time of your program?
c) Let \( B \) equal the number of bits in the binary representation of \( N \). What is the value of \( B \)?
d) In terms of \( B \), what is the worst-case running time of your program?
e) Compare the running times to determine if a 20-bit and a 40-bit number are prime

Solution
a) Test to see if \( N \) is 1, 2 or odd and not divisible by 3, 5, 7, ...\( \sqrt{N} \).

```cpp
#include <cmath>

bool prime_test(int N)
{
    if (N == 1 || N == 2)
        return true;
    if (N % 2 == 0)
        return false;
    for(int i = 3; i < \sqrt{N}; i = i + 2)
        if (N % i == 0)
            return false;
    return true;
}
```
b) One for loop, which iterates at most \( \sqrt{N} \) times, with a constant amount of work each time. Therefore \( O(\sqrt{N}) \).
c) With \( x \) bits one can represent any number from 0 up to \( 2^x - 1 \). Therefore \( N \) can be represented with \( O(\log_2 N) \) bits.
d) \( B \approx \log_2 N \). Therefore \( N \approx 2^B \). So the running time is \( O(\sqrt{2^B}) = O(2^{\frac{B}{2}}) \).
e) A 20-bit number can be tested in time approximately \( 2^{\frac{20}{2}} = 2^{10} \). A 40-bit number can be tested in time approximately \( 2^{\frac{40}{2}} = 2^{20} \). Observe that \( 2^{20} = (2^{10})^2 \).
f) \( B \) is better because it more accurately measures the size of the input.
**Question 2.21**

The *Sieve of Eratosthenes* is a method used to compute all primes less than $N$. We begin by making a table of integers 2 to $N$. We find the smallest integer, $i$, that is not crossed out, print $i$, and cross out $i, 2i, 3i, ...$. When $i > \sqrt{N}$, the algorithm terminates. What is the running time of this algorithm?

Here are two nice graphical explanations of the sieve method

- MathWorld’s [page](#)
- scroll down to table on this [page](#)

**Solution**

// Sieve of Eratosthenes: at end of function the bool array 'prime'
// contains all of the prime numbers marked true; that is
// prime[i] = true if and only if i is prime

```cpp
void sieve(int n) {
    bool prime[n+1]; // indices 0..n
    prime[0] = prime[1] = false; // set these initially
    for (int i = 2; i <= n; i++)
        prime[i] = true; // initialise remainder to true

    for (int p = 2; p <= sqrt(n); p++)
        if (prime[p] == false) continue; // ignore non-primes
            for (int k = 2; k*p <= n; k++) // loop iterates n/p times
                int c = k*p; // c is composite, non-prime
                prime[c] = false;

    // The number of steps taken to work out all the primes from 1 to n is equal
    // to the number of times you cross out an integer (note some integers get crossed
    // out more than crossed out more than once e.g. 21 = 3 \times 7, gets crossed out for
    // i = 3 and i = 7). The inner loop gets triggerred once for every prime less than
    // or equal $\sqrt{n}$. You didn’t know this but there are $\approx \log \sqrt{n}$ prime numbers less
    // than or equal to $\sqrt{n}$. You didn’t know this but there are $\approx \log \sqrt{n}$ prime numbers less
    // than or equal to $\sqrt{n}$.

    // The work done by the inner loop (number of iterations) when working with
    // $p$ is $N/p$. Over the entire lifetime of the algorithm this will amount to $N/2 +
    // N/3 + N/5 + \cdots + N/q$ , where $q$ is the last prime no. less than or equal to $\sqrt{n}$.
    // (There are $\log \sqrt{n}$ terms.) This is equal to $N$ times the sum of the reciprocals
    // of the primes less than or equal to $\sqrt{n}$ and is less than $N$ times the sum of the
    // first $\log \sqrt{n}$ reciprocals, which is the harmonic number $H_{\log \sqrt{n}}$ whose size we
    // said in Lect01 is $\approx \log \log \sqrt{n}$. Therefore the running time of this algorithm is
    // $O(N \log \log \sqrt{n})$.}
```
Question 2.22
Show that $X^{62}$ can be computed with only eight multiplications.

Solution

\[ X^{62} = X^{40} \times X^{20} \times X^{2} \]
\[ X^{40} = X^{20} \times X^{20} \]
\[ X^{20} = X^{10} \times X^{10} \]
\[ X^{10} = X^{8} \times X^{2} \]
\[ X^{8} = X^{4} \times X^{4} \]
\[ X^{4} = X^{2} \times X^{2} \]
\[ X^{2} = X \times X \]

Note that this is better than even the exponentiation by binary decomposition method. It doesn’t work in general though.
Question 2.24
Give a precise count on the number of multiplications used during the fast exponentiation routine. (Hint: consider the binary representation of $N$.)

Solution
For $N = 0$ or $N = 1$ the number of multiplications is zero.
For $N > 1$. Let $b(N) =$ the number of ones in the binary representation of $N$.

It is possible to convert a number in base 10 to a number in base by successive division by 2.
E.g. Converting $135_{10}$ to base 2.

$135/2 = 67$ remainder 1  
$67/2 = 33$ remainder 1  
$33/2 = 16$ remainder 1  
$16/2 = 8$ remainder 0  
$8/2 = 4$ remainder 0  
$4/2 = 2$ remainder 0  
$2/2 = 1$ remainder 0  
$1/2 = 0$ remainder 1

Thus the number $135_{10}$ in binary is 10000111.

To work out $X^{135}$ using the fast exponentiation algorithm the following steps are taken:

$X^{135} = X^{67} \times X^{67} \times X$  
$X^{67} = X^{33} \times X^{33} \times X$  
$X^{33} = X^{16} \times X^{16} \times X$  
$X^{16} = X^8 \times X^8$  
$X^8 = X^4 \times X^4$  
$X^4 = X^2 \times X^2$  
$X^2 = X \times X$  

Notice that the number of times two multiplications are needed at a step, rather than 1, is equal to $b(N) - 1$. Thus the number of multiplications used to evaluate $X^N$ equals the number of steps, $\lceil \log N \rceil$, plus the number of ones in the binary representation, $b(N)$, minus 1.

So the answer is:

$\lceil \log N \rceil + b(N) - 1$
Question 2.25
Programs A and B are analyzed and found to have worst-case running times no greater $150N \log_2 N$ and $N^2$, respectively. Answer the following questions, if possible:

a) Which program has the better guarantee on running time, large values of $N (N > 10,000)$?
b) Which program has the better guarantee on running time, small values of $N (N < 100)$?
c) Which program will run faster on average for $N = 1000$?
d) Is it possible that program B will run faster than program A on all possible inputs?

Solution
a) A
b) B
c) Not enough information given to answer this question.
d) Yes.