Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Notes 2.4.
Assistances and comments will be acknowledged.

2.4 Algorithm Analysis

It has already been described qualitatively that if we intend to store nothing but objects, that this can be done quickly using a hash table; however, if we wish to store relationships and perform queries and data manipulations based on that relationship, it will require more time and memory. The words "quickly" and "more" are qualitative.

In order to make rational engineering decisions about implementations of data structures and algorithms, it is necessary to describe such properties quantitatively: "How much faster?" and "How much more memory?"

The best case to be made is that a new algorithm may be known to be *faster* than another algorithm, but that one word will not be able to allow any professional engineer to determine whether or not the newer algorithm is worth the seven person-weeks required to implement, integrate, document, and test the new algorithm. In some cases, it may be simply easier to buy a faster computer.

2.4.1 Operators

A processor is restricted to performing a limited set of *instructions*. The exact set of instructions differs from processor to processor, however all perform more-or-less the same general operations. In ECE 222, you will see the ColdFire instruction set, but examples of others include MIPS, ARM, x86, 68k, 6800, *etc*. Each instruction executes in a fixed number of cycles.

Assembly language is essentially a human readable representation of assembly instructions, for example, `ADDI.L #$F, D7` maps to the instruction `0x0687000000F`. The strong dependency of assembly language on the hardware results in it being described as a *low-level* language.

Languages that provide a significant level of abstraction from the underlying hardware are described as *high-level* languages. This includes Java, C++, Matlab, and C#. Because of the high dependence on the operators in C to the underlying instructions, it can be described as a *mid-level* language. For example, the primary reason for operators such as `+=` and `++` is that instructions such as `ADD.L D1, D6` replace the value stored in data register 6 with the sum of the values stored in data registers 1 and 6 and there are even more efficient operands that perform autoincrement than, say, `ADDI.L #$1, D7` (the logical translation of `a += 1`).

Therefore, each operator in C++ can be considered to translate to a fixed maximum number of assembly instructions and that collection of assembly instructions will run in a fixed number of cycles. This includes all the operators listed in Table 1.

Table 1. Classification of operators.

Operator Classification	Examples
Assignment	`a=b`
Arithmetic operations	`+a -a a+b a-b a*b a/b a%b a+=b a-=b a*=b a/=b a%=b`
Increment/Decrement	`a++ ++a a-- --a`
Logical	`a&&b a||b !a`
Bitwise	`a&b a|b a^b ~a a<<b a>>b a&=b a|=b a^=b a<<=b a>>=b`
Relational	`a==b a!=b a<b a<=b a=>b a>b`
Pointer	`a[] *a &a a.b a->b a.*b a->*b`
Memory allocation	`new T  new T[]  delete a  delete[] a`
Others	`a() a,b a::b a?b:c  (T)a  sizeof T  typeid T`

Any combination of operands (variables, literals, or constants) and operators forms an *expression*. Each operation always returns a value which may then be used by the next operator. The behaviours of some of the potentially more interesting operators are given in Table 2.

Table 2. Less obvious properties of some operations.

Operator	Example	Description
Assignment	`a = b`	The left operand must evaluate to something that has an address. The return value is the value assigned to the left operand. Thus `a = b = c` is equivalent to `a = (b = c)` (right associative) and the value of `c` is assigned to `b` and the result of this is the value `c` which is then assigned to `a`.

Boolean	`a == b`
	`a < b`
		These compare the operands and return either `0` or `1` representing *true* or *false*. In C++, any non-zero value is *true* while any zero value is *false*. False includes the `bool 0`, any integer value `0`, any floating-point `0.0`, the character `'\0'`, and the `0` or *null* pointer. `return a == b;` compares `a` and `b` and then returns either `0` or `1`, as appropriate.

Increment	`++a`
	`a++`
		The first, `++a`, adds one to `a` and then returns the new value of `a`.
		The second, a++, adds one to `a` but then returns the original value of `a`.
		Like the left operator of =, the operand must evaluate to something that has an address. The statements
		    `int a = 3;`
		    `cout << (++a + a++) << end;`
		will print `8` and `a` will be assigned `5`.

Comma	`a,b`	Outside of the arguments to a function call, the behaviour of the comma operator is to evaluate the operands in order and then return the result of the second operand. For example,
		    `int a = 3;`
		    `int b = (a++, ++a, 3*a);`
		will assign the value of `15` to `b`.

Important: Any expression that does not make a function call (functions, member functions, constructors, copy constructors, *etc*.) executes in Θ(1) time.

2.4.2 Statements

The unit of execution is a *statement*. Any expression followed by a semi-colon forms an e*xpression statement*; however, there are other statements, as well. Table 3 lists the different types of statements and gives examples. Anything in square parentheses is optional.

Table 3. Classification of statements.

Statement Classification	Examples
Expression	`expression;`
Declaration	`type name; type name = initial value;`
Flow-control	`break; continue; return [expression]; goto label;`
Compound	`{ statement statement ... statement }`

Conditional	`if (expression) statement [else statement]`
	`switch (expression) { case constant: statement ... default: }`

Looping	`for ( expression; expression; expression ) statement`
	`while ( expression ) statement`
	`do statement while( expression );`

Null	`;`

Important: Expression statements, declaration statements that do not call constructors, and the null statement all execute in Θ(1) time. Any compound statement comprising only these three classifications also runs in Θ(1) time.

For example, all four of the following compound statements run in Θ(1) time.

Code Samples 1-4

```
{
    delete [] column_index;
    delete [] off_diag;
    capacity = A.capacity;
    column_index = new int[capacity];
    off_diag = new double[capacity];
}
```

```
{
    Tree_node *lrl = left->right->left;
    Tree_node *lrr = left->right->right;
    parent = left->right;
    parent->left  = left;
    parent->right = this;
    left->right = lrl;
    left = lrr;
}
```

```
{
    ++index;
    prev_modulus = modulus;
    modulus = next_modulus;
    next_modulus = modulus_table[index];
}
```

```
{
    Type tmp = array[i];
    array[i] = array[posn];
    array[posn] = tmp;
}
```

We will now proceed to examine the run times of looping and conditional statements, how to determine the run time of a sequence of statements, and finally determining the run times of function calls, specifically, recursive functions.

2.4.3 Looping Statements

Repetition statements in C++ are condition-controlled statements:

```
for ( initialization-expr; conditional-expr; loop-increment-expr )
      body-statement
```

The *initialization-expression* is the first expression that is executed. So long as the *conditional-expression* evaluates to *true* (any non-zero value), the *body-statement* is executed followed by the *loop-increment-expression*. Assuming that all of the expressions run in Θ(1) time (which is normally the case), the run time is the sum of the run times of the body.

The easiest analysis is a counting loop where the body-statement runs in Θ(1) time, for example, the run time of the statement

```
for ( int i = 0; i < n; ++i ) {
    sum += i;
}
```

is $\Theta(1\cdot n) = \Theta(n)$. We would say this statement has a *linear run-time*.

2.4.3.1 Nested Loops

Suppose, however, we are calculating a matrix-vector product **u** = **Mv**:

Code Sample 5

```
// outer loop
for ( int i = 0; i < n; ++i ) {
    u[i] = 0;

    // inner loop
    for ( int j = 0; j < n; ++j ) {
        u[i] += M[i][j] * v[j];
    }
}
```

To examine nested loops such as this, we will examine the inner-most loop first and work our way out. In this case, the inner loop executes a Θ(1) statement n times, thus the run time of this inner loop is $\Theta(n)$. For now we will note that the *body-statement* of the outer loop is a Θ(1) statement followed by a $\Theta(n)$ statement. The execution of the first statement will not affect the more expensive second statement, thus, the body of the outer loop runs in $\Theta(n)$ time. This body is executed n times and therefore the total run time is $\Theta(n^2)$. The statement would also be described as having a *quadratic run-time*.

Similarly, calculating a matrix-matrix product **C** = **AB** where A is an $n_1 \times n_2$ matrix and B is an $n_2 \times n_3$ matrix be implemented as follows:

Code Sample 6

```
// outer loop
for ( int i = 0; i < n_1; ++i ) {
    // intermediate loop
    for ( int k = 0; k < n_3; ++k ) {
        C[i][k] = 0;

        // inner loop
        for ( int j = 0; j < n_2; ++j ) {
            C[i][k] += A[i][j] * B[j][k];
        }
    }
}
```

Again, the inner loop is executing an $\Theta(1)$ statement n_2 times, and thus, the inner loop and the assignment run in $\Theta(n_2)$ time. The intermediate loop executes an $\Theta(n_2)$ statement n_3 times and thus it runs in $\Theta(n_2n_3)$ time. Finally, the outer loop runs a $\Theta(n_3n_2)$ statement n_1 times and therefore the overall run time is $\Theta(n_1n_2n_3)$.

If $n_1 = n_2 = n_3 = n$, we could describe the run time as cubic or $\Theta(n^3)$. If we knew for a particular problem that $n_2 = 2$ for all cases, the run time could be simplified to $\Theta(n_1n_3)$.

In general, if the body of a for-loop of the form

```
for ( int i = 0; i < n; ++i )
    body-statement;
```

has a body that has a run-time of $\Theta(b)$ and b does not depend on i, the run-time of the loop is $\Theta(nb)$.

2.4.3.2 Logarithmic Loops

Here is a different type of loop:

Code Sample 6

```
for ( int i = n; i > 0; i /= 2 ) {
    sum += i;
}
```

Clearly, this does not run in $\Theta(n)$. If n is 100, the values of `i` after successive iterations are 100, 50, 25, 12, 6, 3, 1, 0. The body-statement would only be executed 7 times and on the eighth loop, `i` would be assigned 0. Suppose n is 1000, the values of `i` would be 1000, 500, 250, 125, 62, 31, 15, 7, 3, 1, 0, thus, the body-statement would be executed 10 times. To determine the run time, assume that n is of the form $n = 2^m$. In this case, the values `i` would be $2^m, 2^{m-1}, 2^{m-2}, \ldots, 2^3, 2^2, 2, 1, 0$ and thus, the body would be executed m times; however, $m = \lg(n)$ and therefore the run time would be $\Theta(\lg(n))$.

Recall that all logarithms are scalar multiples of each other, and therefore lg(n) = Θ(ln(n)) and thus will simply describe the run time of this algorithm as Θ(ln(n)) or *logarithmic*: double the size of the problem, the run time increases by a constant.

Note we can always rewrite a function like this in terms of

Code Sample 7

```
for ( int j = 0; j <= std::log( n )/std::log( 2.0 ); ++j ) {
    int i = n/(1 << j);
    sum += i;
}
```

2.4.3.3 Iterative Loops

As a different example, consider the following code from Project 1:

Code Sample 8

```
for ( Single_node<Type> *ptr = head(); ptr != 0; ptr = ptr->next() ) {
    if ( ptr->retrieve() == obj ) {
        return true;
    }
}
```

In this case, you would have to realize that the pointer `ptr` is starting at the head of the linked list and with each iteration, it steps through the linked list until `ptr` is assigned the address of the next node of the tail, that is, the *null pointer* or `0`. Suppose the linked list has n nodes. The loop statement will run in Θ(1) and thus, in the worst case, the loop will run n times. Note however, if the object stored one of the nodes equals `obj`, we end the loop early. Therefore, as the loop will not necessarily run n times, we will say that the run time is O(n). It may run n times, but, if we're lucky, it may run in less time.

2.4.3.4 Θ(n) versus O(n) Loops

In order to reinforce this idea, consider the two examples:

Code Samples 9 and 10

```
int find_max( int *array, int n ) {
    int max = array[0];

    for ( int i = 1; i < n; ++i ) {
        if ( array[i] > max ) {
            max = array[i];
        }
    }

    return max;
}
```

```
bool linear_search( int value, int *array, int n ) {
    for ( int i = 0; i < n; ++i ) {
        if ( array[i] == value ) {
            return true;
        }
    }

    return false;
}
```

The first function finds the maximum entry in an array of size *n* while the second attempts to determine if there is a value *v* in the array of size *n*. In order to find the maximum value, it is necessary to search every entry of the array and thus, the run time of the left function is Θ(*n*). The function on the right, however, may return early, and therefore we will say that its run time may be O(*n*).

2.4.3.5 Variable-dependent Loops

Suppose we now are multiplying a lower-triangular matrix (all the entries above the diagonal are zero) by a vector, that is, **u** = **Lv**:

Code Sample 11

```
// outer loop
for ( int i = 0; i < n; ++i ) {
    u[i] = 0;

    // inner loop
    for ( int j = 0; j <= i; ++j ) {
        u[i] += L[i][j] * v[j];
    }
}
```

In this case, the run time of the inner loop is Θ(*i*); however, *i* changes with each iteration of the outer loop. Therefore, we cannot simply return Θ(*ni*)! Instead, it is now necessary to count the number of operations: first the inner loop runs once, then twice, and so on, up to *n* times in the last case. Because the run time of the inner loop is Θ(*i*), we calculate:

$$\Theta\left(\sum_{i=1}^{n} i\right) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta\left(n^2\right).$$

Thus, while this second function may be faster than multiplying a full matrix by a vector, the growth is still similar.

Try determining the run times of the following six repetition statements. In the latter four, assume *n* is a power of 2.

Code Samples 12 and 13

```
for ( int i = 0; i < n*n; ++i ) {
    for ( int j = 0; j < i; ++j ) {
        ++sum;
    }
}
```

```
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < i*i; ++j ) {
        ++sum;
    }
}
```

Answers: Θ(n^4) and Θ(n^3).

The analysis of while- and do-while-loops is similar to that of a for-loop; however, it will require an understanding of the problem to determine how often the loop will run based on the problem.

2.4.4 Conditional Statements

The format of an if-then statement is

```
if ( conditional-expression )
    consequent-statement
else
    alternative-statement
```

The *conditional-expression* is usually an expression that executes in Θ(1) time.

The analysis of the run time in general is often more difficult and is often situation dependant.

The easiest analysis is if both the *consequent-statement* and the *alternative-statement* run in Θ(1) time—in this case, we may say that the entire conditional statement must run in Θ(1) time. More generally, if both the *consequent-statement* and the *alternative-statement* have the same run time, say, Θ(*f*), it follows that the run time of the conditional statement will also be Θ(*f*).

The issue is when the conditional statement either:

1. Has a *consequent-statement* and an *alternative-statement* that have different run times, or
2. The conditional statement is used to prematurely break from a repetition statement or return from a function (we have already seen two examples of this).

In the first case, there may be no alternative but to consider the characteristics of the problem. Alternatively, one may be able to assign probabilities to either the *conditional-expression* being true or false. For example, suppose the following conditional statement:

Code Sample 14

```
if ( conditional-expression ) {
    for ( int i = 0; i < n; ++i ) {
        sum += i;
    }
}
```

Because there is no alternative statement, we may assume that if the *conditional-expression* returns *false*, the run time is Θ(1). If, however, the *conditional-expression* returns *true*, the run time is Θ(*n*).

Suppose that we embed this statement inside a for loop:

Code Sample 15

```
for ( int j = 0; j < n; ++j ) {
    if ( conditional-expression ) {
        for ( int i = 0; i < n; ++i ) {
            sum += i;
        }
    }
}
```

Thus, the conditional statement is run *n* times.

1. Suppose the conditional-expression has approximately a 50-50 chance of evaluating to true. In this case, *n*/2 times, the body of the repetition statement will run in Θ(1) time, while the other *n*/2 times, the body run in Θ(*n*) time. Thus, the run time of the repetition statement would be $\Theta\left(\frac{n}{2}n+\frac{n}{2}\cdot 1\right)=\Theta\left(\frac{n^2}{2}+\frac{n}{2}\right)=\Theta\left(n^2\right)$.
2. Suppose, however, we are certain that the *conditional-expression* will only ever return true once. In that case, the run time will be $\Theta\left(1\cdot n+\left(n-1\right)\cdot 1\right)=\Theta\left(2n-1\right)=\Theta\left(n\right)$.

Consider again the previous function `find_max`:

Code Sample 16

```
int find_max( int *array, int n ) {
    int max = array[0];

    for ( int i = 1; i < n; ++i ) {
        if ( array[i] > max ) {
            max = array[i];
        }
    }

    return max;
}
```

How often will the *conditional-expression* `array[i] > max` evaluate to true?

1. If the entries of the array are strictly monotonically increasing, the expression will evaluate to *true* every time,
2. If the largest entry of the array is in the first location (for example, if the entries are monotonically decreasing), it will never evaluate to *true*, but
3. What if the entries are random?

This third question is more subtle than one may think. The *obvious* answer is "half the time". This, however, turns out to be significantly wrong. Take some time over the next day or two to try to find a better answer to this question.

2.4.5 The Run-time of Statements Executed Serially

Suppose we are now executing a sequence of statements serially, that is,

```
statement_1
statement_2
    ⋮
statement_m
```

and that we know that the run-time of the kth statement is $\Theta(f_k)$. In this case, the run time of the sequence of statements is $\Theta\left(\sum_{k=1}^{m} f_k\right)$.

2.4.5.1 Example 1: Resizing an Array

For example, given the code

Code Sample 17

```
T *array_old = array;
int capacity_old = array_capacity;
array_capacity += delta;
array = new T[array_capacity];

for ( int i = 0; i < capacity_old; ++i ) {
    array[i] = array_old[i];
}

delete[] array_tmp;
```

we note the first four statements run in $\Theta(1)$ time, the fifth runs in $\Theta(n)$ time, and the last also runs in $\Theta(1)$ time. Therefore, the run time of these six statements is $\Theta(1 + 1 + 1 + 1 + n + 1) = \Theta(n + 5) = \Theta(n)$.

2.4.5.2 Example 2: Sparse Matrix Assignment

Let us examine the body of this function:

Code Sample 18

```
template<int M, int N>
Matrix<M, N> &Matrix<M, N>::operator= ( Matrix<M, N> const &A ) {
     if ( &A == this ) {
        return *this;
     }


     if ( capacity != A.capacity ) {
        delete [] column_index;
        delete [] off_diagonal;
        capacity = A.capacity;
        column_index = new int[capacity];
        off_diagonal = new double[capacity];
     }

     for ( int i = 0; i < std::min(M, N); ++i ) {
        diagonal[i] = A.diagonal[i];
     }

     for ( int i = 0; i <= M; ++i ) {
        row_index[i] = A.row_index[i];
     }

     for ( int i = 0; i < A.size(); ++i ) {
        column_index[i] = A.column_index[i];
        off_diagonal[i] = A.off_diagonal[i];
     }

     return *this;
}
```

This copies a sparse matrix with size n (there are n non-zero entries). Focusing exclusively on the body of the function, the first conditional statement returns if the matrix is being assigned to itself. Let us assume this is an exceptionally rare case and therefore `&A == this` will almost certainly never return true and if it does, it is by accident.

Thus, the first two conditional statements run in $\Theta(1)$ time. The bodies of each of the next three repetition statements is $\Theta(1)$ and thus, the first repetition statement that runs in $\Theta(\min(M, N))$ time, the second runs in $\Theta(M)$ time, and the third in $\Theta(n)$. Finally, the return statement runs in $\Theta(1)$ time.

Therefore, the overall run time is $\Theta(1 + 1 + \min(M, N) + M + n + 1) = \Theta(\min(M, N) + M + n)$. Can we simplify this further? We can note that $M \geq \min(M, N)$, so $\min(M, N) = \mathrm{O}(M)$ and thus, we may discard it. Therefore we are left with a run time of $\Theta(M + n)$.

2.4.6 The Run Time of Functions

Given a function f, the run time of which depends on a variable n, we may write its run time as $T_f(n)$. If function is understood or implied, we may simply write $T(n)$. If the run time of a function is known ,we may simply substitute its run time, for example, in the previous example, $T_=(n, N, M) = \Theta(M + n)$.

If we know the run time of a function is, for example, $\Theta(n)$, we would simply use that value. For example, without proof, I will state that the function

```
bool binary_search( int value, int *array, int n )
```

runs in $O(\ln(n))$ time. Therefore, the run time of the statement

Code Sample 18

```
for ( int i = 0; i < m; ++i ) {
    // Search for i in an array of size n
    std::cout << binary_search( i, entries, n );
}

std::cout << std::endl;
```

would run in $O(m \ln(n))$ time. Alternatively, if we replaced the binary search with a linear search,

Code Sample 19

```
for ( int i = 0; i < m; ++i ) {
    // Search for i in an array of size n
    std::cout << linear_search( i, entries, n );
}

std::cout << std::endl;
```

would run in $O(mn)$ time.

If we do not know the run time of a function, we would simply place the placeholder. Consider, for example, the function `set_union` shown on the next page. This is a function defined on a disjoint set of size n and therefore, we may assume that most functions will have a run-time that depends on n. There are two different functions that are called: `Disjoint_set::find` and `std::min`. By now it should be obvious that the run-time of the later is $\Theta(1)$, but without examining `find`, we do not know what the run time of the first function is. Let us represent its run time by $T_{\text{find}}(n)$. Therefore, after we notice that all expressions in the second conditional statement are $\Theta(1)$, we determine that the run time of this set union function is

$$T_{\text{set\_union}}(n) = 2\ T_{\text{find}}(n) + \Theta(2 + 1 + 1 + 1) = 2T_{\text{find}}(n) + \Theta(1).$$

Because we know nothing about the function $T_{\text{find}}(n)$, it is important that we track the "+ $\Theta(1)$" term—the balance of the function call still performs some work.

Code Sample 20

```
void Disjoint_sets::set_union( int m1, int m2 ) {
    m1 = find( m1 );
    m2 = find( m2 );

    if ( m1 == m2 ) {
        return;
    }

    --num_disjoint_sets;

    if ( tree_height[m1] >= tree_height[m2] ) {
        parent[m2] = m1;

        if ( tree_height[m1] == tree_height[m2] ) {
            ++( tree_height[m1] );
            max_height = std::max( max_height, tree_height[m1] );
        }
    } else {
        parent[m1] = m2;
    }
}
```

We will now look at the most interest case: when a function calls itself.

2.4.7 The Run Time of Recursive Functions

To begin, we will look at two functions: one that computes the factorial function recursively and another that does the same with the Fibonacci numbers.

Code Samples 21 and 22

```
int fact( int n ) {
    // fact( 0 ) = fact( 1 ) = 1, otherwise
    // it is n times the previous value
    if ( n <= 1 ) {
        return 1;
    } else {
        return n * fact( n – 1 );
    }
}
```

```
int fib( int n ) {
    // fib( 0 ) = 0 and fib( 1 ) = 1, otherwise
    // it is the sum of the two previous values.
    if ( n <= 1 ) {
        return n;
    } else {
        return fib( n - 1 ) + fib( n – 2 );
    }
}
```

Now we must be a little more careful. The run time of the functions will depend on the values of *n*. For example, for the factorial function, we can write

$$T_{\text{fact}}(n) = \begin{cases} \Theta(1) & n \le 1 \\ T_{\text{fact}}(n-1) + \Theta(1) & n > 1 \end{cases}.$$

You will note that this looks like a recurrence relation.

If you perform a similar analysis on the Fibonacci function, you get

$$T_{\text{fib}}(n) = \begin{cases} \Theta(1) & n \le 1 \\ T_{\text{fib}}(n-1) + T_{\text{fib}}(n-2) + \Theta(1) & n > 1 \end{cases}.$$

How about selection sort on an array of size *n*? This function finds the largest entry in the array, swaps it with the value in the last position, and then proceeds to recursively call selection sort on the same array but of size *n* – 1. The function is

Code Sample 23

```
void selection_sort( int * array, int n ) {
    if ( n <= 1 ) {
        return;                        // special case:  0 or 1 items are always sorted
    }

    int posn = 0;                      // assume the first entry is the smallest
    int max = array[posn];

    for ( int i = 1; i < n; ++i ) {    // search through the remaining entries
        if ( array[i] > max ) {        // if a larger one is found
            posn = i;                  // update both the position and value
            max = array[posn];
        }
    }

    int tmp = array[n - 1];            // swap the largest entry with the last
    array[n - 1] = array[posn];
    array[posn] = tmp;

    selection_sort( array, n – 1 );    // recursively sort everything else
}
```

Since all the expressions in the repetition statement are Θ(1), it follows that the run time of the loop is Θ(*n*). Consequently, the run time of body of this function Θ(1 + 2 + *n* + 3) + $T_{\text{select}}(n-1)$. Thus, in a similar manner, we may write

$$T_{\text{selection}}(n) = \begin{cases} \Theta(1) & n \le 1 \\ T_{\text{selection}}(n-1) + \Theta(n) & n > 1 \end{cases}$$

All three of these are very similar to recurrence relations; however, all of these contain Landau symbols. Recall that, for example, Θ(*n*) represents the equivalence class of all functions that grow linearly. If this is the case, why not select exactly one of those functions to represent the entire class—that is, replace Θ(*n*) with *n*, Θ(1) with 1, and so on.

This gives us three recurrence relations:

$$T_{\text{fact}}(n) = \begin{cases} 1 & n \le 1 \\ T_{\text{fact}}(n-1)+1 & n > 1 \end{cases}$$

$$T_{\text{fib}}(n) = \begin{cases} 1 & n \le 1 \\ T_{\text{fib}}(n-1)+T_{\text{fib}}(n-2)+1 & n > 1 \end{cases}$$

$$T_{\text{selection}}(n) = \begin{cases} 1 & n \le 1 \\ T_{\text{selection}}(n-1)+n & n > 1 \end{cases}$$

At this point, we could use a symbolic computation language such as Maple to solve these recurrence relations:

```
[> rsolve( {T(1) = 1, T(n) = T(n - 1) + 1}, T(n) );
```

$$n$$

```
[> rsolve( {T(0) = 1, T(1) = 1, T(n) = T(n - 1) + T(n - 2) + 1}, T(n) );
```

$$\left(-\frac{1}{10}\sqrt{5}+\frac{1}{2}\right)\left(\frac{1}{2}-\frac{1}{2}\sqrt{5}\right)^n+\left(\frac{1}{10}\sqrt{5}+\frac{1}{2}\right)\left(\frac{1}{2}+\frac{1}{2}\sqrt{5}\right)^n+\frac{2}{5}\frac{\sqrt{5}\left(-\frac{2}{1+\sqrt{5}}\right)^n}{1+\sqrt{5}}-\frac{2}{5}\frac{\sqrt{5}\left(-\frac{2}{1-\sqrt{5}}\right)^n}{1-\sqrt{5}}-1$$

```
[> asympt( %, n ); # What is the asymptotic behaviour as n grows large?
```

$$\frac{4}{5}\frac{\sqrt{5}\left(1+\sqrt{5}\right)^n}{\left(-1+\sqrt{5}\right)2^n}-1+\frac{4}{5}\frac{e^{in\pi}\sqrt{5}\left(-1+\sqrt{5}\right)^n}{\left(1+\sqrt{5}\right)2^n}$$

```
[> rsolve( {T(1) = 1, T(n) = T(n - 1) + n}, T(n) );
```

$$-1+(n+1)\left(\frac{1}{2}n+1\right)-n$$

```
[> simplify( % ); # Simplify this result...
```

$$\frac{1}{2}n^2+\frac{1}{2}n$$

With a little bit of algebra in the second case, we note that dominant (first) term is of the form ϕ^n where

$$\phi = \frac{\sqrt{5}+1}{2} \approx 1.618$$

is the golden ratio.

Thus, we see that the run time for these functions are:

Function	**Run Time**	**Classification**
Factorial	$\Theta(n)$	linear
Fibonacci	$\Theta(\phi^n)$	exponential
Selection Sort	$\Theta(n^2)$	quadratic

Thus, the factorial function is possibly as good as you can get, selection sort is slower, and using this technique to calculate the Fibonacci numbers is absolutely absurd—especially when you should be able to find the Fibonacci numbers in Θ(*n*) time.

Why use Maple?

In all these examples, we solved the recurrence relation using Maple. Why not use mathematics? Toward the end of this course, we will see the *master theorem* (yes, that is the name of the theorem). This will provide a mathematical theory for most useful recurrence relations.

2.4.8 Case Analysis

For any function that is big-Oh of an expression, we can usually determine three possible cases:

1. The best case,
2. The average case, and
3. The worst case.

We are not interested in the best case—that is often of no use. Instead, the information we need is what will be the performance and memory allocations on average and in the worst-case scenario.

2.4.8.1 Linear Search

Consider Code Sample 10: a linear search. If the object we are searching for does not appear in the list, the run time is Θ(*n*).

If the object appears in location *k*, the run time to find that object is Θ(*k*), but we usually do not know where the object will be found. Suppose however, we know that the object has an equal chance of appearing in any one of the *n* locations; that is, there is a 1/*n* chance of the object appearing in the *k*th location. We first note that the sum of the probabilities is 1: $\sum_{k=1}^{n} \frac{1}{n} = \frac{1}{n} \cdot n = 1$.

In this case, we must sum the products of the probabilities plus the work required to find that object:

$$\sum_{k=1}^{n}\left(\frac{1}{n}\cdot k\right)=\frac{1}{n}\sum_{k=1}^{n}k=\frac{1}{n}\cdot\frac{n(n+1)}{2}=\frac{n+1}{2}.$$

Therefore, the average effort will be $\Theta(n)$. It is true that it *may* require less than $\Theta(n)$ time, but the average time will double if you double the size of the array.

Aside

This is like throwing a six-sided die. The value that appears on the die in any roll could be any number from one to six with equal probability: 1/6. The average value is therefore

$$\sum_{k=1}^{6}\left(\frac{1}{6}\cdot k\right)=\frac{1}{6}\sum_{k=1}^{6}k=\frac{1}{6}\cdot\frac{21}{2}=\frac{7}{2}=3.5.$$

If this doesn't convince you, try rolling a dice 20 times. Add the numbers and divide by 20. On occasion you may get an average value less than three or greater than four, but usually the average will be around 3.5.

Suppose, however, that the probability of finding the object in the first entry is ½, the probability of finding the object in the second entry is ¼, and the probability of finding it in the kth location is half the probability of finding it in the previous location (except in the last location where the probability is the same as that of the previous location. This requires us to calculate:

$$\sum_{k=1}^{n-1}\left(\frac{1}{2^k}\cdot k\right)+\frac{1}{2^{n-1}}n.$$

Now, we do note that the sum of the probabilities is 1: $\sum_{k=1}^{n-1}\frac{1}{2^k}+\frac{1}{2^{n-1}}=\frac{1-\left(\frac{1}{2}\right)^n}{1-\frac{1}{2}}-1+\frac{1}{2^{n-1}}=1$

To calculate this, we, again, use Maple

```
[> sum( k/2^k, k = 1..n - 1 ) + n/2^(n - 1);
```

$$-2\left(\frac{1}{2}\right)^n n-2\left(\frac{1}{2}\right)^n+2+\frac{n}{2^{n-1}}$$

```
[> asympt( %, n );  # What is the asymptotic behaviour as n becomes large
```

$$2-\frac{2}{2^n}$$

This value is less than 2, so on average, we should do less than two searches. That is, the average run time is $\Theta(1)$. It doesn't matter if we quadruple the length of the list, it will still take, on average, only a constant number of searches. It may take *n* searches, but the probability of that event occurring is so small that it does not affect the overall probability.

2.4.8.2 Finding the Maximum Entry

Consider Code Sample 9: the question was posed "If the distribution of values in the list is randomly distributed, on average, how often is the variable max updated?"

Now, if we consider the first *k* entries of the array, the probability that the last entry is actually the largest is 1/*k*. Thus,

1. Upon checking the second entry, there is a 50-50 chance that the second entry is larger than the first, so there is a ½ chance that the update will be executed,
2. With the third entry, there is a one-in-three chance the update will be executed, and so on...

Now, the update runs in $\Theta(1)$ time, and therefore the total work required would be the sum-of-the-products of the probabilities and the corresponding work:

$$\sum_{k=2}^{n}\left(\frac{1}{k}\cdot 1\right).$$

Now, this is the *harmonic series* which is divergent (it grows to infinity as *n* approaches infinity); however, what is the behaviour for small *n*? One could consider the function in Figure 1 where the area of the rectangles is 1/2, 1/3, 1/4, *etc*. Therefore, if we could integrate this piecewise constant function from 1 to *n* + 1, we could calculate the sum.

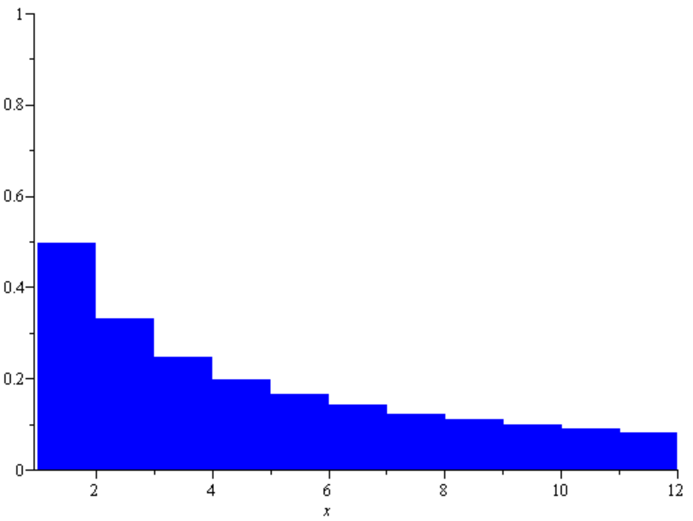

Figure 1. The harmonic series.

Unfortunately, integrating a piecewise constant function is no different from finding the sum. Note, however, that we can approximate the piecewise constant function by the continuous function $\frac{1}{x}$, as is shown in Figure 2.

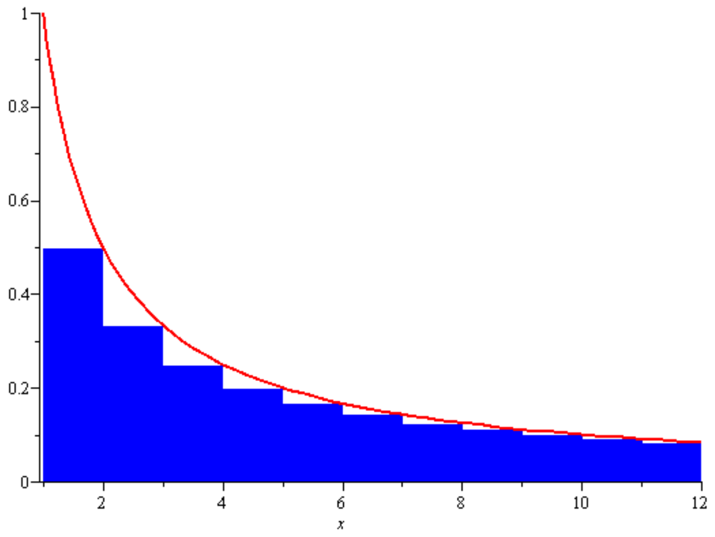

Figure 2. An approximation of Figure 1 using the function $\frac{1}{x}$.

Therefore, the area of the blue region is less than the integral of the red function which equals

$$\int_1^{n+1} \frac{1}{x}dx = \ln(x)\Big|_1^{n+1} = \ln(n+1) - \ln(1) = \ln(n+1).$$

Therefore, the sum is no more than the integral; however, could it be significantly less? We note that

$$\lim_{n\to\infty}\frac{\ln(n+1)}{\ln(n)} = \lim_{n\to\infty}\frac{\frac{1}{n+1}}{\frac{1}{n}} = \lim_{n\to\infty}\frac{n}{n+1} = \lim_{n\to\infty}\frac{1}{1} = 1$$

and therefore, the integral suggests that

$$\sum_{k=2}^{n}\frac{1}{k} = \mathrm{O}(\ln(n)).$$

Question: is it o(ln(*n*)) or Θ(ln(*n*))? To determine this, we must determine the error shown in Figure 3.

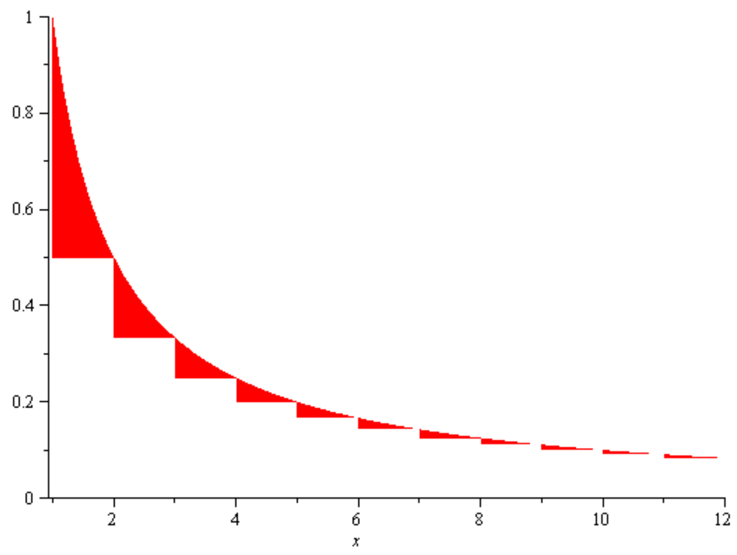

Figure 3. The error in the approximation.

Fortunately, note that the errors can be shown to form a subset of a 1 × 1 square, as is shown in Figure 4.

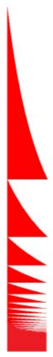

Figure 4. The accumulated errors.

Therefore, the error must be less than 1. As *n* becomes very large, the error converges to $1 - \gamma$ where γ is Euler's constant and $\gamma \approx 0.5772156649$.

Therefore, the run time of the find_max function is

$$\Theta\left(1+\sum_{k=2}^{n}\left(1+\frac{1}{k}\right)\right)=\Theta\left(1+n+\ln(n)+1-\gamma\right)=\Theta(n).$$

This does not change the overall run time; however, that particular assignment is not called, relatively speaking, that often.