Homework #4

Problem One (2.1.2)

Determine which characteristics of an algorithm the following procedures have and which they lack.

a)

procedure *double*(*n*: positive integer)
while $n > 0$
    $n := 2n$

Since n is a positive number, the while loop in this algorithm will run forever, therefore this algorithm is not finite.

b)

procedure *divide*(*n*: positive integer)
while $n >= 0$
begin
    $m := 1/n$
    $n := n - 1$
end

Since algorithm is not effective since the line "m := 1/n" cannot be executed when n=0, which will eventually be the case. It can also be argued that this algorithm is not finite: if a line in the algorithm cannot be completed, the algorithm as a whole cannot be completed. It can also be argued that the algorithm lacks correctness since the "m := 1/n" line will also keep the algorithm from arriving at a correct answer.

c)

procedure *sum*(*n*: positive integer)
sum := 0
while $j < 10$
    sum := *sum* + *j*

The value of j is never set in the algorithm, so the algorithm lacks definiteness. Without knowing the initial value of j, the behavior of this algorithm is undetermined.

d)

procedure *choose*(*a*, *b*: integers)
x := either *a* or *b*

The only line in the algorithm is ambiguous, how does the algorithm decide which value (a or b) to assign to x? Without knowing how this decision is made, the behavior of this algorithm is undetermined; therefore, this algorithm lacks definiteness.

Problem Two (2.1.16)

Describe an algorithm for finding the smallest integer in a finite sequence of natural numbers.

procedure find_min(a_0, a_1, …, a_n: positive integer)
min := a0
for j := 1 **to** n
    if (min > aj) **then** :min = aj

Problem Three (2.1.18)

Describe an algorithm that locates the last occurrence of the smallest element in a finite list of integers, where the integers in the list are not necessarily distinct.

```
procedure find_min(a0, a1, …, an: positive integer)
min := a0
index := 0
for j := 1 to n
        if (min >= aj)
        then
        begin
                min := aj
                index := j
        end
```

Problem Four (2.2.2)

Determine whether each of these functions is $O(x^2)$

a) $f(x) = 17x + 11$ Yes, the determining factor in f(x) is less than or equal to x^2.

b) $f(x) = x^2 + 1000$ Yes, the determining factor in f(x) is x^2 which is equal to x^2.

c) $f(x) = x\log(x)$ Yes, the determining factor in f(x) is xlog(x) which is less than x^2.

d) $f(x) = x^4/2$ No, the determining factor in f(x) is x^4 which is greater than x^2.

e) $f(x) = 2^x$ No, the determining factor in f(x) is 2^x which is greater than x^2.

f) $f(x) = \lfloor x \rfloor \bullet \lceil x \rceil$ Yes, the determining factor in f(x) is approximately x^2 which is equal to x^2.

Problem Five (2.2.6)

Show that $(x^3 + 2x)/(2x + 1)$ is O(x2)

Let: $f(x) = (x^3 + 2x)/(2x + 1) < (x^3 + 2x)/2x = (½)x^2 + 1$ $f_2(x) = (½)x^2 + 1$ $g(x) = x^2$

Since $f(x) < f_2(x)$, if $f_2(x) = O(g(x))$ then it must also be true that $f(x) = O(g(x))$.
If $f_2(x) = O(g(x))$, then $|f_2(x)| \leq C^*|g(x)|$ when $n > k$ for some integer value C and some real number k.

Our building blocks are: x2

$(½)x^2 + 1$	$\leq^?$	_____	
$(½)x^2 + 1$	$\leq^?$	x^2	$(½) \leq 1$, is always true, therefore, $(½)x^2 \leq x^2$ is true for all x.
$(½)x^2 + 1$	$\leq^?$	$x^2 + x^2$	$1 \leq x^2$, is true for all $x \neq 0$.
$(½)x^2 + 1$	$\leq$	$2x^2$	

Therefore, let C=2 and k=1, with these values "$|f_2(x)| \leq C^*|g(x)|$ when $n > k$" is a true statement. Since $f_2(x) = O(g(x))$, it necessarily follows that $f(x) = O(g(x))$ as well.

Problem Six (2.2.18)

Let k be a positive integer. Show that $1^k + 2^k + \ldots + n^k$ is $O(n^{k+1})$

$1^k + 2^k + \ldots + n^k \quad < \quad n^k + n^k + \ldots + n^k \quad = n*(n^k) = n^{k+1}$

Since the sum $(n^k + n^k + \ldots + n^k)$ is greater than the sum $(1^k + 2^k + \ldots + n^k)$ and it is clearly $O(n^{k+1})$ since the sum is, exactly, n^{k+1}, it necessarily follows that the smaller sum of $(1^k + 2^k + \ldots + n^k)$ is also $O(n^{k+1})$.

Problem Seven (2.2.24)

Note that the solutions for this problem demonstrate that there are two methods for solving these types of problems: one that is systematic and more "math-heavy" and one that that is fluid and relies more on English. Some of the solutions use the systematic method and some use the English method. The point is that both are valid methods for solving the problem.

a) Show that $(3x + 7)$ is $\theta(x)$

Let $f = 3x + 7$ and $g = x$

$f = O(g)$		$g = O(f)$	
$3x + 7 \leq^? 3x$	$3 \leq 3$ is always true so $3x \leq 3x$ is true for all x	$x \leq^? 3x$	$x \leq 3x$ is true for all $x > 0$.
$3x + 7 \leq^? 3x + x$	$7 \leq x$ is true when $x \geq 7$	$x \leq 3x + 7$	$0 \leq 7$ is always true
$3x + 7 \leq 4x$			

Therefore, let $C_1=4$, $k_1=7$, $C_2=1$, and $k_2=7$. With these values "$|f(x)| \leq C_1*|g(x)|$ when $n > k_1$" and "$|g(x)| \leq C_2*|f(x)|$ when $n > k_2$" are true statements and $f = O(g)$ and $g = O(f)$. There fore, $f = \theta(g)$.

b) Show that $(2x^2 + x - 7)$ is $\theta(x^2)$

Let $f = 2x^2 + x - 7$ and $g = x^2$

$f = O(g)$		$g = O(f)$	
$2x^2 + x - 7 \leq^? 2x^2$	$2 \leq 2$ is always true so $2x^2 \leq 2x^2$ is true for all x	$x^2 \leq^? 2x^2$	$x^2 \leq 2x^2$ is true for all x.
$2x^2 + x - 7 \leq^? 2x^2 + x^2$	$1 \leq x$ is true for $x \geq 1$ so $x \leq x^2$ is true for $x \geq 1$	$x^2 \leq 2x^2 + x - 7$	$0 \leq (x-7)$ for $x \geq 7$
$2x^2 + x - 7 \leq 3x^2$	$-7 < 0$ is always true, so the inequality is true		

Therefore, let $C_1=3$, $k_1=1$, $C_2=1$, and $k_2=7$. With these values "$|f(x)| \leq C_1*|g(x)|$ when $n > k_1$" and "$|g(x)| \leq C_2*|f(x)|$ when $n > k_2$" are true statements and $f = O(g)$ and $g = O(f)$. There fore, $f = \theta(g)$.

c) Show that $\text{floor}(x + \frac{1}{2})$ is $\theta(x)$

Let $f = \text{floor}(x + \frac{1}{2})$ and $g = x$

Notice if x has a fractional component less than ½, $\text{floor}(x + \frac{1}{2}) = x$, and if x has a fractional component greater than or equal to ½ then $\text{floor}(x + \frac{1}{2}) = x+1$, in either case, $\text{floor}(x + \frac{1}{2})$ is strictly less than $2x$ for any positive value of x. So, let $C=2$ and $k=1$ and we have that $f = O(g)$. By similar reasoning, x is inherently less than or equal $\text{floor}(x + \frac{1}{2})$ for any positive value of x. So, let $C=1$ and $k=1$ and we have that $g = O(f)$. Since $f=O(g)$ and $g=O(f)$, it is also true that $f = \theta(g)$.

d) Show that $\log_2(x^2 + 1)$ is $\theta(\log_2(x))$

Let $f = \log_2(x^2 + 1)$ and $g = \log_2(x)$

Notice that $\log_2(x^2 + 1)$ is strictly less than $\log_2(2x^2)$ for any positive value of x. $\log_2(2x^2) = \log_2(2) + \log_2(x^2) = 1 + 2\log^2(x)$. This value is definitely less than $3\log_2(x)$. So, let $C=3$ and $k=1$ and we have that $\log(2x^2) = O(g)$, therefore it must also be the case that $f = O(g)$ as well. Also, since x is inherently less than or equal $(x^2 + 1)$ simply let $C=1$ and $k=1$ and we have that $g = O(f)$. Since $f=O(g)$ and $g=O(f)$, it is also true that $f = \theta(g)$.

e) Show that $\log_{10}(x)$ is $\theta(\log_2(x))$

It is known fact of logs that a log of one base can be converted to the log of another base, therefore, since the parameters of the log are equal, simply let $C1=1/\log_2(10)$ and $C_2=1/\log_{10}(2)$. These values of C1 and C2 will, in fact, make the to logs equal, therefore it must be that they are theta of each other as well.

Problem Eight (2.3.8)

There is a more efficient algorithm (in terms of the number of multiplications and additions used) for evaluating polynomials than the conventional algorithm described in the previous exercise. It is called **Horner's method**. This pseudocode shows how to use this method to find the value of $a_nx^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0$ at $x = c$.

```
procedure Horner(c, a0, a1, a2, ... an: real numbers)
y := an
for j := 1 to n
        y := y * c + an-j
{y = ancn + an-1cn-1 + ... + a1c + a0}
```

a) *Evaluate $3x^2 + x + 1$ at $x=2$ by working through every step in the algorithm.*

On the initial call to the algorithm:

$n = 2$ $c=2$ $a_0 = 1$ $a_1 = 1$ $a_2 = 3$ $y = a_2 = 3$

first iteration (j=1)	second iteration (j=2)	FINAL ANSWER
$y = y*c + a_{n-1}$	$y = y*c + a_{n-2}$	$y = 15$
$y = 3*2 + a_1$	$y = 7*2 + a_0$	
$y = 3*2 + 1$	$y = 7*2 + 1$	
$y = 7$	$y = 15$	

check: $3(2^2) + 2 + 1 = (3*4) + 2 + 1 = 12 + 2 + 1 = 15$

b) *Exactly how many multiplications and additions are used by this algorithm to evaluate a polynomial of degree n at x=c? (Do not count additions used to increment the loop variable).*

- The loop iterates n times
- In one iteration of the loop:
    - 1 multiplication is done
    - 1 addition is done
- The total number of multiplications and additions done is $n*(1+1) = 2n$ operations total (n additions and n multiplications).

Problem Nine (2.3.10)

How much time does an algorithm take to solve a problem of size n if this algorithm uses $2n^2 + 2^n$ bit operations, each requiring 10^{-9} seconds, with these values of n?

a) 10 1.224×10^{-6} seconds

b) 20 $\approx 1.05 \times 10^{-3}$ seconds

c) 50 $\approx 1.13 \times 10^{6}$ seconds (roughly 13 **days** non-stop)

d) 100 ≈1.27 x 10^{21} seconds (roughly 4 x 1013 **years**, non-stop)

Problem Ten (2.3.12)

Determine the least number of comparisons, or best-case performance,

a) *required to find the maximum of a sequence of n integers, using the following algorithm:*

```
procedure max(a, a2, … , an: integers)
max := a1
for j:=2 to n
        if max < aj then max := aj
```

Since the algorithm simply runs through the list of numbers completely, it will always do exactly n-1 useful comparisons, therefore the best case performance is Ω(n) where n is the size of the list.

b) *used to locate an element in a list of n terms with a linear search.*

In the best case, the item we are looking for is the very first element in the list and we find it immediately. This means the best case running time of the algorithm is Ω(1).

c) *used to locate an element in a list of n terms using a binary search.*

Note that for binary search elements are assumed to be ordered in some way. Roughly speaking, a binary search divides the list in half every time the item is not found and discards the half known to not contain the element. For example, if the elements are sorted in ascending order and we are looking for element '5', if we pick an element, and it turns out to be '9', we know anything that comes after this element cannot be 5 (since 5 <9); so we ignore that half of the list. Since we are dividing the list in half each time, the number of needed steps will be $\Omega(\log_2(n))$ where n is the size of the list.