# COMP 350 Solutions to Assignment 2

1. (a)
```
#include <stdio.h>
int main(){

float x = 1;
float y = 1;
float z;
while (y == x) {
x++;
z = 1/x;
y = 1/z;
}

int result = x;
printf("The smallest positive integer x is %d\n", result);

return 0;
}

Result:
The smallest positive integer x is 7
```

   (b)
```
#include <stdio.h>
int main(){

double x = 1;
double y = 1;
double z;
while (y == x) {
x++;
z = 1/x;
y = 1/z;
}

int result = x;
printf("The smallest positive integer x is %d\n", result);

return 0;
}
Result:
The smallest positive integer x is 49
```

2. (a)

```
#include <stdio.h>
#include <math.h>

int main () {
int i;
float x = 1;

//i should be large enough to evaluate the limit
for (i = 1;i <= 70;i++) {
x = 100*x/i;
printf("x%d = %.6e\n",i,x);
}

return 0;
}
```

The output:

```
x1 = 1.000000e+02
x2 = 5.000000e+03
x3 = 1.666667e+05
x4 = 4.166667e+06
x5 = 8.333334e+07
x6 = 1.388889e+09
x7 = 1.984127e+10
x8 = 2.480159e+11
x9 = 2.755732e+12
x10 = 2.755732e+13
x11 = 2.505211e+14
x12 = 2.087676e+15
x13 = 1.605904e+16
x14 = 1.147075e+17
x15 = 7.647165e+17
x16 = 4.779478e+18
x17 = 2.811458e+19
x18 = 1.561921e+20
x19 = 8.220636e+20
x20 = 4.110318e+21
x21 = 1.957294e+22
```

```
x22 = 8.896793e+22
x23 = 3.868171e+23
x24 = 1.611738e+24
x25 = 6.446951e+24
x26 = 2.479597e+25
x27 = 9.183691e+25
x28 = 3.279890e+26
x29 = 1.130996e+27
x30 = 3.769988e+27
x31 = 1.216125e+28
x32 = 3.800391e+28
x33 = 1.151634e+29
x34 = 3.387158e+29
x35 = 9.677595e+29
x36 = 2.688221e+30
x37 = 7.265461e+30
x38 = 1.911963e+31
x39 = 4.902470e+31
x40 = 1.225617e+32
x41 = 2.989311e+32
x42 = 7.117406e+32
x43 = 1.655211e+33
x44 = 3.761843e+33
x45 = 8.359651e+33
x46 = 1.817315e+34
x47 = 3.866628e+34
x48 = 8.055475e+34
x49 = 1.643975e+35
x50 = 3.287949e+35
x51 = 6.446959e+35
x52 = 1.239800e+36
x53 = 2.339245e+36
x54 = 4.331935e+36
x55 = inf
x56 = inf
x57 = inf
x58 = inf
x59 = inf
x60 = inf
x61 = inf
```

```
x62 = inf
x63 = inf
x64 = inf
x65 = inf
x66 = inf
x67 = inf
x68 = inf
x69 = inf
x70 = inf
```

The final infinity result is due to overflow when single precision is used. To overcome the difficulty, we use double precision, see below.

(b)
```
#include <stdio.h>
#include <math.h>
int main() {
int i;
double x = 1;
for (i = 1;i <= 300;i++) {   //i should be large enough to evaluate the limit
x = 100*x/i;
printf("x%d=%.6e\n",i,x);
}
return 0;
}
```

The output:

```
x1=1.000000e+02
x2=5.000000e+03
x3=1.666667e+05
x4=4.166667e+06
x5=8.333333e+07
...
x295=7.678101e-13
x296=2.593953e-13
x297=8.733849e-14
x298=2.930822e-14
x299=9.802079e-15
x300=3.267360e-15
```

We could see the result is converge to 0, instead of $\infty$

3. (a)
```c
#include<stdio.h>
#include<math.h>
int main() {
int n;
double p;           /* store the current value of p(n) */
double t;           /* store the current value of 2^n  */
double temp;

p = 2 * sqrt(2);                    /* initiate values of g and t     */
t = 4;

for (n=2; n<35; n++) {
temp = p/t;
temp = 1.0 - temp * temp;
temp = 2 * (1.0 - sqrt(temp));
p = t * sqrt(temp);
t *= 2;
printf("p%d = %.15f\n",n+1,p);
}
return 0;
}
```

The output:

```
p3 = 3.061467458920719
p4 = 3.121445152258053
p5 = 3.136548490545941
p6 = 3.140331156954739
p7 = 3.141277250932757
p8 = 3.141513801144145
p9 = 3.141572940367883
p10 = 3.141587725279961
p11 = 3.141591421504635
p12 = 3.141592345611077
p13 = 3.141592576545004
p14 = 3.141592633463248
p15 = 3.141592654807589
p16 = 3.141592645321215
p17 = 3.141592607375720
p18 = 3.141592910939673
p19 = 3.141594125195191
```

```
p20 = 3.141596553704820
p21 = 3.141596553704820
p22 = 3.141674265021758
p23 = 3.141829681889202
p24 = 3.142451272494134
p25 = 3.142451272494134
p26 = 3.162277660168380
p27 = 3.162277660168380
p28 = 3.464101615137754
p29 = 4.000000000000000
p30 = 0.000000000000000
p31 = 0.000000000000000
p32 = 0.000000000000000
p33 = 0.000000000000000
p34 = 0.000000000000000
p35 = 0.000000000000000
```

We can see that estimates initially seem to converge, increasing toward the true value of $\pi$, but then deteriorate after the 15th iteration, then dramatically drop to zero. The reason for the failure of the formula is that when $n$ is large, the term $(p_n/2^n)^2$ is small, then $\sqrt{1-(p_n/2^n)^2}$ is close to 1 and there is cancellation error in the subtraction $1-\sqrt{1-(p_n/2^n)^2}$ (the first subtraction). The factor $2^n$ in the front of the expression of $p_{n+1}$ makes the error larger. When $n$ is large enough, $\text{round}(1-(p_n/2^n)^2) = 1$, so finally $p_n$ becomes zero.

(b) To avoid the cancellation issue in the original formula, we modify the formula. Let $g_n = (\frac{p_n}{2^{n-1}})^2$. Then $p_{n+1} = 2^n\sqrt{2(1-\sqrt{1-(p_n/2^n)^2})}$ can be written as

$$g_{n+1} = 2(1-\sqrt{1-g_n/4}) = \frac{(2-\sqrt{4-g_n})(2+\sqrt{4-g_n})}{(2+\sqrt{4-g_n})} = \frac{g_n}{2+\sqrt{4-g_n}}.$$

where the initial $g_2 = 2$. Note that the cancellation problem does not exist anymore. In the implementation, we do not use the function pow to avoid unnecessary computations.

program:

```
#include<stdio.h>
#include<math.h>

int main() {

int n;
double p;              /* store the current value of p(n) */
double g;              /* store the current value of g(n) */
double t;              /* store the current value of 2^n  */

g = 2;                 /* initiate values of g and t     */
t = 4;

for (n=2; n<40; n++) {
g = g/(2 + sqrt(4 - g));
p = t*sqrt(g);
t *= 2;
printf("p(%2i) = %.15f\n",n+1,p);
}
return 0;
}
```

output:

```
p( 3) = 3.061467458920718
p( 4) = 3.121445152258052
p( 5) = 3.136548490545939
p( 6) = 3.140331156954753
p( 7) = 3.141277250932773
p( 8) = 3.141513801144301
p( 9) = 3.141572940367091
p(10) = 3.141587725277160
p(11) = 3.141591421511200
p(12) = 3.141592345570118
p(13) = 3.141592576584872
p(14) = 3.141592634338563
```

```
p(15) = 3.141592648776986
p(16) = 3.141592652386591
p(17) = 3.141592653288993
p(18) = 3.141592653514593
p(19) = 3.141592653570993
p(20) = 3.141592653585093
p(21) = 3.141592653588618
p(22) = 3.141592653589500
p(23) = 3.141592653589720
p(24) = 3.141592653589775
p(25) = 3.141592653589789
p(26) = 3.141592653589793
p(27) = 3.141592653589794
p(28) = 3.141592653589794
p(29) = 3.141592653589794
p(30) = 3.141592653589794
p(31) = 3.141592653589794
p(32) = 3.141592653589794
p(33) = 3.141592653589794
p(34) = 3.141592653589794
p(35) = 3.141592653589794
p(36) = 3.141592653589794
p(37) = 3.141592653589794
p(38) = 3.141592653589794
p(39) = 3.141592653589794
p(40) = 3.141592653589794
```

We see because we avoided the cancellation, we have got high accurate approximations.