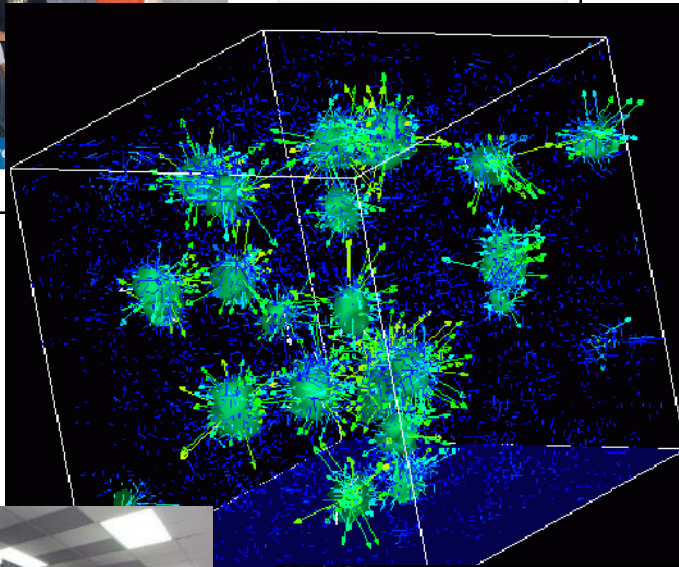




Survey of Scientific Computing (SciComp 301)

Dave Biersach
Brookhaven National
Laboratory
dbiersach@bnl.gov



Session 03
Loops, Conditionals,
Modulus

```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Windows.Forms;
9
10 namespace SimpleEvents
11 {
12     public partial class Form1 : Form
13     {
14         Person person = new Person();
15
16         public Form1()
17         {
18             InitializeComponent();
19             person.FirstName = "Christian";
20             person.LastName = "Pano";
21         }
22
23         private void button1_Click(object sender, EventArgs e)
24         {
25             person.MainColor = textBox1.Text;
26         }
27     }
28 }
```

Session Goals

- Introduce the **bool** data type and **logical operators**
- Use the **if()** statement for ***conditional*** code execution
- Learn about the **while()** loop
- Appreciate the **%** “modulus” (remainder) operator
- Generate a list of **perfect numbers**
- Create an algorithm to find **square roots**
- Consider one approach to handling very large integers
- Write code to **factor** any **quadratic** with *integer* coefficients
- Use **Simpson’s Rule** to calculate area under a polynomial

Logical Operators

- A variable of type **bool** (Boolean) can store only **true** or **false** values. The default value for a **bool** is **false**
- Use the **&&** operator to calculate a Boolean **AND**
 - $(A \ \&\& \ B) == \text{true}$ only if both A and B are **true**
- Use the **||** operator to calculate a Boolean **OR**
 - $(A \ || \ B) == \text{true}$ if either A or B are **true**
- Use the **!** operator to calculate a Boolean **NOT**
 - If $A == \text{true}$, then $!A == \text{false}$
 - If $A == \text{false}$, then $!A == \text{true}$

if() Statement

- An **if()** statement identifies which code block (scope) to run based upon the value of a **Boolean expression**
- The expression (the *condition*) between the parenthesis **must evaluate** to either a **true** or a **false** value
- If the condition is **true**, then the scope immediately following the **if()** statement is executed
- If the condition is **false**, and there is an **else** clause, then the scope immediately following the **else** statement is executed
- Every **if()** statements does not need to have an **else** clause

Two types of `if()` Statements

An `if()` without an else

```
// if statement without an else
if (condition)
{
    then-statement;
}
// Next statement in the program.
```

An `if()` with an else

```
// if-else statement
if (condition)
{
    then-statement;
}
else
{
    else-statement;
}
// Next statement in the program.
```

while() Loop

- A **while()** loop executes all the statements within its scope as long as loop conditional remains **true**

The loop
conditional

```
while (abs(estimateSquared - x) > epsilon)
{
    if (estimateSquared > x)
        highEnd = estimate;
    else
        lowEnd = estimate;

    estimate = (highEnd + lowEnd) / 2;
    estimateSquared = pow(estimate, 2);

    if (highEnd == lowEnd)
        break;
}
```

$$|estimate^2 - x| > \epsilon$$

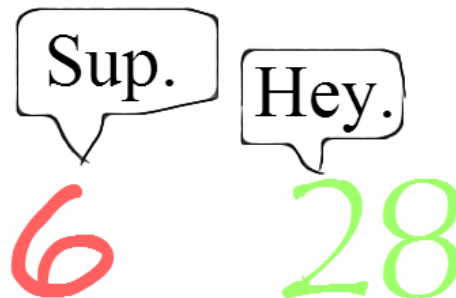
The Modulus (%) Operator

- The “**mod**” operator (%) returns the integer **remainder** of an implicit division operation, e.g. **37 % 5 = 2**
- Use double equals operator (==) when testing for **equality**

```
int sumOfFactors{ 1 };  
for (int factor{ 2 }; factor < n; factor++)  
    if (n % factor == 0)  
        sumOfFactors += factor;
```

Perfect Numbers

- Write a program to calculate and display all the perfect numbers n ($n \in \mathbb{Z}^+$) between **2** and **10,000**
- An integer n is **perfect** when the sum of *almost all* of its divisors (including **1**, but not including n itself) is equal to n
- Example: **6 = 1 + 2 + 3**



Perfect Numbers

Number	Positive Factors	Sum of all factors excluding itself
1	1	0
2	1, 2	1
3	1, 3	1
4	1, 2, 4	3
5	1, 5	1
6	1, 2, 3, 6	6 Perfect!
7	1, 7	1
8	1, 2, 4, 8	7
9	1, 3, 9	4
10	1, 2, 5, 10	8
11	1, 11	1
12	1, 2, 3, 4, 6, 12	16

Edit Lab 1 – Perfect Numbers

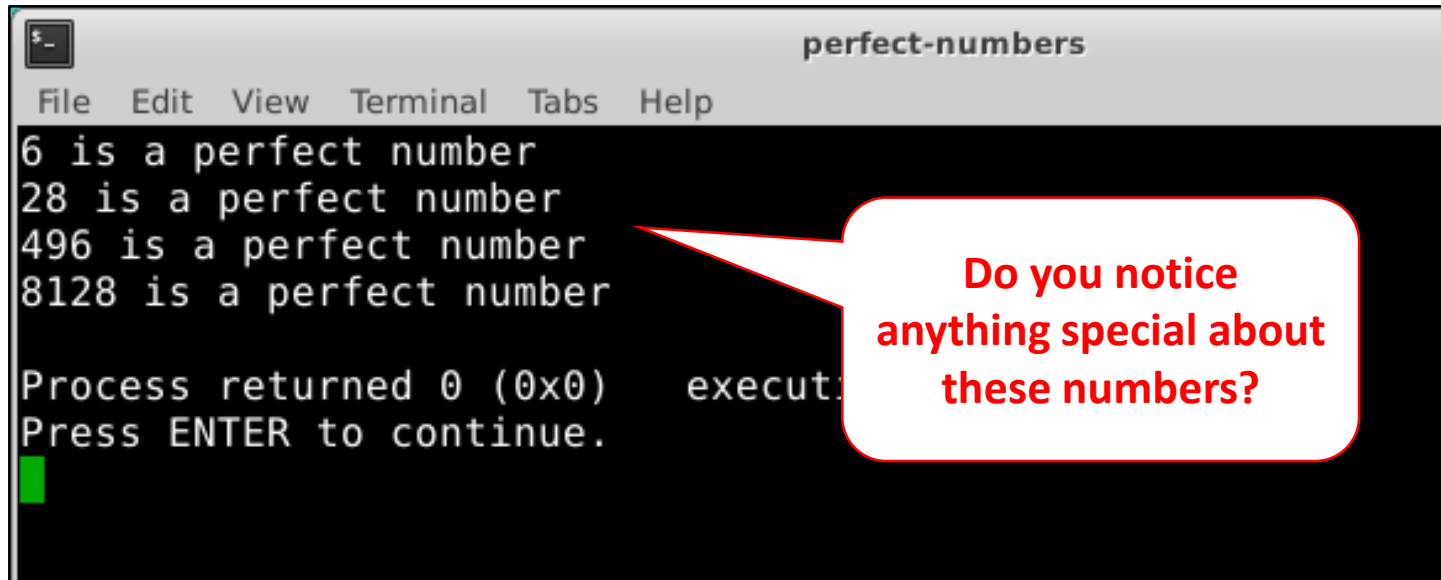
```
main.cpp [X]
1  #include "stdafx.h"
2
3  using namespace std;
4
5  int main()
6  {
7      for (int n{ 2 }; n < 10000; ++n)
8      {
9          int sumOfFactors{ 1 };
10
11         // Insert your code here
12
13         if (sumOfFactors == n)
14             cout << n << " is a perfect number"
15                 << endl;
16     }
17     return 0;
18 }
19
```



Run Lab 1 – Perfect Numbers

```
main.cpp [x]
1  #include "stdafx.h"
2
3  using namespace std;
4
5  int main()
6  {
7      for (int n{ 2 }; n < 10000; ++n)
8      {
9          int sumOfFactors{ 1 };
10
11         for (int factor{ 2 }; factor < n; factor++)
12             if (n % factor == 0)
13                 sumOfFactors += factor;
14
15         if (sumOfFactors == n)
16             cout << n << " is a perfect number"
17                 << endl;
18     }
19     return 0;
20 }
```

Check Lab 1 – Perfect Numbers



A terminal window titled "perfect-numbers" with a menu bar (File, Edit, View, Terminal, Tabs, Help). The terminal output shows:

```
6 is a perfect number
28 is a perfect number
496 is a perfect number
8128 is a perfect number

Process returned 0 (0x0)   execut
Press ENTER to continue.
```

A red callout box with a pointer to the list of numbers contains the text: "Do you notice anything special about these numbers?"

Bonus points: Given a perfect number n , what is the **sum** of the **reciprocals** of its divisors (including **1** and **n**) ?

Perfect Numbers

*Euclid–Euler
theorem*

$n = 2^{(p-1)}(2^p - 1)$ is perfect

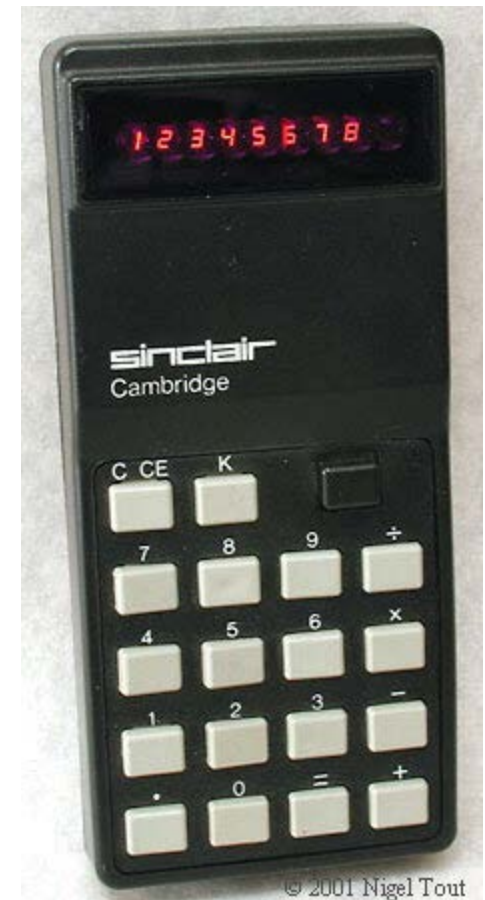
if and only if $\{p, (2^p - 1)\} \in \text{primes}$

p	$2^p - 1$	n
2	3	6
3	7	28
5	31	496
7	127	8,128
11	2,047	2,096,128
13	8,191	33,550,336
17	131,071	8,589,869,056

2047 = 23 x 89

Old School Square Roots

- My first calculator back in 1977 could only add, subtract, multiply, and divide
- As a 6th grader, I had heard of “Square Roots” and I knew that $\sqrt{25} = 5$.
- But what is $\sqrt{1977}$?
- How can we find the square root of a number using only the *elementary* (+, -, *, /) operations?
- Newton had solved that **313** years before me!

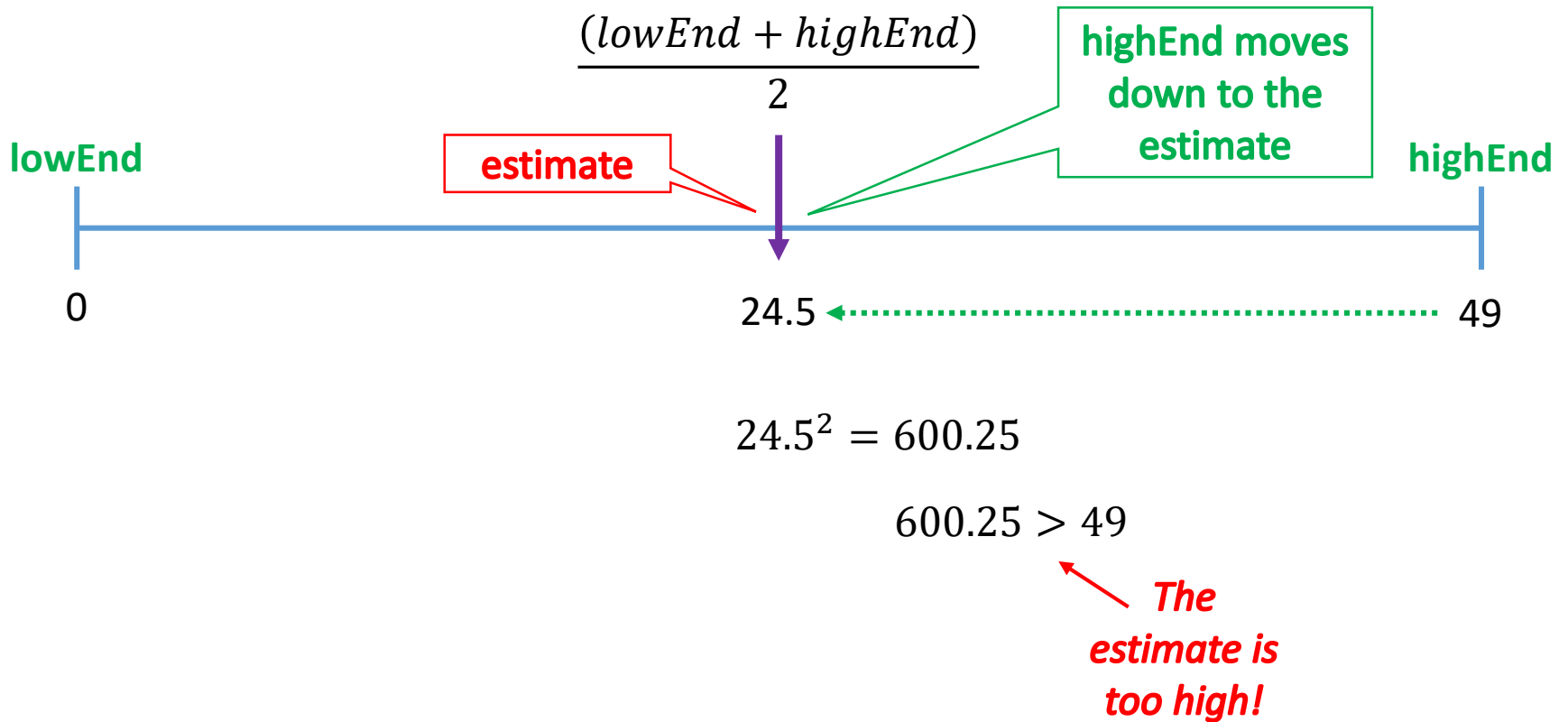


Old School Square Roots

- Newton's method for calculating the square root of any real x involves keeping track of “low end” and “high end” brackets which get successively closer to the actual root
 - We start with the $lowEnd = 0$ and $highEnd = x$
 - The process **brackets inward** keeping $lowEnd \leq \sqrt{x} \leq highEnd$
- During each loop iteration, our *estimate* is the **mean** of the current $lowEnd$ & the $highEnd$ brackets
 - Then if the $estimate^2 > x$, move $highEnd$ down to *estimate*
 - Alternatively, if the $estimate^2 < x$, move $lowEnd$ up to *estimate*
- Stop when the $| (estimate^2 - x) | \leq \epsilon$

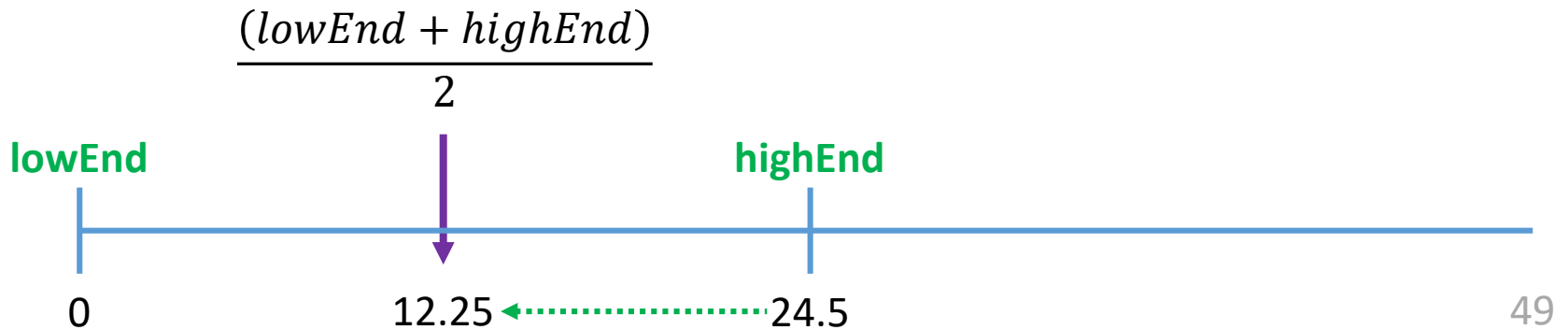
Newton's Method for $\sqrt{49}$

$$\varepsilon = .0003$$



Newton's Method for $\sqrt{49}$

$$\varepsilon = .0003$$



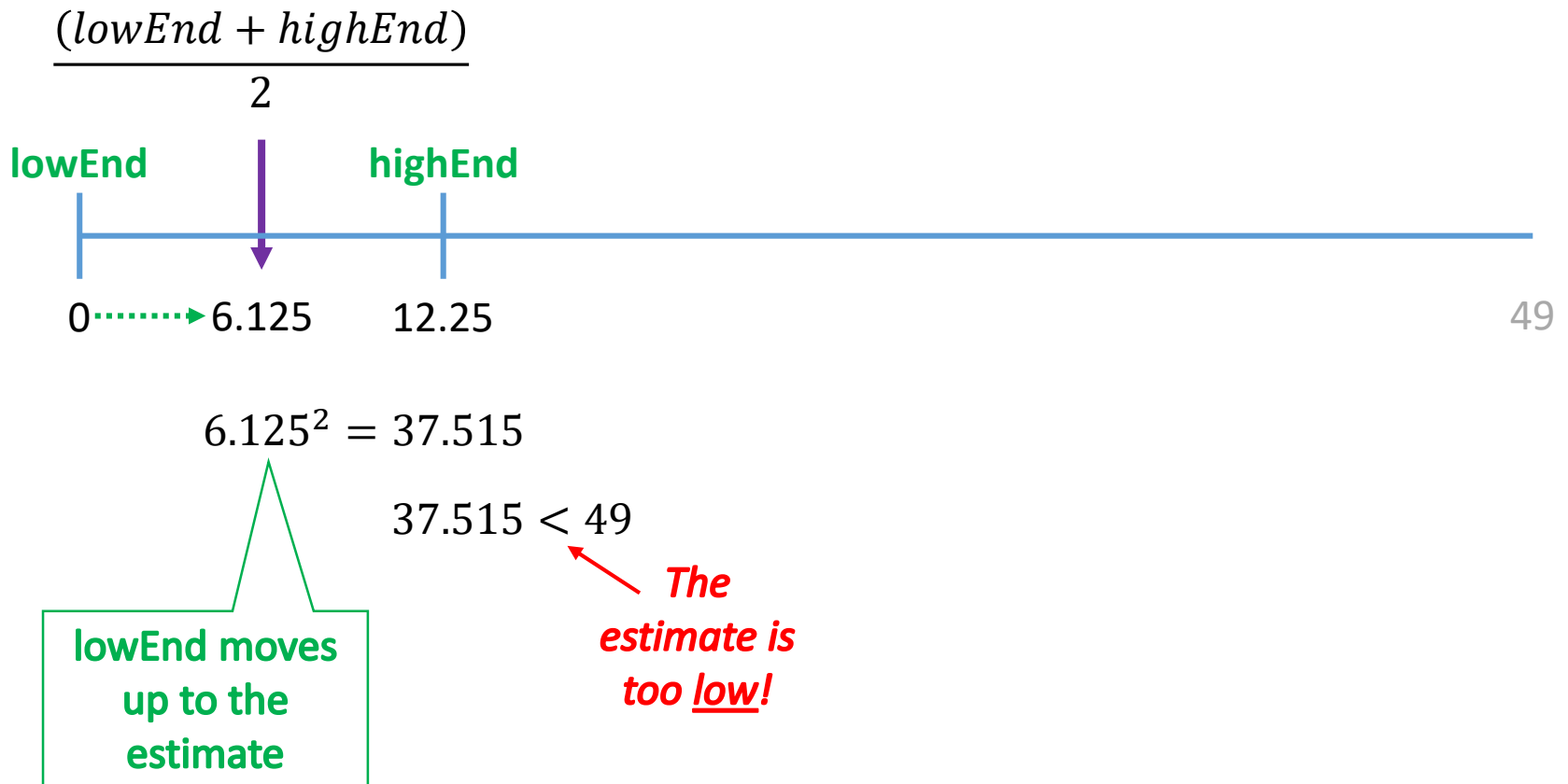
$$12.25^2 = 150.0625$$

$$150.0625 > 49$$

*The
estimate is
too high!*

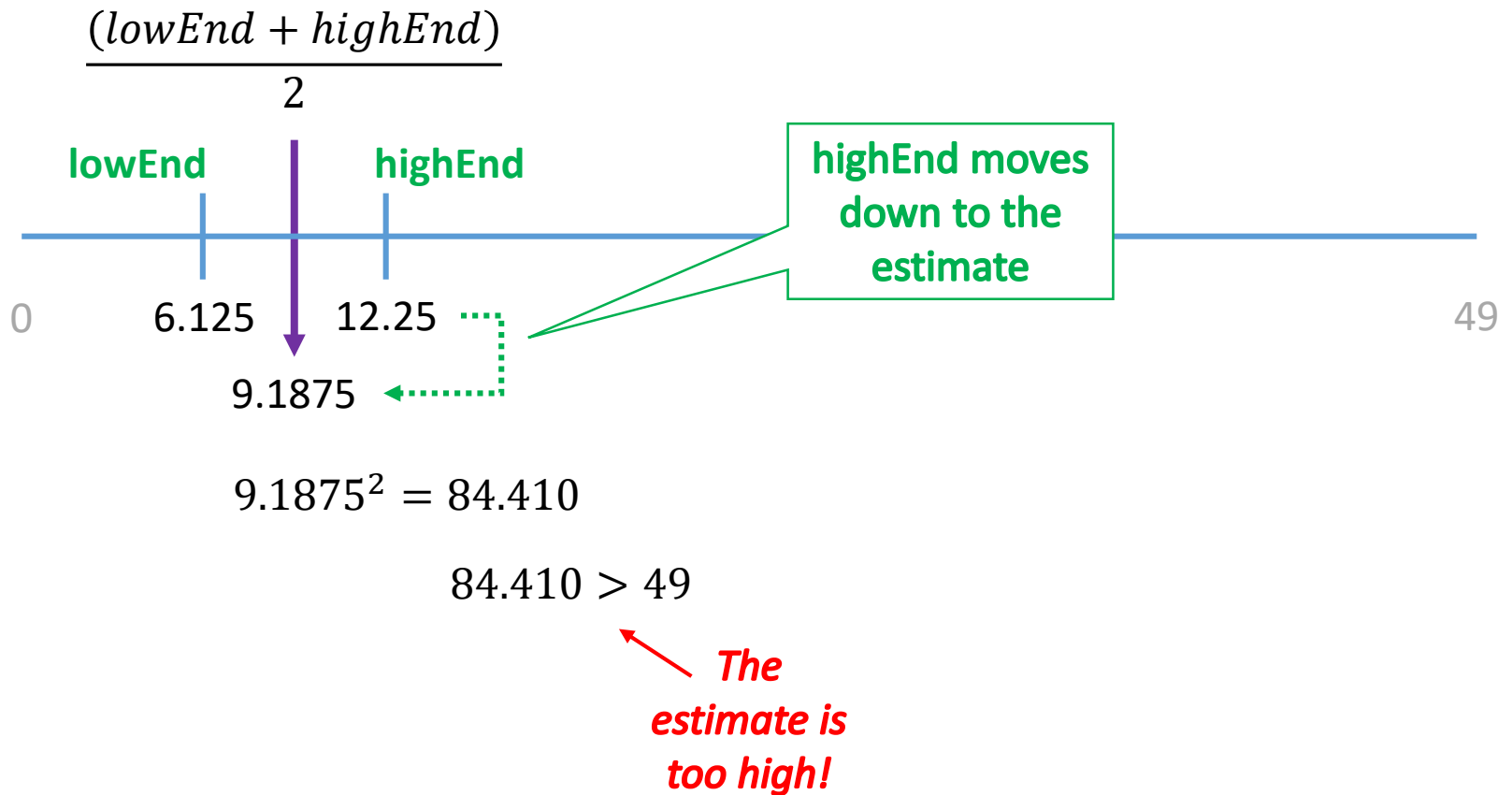
Newton's Method for $\sqrt{49}$

$$\varepsilon = .0003$$



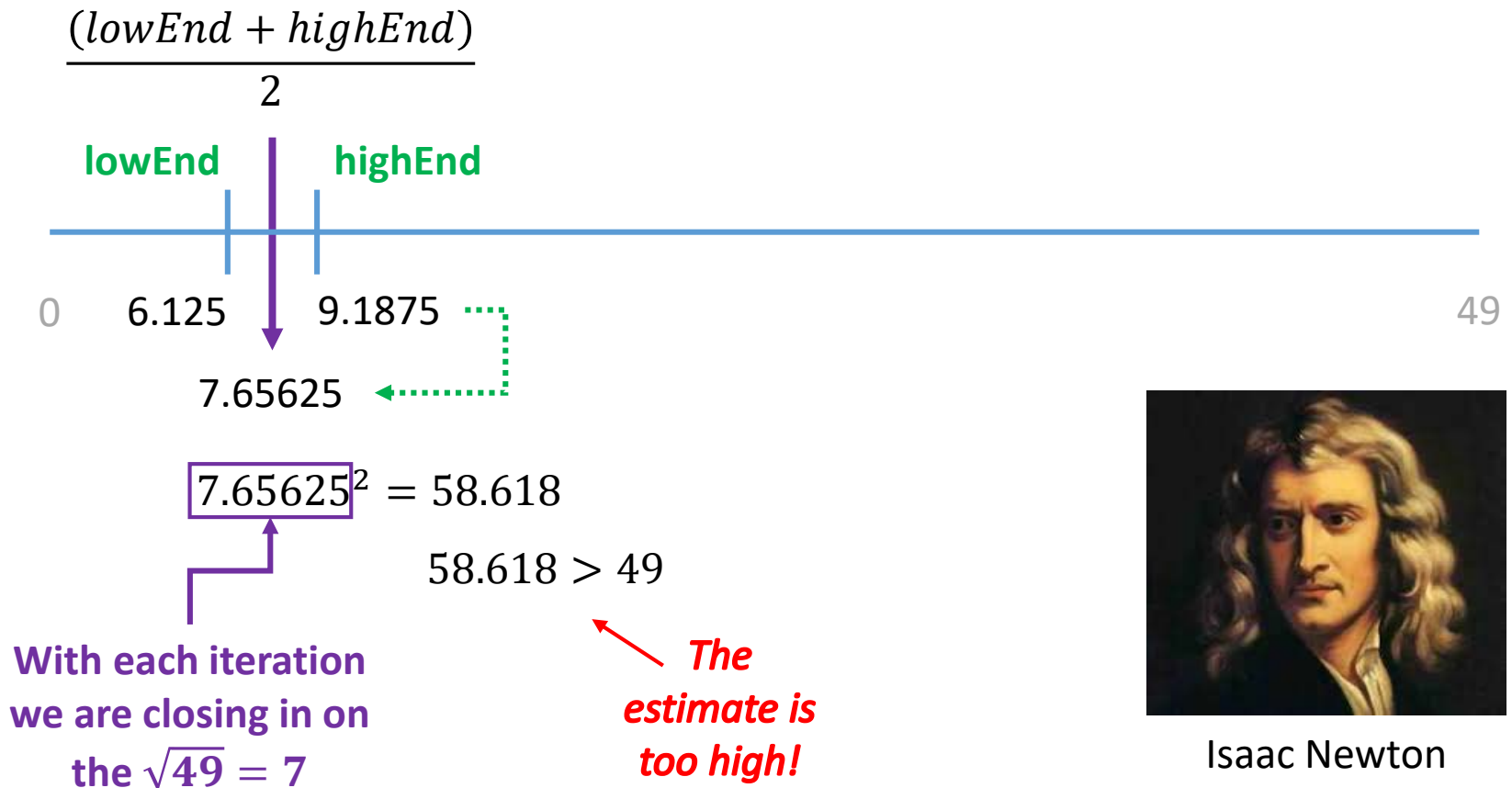
Newton's Method for $\sqrt{49}$

$\varepsilon = .0003$



Newton's Method for $\sqrt{49}$

$$\varepsilon = .0003$$



Isaac Newton
(1642-1726)

Newton's Method for $\sqrt{49}$

$\varepsilon = .0003$

lowEnd	highEnd	estimate	estimate ²	error	result
0.00000000	49.00000000	24.50000000	600.25000000	551.25000000	Too high
0.00000000	24.50000000	12.25000000	150.06250000	101.06250000	Too high
0.00000000	12.25000000	6.12500000	37.51562500	-11.48437500	Too low
6.12500000	12.25000000	9.18750000	84.41015625	35.41015625	Too high
6.12500000	9.18750000	7.65625000	58.61816406	9.61816406	Too high
6.12500000	7.65625000	6.89062500	47.48071289	-1.51928711	Too low
6.89062500	7.65625000	7.27343750	52.90289307	3.90289307	Too high
6.89062500	7.27343750	7.08203125	50.15516663	1.15516663	Too high
6.89062500	7.08203125	6.98632813	48.80878067	-0.19121933	Too low
6.98632813	7.08203125	7.03417969	49.47968388	0.47968388	Too high
6.98632813	7.03417969	7.01025391	49.14365983	0.14365983	Too high
6.98632813	7.01025391	6.99829102	48.97607714	-0.02392286	Too low
6.99829102	7.01025391	7.00427246	49.05983271	0.05983271	Too high
6.99829102	7.00427246	7.00128174	49.01794598	0.01794598	Too high
6.99829102	7.00128174	6.99978638	48.99700932	-0.00299068	Too low
6.99978638	7.00128174	7.00053406	49.00747709	0.00747709	Too high
6.99978638	7.00053406	7.00016022	49.00224307	0.00224307	Too high
6.99978638	7.00016022	6.99997330	48.99962616	-0.00037384	Too low
6.99997330	7.00016022	7.00006676	49.00093461	0.00093461	Too high
6.99997330	7.00006676	7.00002003	49.00028038	0.00028038	Too high

Open Lab 2 – Newton's Square Root

- Write a program to calculate the square root of a given **double** x , using only **elementary** (+, -, *, /) operations
- Use Newton's method to display the value of $\sqrt{168923}$
- Use $\varepsilon = 1 \times 10^{-14}$
- Your current $estimate = \frac{(highEnd + lowEnd)}{2}$
- If $(estimate)^2 > x$ then $highEnd$ value must move **down** to $estimate$
- If $(estimate)^2 < x$ then $lowEnd$ value must move **up** to $estimate$

Edit Lab 2

Newton's Square Root

main.cpp

```
5  int main()
6  {
7      double x{ 168923 };
8
9      double lowEnd{};
10     double highEnd{ x };
11
12     double estimate = (highEnd + lowEnd) / 2;
13     double estimateSquared = pow(estimate, 2);
14
15     double epsilon{ 1e-14 };
16
17     while (abs(estimateSquared - x) > epsilon)
18     {
19         if (estimateSquared > x)
20             highEnd = 0;
21         else
22             lowEnd = 0;
23
24         estimate = (highEnd + lowEnd) / 2;
25         estimateSquared = pow(estimate, 2);
26
27         if (highEnd == lowEnd)
28             break;
29     }
30
31     cout << "Estimated Square Root of "
32          << x << " = " << fixed
33          << setprecision(14) << estimate
34          << endl;
35
36     return 0;
37 }
38
```

Fix these two
lines of code

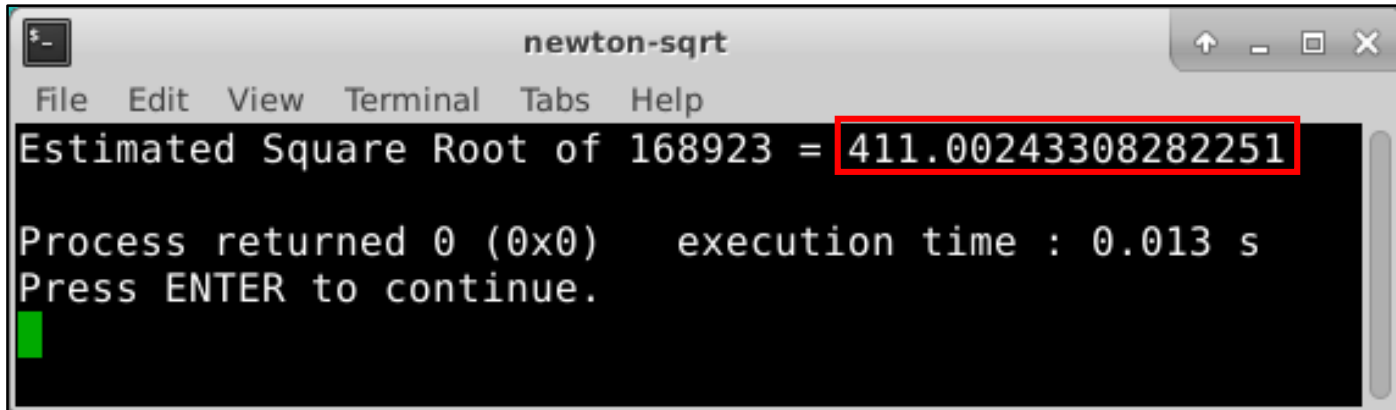


Run Lab 2

Newton's Square Root

```
main.cpp
5  int main()
6  {
7      double x{ 168923 };
8
9      double lowEnd{};
10     double highEnd{ x };
11
12     double estimate = (highEnd + lowEnd) / 2;
13     double estimateSquared = pow(estimate, 2);
14
15     double epsilon{ 1e-14 };
16
17     while (abs(estimateSquared - x) > epsilon)
18     {
19         if (estimateSquared > x)
20             highEnd = estimate;
21         else
22             lowEnd = estimate;
23
24         estimate = (highEnd + lowEnd) / 2;
25         estimateSquared = pow(estimate, 2);
26
27         if (highEnd == lowEnd)
28             break;
29     }
30
31     cout << "Estimated Square Root of "
32          << x << " = " << fixed
33          << setprecision(14) << estimate
34          << endl;
35
36     return 0;
37 }
38
```


Check Lab 2 – Newton's Square Root



```
newton-sqrt
File Edit View Terminal Tabs Help
Estimated Square Root of 168923 = 411.00243308282251
Process returned 0 (0x0)    execution time : 0.013 s
Press ENTER to continue.
```

A terminal window titled "newton-sqrt" with a menu bar (File, Edit, View, Terminal, Tabs, Help) and standard window controls. The output shows the estimated square root of 168923 as 411.00243308282251, which is highlighted with a red box. Below this, it states "Process returned 0 (0x0) execution time : 0.013 s" and "Press ENTER to continue." with a green cursor.

Roots of Googol

- Create a program to calculate the **integer square root** of a number with **100** random digits
 - This is bigger than a **googol** (1×10^{100})
 - The program can still use **Newton's method** to calculate $\sim \lfloor \sqrt{x} \rfloor$
- Specifically, the code must compute the **mean** of two very large **Base 10** numbers represented as **vectors** of digits
 - The code will implement *column wise* addition and multiplication, just as you learned in grade school
 - The challenge is there is an **add()** & **multiply()** function available for big integers, but there is no **divide()** function 😊
 - The mean of a & b = $(a+b)/2 = (a+b)*5/10$ and the “/10” is simulated by simply *shifting* all digits one position **to the right!**

Treating Large Integers as vector<int>

		b	21,985
		a	6,443

pos	00	01	02	03	04	05	06	07	08
vector<int> b	2	1	9	8	5				
vector<int> a	6	4	4	3					

Reverse b	5	8	9	1	2				
Reverse a	3	4	4	6					

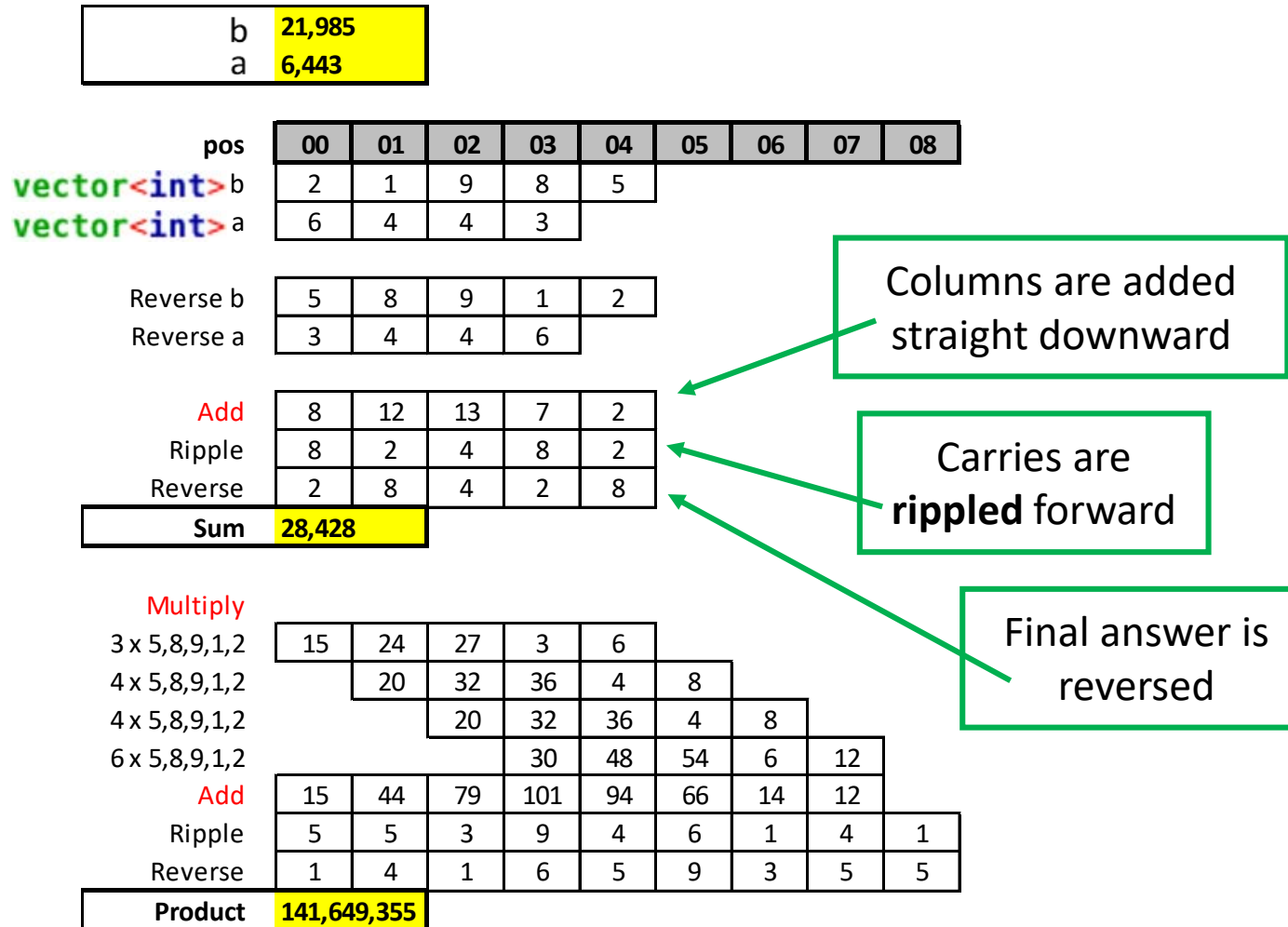
Add	8	12	13	7	2				
Ripple	8	2	4	8	2				
Reverse	2	8	4	2	8				
Sum	28,428								

Multiply	3 x 5,8,9,1,2	15	24	27	3	6				
	4 x 5,8,9,1,2		20	32	36	4	8			
	4 x 5,8,9,1,2			20	32	36	4	8		
	6 x 5,8,9,1,2				30	48	54	6	12	
Add		15	44	79	101	94	66	14	12	
Ripple		5	5	3	9	4	6	1	4	1
Reverse		1	4	1	6	5	9	3	5	5
Product	141,649,355									

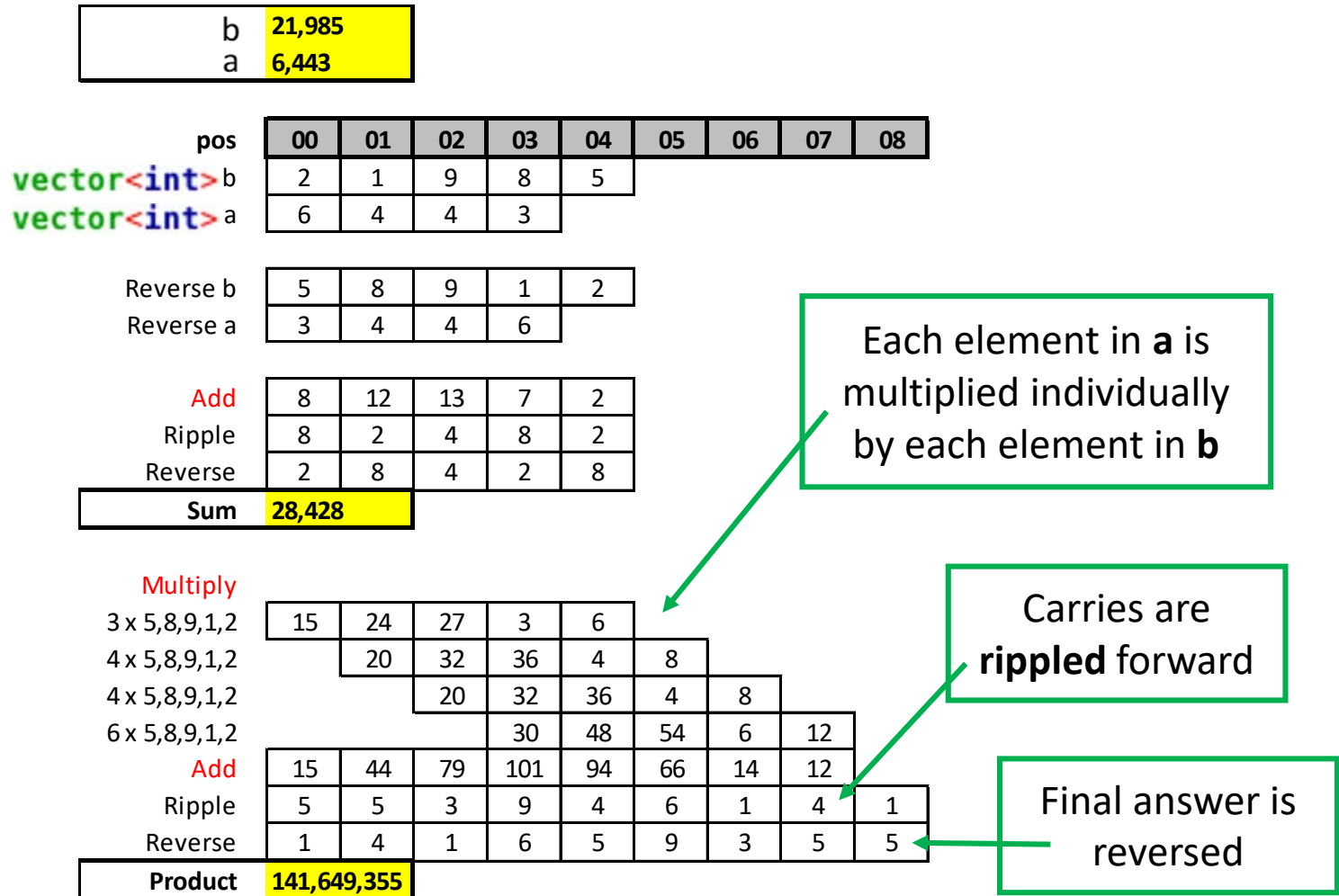
vectors are arranged so **b** is longer than **a**

vectors are reversed to easily align places

Treating Large Integers as vector<int>



Treating Large Integers as vector<int>



Open Lab 3 – Big Integer Square Root

```
main.cpp [x]
1  #include "stdafx.h"
2
3  using namespace std;
4
5  static vector<int> five{ 5 };
6
7  vector<int>* getDigits(const string& s)
8  {
9      size_t len = s.size();
10     vector<int>* digits = new vector<int>(len);
11     for (size_t i{}; i < len; ++i)
12         digits->at(i) = s.at(len - i - 1) - '0';
13     return digits;
14 }
15
16 string makeString(const vector<int>* digits)
17 {
18     string s{};
19     for (size_t i{ digits->size() }; i > 0; --i)
20         s += digits->at(i - 1) + '0';
21     while (s.size() > 1 && s.at(0) == '0')
22         s.erase(0, 1);
23     return s;
24 }
25
```

This function creates a vector<int> from the “digits” in a string

This function creates a string from a vector<int> of digits

View Lab 3 – Big Integer Square Root

```
129 int main()
130 {
131     seed_seq seed{ 2016 };
132     default_random_engine generator{ seed };
133     uniform_int_distribution<int> distribution(0, 9);
134
135     string s = "1";
136     for (int i{}; i < 99; ++i)
137         s += distribution(generator) + '0';
138
139     cout << "The Integer Square Root of "
140          << endl << endl
141          << s << endl << endl
142          << "is" << endl << endl;
143
144     cout << intSqrt(getDigits(s))
145          << endl << endl;
146
147     return 0;
148 }
149
```

This code creates
a “number” from
a string containing
100 random digits

View Lab 3 – Big Integer Square Root

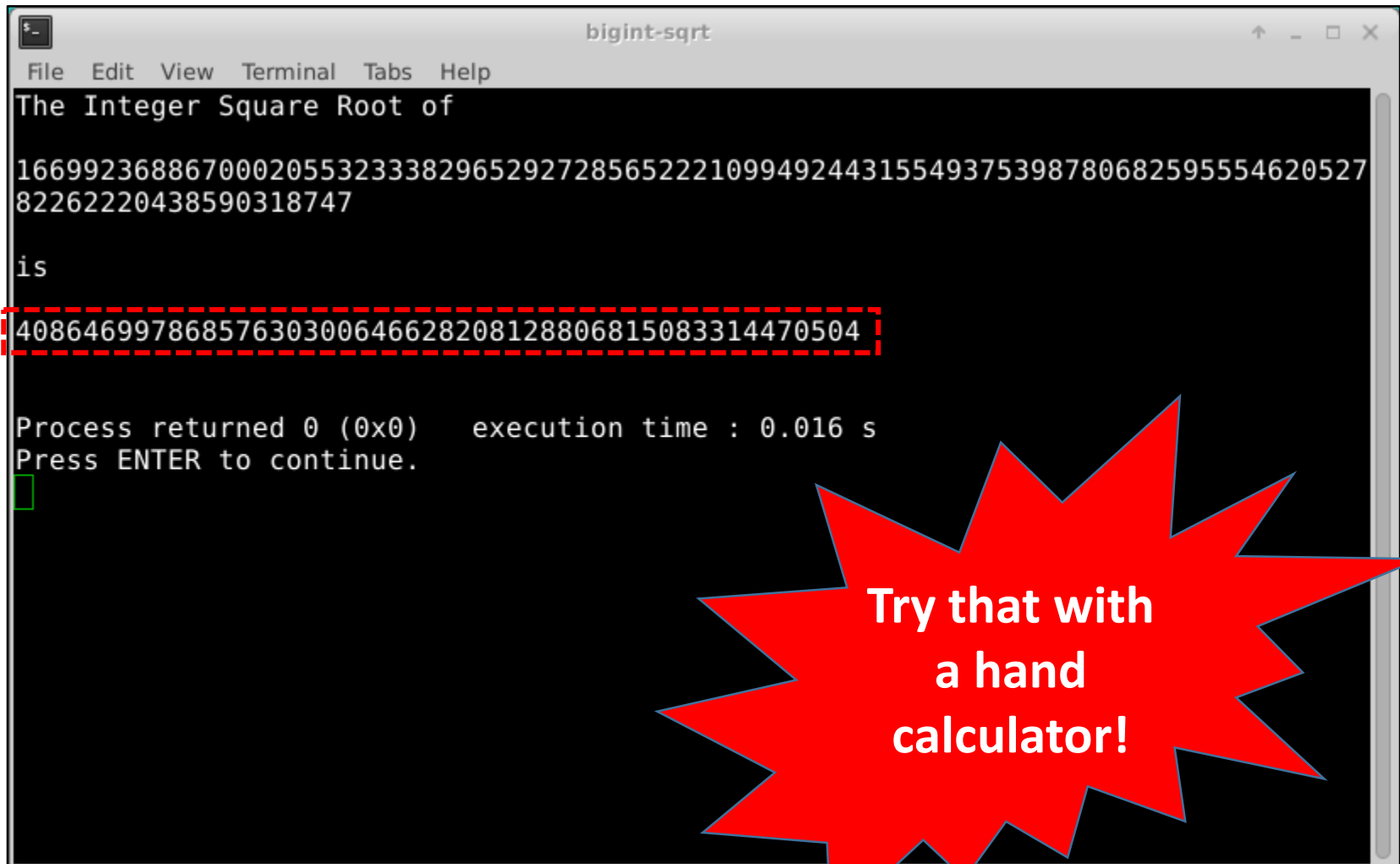
```

98  string intSqrt(vector<int>* x)
99  {
100      vector<int>* lowEnd = new vector<int>{ 0 };
101      vector<int>* highEnd = x;
102
103      vector<int>* lastEstimate = new vector<int>{ 0 };
104
105      vector<int>* estimate = average(lowEnd, highEnd);
106
107      while (!isEqual(lastEstimate, estimate))
108      {
109          vector<int>* estimateSquared = multiply(estimate, estimate);
110
111          if (isGreater(estimateSquared, x))
112              highEnd = estimate;
113          else
114              lowEnd = estimate;
115
116          lastEstimate = estimate;
117          estimate = average(lowEnd, highEnd);
118          delete estimateSquared;
119      }
120
121      string s = makeString(estimate);
122      delete estimate;
123      delete highEnd;
124      delete lowEnd;
125
126      return s;
127  }
128
129  vector<int>* average(vector<int>* x, vector<int>* y)
130  {
131      vector<int>* z = multiply(&five, add(x, y));
132      z->erase(z->begin());
133      return z;
134  }
```

We can still use Newton's algorithm!

Shift digits right to simulate divide by 10

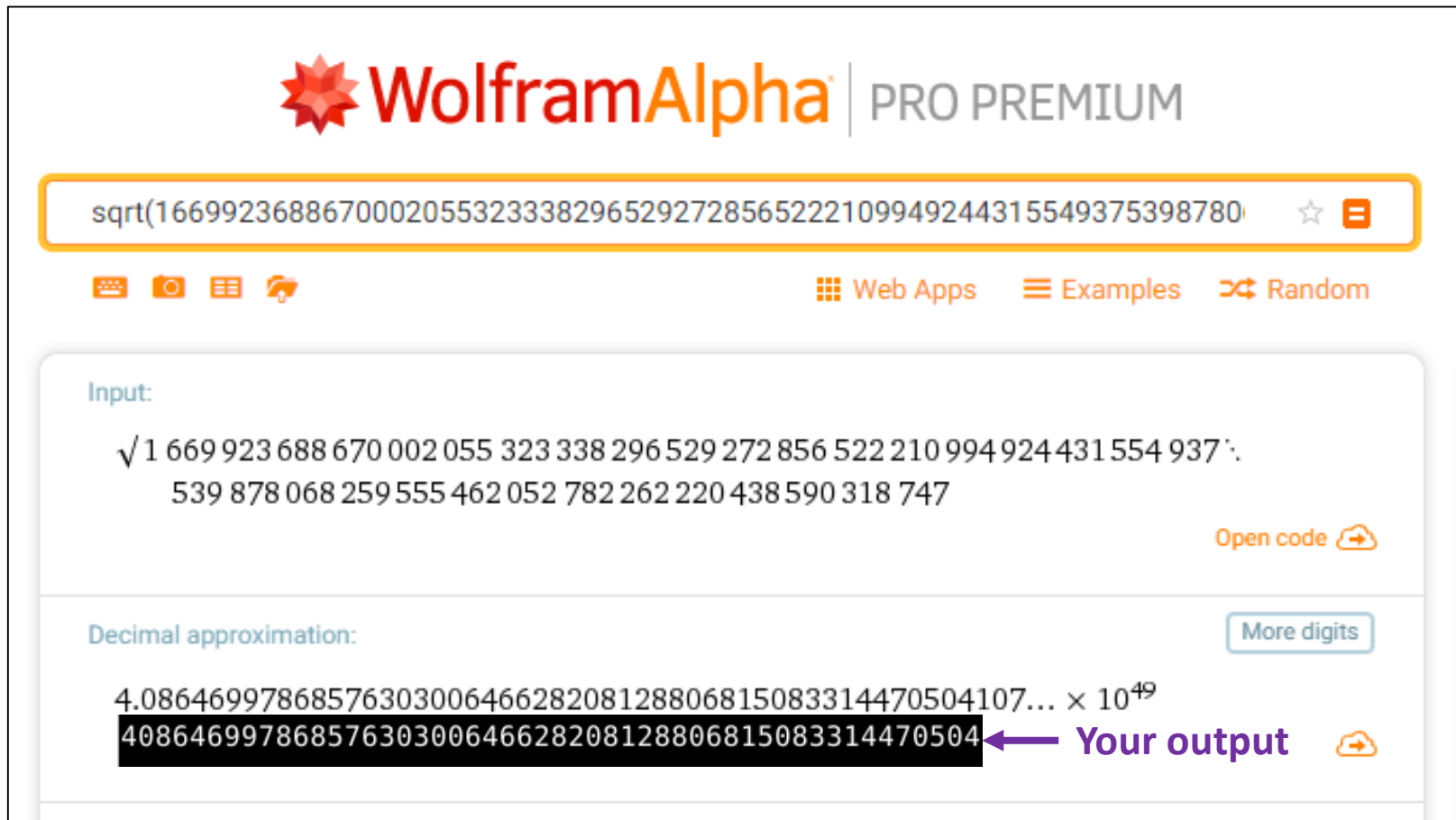
Run Lab 3 – Big Integer Square Root



```
bigint-sqrt
File Edit View Terminal Tabs Help
The Integer Square Root of
16699236886700020553233382965292728565222109949244315549375398780682595554620527
82262220438590318747
is
40864699786857630300646628208128806815083314470504
Process returned 0 (0x0)    execution time : 0.016 s
Press ENTER to continue.
█
```

**Try that with
a hand
calculator!**

Check Lab 3 – Big Integer Square Root



The screenshot shows the WolframAlpha PRO PREMIUM interface. At the top, the WolframAlpha logo is followed by "PRO PREMIUM". Below this is a search bar containing the input `sqrt(16699236886700020553233382965292728565222109949244315549375398780539878068259555462052782262220438590318747)`. Below the search bar are icons for keyboard, camera, list, and voice input, and links for "Web Apps", "Examples", and "Random". The "Input:" section shows the same large integer under a square root symbol. To the right of the input is an "Open code" link. The "Decimal approximation:" section shows the result $4.0864699786857630300646628208128806815083314470504107... \times 10^{49}$. Below this, the integer part of the result is highlighted in a black box: `40864699786857630300646628208128806815083314470504`. A purple arrow points from the text "Your output" to this highlighted box. To the right of the highlighted box is a "More digits" button and a cloud icon.

Always seek independent *confirmation* of your program's output!

Representing a Quadratic Polynomial

- The Fundamental Theorem of Algebra shows a polynomial of degree **2** will have exactly **2** roots
 - $Jx^2 + Kx + L = 0$, the roots can be unique or repeated
 - Either one (or both) of the roots can be a real or a complex number
- Assume we have factored a quadratic polynomial

$$(ax + b)(cx + d) = 0$$

$$(ac)x^2 + (ad + bc)x + (bd) = 0$$


$$Jx^2 + Kx + L = 0$$

$$J = (ac), \quad K = (ad + bc), \quad L = (bd)$$

Factoring a Quadratic Polynomial

$$Jx^2 + Kx + L = 0$$

$$J = (ac), \quad K = (ad + bc), \quad L = (bd)$$

- To factor J we need to try every integer a where $1 \leq a \leq J$
 - If $J \% a == 0$ (no remainder) then set $c = J / a$
- To factor L we need to try every integer b where $1 \leq b \leq L$
 - If $L \% b == 0$ (no remainder) then set $d = L / b$
- If $(ad + bc) == K$ then we have found a factorization!
 - If the sum does not equal K , then we have to keep trying more factors

Lab 4 – Factor Quadratic Polynomial

- Write a C++ console application to display **only** (but all) correct factorizations of a given quadratic polynomial

$$Jx^2 + Kx + L = 0$$

- You may assume in all cases $\{J, K, L\} \in \mathbb{Z}^+$
- Your scientist needs you to factor **this quadratic**:

$$115425x^2 + 3254121x + 379020$$

Edit Lab 4 – Factor Quadratic Polynomial

```
main.cpp
1  #include "stdafx.h"
2
3  using namespace std;
4
5  int main()
6  {
7      int J{ 115425 };
8      int K{ 3254121 };
9      int L{ 379020 };
10
11     cout << "Given the quadratic:" << endl
12         << J << "x^2 + " << K << "x + " << L
13         << " = 0" << endl << endl
14         << "The factors are:"
15         << endl << endl;
16
17     // TODO: Insert your code here
18
19     return 0;
20 }
21
22
```



```

1  #include "stdafx.h"
2
3  using namespace std;
4
5  int main()
6  {
7      int J{ 115425 };
8      int K{ 3254121 };
9      int L{ 379020 };
10
11     cout << "Given the quadratic:" << endl
12         << J << "x^2 + " << K << "x + " << L
13         << " = 0" << endl << endl
14         << "The factors are:"
15         << endl << endl;
16
17     for (int a{ 1 }; a <= J; ++a)
18     {
19         if (J % a == 0)
20         {
21             int c = J / a;
22             for (int b{ 1 }; b <= L; ++b)
23             {
24                 if (L % b == 0)
25                 {
26                     int d = L / b;
27                     if (a*d + b*c == K)
28                     {
29                         cout << "(" << a << "x + " << b << ")"
30                             << "(" << c << "x + " << d << ")"
31                             << endl;
32                     }
33                 }
34             }
35         }
36     }
37
38     return 0;
39 }

```

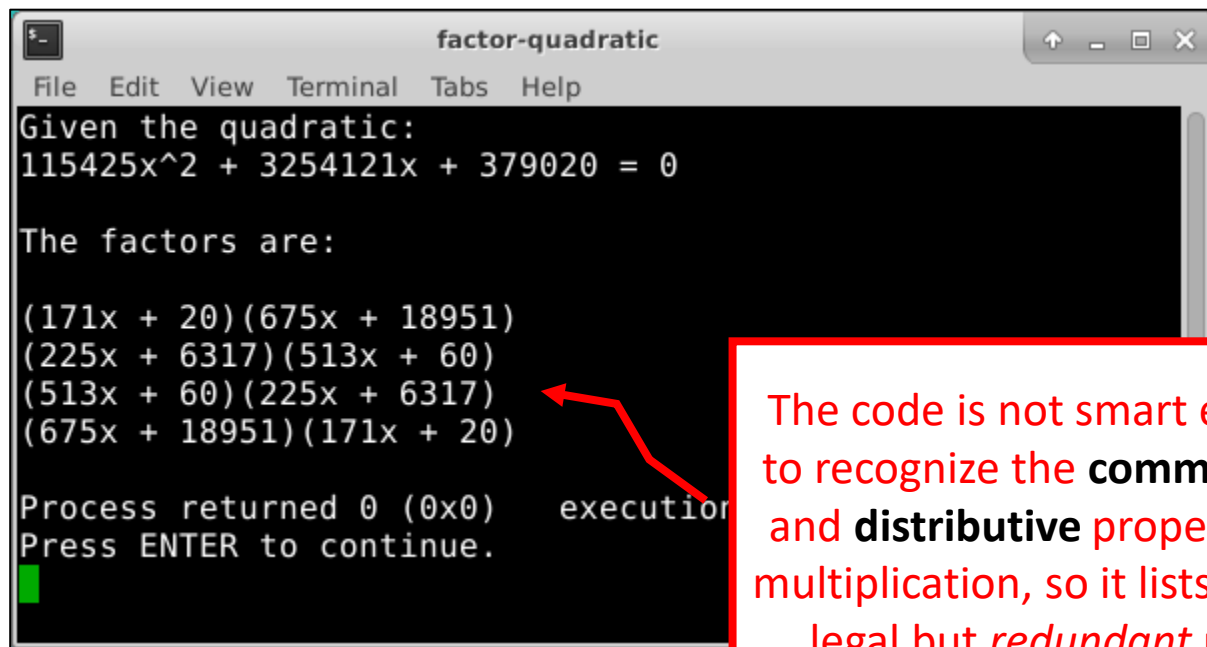
Run Lab 4

Factor Quadratic Polynomial

Enter all this code carefully - watch out for punctuation marks!

Check Lab 4 – Factor Quadratic Polynomial

$$115425x^2 + 3254121x + 379020$$



```
factor-quadratic
File Edit View Terminal Tabs Help
Given the quadratic:
115425x^2 + 3254121x + 379020 = 0

The factors are:

(171x + 20)(675x + 18951)
(225x + 6317)(513x + 60)
(513x + 60)(225x + 6317)
(675x + 18951)(171x + 20)

Process returned 0 (0x0)    execution
Press ENTER to continue.
█
```

The code is not smart enough to recognize the **commutative** and **distributive** properties of multiplication, so it lists several legal but *redundant* roots

Edit Lab 4 – Factor Quadratic Polynomial

- Edit the code to factor this *prime* polynomial:

$$2x^2 + 14x + 3 ?$$

- What is shown in the output? Why does this happen?
- The code as currently written can handle only *positive* coefficients - how could we strengthen the code to process *negative* coefficients?
- How could we avoid displaying simple commutative interchanges of the previously found factors?

Strassen's Method

- Multiplication is repeated addition, so a computer is **much faster at adding** two numbers than *multiplying* them
- From our first days in Algebra we are taught that you can only add “**like**” terms (those terms where each variable and exponent are the same)
- Hence we are taught that the **FOIL** method of expanding the product of two monomials requires **four (4)** multiplications: **first, outside, inside, last**:

$$(ax + b)(cx + d) = (ac)x^2 + (ad + bc)x + (bd)$$

Strassen's Method

- Volker Strassen showed in 1969 that you only need **three** (3) multiplications

$$(3x + 5)(7x + 9)$$

$$3 * 7 = 21$$

$$5 * 9 = 45$$

$$8 * 16 = 128$$

- The solution is then:

$$(21)x^2 + (128 - 45 - 21)x + (45)$$

$$21x^2 + 62x + 45$$



Strassen's Method

- We break the rules by **adding** the $3 + 5 = \mathbf{8}$ and $7 + 9 = \mathbf{16}$, even though they are not like terms!

$$(3x + 5)(7x + 9)$$

$$3 * 7 = 21$$

$$5 * 9 = 45$$

$$\mathbf{8} * \mathbf{16} = 128$$

- Essentially **we trade one multiplication for two subtractions**

$$(21)x^2 + (128 - 45 - 21)x + (45)$$

$$21x^2 + 62x + 45$$

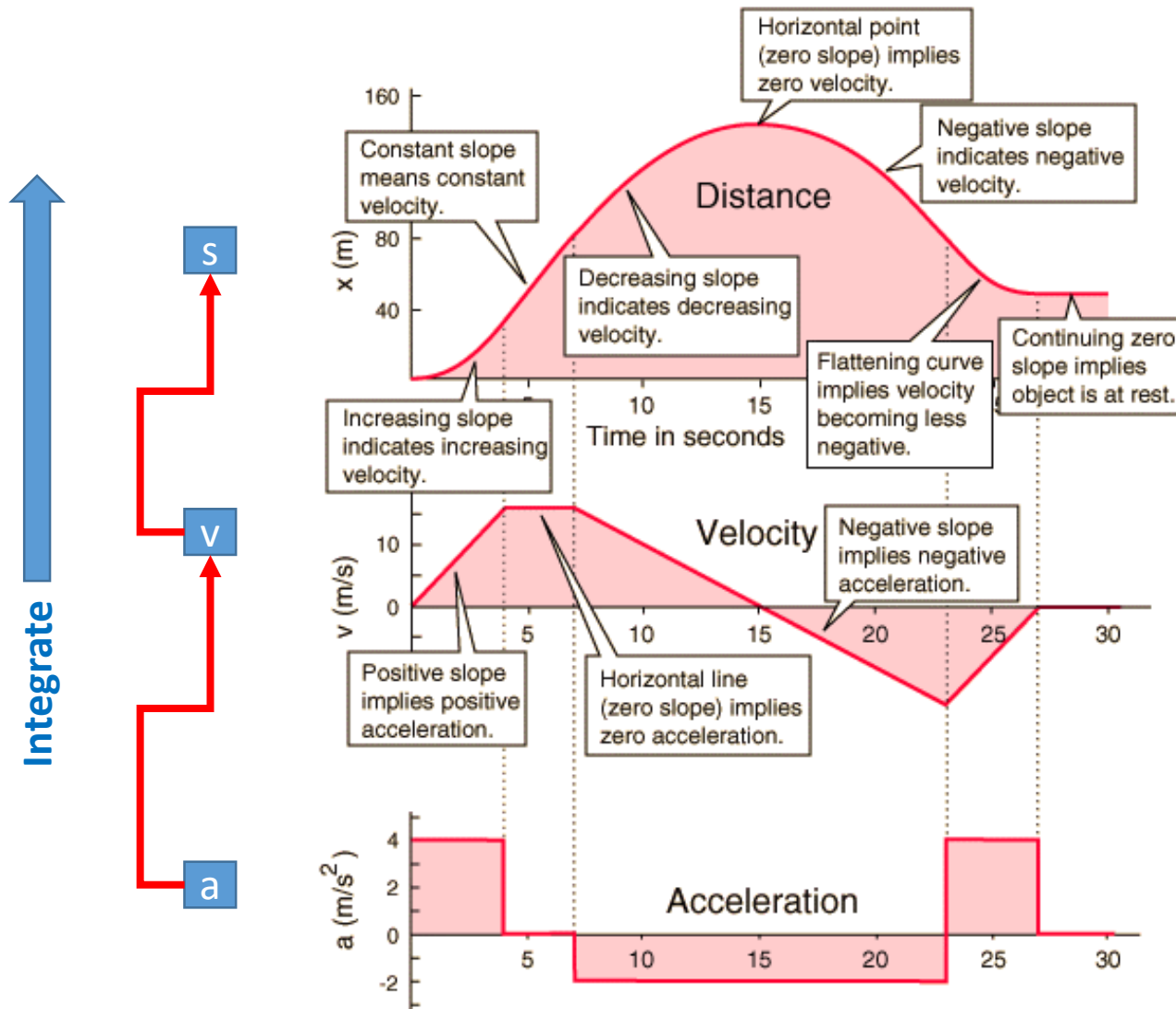
Strassen's Method

- When we cover matrix multiplication, you will find that the naïve approach requires N^3 operations, so a 3 x 3 matrix multiply requires 27 multiplications
- Volker Strassen showed in his 1969 paper that the exponent is less than 3. In fact further improvements on Strassen's method has brought this down to $N^{2.375477}$
- **Why is this important?** Because if you have really large matrices (think about solving **1,000** equations with **1,000** unknowns) the difference adds up *quickly*
- With $N = 1000$, Strassen's method is **74x faster!** (not just a mere 74% faster) than the naïve approach to matrix multiplication!

Why do we need integrals?

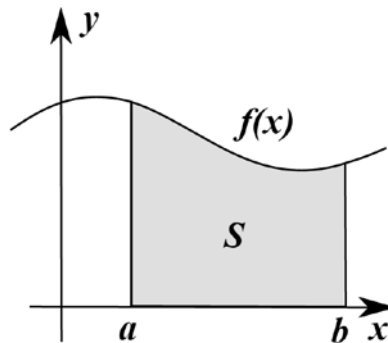
- How to calculate the ***total*** change in a variable X
 - When variable X *depends* on the changes in variable Y...
 - ... and variable Y *depends* on the changes in variable Z...
 - ... and variable Z is constantly changing...
- Think about an accelerating car and the total distance it will travel in a given number of seconds
 - The total distance *depends* on the velocity of the car...
 - ... the velocity of the car *depends* on the acceleration
 - ... and the acceleration is constantly changing

Why do we need integrals?



Why do we need integrals?

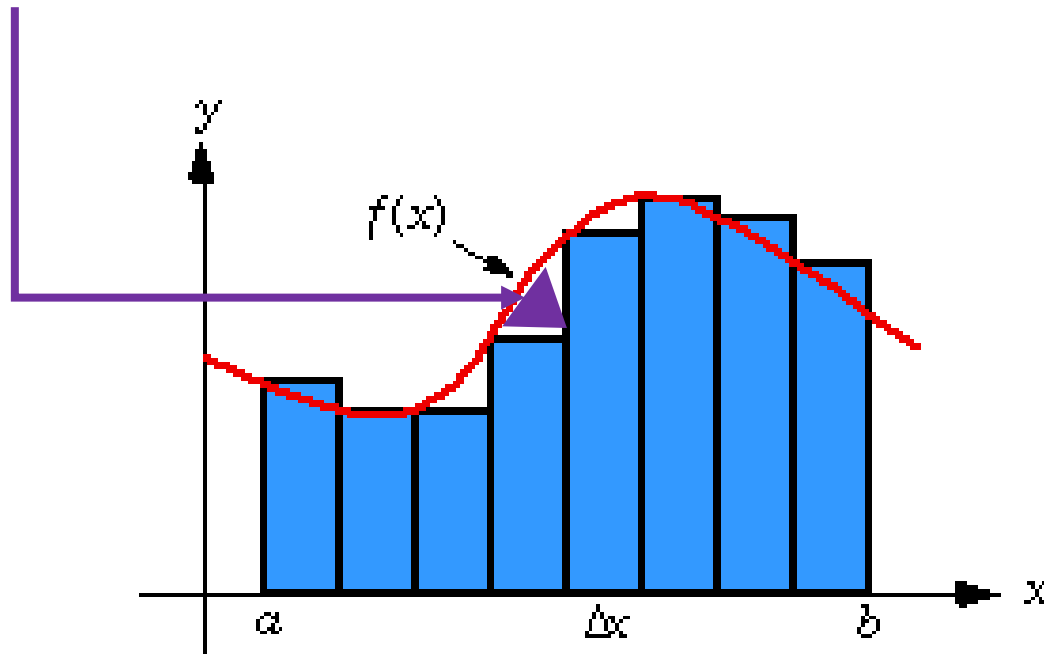
- The **integral** of a function can be defined as the **area under the curve** $f(x)$ within the region $[a,b]$



- Sometimes there are methods to determine *exactly* the value of the integral of $f(x)$ which we would write $F(x) = \int_a^b f(x)$
- However, sometimes it is not possible to find an *analytic* expression for $F(x)$ — so we use **numerical integration**

Riemann Sums

- One way we can integrate $f(x)$ is to divide the area under the curve into strips (**intervals**) and sum the area of each strip
- This estimate may not be totally accurate because we might have **gaps** between the true value of $f(x)$ and the top of a strip

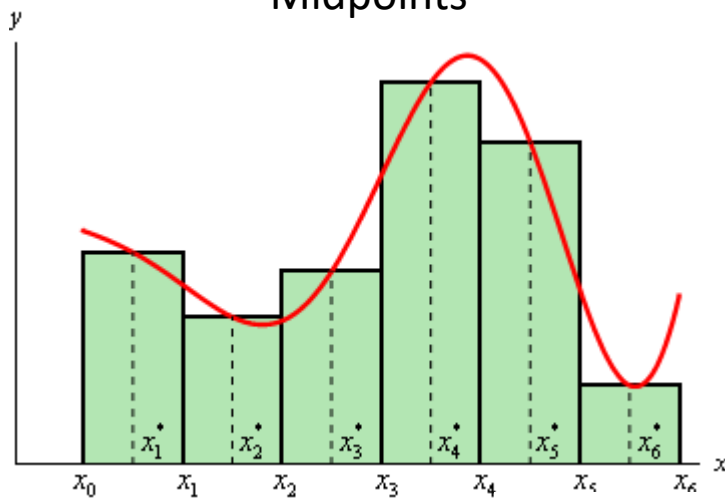


Riemann Sums

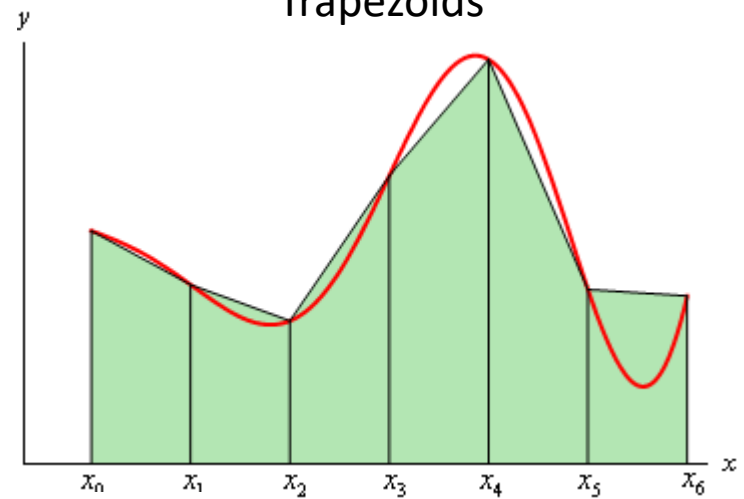
- The width of each strip is $\Delta x = \frac{(b-a)}{\# \text{ of intervals}}$
- We can minimize the **gaps** by increasing the number of intervals, which makes the Δx **smaller**
- There are different strategies for determine the shape and height of each strip
 - Left-hand Rule, Right-hand Rule, Midpoint Rule
 - Fit Trapezoids
 - Fit Parabolas (Simpson's Rule)
- Depending upon the particular *shape* of $f(x)$, one method might be more accurate than the others

Riemann Sums

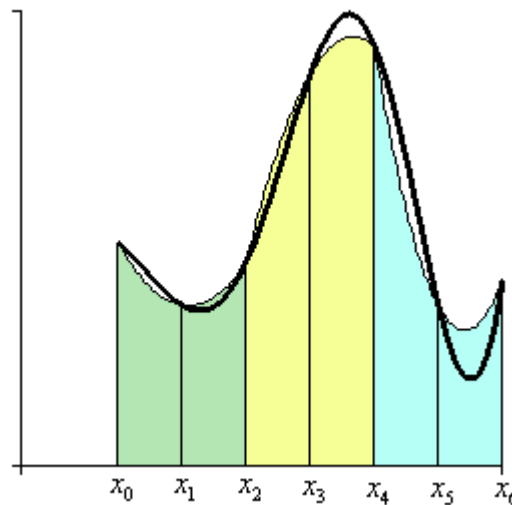
Midpoints



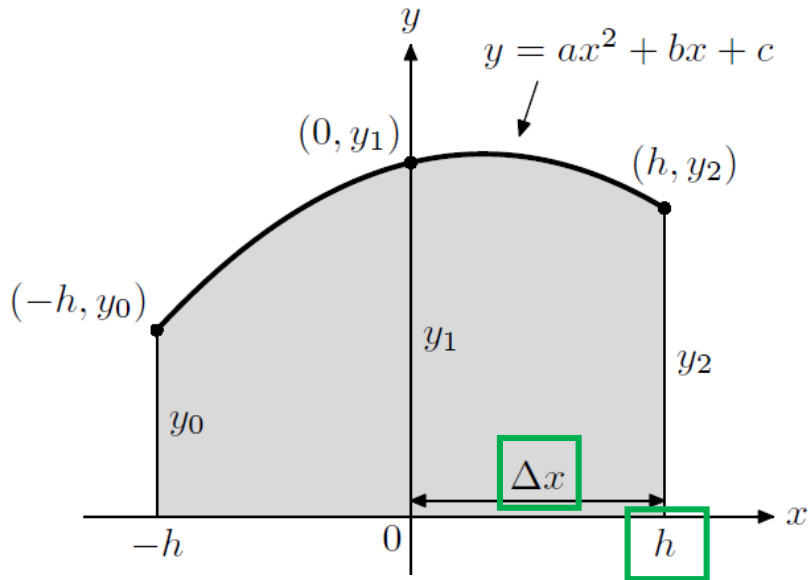
Trapezoids



Parabolas
(Simpson's Rule)



Why is Simpson's Rule more accurate?



$$y_0 = ah^2 - bh + c$$

$$y_1 = c$$

$$y_2 = ah^2 + bh + c$$

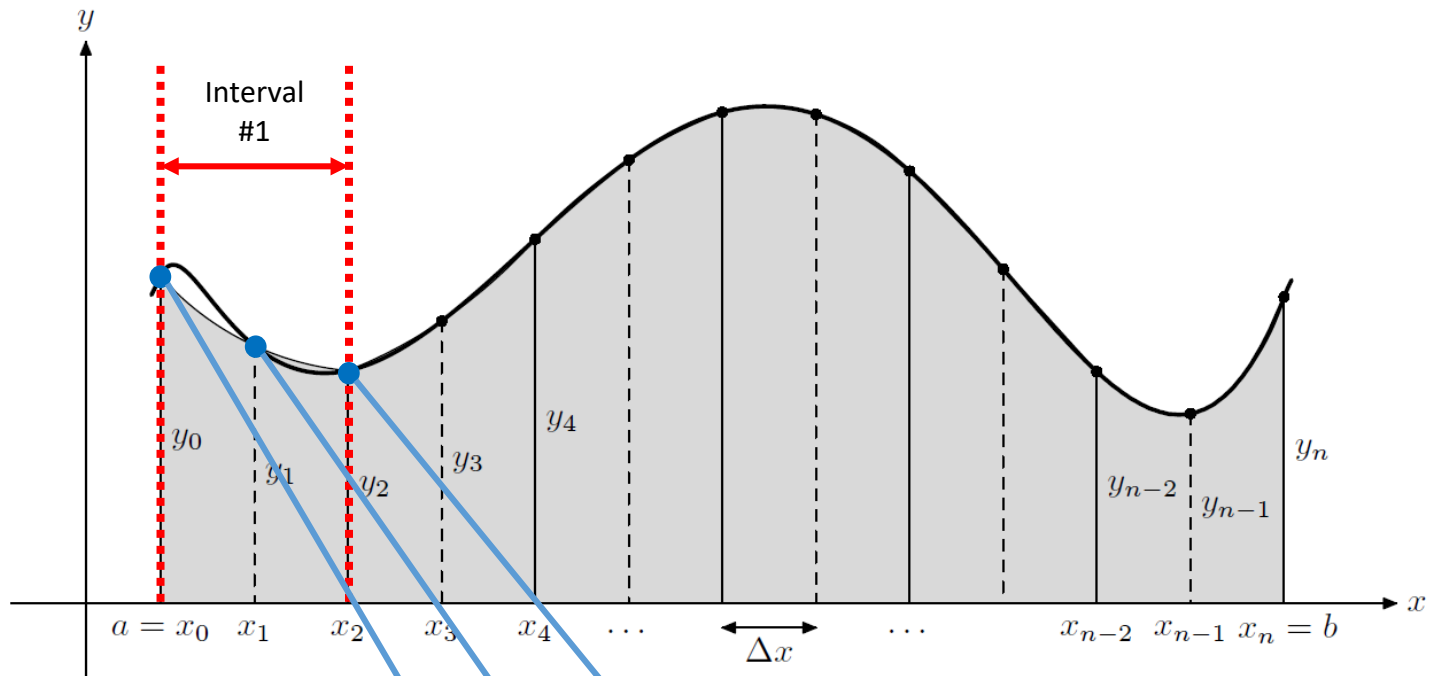
$$y_0 + 4y_1 + y_2 = (ah^2 - bh + c) + 4c + (ah^2 + bh + c) = 2ah^2 + 6c$$

$$\begin{aligned} A &= \int_{-h}^h (ax^2 + bx + c) dx \\ &= \left(\frac{ax^3}{3} + \frac{bx^2}{2} + cx \right) \Big|_{-h}^h \\ &= \frac{2ah^3}{3} + 2ch \\ &= \frac{h}{3} (2ah^2 + 6c) \end{aligned}$$

$$A = \frac{h}{3} (y_0 + 4y_1 + y_2) = \frac{\Delta x}{3} (y_0 + 4y_1 + y_2)$$

Why is Simpson's Rule more accurate?

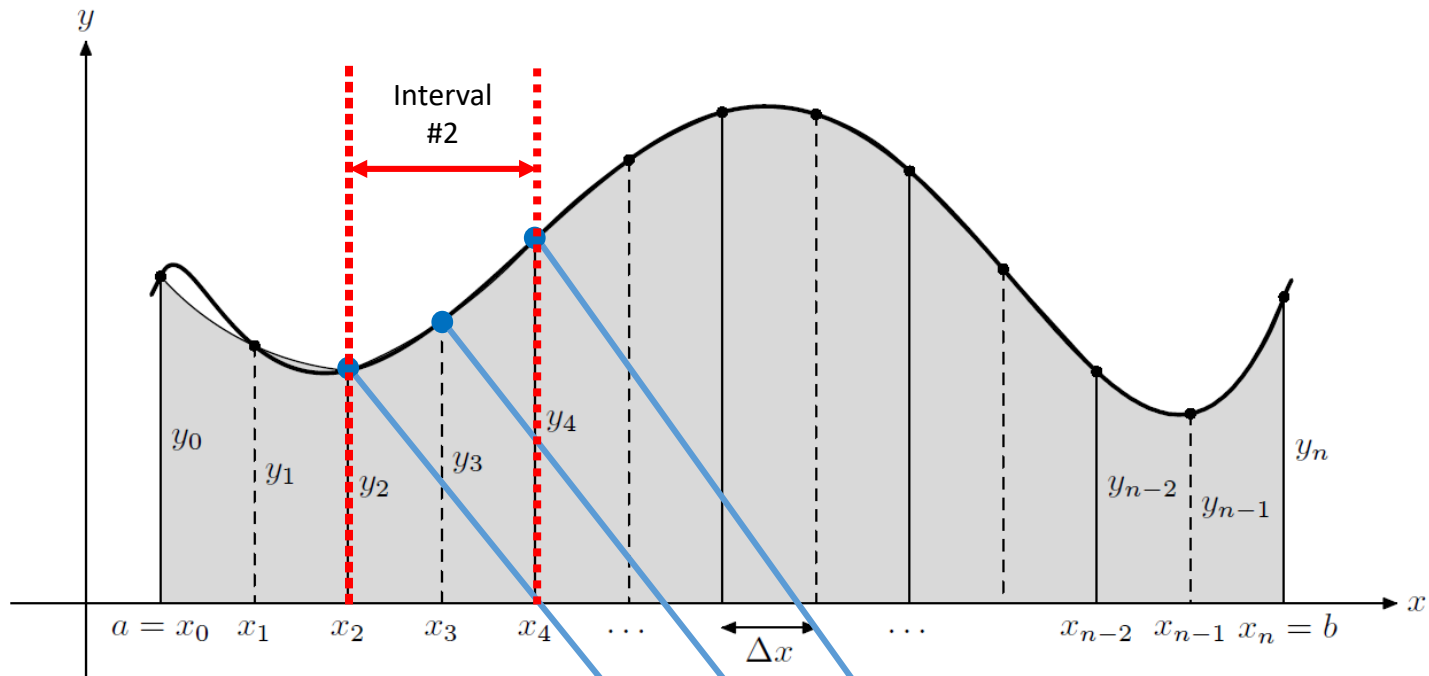
$$y_0 = f(x_0), \quad y_1 = f(x_1), \quad y_2 = f(x_2), \quad \dots, \quad y_n = f(x_n).$$



$$\int_a^b f(x) dx \approx \frac{\Delta x}{3} (y_0 + 4y_1 + 1y_2 + \dots)$$

Why is Simpson's Rule more accurate?

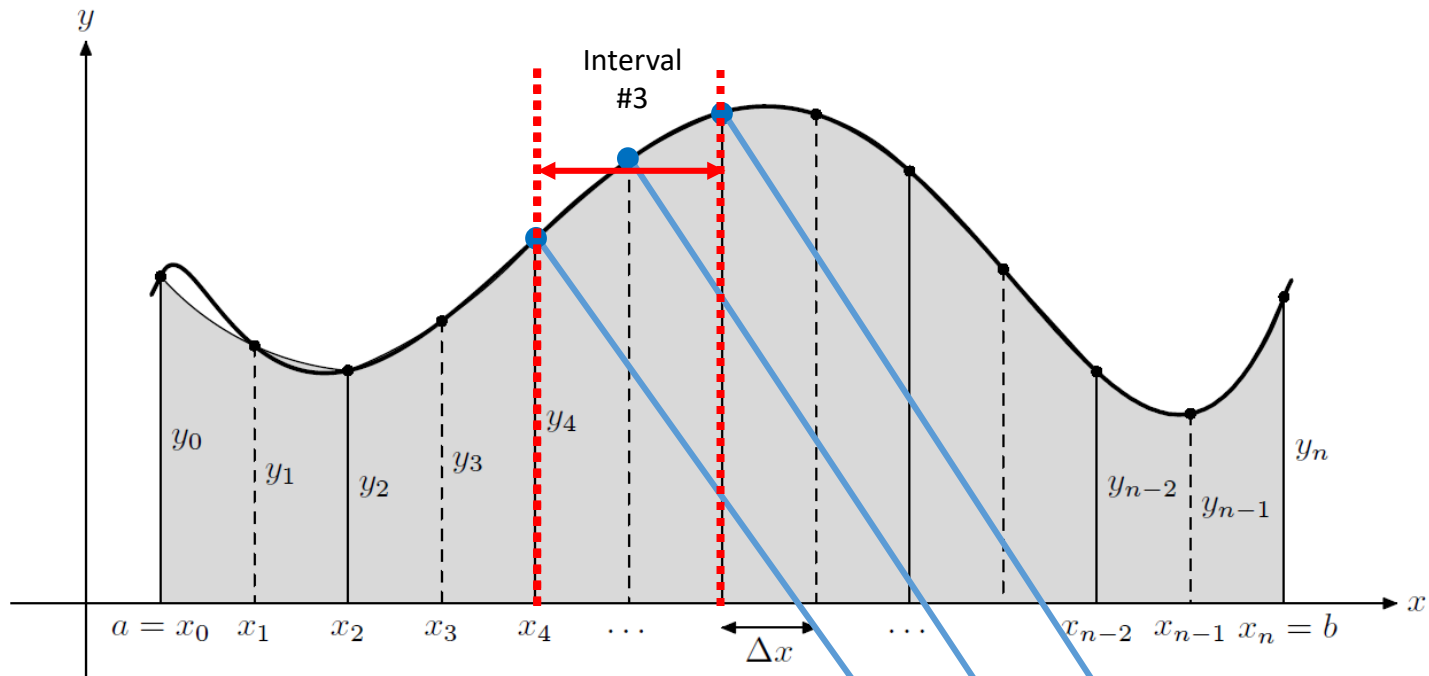
$$y_0 = f(x_0), \quad y_1 = f(x_1), \quad y_2 = f(x_2), \quad \dots, \quad y_n = f(x_n).$$



$$\int_a^b f(x) dx \approx \frac{\Delta x}{3} (y_0 + 4y_1 + \boxed{2y_2} + \boxed{4y_3} + \boxed{1y_4} \dots)$$

Why is Simpson's Rule more accurate?

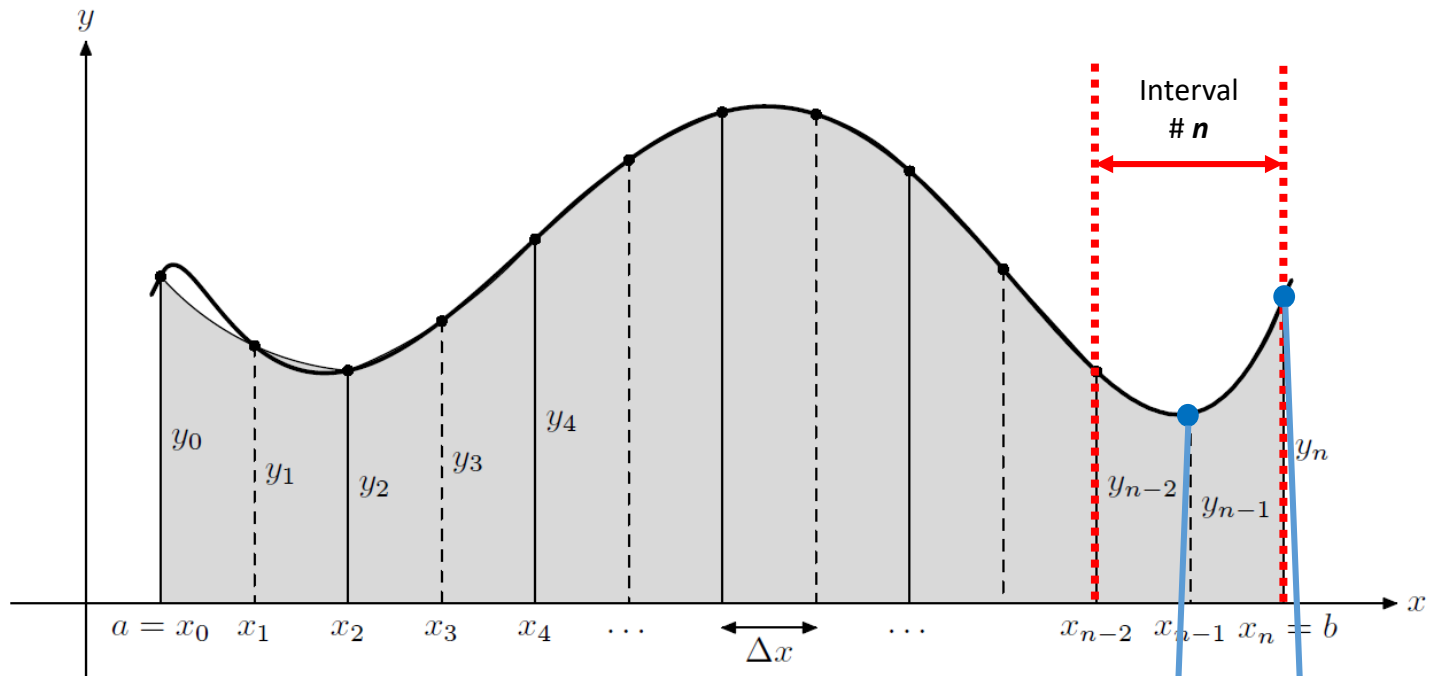
$$y_0 = f(x_0), \quad y_1 = f(x_1), \quad y_2 = f(x_2), \quad \dots, \quad y_n = f(x_n).$$



$$\int_a^b f(x) dx \approx \frac{\Delta x}{3} (y_0 + 4y_1 + 2y_2 + 4y_3 + \boxed{2y_4} + \boxed{4y_5} + \boxed{1y_6} + \dots)$$

Why is Simpson's Rule more accurate!

$$y_0 = f(x_0), \quad y_1 = f(x_1), \quad y_2 = f(x_2), \quad \dots, \quad y_n = f(x_n).$$



$$\int_a^b f(x) dx \approx \frac{\Delta x}{3} (y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \dots + \boxed{4y_{n-1}} + \boxed{y_n})$$

Why is Simpson's Rule more accurate?

$$\int_a^b f(x) dx \approx \frac{\Delta x}{3} (y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 4y_{n-1} + y_n)$$

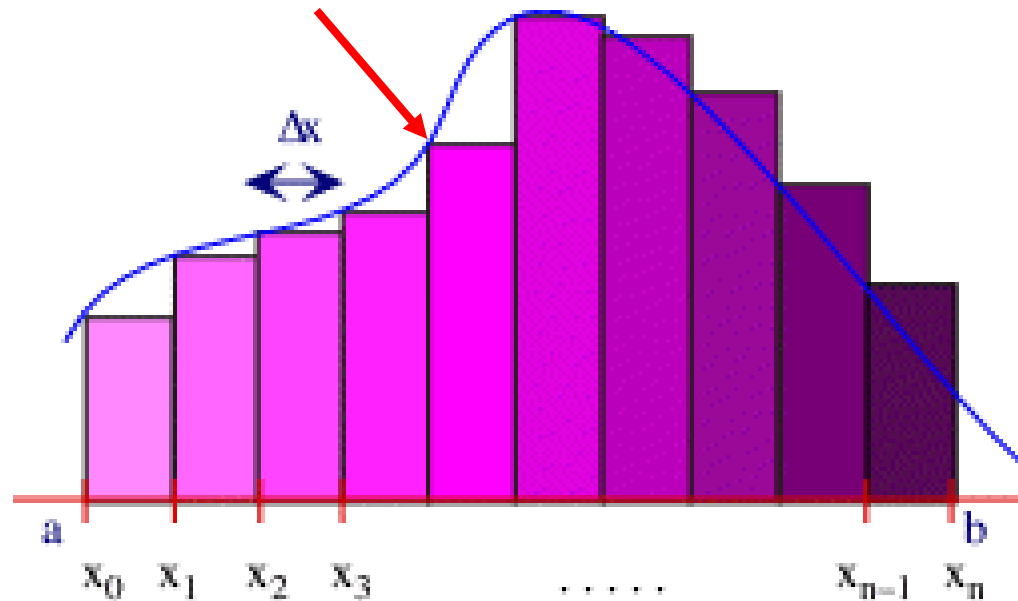
Point	y0	y1	y2	y3	y4	y5	y6
Coeff	1	4	2	4	2	4	1

The first and last point have coefficient = 1
Every point with an odd index has coefficient = 4
Every point with an even index has coefficient = 2

```
double simpsons(double a, double b)
{
    const double dx{(b-a)/intervals};
    double sum{f(a)+f(b)};
    a+=dx;
    for(int i{1}; i<intervals; ++i,a+=dx)
        sum+=f(a)*(2*(i%2+1));
    return (dx/3)*sum;
}
```

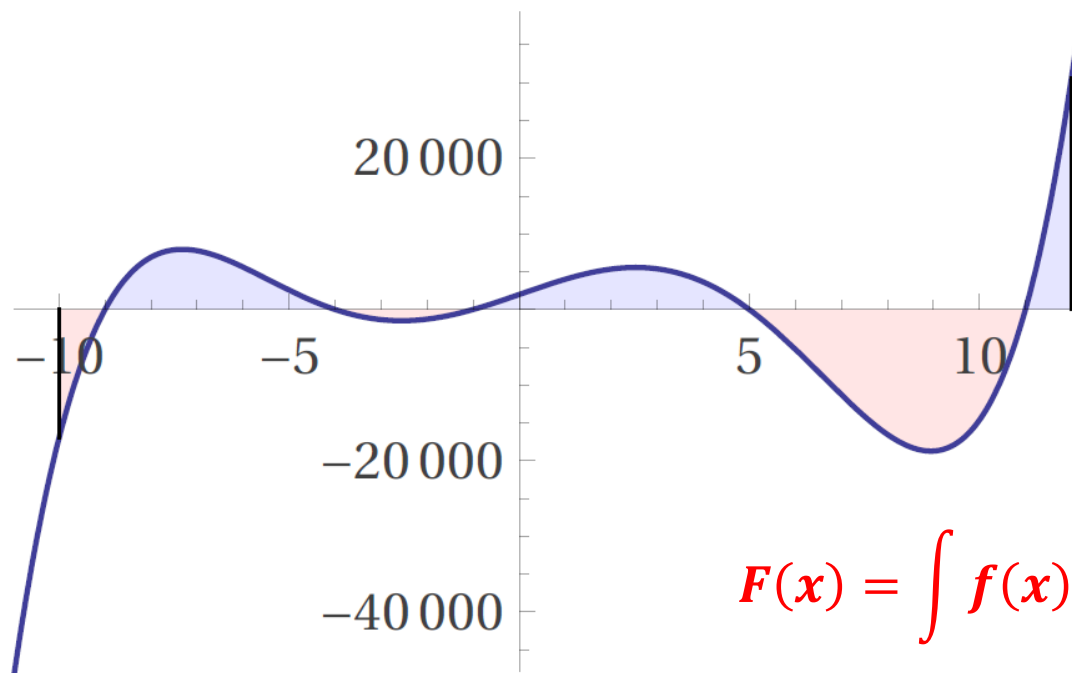
Lab 5 – Simpson's Rule

- Compare the percent error in the estimate of the integral provided by the **left-hand rule** vs. **Simpson's rule**



Lab 5 – Simpson's Rule

$$y = f(x) = x^5 - 2x^4 - 120x^3 + 22x^2 + 2119x + 1980$$
$$= (x + 9)(x + 4)(x + 1)(x - 5)(x - 11)$$



$$F(x) = \int f(x) = ?$$

Lab 5 – Simpson's Rule

- Perform **numerical** integration with respect to x using a million intervals on the following polynomial over the domain **$[-10, 12]$** :

$$F(x) = \int_{-10}^{12} (x^5 - 2x^4 - 120x^3 + 22x^2 + 2119x + 1980) dx$$

$$F(x) = \frac{x^6}{6} - \frac{2x^5}{5} - 30x^4 + \frac{22x^3}{3} + \frac{2119x^2}{2} + 1980x$$

$$F(x) = \left[\frac{-174744}{5} - \frac{-43550}{3} \right] = \frac{-306482}{15} = \mathbf{-20432.13333}$$

Open Lab 5

Simpson's Rule

```
#include "stdafx.h"

using namespace std;

const double a{-10};
const double b{12};
const int intervals = 1e6;
```

Variables declared outside the scope of any function are considered global variables and are initialized *before* main() is called

View Lab 5

Simpson's Rule

```
#include "stdafx.h"

using namespace std;

const double a{-10};
const double b{12};
const int intervals = 1e6;

inline double f(double x)
{
    return (x+9)*(x+4)*(x+1)*(x-5)*(x-11);
}

double lefthand(double a, double b)
{
    const double dx{(b-a)/intervals};
    double sum{};
    while(a<=b)
    {
        sum+=f(a);
        a+=dx;
    }
    return sum*dx;
}
```

```
int main()
{
    cout.imbue(locale(""));

    cout << "Integrating "
        << "x^5 - 2x^4 - 120x^3 + 22x^2 + 2199x +1980"
        << endl << " over [" << a << ", " << b << "]"
        << " using " << intervals << " intervals:"
        << endl << endl;

    double i1{-306482./15};
    cout << "Analytic (Exact): "
        << fixed << setprecision(14)
        << i1 << endl << endl;

    double i2{lefthand(a,b)};
    cout << "Left-hand Rule : "
        << fixed << setprecision(14)
        << i2 << endl
        << scientific << setprecision(4)
        << "% Error = " << (i2-i1)/i1
        << endl << endl;

    double i3{simpsons(a,b)};
    cout << "Simpson's Rule : "
        << fixed << setprecision(14)
        << i3 << endl
        << scientific << setprecision(4)
        << "% Error = " << (i3-i1)/i1
        << endl << endl;

    return 0;
}
```

View Lab 5

Simpson's Rule

```
#include "stdafx.h"

using namespace std;

const double a{-10};
const double b{12};
const int intervals = 1e6;

inline double f(double x)
{
    return (x+9)*(x+4)*(x+1)*(x-5)*(x-11);
}

double lefthand(double a, double b)
{
    const double dx{(b-a)/intervals};
    double sum{};
    while(a<=b)
    {
        sum+=f(a);
        a+=dx;
    }
    return sum*dx;
}

double simpsons(double a, double b)
{
    const double dx{(b-a)/intervals};
    double sum{f(a)+f(b)};
    a+=dx;
    for(int i{1}; i<intervals; ++i, a+=dx)
        sum+=f(a)*(2*(i%2+1));
    return (dx/3)*sum;
}
```

```
int main()
{
    cout.imbue(locale(""));

    cout << "Integrating "
        << "x^5 - 2x^4 - 120x^3 + 22x^2 + 2199x +1980"
        << endl << " over [" << a << ", " << b << "]"
        << " using " << intervals << " intervals:"
        << endl << endl;

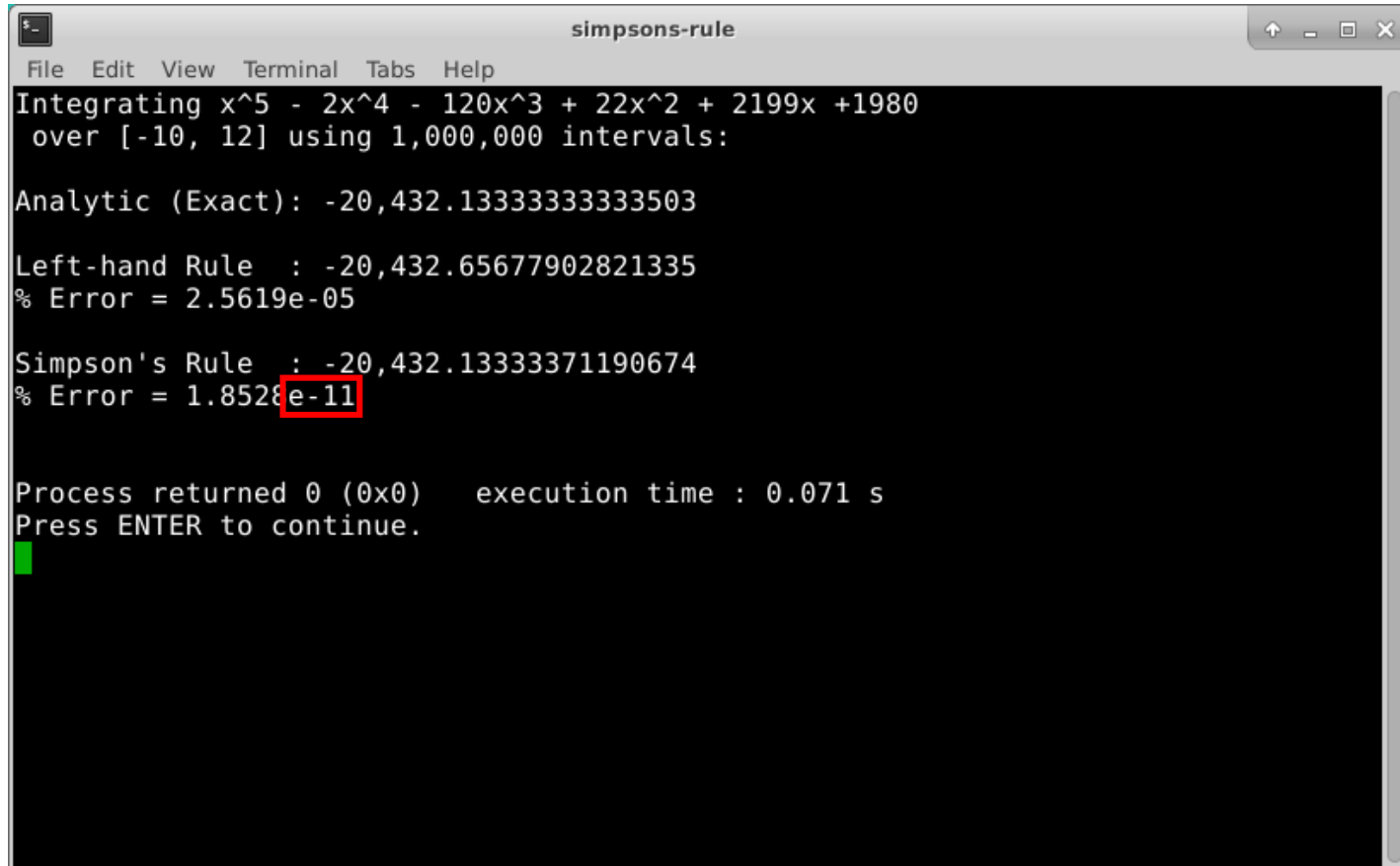
    double i1{-306482./15};
    cout << "Analytic (Exact): "
        << fixed << setprecision(14)
        << i1 << endl << endl;

    double i2{lefthand(a,b)};
    cout << "Left-hand Rule : "
        << fixed << setprecision(14)
        << i2 << endl
        << scientific << setprecision(4)
        << "% Error = " << (i2-i1)/i1
        << endl << endl;

    double i3{simpsons(a,b)};
    cout << "Simpson's Rule : "
        << fixed << setprecision(14)
        << i3 << endl
        << scientific << setprecision(4)
        << "% Error = " << (i3-i1)/i1
        << endl << endl;

    return 0;
}
```

Run Lab 5 – Simpson's Rule

A terminal window titled "simpsons-rule" with a menu bar (File, Edit, View, Terminal, Tabs, Help) and standard window controls. The terminal displays the integration of the polynomial $x^5 - 2x^4 - 120x^3 + 22x^2 + 2199x + 1980$ over the interval $[-10, 12]$ using 1,000,000 intervals. It compares the analytic (exact) result with the Left-hand Rule and Simpson's Rule. The error for Simpson's Rule is highlighted with a red box. The process returns 0 (0x0) with an execution time of 0.071 s, and prompts the user to press ENTER to continue.

```
simpsons-rule
File Edit View Terminal Tabs Help
Integrating x^5 - 2x^4 - 120x^3 + 22x^2 + 2199x + 1980
over [-10, 12] using 1,000,000 intervals:

Analytic (Exact): -20,432.13333333333503

Left-hand Rule : -20,432.65677902821335
% Error = 2.5619e-05

Simpson's Rule : -20,432.13333371190674
% Error = 1.8528e-11

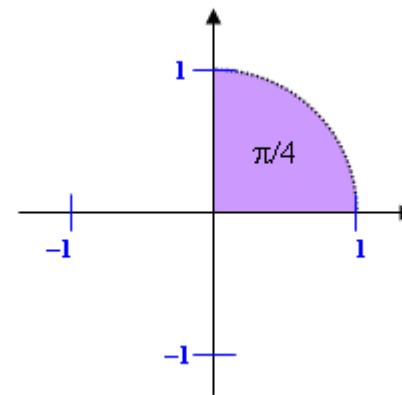
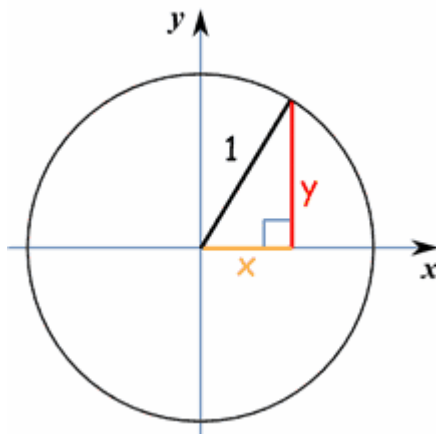
Process returned 0 (0x0)   execution time : 0.071 s
Press ENTER to continue.
█
```


Lab 6 – Circle Area

- Specify in the code the correct $f(x)$, limits $[a, b]$, and exact analytic value for **the area of a unit circle**:

$$F(x) = 4 \int_0^1 \sqrt{1 - x^2} dx$$

**Note: in C++
the constant
 $\text{M_PI} = \pi$**

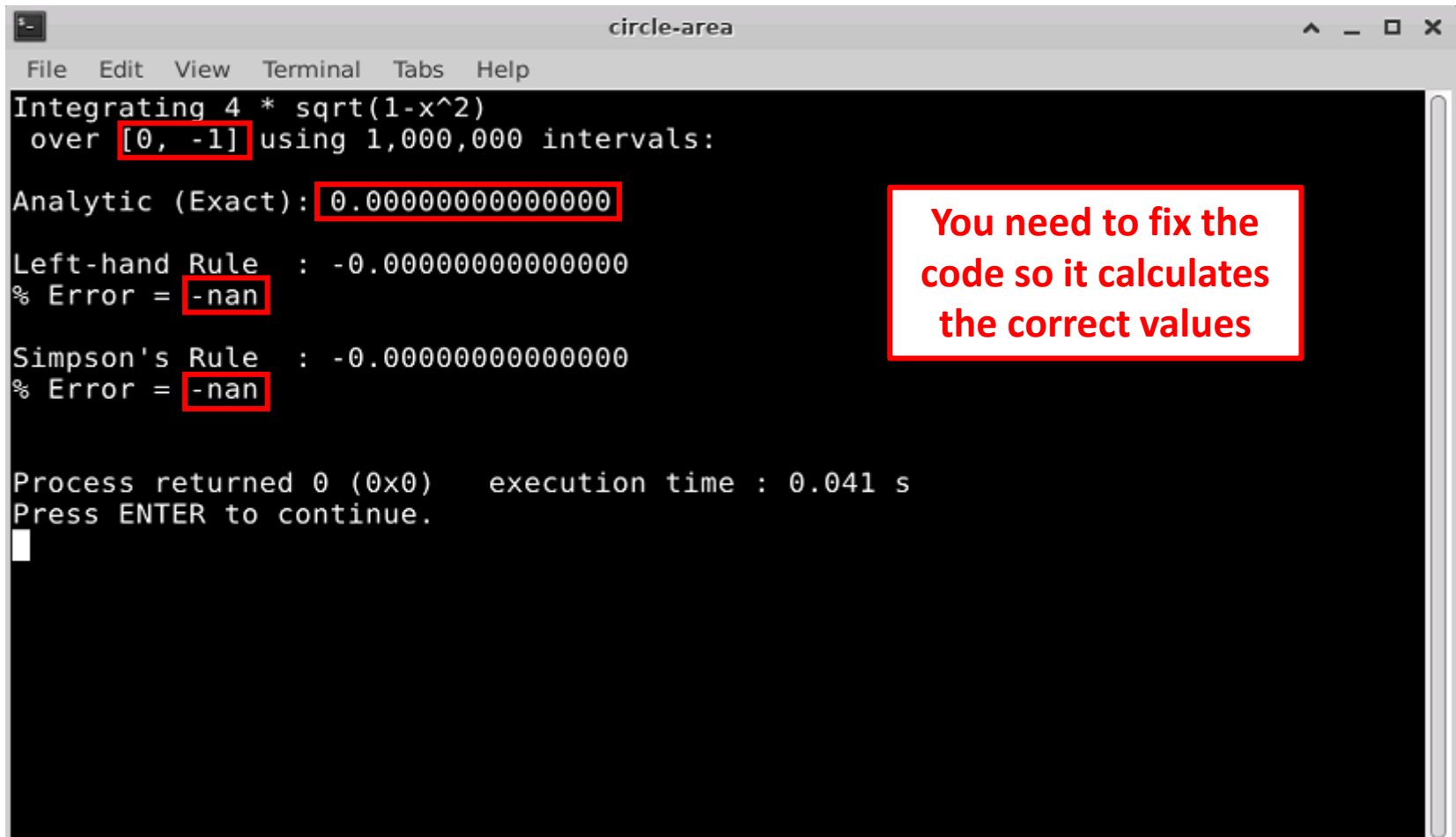


Open Lab 6 – Circle Area

```
main.cpp
1  #include "stdafx.h"
2
3  using namespace std;
4
5  const double a{0};
6  const double b{-1};
7  const int intervals = 1e6;
8
9  inline double f(double x)
10 {
11     return 0;
12 }
13
14 double lefthand(double a, double b)
15 {
16
17
18
19
20
21
22
23
24
25
26 double simpsons(double a, double b)
27 {
28
29
30
31
32
33
34
35
36 int main()
37 {
38     cout.imbue(locale(""));
39
40     cout << "Integrating "
41          << "4 * sqrt(1-x^2)"
42          << endl << " over [" << a << ", " << b << "]"
43          << " using " << intervals << " intervals:"
44          << endl << endl;
45
46     double i1{0};
47     cout << "Analytic (Exact): "
48          << fixed << setprecision(14)
49          << i1 << endl << endl;
50
```

Why are these lines of code incorrect?

Run Lab 6 – Circle Area



```
circle-area
File Edit View Terminal Tabs Help
Integrating 4 * sqrt(1-x^2)
over [0, -1] using 1,000,000 intervals:
Analytic (Exact): 0.0000000000000000
Left-hand Rule : -0.0000000000000000
% Error = -nan
Simpson's Rule : -0.0000000000000000
% Error = -nan

Process returned 0 (0x0)   execution time : 0.041 s
Press ENTER to continue.
```

You need to fix the code so it calculates the correct values

Open Lab 6 – Circle Area

```
main.cpp
1  #include "stdafx.h"
2
3  using namespace std;
4
5  const double a{0};
6  const double b{-1};
7  const int intervals = 1e6;
8
9  inline double f(double x)
10 {
11     return 0;
12 }
13
14 double lefthand(double a, double b)
15 {
16
17
18
19
20
21
22
23
24
25
26 double simpsons(double a, double b)
27 {
28
29
30
31
32
33
34
35
36 int main()
37 {
38     cout.imbue(locale(""));
39
40     cout << "Integrating "
41          << "4 * sqrt(1-x^2)"
42          << endl << " over [" << a << ", " << b << "]"
43          << " using " << intervals << " intervals:"
44          << endl << endl;
45
46     double i1{0};
47     cout << "Analytic (Exact): "
48          << fixed << setprecision(14)
49          << i1 << endl << endl;
50
```

What should these lines of code do?

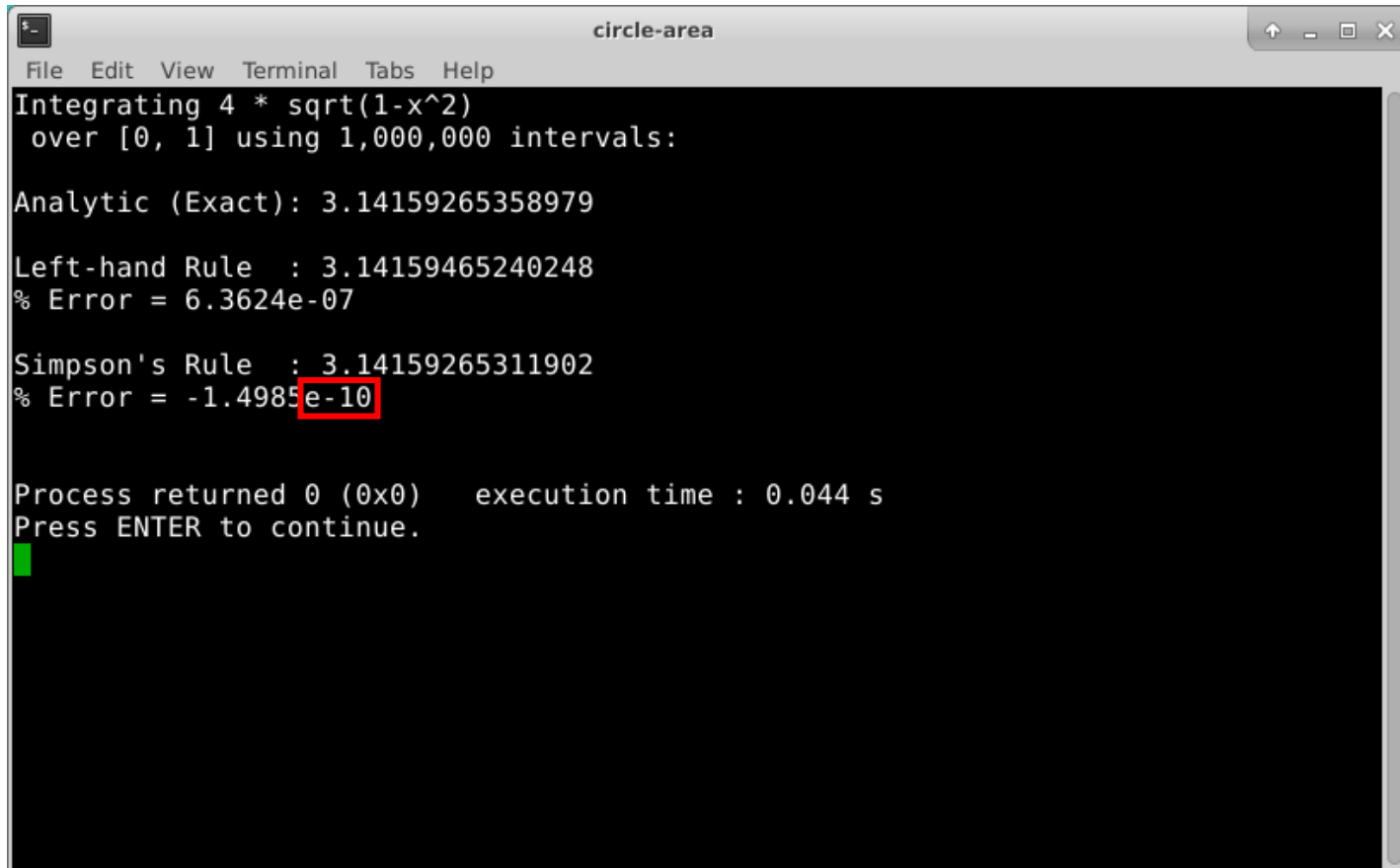


Edit Lab 6 – Circle Area

```
main.cpp
1  #include "stdafx.h"
2
3  using namespace std;
4
5  const double a{0};
6  const double b{1};
7  const int intervals = 1e6;
8
9  inline double f(double x)
10 {
11     return sqrt(1-x*x);
12 }
13
14 double lefthand(double a, double b)
15 {
25
26 double simpsons(double a, double b)
27 {
35
36 int main()
37 {
38     cout.imbue(locale(""));
39
40     cout << "Integrating "
41         << "4 * sqrt(1-x^2)"
42         << endl << " over [" << a << ", " << b << "]"
43         << " using " << intervals << " intervals:"
44         << endl << endl;
45
46     double i1{M_PI};
47     cout << "Analytic (Exact): "
48         << fixed << setprecision(14)
49         << i1 << endl << endl;
50
```

These are the
correct lines of code

Run Lab 6 – Circle Area



```
circle-area
File Edit View Terminal Tabs Help
Integrating 4 * sqrt(1-x^2)
over [0, 1] using 1,000,000 intervals:

Analytic (Exact): 3.14159265358979

Left-hand Rule : 3.14159465240248
% Error = 6.3624e-07

Simpson's Rule : 3.14159265311902
% Error = -1.4985e-10

Process returned 0 (0x0)   execution time : 0.044 s
Press ENTER to continue.
█
```

Now you know...

- An **algorithm** is a recipe, often with loops, that changes inputs to outputs
- There are many **simple to state**, but hard to *solve*, open problems in **number theory**
- It is not known if there are any **odd** perfect numbers
- It is not known if there are *infinitely* many perfect numbers
- The **bool** data type to store true or false values
- Use the **if()** statement for ***conditional code execution***
- The **if()** statement introduces a scope {}, and can have an optional **else** {} scope
- The **while()** is like an **if()** statement that loops
- The **while()** loop a simplified **for()** loop

Now you know...

- The **%** operator returns the **remainder**
- Use **double** equals **==** operator to test for equality
- Use single equal **=** to define the value of a variable
- The **&&** operator performs a **logical AND** of two Boolean values
- Numerical Integration finds **the area under the curve** using **successively smaller** and smaller strips
- The strips can be sized according to the **Left, Right, Trapezoid, or Midpoint** rules
- Simpson's method is the more accurate due to fitting parabolas