Computer Science

Y. Liow (September 11, 2023)

Contents

Chapter 102

Asymptotics and algorithmic runtime analysis

102.1 Introduction debug: introduction.tex

Here are two functions, one for computing fibonacci numbers and another for computing factorials:

```
int fib(int n)
{
    if (n <= 1)
    {
        return 1;
    }
    else
    {
        return fib(n - 1) + fib(n - 2);
    }
}
```

```
int factorial(int n)
{
    int p = 1;
    for (int i = 1; i <= n; i++)
    {
        p *= i;
    }
    return p;
}
```

Exercise 102.1.1. Use the above functions and compute `fib(n)` and `factorial(n)` and for `n` = 0, 1, 2, 3, 4, 5, 6 *by hand*. Check your work by running the above. □

Here's the big questions: which of the above two is faster?

Yes, you can run the two functions and time it with your watch. You want to test the above for several values for n of course. If the set of values is too small, you might not be able to establish a pattern. Of course when n is small, the functions will probably finish too fast for you to tell any difference. So you should try large values. But the problem is ... what is "large"? For a different pair of functions maybe n has to be at least 10. For another different pair, you might need n to be at least 1000000. Also, timings can change depending on what your computer is doing at the same time. So you should probably shutdown all other programs while you're doing the tests.

Or ...

You can take a course on data structures and algorithms and tell me right away which is better ... yes, right away ... without running the functions.

That's one of the benefits of understanding data structures and algorithms. Besides knowing how to compare performance, of course you will also learn different algorithms (not just to time them) and create different data structures. You can think of data structures as container (think of a bag) and you are basically interested in putting things into this container, taking things ouf of this container, checking if the container has a specific data, etc. You will learn how to build different types of containers with different performance characteristics. Such containers are important in the real world. You can think of google for instance as being a massive container of webpages. Google has to scan the world for webpages and put relevant data about these webpages into the container. More importantly, google wants to be able to tell you as fast and as accurate as possible which are the most relevant webpages that you might want to look at when you search for (say) "scifi movies 1980s".

102.2 Timing debug: timing.tex

Although we want to be mathematical and scientific and compute algorithmic performance abstractly, sometimes it's still a good idea to get our hands dirty and measure performance based on real time. Also, the runtime of some algorithms are actually very compute mathematically. For such cases, measure runtime experimentally is helpful.

Let me show you two ways of measuring times. The second method is more accurate.

102.2.1 First method

Here's a program that prints the time, sleep for 1 second, prints the time again:

```
#include <ctime>
#include <iostream>
#include <unistd.h>

int main()
{
    std::cout << time(NULL) << std::endl;
    sleep(1);
    std::cout << time(NULL) << std::endl;

    return 0;
}
```

If you want to store the return value of `time`, you can use the time type `time_t`:

```
#include <ctime>
#include <iostream>

int main()
{
    time_t start = time(NULL);
    sleep(1);
    time_t end = time(NULL);
    double diff = difftime(end, start);
    std::cout << start << ' ' << end << ' '
              << diff
              << std::endl;

    return 0;
}
```

If the time to execute a section of code is too short, you can of course execute the code 1000 times. You can then divide by 1000 to get the time. In fact such an averaging will probably give you a more accurate approximation.

It's also a good idea to make your test environment as similar as possible. So if you test program A while watching a DVD and then test program B overnight while you're asleep ... it wouldn't be fair to say that program A is a terribly inefficient program, right???

Note that the `time` function returns the real time. Even if you try real hard not to watch a movie on your laptop (and a million other things), it would still be difficult to measure the time taken for your program to run because there are many other processes running in your machine. Unless if you're using a really really really old machine! This means that the time difference reported might include time spent doing something else. If you like, you can google for Unix/Linux support for querying user time, system CPU time, etc. for a process.

102.2.2 Second method

The second method is lot more accurate and does not depend on what else your computer is doing. In other words, it actually measures CPU usage. (Unfortunately this method is platform dependent and works only on unix/linux systems.) Try this program ...

```
#include <iostream>
#include <sys/time.h>
#include <sys/resource.h>


int main()
{
    rusage start, end;

    getrusage(RUSAGE_SELF, &start);

    for (unsigned long int i = 0; i < 1000000000; i++)
    {
        double t = 3.14;
        t * t * t;
    }

    getrusage(RUSAGE_SELF, &end);
```

```
    double endtime = end.ru_utime.tv_sec
                     + end.ru_utime.tv_usec * 1e-6;
    double starttime = start.ru_utime.tv_sec
                       + start.ru_utime.tv_usec * 1e-6;
    double diff = endtime - starttime;
    std::cout << diff << "secs \n";

    return 0;
}
```

Exercise 102.2.1. Clearly it's more convenient for you to create a class to do the above. Call the class `Timer`. Here's an example usage:

debug: exercises/timer/question.tex

```
#include "Timer.h"

int main()
{
    Timer timer;
    timer.start();

    for (unsigned long int i = 0; i < 1000000000; i++)
    {
        double t = 3.14;
        t * t * t;
    }

    timer.stop();
    double diff = timer.read();
    std::cout << diff << "secs \n";

    return 0;
}
```

(Go to solution, page 4007) □

Solutions

Solution to Exercise 102.2.1.

debug: exercises/timer/answer.tex

Solution not provided.

102.3 What is asymptotic analysis? debug:

what-is-asymptotic-analysis.tex

What is asymptotic analysis? It is the study of growth behavior of functions as the independent variable gets larger and larger.

(More generally, there can be more than one independent variable and the limit(s) can be other than infinity).

It's a kind of math tool that allows you to say

informally	notation
$f(n)$ "is roughly $\leq$" $g(n)$ for large n	$f(n) = O(g(n))$
$f(n)$ "is roughly $\geq$" $g(n)$ for large n	$f(n) = \Omega(g(n))$
$f(n)$ "is roughly $=$" $g(n)$ for large n	$f(n) = \Theta(g(n))$

I'll get more specific later. The big picture to keep in mind is that the asymptotic notation is a notation for comparing functions.

I'm going to use this tool to measure various things about algorithms, including their runtime and resource usage such as memory usage. I'll start with big-O. Once you get the hang of big-O, the big-Ω and big-Θ is easy.

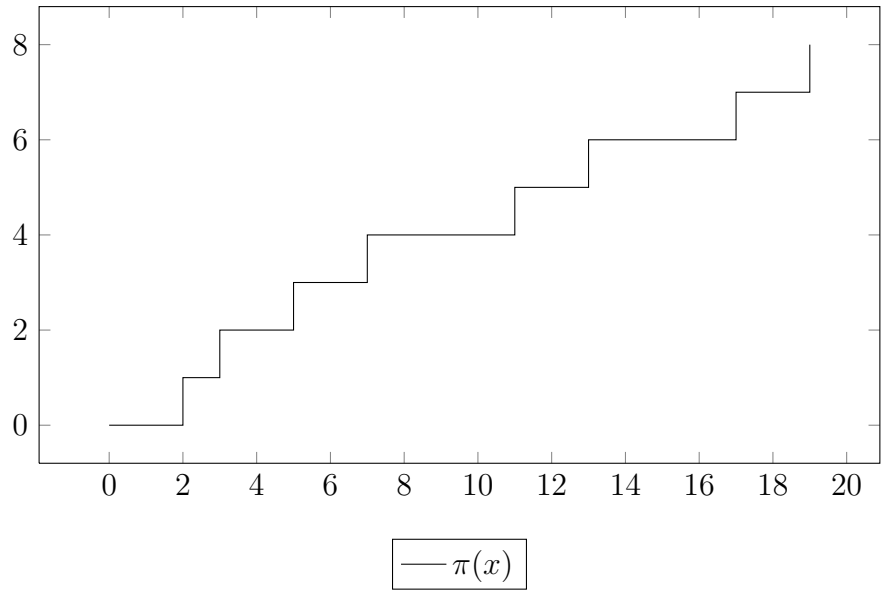
102.4 Roots of Asymptotics (DIY) debug: roots-of-asymptotics.tex

Historically, the use of asymptotics to analyze the growth of functions started around 1800s and was used extensively by mathematicians studying analysis (think Calculus) and especially in the study of analytic number theory. In fact it was introduced by Paul Bachmann who was an analytic number theorist.

Number theory is the study of integers. One big goal of number theorists is to know everything about prime numbers 2, 3, 5, 7, 11,13, ... An example of a question a number theorist might be interested in is this: What is the n–th prime p_n for positive n? So $p_1 = 2$, $p_5 = 11$. They want to know if there is a formula for p_n so that for instance they can work out very quickly what is $p_{1000000}$.

Although number theory involved the study of whole numbers, later on techniques involving numbers which are not integers are used to analyze integers. Analytic number theory uses methods from analysis. Here "analysis" means more or less calculus, which means analytic number theory uses real numbers.

Number theorist, especially analytic number theorists, are interested also in the number of primes less then a fixed number say x. This is called $\pi(x)$, the prime counting function. For instance $\pi(1) = 0$, $\pi(2) = 1$, $\pi(3) = 2$, $\pi(4) = 2$, and $\pi(6.2) = 3$. Here's $\pi(x)$ for $0 \leq x \leq 20$:



By the way I hope you see that the two concepts p_n and $\pi(x)$ are related. Do

you see that $n = \pi(p_n)$? So if for instance I have a formula for $\pi(x)$:

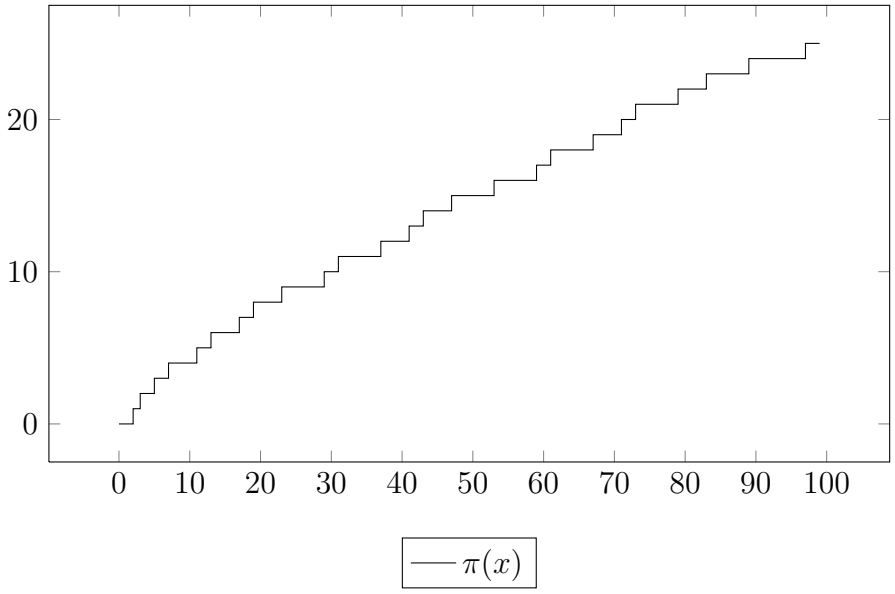
$$\pi(x) = \ldots \text{ formula in } x \ \ldots$$

I just replace the x in the formula by p_n to get:

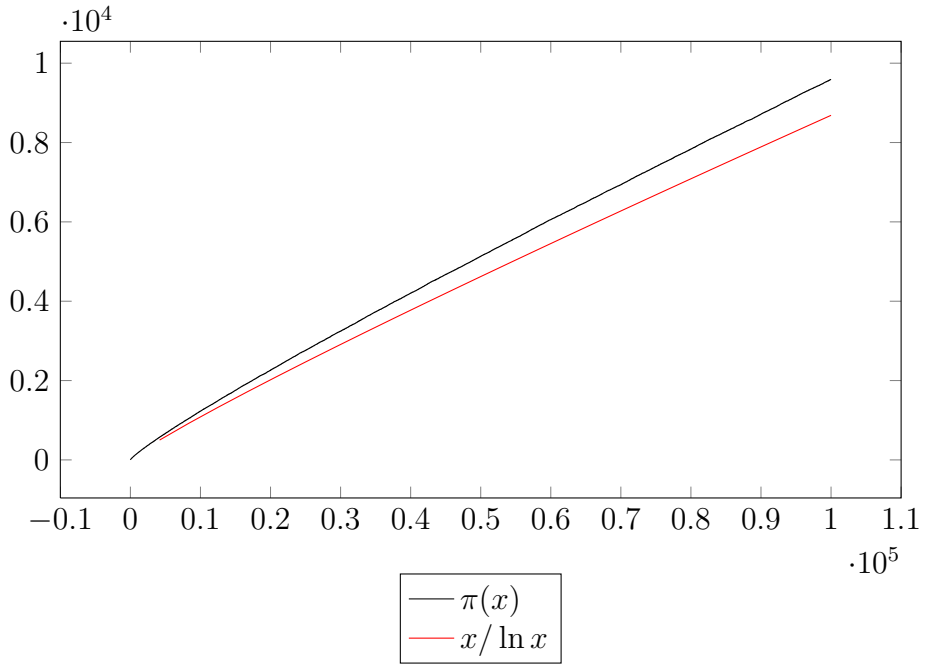
$$n = \pi(p_n) = \ldots \text{ formula in } p_n \ \ldots$$

This is an equation involving n and p_n. I then try to solve for p_n to get a formula for p_n in terms of n.

From the above graph of $\pi(x)$, you see that the graph is not smooth. Here's $\pi(x)$ for $0 \leq x \leq 100$:



Here's $\pi(x)$ for $0 \leq x \leq 100000$ compared against the function $x/\ln x$:



As you can see, $\pi(x)$, which started as a very jagged and unpredictable function becomes extremely well-behaved and almost smooth when you zoom out. It seems that $x/\ln x$ is a tad too small. However if you plot the same graph but for larger and larger values of x (i.e. if you zoom out to see more and more of the graph) you would notice that the gap narrows.

Around 1792, Gauss, when analyzing a table of primes up to 100,000, found that the function $f(x) = x/\ln x$ approximates $\pi(x)$ function. However this was a numerical verification – Gauss was not able to prove mathematically that his "nice" $f(x)$ does approximate $\pi(x)$ for all large x.

For a very long time, no one was able to prove that Gauss' $f(x)$ is very close to $\pi(x)$. In 1850, Chebychev was able to proved that there are constants C_1 and C_2 such that for large values of x

$$C_1 f(x) \leq \pi(x) \leq C_2 f(x)$$

where $f(x)$ is Gauss's conjectured approximation. By improving on proof techniques, mathematicians were able to improve on the constants C_1 and C_2 to get Gauss' $f(x) = x/\ln x$ closer and closer to $\pi(x)$.

Finally, in 1896, Hadamard and de la Vallé Poussin independently proven that

the approximation $f(x) = x/\ln x$ is extremely good and tight. In fact:

$$\lim_{x\to\infty} \frac{\pi(x)}{x/\ln x} = 1$$

This is the famous Prime Number Theorem, PNT. One can conclude from PNT that the n–th prime is "roughly"

$$n \ln n$$

In the above statement:

$$\text{“}C_1 f(x) \leq \pi(x) \leq C_2 f(x), \quad \text{for large } x\text{”}$$

the right half of the inequality:

$$\text{“}\pi(x) \leq C_2 f(x), \quad \text{for large } x\text{”}$$

is in fact our big-O:

$$\pi(x) = O(f(x))$$

The left half:

$$\text{“}C_1 f(x) \leq \pi(x) \quad \text{for large } x\text{”}$$

is the big-Ω: $\pi(x) = \Omega(f(x))$. And because $\pi(x)$ is big-O and big-Ω of $f(x)$, we say $\pi(x) = \Theta(f(x))$. The statement

$$\lim_{x\to\infty} \frac{\pi(x)}{x\ln x} = 1$$

is also another type of asymptotic statement. In this case, we say that $\pi(x)$ and $x/\ln x$ are **asymptotically equivalent** and the notation is $\pi(x) \sim x/\ln n$.

asymptotically equivalent

The key point to take away is that the asymptotic notations and concepts (O, Ω, Θ, $\sim$) are for the comparison of function when the x is huge, which informally is the same as looking the graphs of the functions *when you zoom out*.

D. Knuth popularized the use of asymptotic notation in computer science (around 1960s).

102.5 Algorithmic analysis: how fast is an algorithm?

debug: algorithm-analysis-how-fast-is-an-algorithm.tex

Why do we study asymptotics? For the computer scientists, asymptotics tell us what to focus on and what to ignore. It creates a useful classification of functions that grow in the same manner.

For instance look at the following algorithm that computes the sum of integers from 1 to n (the value of n is stored in variable n) and the sum is stored in s.

```
s = 0
for i = 1, 2, ..., n:
     s = s + i
```

This algorithms solves the following problem:

$$P(n) : \text{“Compute the sum of integers from 1 to } n\text{”}$$

This problem is of course made up of many specific problem instances. For example it includes:

$$P(10) : \text{“Compute the sum of integers from 1 to 10”}$$

and

$$P(1135452) : \text{“Compute the sum of integers from 1 to 1135452”}$$

Our algorithm:

```
s = 0
for i = 1, 2, ..., n:
     s = s + i
```

is an algorithmic solution to our problem $P(n)$. (It's not the only one.)

In the above, you can think of n as the *size* of each problem instance. And of course you would expect, without even analyzing our algorithm in detail, that the above algorithm will need more time for a large n. Correct?

Different people design different algorithms to solve the same problem.

We are interested in measuring how fast an algorithm runs so that we can pick the best. It's clear that the crucial thing to focus on is the performance of an

algorithm when n is large. After all, the sum from 1 to 3 is easy – in fact we can do $1 + 2 + 3$ in our heads and not bother with running a program at all, right? It takes more time just to let your computer boot up or wake up!!!

In the real world, after the algorithm is designed (and you've checked that it's correct!!!), you still have to implement the algorithm with a programming language and run the program on a specific computer. In the real world, the performance of the algorithm can be measured by the time taken by the computer to run the program. By "time taken" in this case, I meant measuring time with a watch or the clock. This is sometimes called **wall-clock time**.

wall-clock time

On some OS, you can also measure processor time for a program, i.e., the amount of time that the program actually uses the CPU.

However using the wall-clock or processor time is problematic because it depends on the hardware used, the programming language used, the operating system, etc. (No, it does not depend on how many planets are lined up.)

What I want to do is to measure the performance independent of external factors, i.e., I want to measure the performance of the algorithm. This does not mean that the external factors are not important. But rather, we want to solve the performance issue at the root first. In fact this is usually *the* most important factor in the performance of any software.

Now let's get back to our sum from 1 to n algorithm and see how we can measure the runtime performance of an algorithm without even running it on a piece of hardware.

I'm going to rewrite my algorithm like this (apologies to those who disapprove of goto statements):

```
        s = 0
        i = 1
LOOP:   if i > n:
            goto ENDLOOP
        s = s + i
        i = i + 1
        goto LOOP
ENDLOOP:
```

In this simplified language, we have basic operations such as assignment operators, arithmetic operators (such as `+`), and boolean operators (such as `>`). Besides that we have goto statements and conditional branching statements `if [boolean]: goto [label]`. In the above pseudocode, I've implement a

for-loop using a conditional branching and goto statement. Control structures such as for-loops, while-loops, do-while loops, if statements, and if-else statements are actually implemented at the machine code level with goto and conditional branching statements (see CISS360). Therefore goto and conditional branching statements are actually more fundamental.

Now I'm going to attach time taken to execute each statements:

```
                            time
        s = 0               t1
        i = 1               t2
LOOP:   if i > n:           t3
            goto ENDLOOP    t4
        s = s + i           t5
        i = i + 1           t6
        goto LOOP           t7
ENDLOOP:
```

This is how you read the above time "accounting": The statement

```
        s = 0               t1
```

takes t_1 seconds (or whatever unit of time you like ... you'll see later that the specific value of t_1 and the units are not important at all). For the `if`–statement (which is made of a header and a body):

```
LOOP:   if i > n:           t3
            goto ENDLOOP    t4
```

it takes time t_3 for the `if`–statement to compute the boolean value of $i > n$ and then to decide to execute `goto ENDLOOP` or not. So if the boolean condition is true, then the time taken for the whole `if` statement is $t_3 + t_4$; if the boolean condition is false, then the time taken is t_3.

Note that the times taken to execute each of the above statement, t_1, t_2, ... are *constants with respect to* n. What this mumbo-jumbo meant was, whether I run the above algorithm with $n = 10$ or $n = 1000000$, the time taken this:

```
        s = 0               t1
```

is still t_1. Makes sense, right?

Next, we count the number of times each statement is executed:

```
                              time    number of times
         s = 0                t1      1
         i = 1                t2      1
LOOP:    if i > n:            t3      n + 1
             goto ENDLOOP     t4      1
         s = s + i            t5      n
         i = i + 1            t6      n
         goto LOOP            t7      n
ENDLOOP:
```

For instance

```
         s = 0                t1      1
```

is executed once (regardless of the value of n). For the `if`–statement

```
LOOP:    if i > n:            t3      n + 1
             goto ENDLOOP     t4      1
```

since i runs through 1, 2, ..., $n-1$, n, $n+1$ (the boolean condition evaluates to true for the first n values and false for the last value of $n+1$), the header of the `if` is executed $n+1$ times and the body is executed only once.

Finally (phew!) we compute the time taken to execute the code. The total time taken is

$$\begin{aligned}\text{Total time} &= t_1 + t_2 + (n+1)t_3 + t_4 + n(t_5 + t_6 + t_7) \\ &= (t_3 + t_5 + t_6 + t_7)n + (t_1 + t_2 + t_3 + t_4)\end{aligned}$$

Therefore the total time taken is

$$An + B$$

for some constants A and B. Note that $t_1, t_3, t_3, ...$, and therefore A and B, depends mainly on the machine that is executing the algorithm.

It's common to use $T(n)$ to denote the runtime of an algorithm. So I might write:

$$\begin{aligned}T(n) &= (t_3 + t_5 + t_6 + t_7)n + (t_1 + t_3 + t_3 + t_4) \\ &= An + B\end{aligned}$$

If I'm talking about several algorithmic runtimes together I would decorate

the $T(n)$ notation. For instance I might write

$$T^P(n)$$

or

$$T^{\text{sum-to}}(n)$$

or

$$T^{\text{CONVERT-DIRT-TO-GOLD}}(n)$$

Now as n grows (and this is the situation we do want to worry about), An is of course going to be larger than B. So ultimately when n is large the time taken to carry out the algorithm is mainly and roughly due to An. And since we don't really care to specify the hardware we're using, we can also fudge away the constant A and conclude the time take to run our algorithm or program is roughly (or rather proportional to) the function

$$n$$

The technical thing to do is this: We write

$$T(n) = An + B = O(n)$$

and say "$T(n)$ is the big-O of n". The big-O tells the reader that we're only expressing what the runtime function looks like for large n and when we ignore multiples. In this case I will say that this algorithm has **linear runtime**. Instead of say "the big-O runtime of the algorithm is $O(n)$", you can also say the **time complexity** of the algorithm is $O(n)$.

linear runtime

time complexity

Warning: Make sure you see the difference between

$$T(n) = O(n)$$

and

$$T(n) = n$$

(which is wrong).

There are therefore *two* elements to measuring the runtime performance of an algorithm:

(1) You need to be able to compute the runtime as a formula in the size of the problem (which in the above is n)
(2) You need to do some fudging to get an "approximation", i.e., the big-O of

the function from (1). If the formula from (1) is a sum of functions, you choose the one that is largest when n is huge and you replace constants with 1.

Notice in the above example, the fudging *simplifies* the function from

$$An + B$$

to

$$n$$

Now, as I said before, I'm ignoring multiples so that I'm considering An the same as $n = 1 \cdot n$. Of course I could have chosen $2n$ as well. But since I consider all multiples the same, I prefer to use n since it's simpler than $2n$.

Let me summarize the above steps carried out in the computation of the big-O of the runtime of our sum to n algorithm:

STEP 1. First I assign times to each statement and compute the time taken to be

$$\begin{aligned} T(n) &= t_1 + t_2 + (n+1)t_3 + t_4 + n(t_5 + t_6 + t_7) \\ &= t_1 + t_2 + t_3 + t_4 + n(t_3 + t_5 + t_6 + t_7) \end{aligned}$$

STEP 2. I clean up and say that the time taken is a function of the form

$$T(n) = An + B$$

where A and B are constants.

STEP 3. The first fudging step is where I looked at the functions An and B and conclude that for large n, the function is roughly the function

$$An$$

STEP 4. The second fudging step is when I throw away the A (i.e., replace the A with 1) because the constant A is hardware dependent and say that the time taken is roughly the function

$$n$$

and I conclude with this statement:

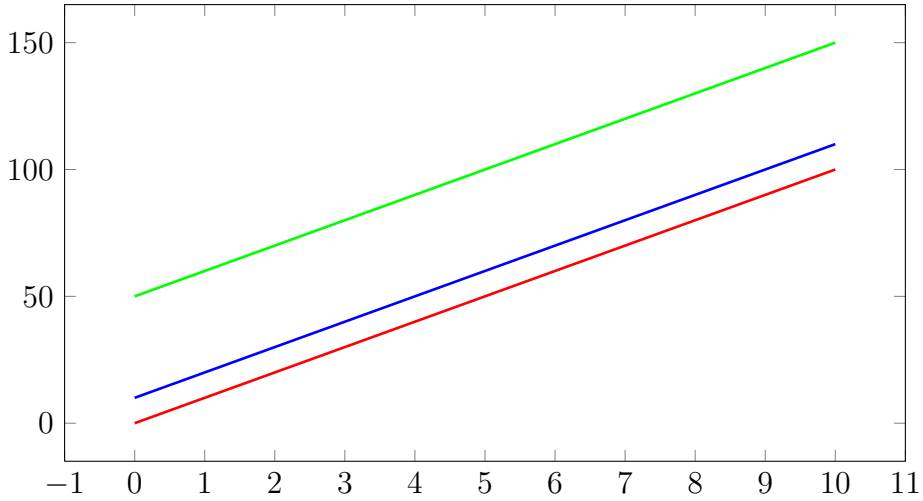
$$T(n) = O(n)$$

In general, to compute the big-O of any given function $f(n)$,

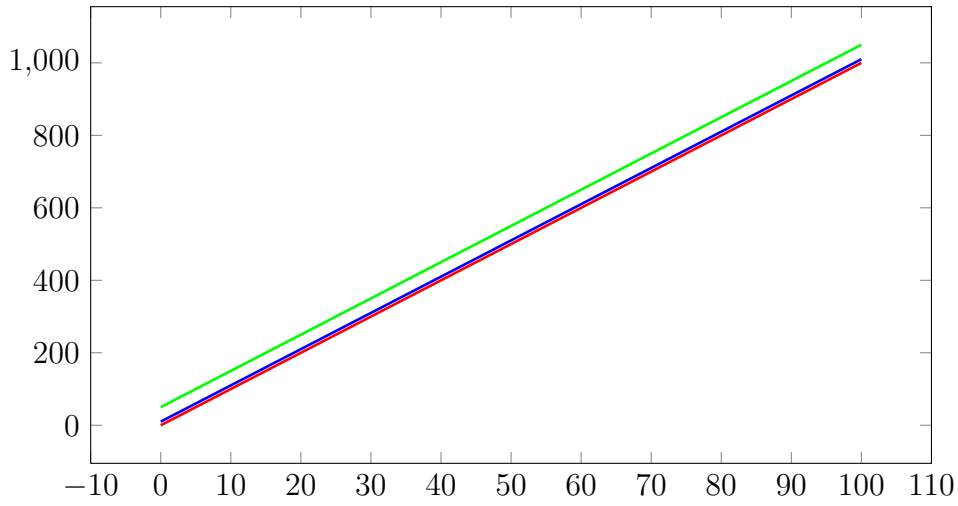
- You only keep the term that is the largest in the long run (i.e. for large n). The growth of $f(n)$ is determined by this term. Typically, your runtime function might have more than 2 terms.
- You replace constant(s) by 1 because different constants indicates different hardware being used.

Later we'll see that there are other simplifications and computational tools. The above steps are good enough for now.

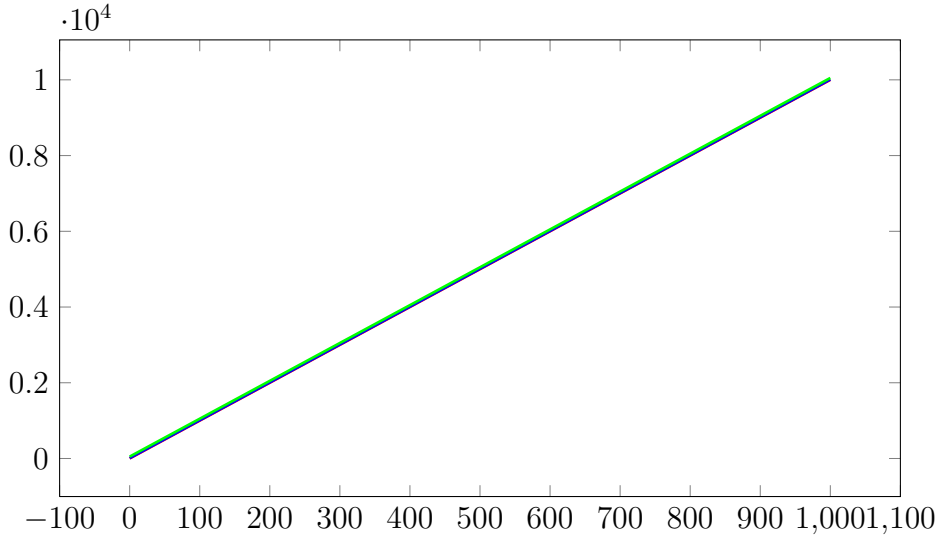
OK, let me show you *graphically* what the fudging does to a function. Here are the plots of $y = 10n, y = 10n + 10, y = 10n + 50$:



(I'm not labeling the graphs because you should be able to tell which is which ... right?) For small values of n, i.e., $0 \leq n \leq 10$, the functions are different and separated from each other. But now if I increase the values for n, say, $0 \leq n \leq 100$, they look like this:



And here's the plot for the domain of $0 \leq n \leq 1000$:



All three graphs more or less collapse into a single line, right? You see that for large values of n, the functions:

$$y = 10n, \quad y = 10n + 10, \quad y = 10n + 50$$

really behave very much like each other: they all grow as fast as $10n$.

The second fudging step is when I throw away the A in the function

$$An$$

and replace it by 1 to get the function

$$n$$

because the constant A is hardware dependent and has nothing to do with the inherent with the algorithm of the pseudocode. If you change your machine, your t_5 will either larger or smaller. On the other hand, the n in An is inherent to the pseudocode – it comes from the loop. You cannot remove the loop (or the n) just by changing your machine. Also, you will see later that when we compare multiples of n with multiples of n^2 and multiples of n^3, you will see that the when n is huge, multiples of n clump together closely and away from the multiples of n^2 and n^3. You will also see that multiples of n^2 clump up together away from the multiples of n^3. I'll show you some examples in the next section.

By the way, you can measure any resource taken up by an algorithm. For instance you can also measure the **space complexity** of an algorithm, i.e., the amount of *extra* memory needed to carry out an algorithm.

space complexity

For the sum from 1 to n algorithm, you are given n and you need to store the result in s. Any memory usage besides n and s (the memory usage for input and output) is considered extra memory. For our algorithm

```
s = 0
for i = 1, 2, ..., n:
     s = s + i
```

The extra memory usage is due to variable i. The variable `s` is not considered extra – you have to compute `s` since that's the goal of the algorithm. Memory is frequently measured in bytes or bits. (For serious theoretical CS, bits is used.) Being an integer, i might use up 4 bytes. I would then say the space complexity of our sum from 1 to n algorithm is

$$\textsc{Space}(n) = 4$$

Since $4 = 4 \cdot n^0$, I will use the big-O notiation and write

$$\textsc{Space}(n) = O(n^0) = O(1)$$

In you prefer to use bits instead of bits, 4 bytes is 32 bits, which is then

$$\textsc{Space}(n) = 32$$

But $32 = 32 \cdot n^0$, so

$$\textsc{Space}(n) = O(n^0) = O(1)$$

which gives the same space complexity as when I use bytes to measure memory usage. It's also clear that you can also count the number of integer variables (i.e., count 4-bytes) and arrive at the same space complexity. In this case, I will say that the algorithm has **constant space complexity**. Altogether for my sum from 1 to n algorithm:

constant space complexity

$$\begin{aligned} T(n) &= O(n) \\ \textsc{Space}(n) &= O(1) \end{aligned}$$

Exercise 102.5.1. The following computes the sum of squares from 1^2 to n^2:

debug: exercises/runtime-of-sum-of-squares/question.tex

```
s = 0
for i = 1, ..., n:
    term = i * i
    s = s + term
```

Here's the program with goto statements and timing for each statement:

```
                              time
         i = 1                t1
         s = 0                t2
LOOP:    if i > n:            t3
             goto ENDLOOP     t4
         term = i * i         t5
         s = s + term         t6
         i = i + 1            t7
         goto LOOP            t8
ENDLOOP:
```

(a) Compute the time taken $T(n)$ as a function of n with constants t_1, ..., t_8.

(b) Simplify the runtime function by giving names A, B, ... to the constants of the function from (a).

(c) Fudge away the constants and write down the simplest $g(n)$ such that the time in (b) is a big-O of your $g(n)$. Your $g(n)$ should be either n or n^2 or n^3 or ...

(d) What is the space complexity of the algorithm?

(Go to solution, page 4029) □

Exercise 102.5.2. The following sums up all the values in array x of size n and sets the values of the array to 0:

debug: exercises/runtime-of-sum-of-array-and-clear/question.tex

```
s = 0
for i = 0, ..., n - 1:
    s = s + x[i]
    x[i] = 0
```

Assume each of the statements

```
    s = s + x[i]
    x[i] = 0
```

take constant time. Here's the algorithm with goto statements:

```
          s = 0
          i = 0
LOOP:     if i >= n:
              goto ENDLOOP
          s = s + x[i]
          x[i] = 0
          i = i + 1
          goto LOOP
ENDLOOP:
```

(a) Assign constant times to each statement and compute the time taken as a function of n, t_1, ...

(b) Simplify the runtime function by giving names A, B, ... to the constants of the function from (a).

(c) Fudge away the constants and write down the simplest $g(n)$ such that the time in (b) is a big-O of your $g(n)$. Your $g(n)$ should be either n or n^2 or n^3 or ...

(d) What is the space complexity of the algorithm?

(Go to solution, page 4030) □

Exercise 102.5.3. The following pseudocode is similar to the above.

debug: exercises/runtime-of-sum-of-array-then-clear/question.tex

```
s = 0
for i = 0, ..., n - 1:
    s = s + x[i]

for i = 0, ..., n - 1:
    x[i] = 0
```

(a) Rewrite the above algorithm with goto statements, assign constant times to each statement and compute the time taken as a function of n, t_1, ...

(b) Simplify the runtime function by giving names A, B, ... to the constants of the function from (a).

(c) Fudge away the constants and write down the simplest $g(n)$ such that the time in (b) is a big-O of your $g(n)$. Your $g(n)$ should be either n or n^2 or n^3 or ...

(d) What is the space complexity of the algorithm?

(Go to solution, page 4031) □

Exercise 102.5.4.

debug:
exercises/big-O-100-3n2+sinn/question.tex

Let

$$f(n) = 100 + 3n^2 + \sin(n)$$

(a) Plot the graphs of

$$y = 300$$
$$y = 3n^2$$
$$y = \sin(n)$$

for $0 \leq n \leq 20$.

(b) From (a), which term of $f(n)$ grows the fastest and therefore will ultimately control the growth of $f(n)$ as n keeps growing?

(c) Compute the big-O of $f(n)$.

Note: Using graphs does not really provide a proof. Big-O requires you to say something about functions for *all* large values of n. Your graph can only show a finite range of n. Later I'll show you how to prove big-O statements.

(Go to solution, page 4032) □

Exercise 102.5.5.

debug:
exercises/40000-1-10-n2-nsinn/question.tex

Repeat the previous problem with this function:

$$f(n) = 40000 + \frac{1}{10}n^2 + n\sin(n)$$

Use a sufficiently large domain for n.

(Go to solution, page 4033) □

Exercise 102.5.6. Repeat the previous problem with this function:

debug: exercises/problem-1/question.tex

$$f(n) = 10000 + \frac{1}{10000}n^2 + \frac{n^2}{1+n}\ln n$$

Use a sufficiently large domain for n. (Go to solution, page 4035) □

Solutions

Solution to Exercise 102.5.1.

debug: exercises/runtime-of-sum-of-squares/answer.tex

(a) The following gives the number of times each statement is executed in terms of n:

```
                              time    number of times
         i = 1                t1      1
         s = 0                t2      1
LOOP:    if i > n:            t3      n + 1
             goto ENDLOOP     t4      1
         term = i * i         t5      n
         s = s + term         t6      n
         i = i + 1            t7      n
         goto LOOP            t8      n
ENDLOOP:
```

Therefore the runtime $T(n)$ is

$$T(n) = (t_3 + t_5 + t_6 + t_7 + t_8)n + (t_1 + t_2 + t_3 + t_4)$$

(b)

$$T(n) = An + B$$

where A, B are constants.

(c)

$$T(n) = O(n)$$

(d)

$$\textsc{Space}(n) = O(1)$$

□

Solution to Exercise 102.5.2.

debug: exercises/runtime-of-sum-of-array-and-clear/answer.tex

(a) The following gives the number of times each statement is executed in terms of n:

```
                              time    number of times
         s = 0                t1      1
         i = 0                t2      1
LOOP:    if i >= n:           t3      n + 1
             goto ENDLOOP     t4      1
         s = s + x[i]         t5      n
         x[i] = 0             t6      n
         i = i + 1            t7      n
         goto LOOP            t8      n
ENDLOOP:
```

Therefore the runtime $T(n)$ is

$$T(n) = (t_3 + t_5 + t_6 + t_7 + t_8)n + (t_1 + t_2 + t_3 + t_4)$$

(b)

$$T(n) = An + B$$

where A, B are constants.

(c)

$$T(n) = O(n)$$

(d)

$$\text{SPACE}(n) = O(1)$$

□

Solution to Exercise 102.5.3.

debug: exercises/runtime-of-sum-of-array-then-clear/answer.tex

(a) The following gives the number of times each statement is executed in terms of n:

```
                              time    number of times
          s = 0               t1      1
          i = 0               t2      1
LOOP1:    if i >= n:          t3      n + 1
              goto ENDLOOP1   t4      1
          s = s + x[i]        t5      n
          i = i + 1           t6      n
          goto LOOP1          t7      n
ENDLOOP1:
          i = 0               t8      1
LOOP2:    if i >= n:          t9      n + 1
              goto ENDLOOP2   t10     1
          x[i] = 0            t11     n
          i = i + 1           t12     n
          goto LOOP2          t13     n
ENDLOOP2:
```

Therefore the runtime $T(n)$ is

$$T(n) = (t_3 + t_5 + t_6 + t_7 + t_9 + t_{11} + t_{12} + t_{13})n + (t_1 + t_2 + t_3 + t_4 + t_8 + t_9 + t_{10})$$

(b)

$$T(n) = An + B$$

where A, B are constants.

(c)

$$T(n) = O(n)$$

(d)

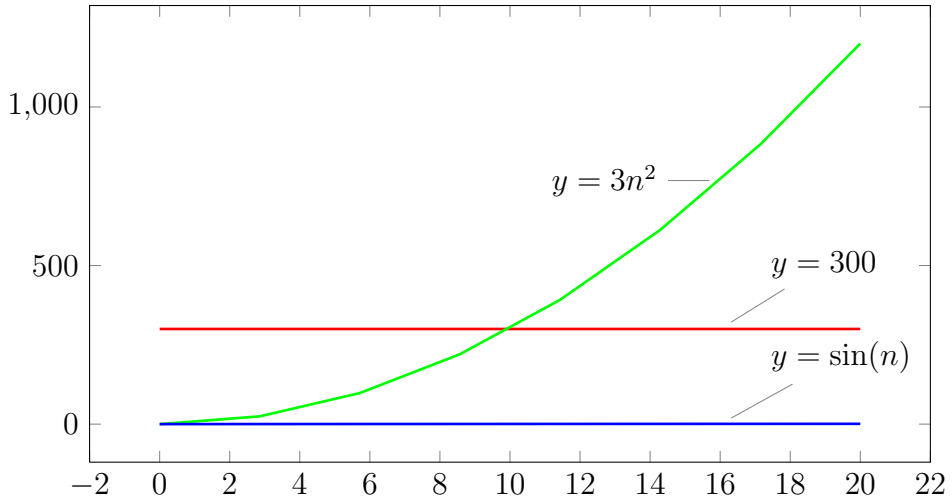
$$\text{Space}(n) = O(1)$$

Note that every if you use a different variable for the second for-loop, say variable j, the space complexity is still $O(1)$. □

Solution to Exercise 102.5.4.

debug:
exercises/big-O-100-3n2+sinn/answer.tex

(a)



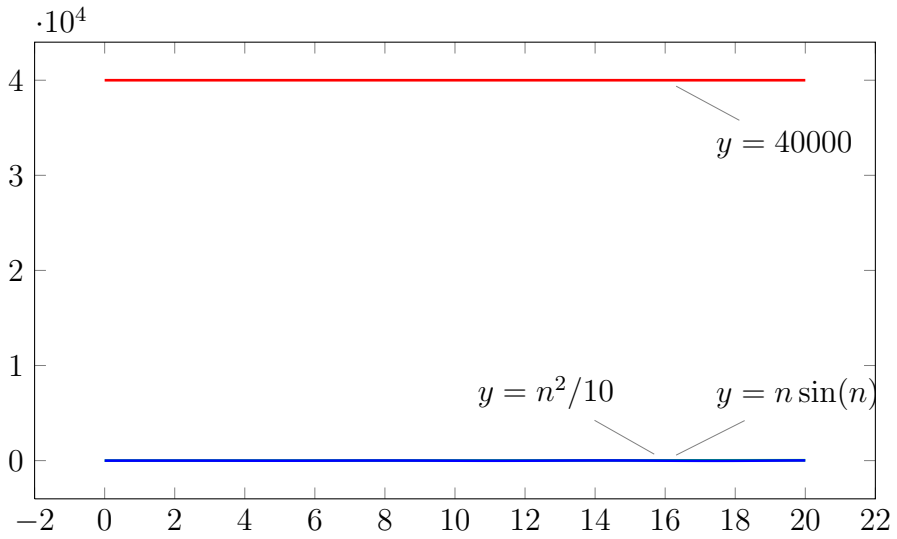
(b) Graphically, for $n \geq 11$, $3n^2$ dominates 300 and $\sin(n)$.

(c) $f(n) = O(n^2)$.

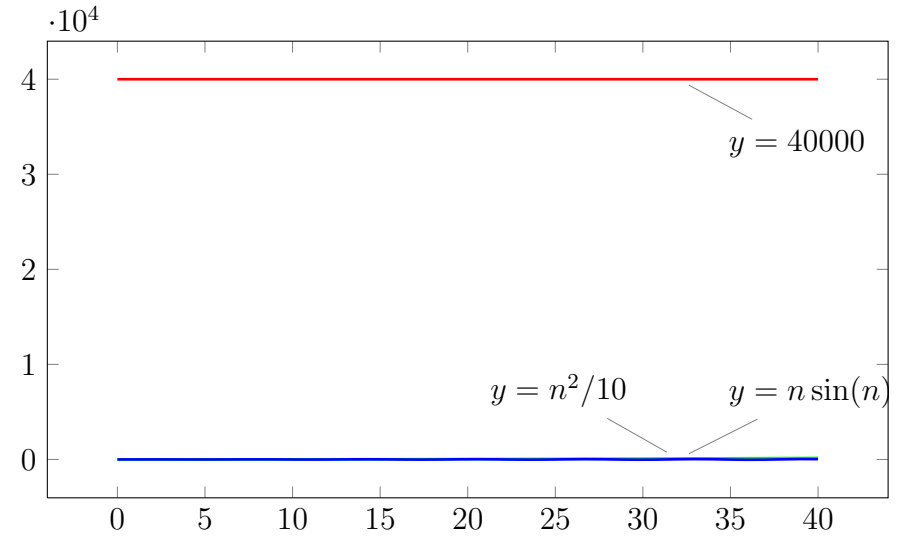
Solution to Exercise 102.5.5.

debug:
exercises/40000-1-10-n2-nsinn/answer.tex

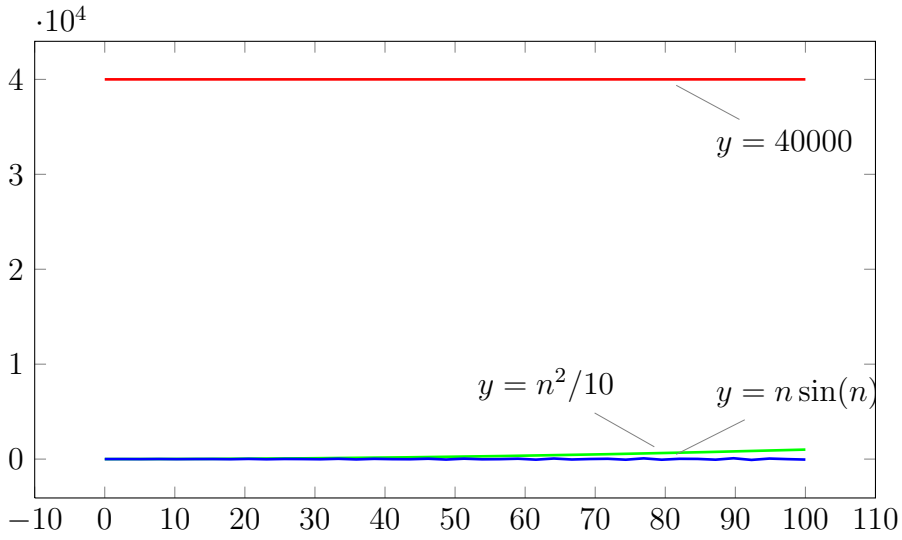
(a) For n in $[0, 20]$, we have



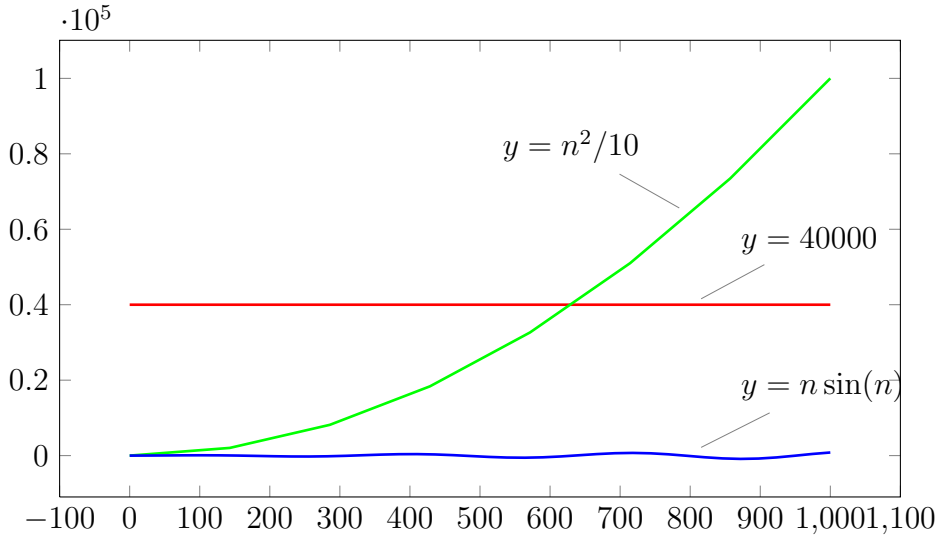
For n in $[0, 40]$, we have



For n in $[0, 100]$, we have



For n in $[0, 1000]$, we have



(b) Graphically, for $n \geq 700$, $n^2/10$ dominates 40000 and $n\sin(n)$. Therefore the big-O of $f(n)$ is determined by $n^2/10$.

(c) The big-O of $n^2/10$ is $O(n^2)$ (by replacing 1/10 by 1). Therefore $f(n) = O(n^2)$. □

Solution to Exercise 102.5.6.

debug: exercises/problem-1/answer.tex

Answer: $f(n) = O(n^2)$

Solution not provided.

102.6 Best, average, and worst runtime debug:

best-average-and-worst-time.tex

Now let me consider an algorithm when the body of the for-loop contains an `if`-statement.

The following computes the index in an (unsorted) array `x` of size `n` where `target` is first found, i.e., this is the linear search algorithm:

```
index = -1
for i = 0, 1, 2, ..., n - 1:
    if x[i] is target:
        index = i
        break
```

Here's the above written in a way that makes timing calculation easier:

```
                                time
         index = -1             t1
         i = 0                  t2
LOOP:    if i >= n:             t3
             goto ENDLOOP       t4
         if x[i] is not target: t5
             goto ELSE          t6
         index = i              t7
         goto ENDLOOP           t8
ELSE:    i = i + 1              t9
         goto LOOP              t10
ENDLOOP:
```

(For non-programmers: when I say array `x` is an array of size `n` I mean that you have `x[0]`, `x[1]`, ..., `x[n-1]` which is similar to the mathematical idea of a bunch of variables with scripts x_0, x_1, ..., x_{n-1}. `break` means to get out of the current loop. The time to access the `i`–th element `x[i]` of `x` is constant.)

The amount of time needed of course depends on how fast we hit `target`: if `target` happens to be at index 0, of course the algorithm ends quickly. This is the best case scenario. And if `target` is at index $n-1$ or if it's not even in the array, the algorithm would run longer since you would have to scan the whole array. These are the worst case scenarios.

Note that in my first timing example that computes the sum from 1 to n, I converted a for-loop into statements of a simplified language that uses the goto and the conditional branching statement. For the above example, it's easy to see that if you have an algorithm that has an if-else statement such as

```
if x > 1:
    statement-1
    statement-2
else:
    statement-3
    statement-4
```

you can rewrite that in the simplified language as

```
          if x <= 1:
              goto ELSE
          statement-1
          statement-2
          goto ENDIF
ELSE:     statement-3
          statement-4
ENDIF:
```

Note that the conditional branching leads to the else case, and therefore the boolean condition is the opposite of the boolean condition of the if-else statement. Now let's go back to the timing calculations.

Here's the timing calculation for the best scenario:

```
                                  time    number of times
          index = -1              t1      1
          i = 0                   t2      1
LOOP:     if i >= n:              t3      1
              goto ENDLOOP        t4      0
          if x[i] is not target:  t5      1
              goto ELSE           t6      0
          index = i               t7      1
          goto ENDLOOP            t8      1
ELSE:     i = i + 1               t9      0
          goto LOOP               t10     0
ENDLOOP:
```

The time taken is

$$\text{Time taken } = A$$

for some constant A. In this case the constant function $f(n) = A$ is a constant multiple of the simpler function $g(n) = 1$. Since we ignore multiples, I can say that for this "optimistic" case, the runtime is $O(1)$. Mathemtically, I can

write this:

$$\text{Time taken } = A = O(1)$$

For the worst case where the `target` is the last element of the array, i.e., at index `n - 1`, we have the following:

```
                                  time     number of times
        index = -1                t1       1
        i = 0                     t2       1
LOOP:   if i >= n:                t3       n
            goto ENDLOOP          t4       0
        if x[i] is not target:    t5       n
            goto ELSE             t6       n - 1
        index = i                 t7       1
        goto ENDLOOP              t8       1
ELSE:   i = i + 1                 t9       n - 1
        goto LOOP                 t10      n - 1
ENDLOOP:
```

The time taken is

$$\begin{aligned}\text{Time taken } &= t_1 + t_2 + t_7 + t_8 + (t_3 + t_5)n + (t_6 + t_9 + t_{10})(n-1) \\ &= (t_3 + t_5 + t_6 + t_9 + t_{10})n + (t_1 + t_2 - t_6 + t_7 + t_8 - t_9 - t_{10}) \\ &= An + B\end{aligned}$$

for constants A and B. In this case the runtime function is big-O of n, i.e., it is $O(n)$. I write: The time taken is

$$\text{Time taken } = An + B = O(n)$$

For the worst case where the `target` is not found we have the following:

```
                                  time     number of times
        index = -1                t1       1
        i = 0                     t2       1
LOOP:   if i >= n:                t3       n + 1
            goto ENDLOOP          t4       1
        if x[i] is not target:    t5       n
            goto ELSE             t6       n
        index = i                 t7       0
        goto ENDLOOP              t8       0
ELSE:   i = i + 1                 t9       n
        goto LOOP                 t10      n
```

```
ENDLOOP:
```

The time taken is

$$\begin{aligned}\text{Time taken} &= t_1 + t_2 + t_4 + n(t_3 + t_5 + t_6 + t_9 + t_{10}) + (n+1)t_3 \\ &= n(t_3 + t_5 + t_6 + t_9 + t_{10}) + t_1 + t_2 + t_3 + t_4 \\ &= An + B \\ &= O(n)\end{aligned}$$

for constants A and B. In the second worst case scenario, the runtime function is also big-O of n, i.e., it is $O(n)$.

In summary the best runtime and the two worst runtimes are

$$\begin{aligned}A_1 &= O(1) \\ A_2 n + B_2 &= O(n) \\ A_3 n + B_3 &= O(n)\end{aligned}$$

Usually performance of algorithms are described in terms of best, worst, and average times. If you want to lump up all the runtimes (i.e., you don't really want to be that specific), we want to say that runtime is $O(n)$, referring to the absolute worst scenario.

In other words you should think of the big-O notation $O(n)$ as some kind of *upper* bound approximation, i.e., all the above times are bounded above by a large enough multiple of n:

$$\begin{aligned}A_1 &\leq C \cdot n \\ A_2 n + B_2 &\leq C \cdot n \\ A_3 n + B_3 &\leq C \cdot n\end{aligned}$$

where C is some humongous fixed number and the above inequalities are true for large values of n.

While talking about a specific algorithm, to distinguish between the best, worst, and average runtime, I will write $T_b(n)$, $T_w(n)$, and $T_a(n)$. (Technically speaking there are two different worst case scenarios for the linear search above. But they both yield the same big-O anyway.)

If I don't say which of the three cases, I always mean the worst case $T_w(n)$.

I have shown you above that for the linear search

$$T_b(n) = O(1)$$
$$T_w(n) = O(n)$$

The average case is a little more complicated. In the above linear search algorithm, we looked at three cases (one best and two worst). But in general, the `target` can be anywhere and you would have to account for all of them, measure their times, say we call them $T_0(n)$, ..., $T_n(n)$ where the algorithm has $n+1$ cases ($T_0(n)$ corresponding to the time where the `target` is at index 0, ... and $T_n(n)$ corresponding to the time where the `target` is not in the array at all) and then take the average:

$$\frac{T_0(n) + \cdots + T_n(n)}{n+1}$$

But that assume something: That the cases corresponding to times $T_0(n)$, ..., $T_n(n)$ are equally likely to occur.

Depending on specific scenario, there are cases that might occur more frequently. For instance if in the above, the case where the index 0 occurs twice as frequently as the rest, then the average would be

$$\frac{2T_0(n) + \cdots + T_n(n)}{n+2}$$

To analyze the average runtime for complicated cases require a little more probability theory. For now we will only handle very simplisitic average cases.

In general, to compute *an* (not *the*) average runtime, you have to

(1) state what cases you're average over and
(2) what is the likelihood of each case.

When the cases are not stated, they are usually obvious. Also, if the likelihood of each case is not stated, then it is assumed that all cases are equally likely.

For many algorithms and many average scenarios, the average runtimes tend to be the same as the worst runtime when you fudge the functions using big-O.

Now that I've explained how to compute the average runtime, let's compute the average runtime of the linear search assuming that we are only averaging over the cases where `target` is at index $0, 1, 2, ..., n-1$. Note that I'm not considering the case where `target` is not in the array. If you like you can think

of this as the "average runtime for a successful search". (In a later section, I'll include the case where the `target` is not in the array – this is just slightly more complicated.)

Let me assume that `target` is at index value k where $0 \leq k \leq n-1$. Here's the number of times each statement will execute in this case:

```
                                  time    number of times
         index = -1               t1      1
         i = 0                    t2      1
LOOP:    if i >= n:               t3      k + 1
             goto ENDLOOP         t4      0
         if x[i] is not target:   t5      k + 1
             goto ELSE            t6      k
         index = i                t7      1
         goto ENDLOOP             t8      1
ELSE:    i = i + 1                t9      k
         goto LOOP                t10     k
ENDLOOP:
```

Therefore time taken for this case is

$$\begin{aligned} T_k(n) &= (t_1 + t_2 + t_7 + t_8) + (t_3 + t_5)(k+1) + (t_6 + t_9 + t_{10})k \\ &= (t_3 + t_5 + t_6 + t_9 + t_{10})k + (t_1 + t_2 + t_3 + t_5 + t_7 + t_8) \end{aligned}$$

for $k = 0, 1, 2, ..., n-1$. Let T_n be the time corresponding to the case where `target` is *not* in the array. Earlier, I have already compute the runtime for the case where `target` is not in the array:

$$T_n(n) = (t_3 + t_5 + t_6 + t_9 + t_{10})n + t_1 + t_2 + t_3 + t_4$$

To simplify the constants (because later, we'll be computing by big-O anyway), let

$$\begin{aligned} A &= t_3 + t_5 + t_6 + t_9 + t_{10} \\ B &= t_1 + t_2 + t_3 + t_5 + t_7 + t_8 \\ C &= t_3 + t_5 + t_6 + t_9 + t_{10} \\ D &= t_1 + t_2 + t_3 + t_4 \end{aligned}$$

Here's a summary:

$$\begin{aligned} T_k(n) &= Ak + B, \quad (k = 0, 1, 2, ..., n-1) \\ T_n(n) &= Cn + D \end{aligned}$$

OK, now I'm going to compute the average runtime assuming

- the `target` is in the array with equal likelihood at all index positions.

Remember: This average runtime scenario does *not* include the case where the `target` is not in array `x`. So I need to average over $T_0(n), \ldots, T_{n-1}(n-1)$ ... do *not* include $T_n(n)$.

This average runtime is

$$\begin{aligned}
T_a(n) &= \frac{1}{n}\left(T_0(n) + T_1(n) + \cdots + T_{n-1}(n)\right) \\
&= \frac{1}{n}\left((A\cdot 0 + B) + (A\cdot 1 + B) + \cdots (A\cdot(n-1) + B)\right) \\
&= \frac{1}{n}\left(A\cdot(0 + 1 + \cdots + (n-1)) + Bn\right) \\
&= \frac{1}{n}\left(A\cdot\frac{n(n-1)}{2} + Bn\right) \\
&= \frac{1}{n}\left(\frac{A}{2}n^2 + \left(B - \frac{1}{2}A\right)n\right) \\
&= \frac{A}{2}n + \left(B - \frac{1}{2}A\right) \\
&= O(n)
\end{aligned}$$

That's it!

debug: exercises/best-average-worst-0/question.tex

Exercise 102.6.1. The linear search algorithm searches from index value 0 to the last index value. The *reverse* linear search is pretty much the same as the linear search except that it starts with the last index value and moves toward the 0 index value. For the following, assume as before the the array is `x` and the size of the array is `n`.

```
index = -1
for i = n - 1, n - 2, ..., 1, 0:
    if x[i] is target:
        index = i
        break
```

Here's the reverse linear search algorithm with goto statements:

```
                                  time
         index = -1               t1
         i = n - 1                t2
LOOP:    if i <= -1:              t3
             goto ENDLOOP         t4
         if x[i] is not target:   t5
             goto ELSE            t6
         index = i                t7
         goto ENDLOOP             t8
ELSE:    i = i - 1                t9
         goto LOOP                t10
ENDLOOP:
```

(a) Assume the best case, i.e., the `target` is at index `n - 1`. Write down the number of times each of the statement is executed. Compute the total runtime for this case, $T_b(n)$ and then write down the big-O of this function.

(b) Assume the worst case where the `target` is not found in the array. Write down the number of times each of the statement is executed. Compute the total runtime for this case, $T_w(n)$ and then write down the big-O of this function.

(c) Assume the worst case where the `target` is only at index 0. Write down the number of times each of the statement is executed. Compute the total runtime for this case, $T_w(n)$ and then write down the big-O of this function.

(d) Assume now that `target` is at index value k. What is the runtime for this case? This should be a formula involving k and the constants `t1`, `t2`, ... (When you set k to 0, you should get the answer in (c) and when you set i to $n-1$, you should get the answer in (a).) Write it as a polynomial in k, given the coefficient of the polynomial simple constant names A, B, ...

(e) Part (d) should give you n values, say T_0, ..., T_{n-1}, i.e., T_k is the runtime for the case where the `target` is at index `k`. Assume that all the above cases are equally likely. Compute the average of these n values to obtain the average runtime $T_a(n)$. (For now we'll forget about the case where `target` is not in the array.) (Go to solution, page 4049) □

Exercise 102.6.2. The following is the maximum of an array algorithm. The algorithm computes and stores the maximum of array `x` of size `n` and stores it in `M`.

debug: exercises/max/question.tex

```
M = x[0]
for i = 1, 2, ..., n - 1:
    if x[i] > M:
        M = x[i]
```

(a) Compute the best runtime where the body of the `if` statement is never executed.

(b) Compute the worst runtime (where the body of the `if` statement is always executed for each iteration of the loop.

(Go to solution, page 4050) □

debug: exercises/best-average-worst-1/question.tex

Exercise 102.6.3. The following algorithm computes the minimum value in an array `x` of size `n`:

```
m = x[0]
for i = 1, 2, ..., n - 1:
    if x[i] < m:
        m = x[i]
```

(a) Compute the best runtime where the body of the `if` statement is never executed.

(b) Compute the worst runtime (where the body of the `if` statement is always executed for each iteration of the loop. (Go to solution, page 4051) □

debug: exercises/best-average-worst-2/question.tex

Exercise 102.6.4. The following algorithms performs some kind of shuffling on an array `x` of size `n`:

```
seed(0)
for i = 0, 1, 2, ..., n * n - 1:
    j = rand() % n
    k = rand() % n
    if j is not k:
        t = x[j]
        x[j] = x[i]
        x[i] = t
```

The basic idea is very simple: Pick two indices and swap the values at those indices if the indices are different. Repeat.

(a) Rewrite the above with goto statements. Assume that each line requires constant time. Assign times to each statement.
(b) Compute the big-O of the best runtime.
(c) Compute the big-O of the worst runtime.
(d) Under the assumption that 1/4 of the iterations of the for-loop has a boolean value of FALSE for the boolean expression in the if-statement:

```
    if j is not k:
```

compute the average runtime.
(e) Under the assumption that $1/n$ of the iterations of the for-loop has a boolean value of FALSE for the boolean expression in the if-statement:

```
    if j is not k:
```

compute the average runtime.

(Go to solution, page 4052) □

debug: exercises/best-average-worst-3/question.tex

Exercise 102.6.5. Here's anothering shuffling. Here I'm assuming that the original array `x` does not contain the value `-1`. I'm going to use another array `y` of the same size `n`. The idea is very simple: `-1` is used to denote "unoccupied" in array `y`. I will put the values from `x` into `y` at a random index position. If the index position in `y` is already occupied, I will move to the next, cycling back to index 0 if necessary. Once this is done, I copy the values in `y` back to `x`.

```
for i = 0, 1, 2, ..., n - 1:
    y[i] = -1

seed(0)
for i = 0, 1, 2, ..., n - 1:
    j = rand() % n
    while y[j] != -1:
        j = (j + 1) % n
    y[j] = x[i]

for i = 0, 1, 2, ..., n - 1:
    x[i] = y[i]
```

(a) Compute the best runtime of the above and then the big-O.
(b) Compute the worst runtime of the above and the big-O. (Be careful now!!! What exactly is the worst case???)

The translation of a while-loop into goto and conditional branching statements is similar to the for-loop. Here's the translation of the above while-loop:

```
...
    while y[j] != -1:
        j = (j + 1) % n
    y[j] = x[i]
...
```

into goto statements:

```
...
LOOP3:      if y[j] == -1:
                goto ENDLOOP3
            j = (j + 1) % n
            goto LOOP3

ENDLOOP3:   y[j] = x[i]
...
```

(Go to solution, page 4053) □

Solutions

Solution to Exercise 102.6.1.

debug: exercises/best-average-worst-0/answer.tex

Solution not provided.

Solution to Exercise 102.6.2.

debug: exercises/max/question.tex

(a)

```
                              time    number of times
         M = x[0]             t1      1
         i = 1                t2      1
LOOP:    if i >= n:           t3      n
            goto ENDLOOP      t4      1
         if x[i] <= M:        t5      n - 1
            goto ELSE         t6      n - 1
         M = x[i]             t7      0
ELSE:    i = i + 1            t8      n - 1
         goto LOOP            t9      n - 1
ENDLOOP:
```

$$T_b(n) = (t_3 + t_5 + t_6 + t_8 + t_9)n + (t_1 + t_2 + t_4 - t_5 - t_6 - t_8 - t_9) \quad = An + B$$

where A, B are constants. Therefore

$$T_b(n) = O(n)$$

(b)

```
                              time    number of times
         M = x[0]             t1      1
         i = 1                t2      1
LOOP:    if i >= n:           t3      n
            goto ENDLOOP      t4      1
         if x[i] <= M:        t5      n - 1
            goto ELSE         t6      0
         M = x[i]             t7      n - 1
ELSE:    i = i + 1            t8      n - 1
         goto LOOP            t9      n - 1
ENDLOOP:
```

$$T_w(n) = (t_3 + t_5 + t_7 + t_8 + t_9)n + (t_1 + t_2 + t_4 - t_5 - t_7 - t_8 - t_9) \quad = Cn + D$$

where C, D are constants. Therefore

$$T_w(n) = O(n)$$

Solution to Exercise 102.6.3.

debug: exercises/best-average-worst-1/answer.tex

Solution not provided. See answers below.

(a) $T_b(n) = O(n)$

(b) $T_w(n) = O(n)$

Solution to Exercise 102.6.4.

debug: exercises/best-average-worst-2/answer.tex

Solution not provided. See answers below.

(b) $T_b(n) = O(n^2)$

(c) $T_w(n) = O(n^2)$

(d) $T(n) = O(n^2)$

(e) $T(n) = O(n^2)$

Solution to Exercise 102.6.5.

debug: exercises/best-average-worst-3/answer.tex

Solution not provided. See answers below.

(a) $T_b(n) = O(n)$

(b) $T_w(n) = O(n^2)$

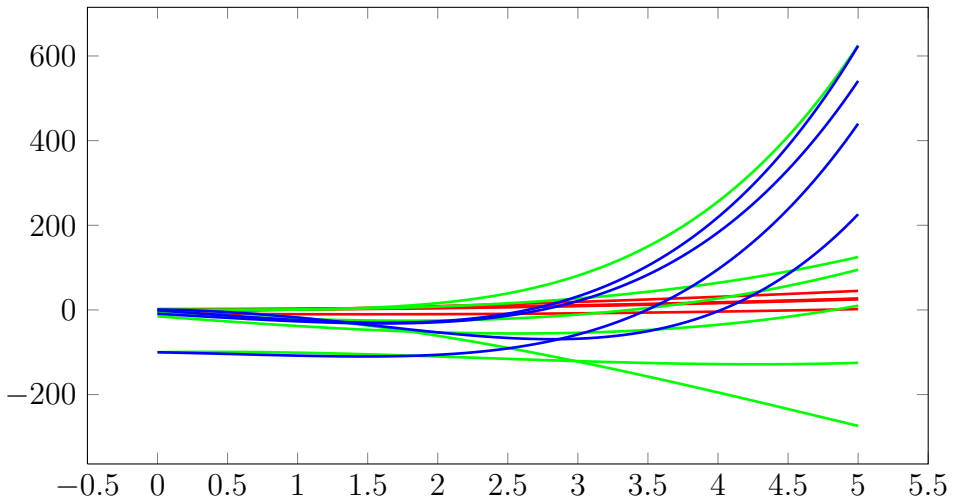
102.7 Separating polynomial functions debug:

separating-polynomial-functions.tex

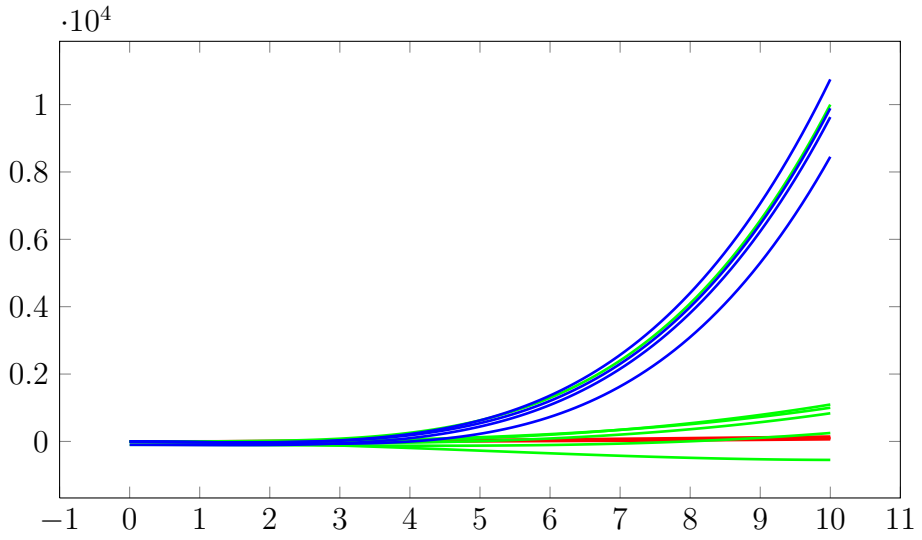
I'll be giving you the formal definition of big-O soon. But before that, I'm going to motivate the (formal) definition of big-O by talking about the way the graphs of polynomial climb. The rate at which they climb essential tells you the story of big-O among polynomials. This will give you the intuitive idea behind big-O before I hit you with the formal definition.

In this section, I will show you that when you plot polynomial functions, they bunch up into groups. These groups are very well-defined and simple: they are determined by the *degree* of polynomials.

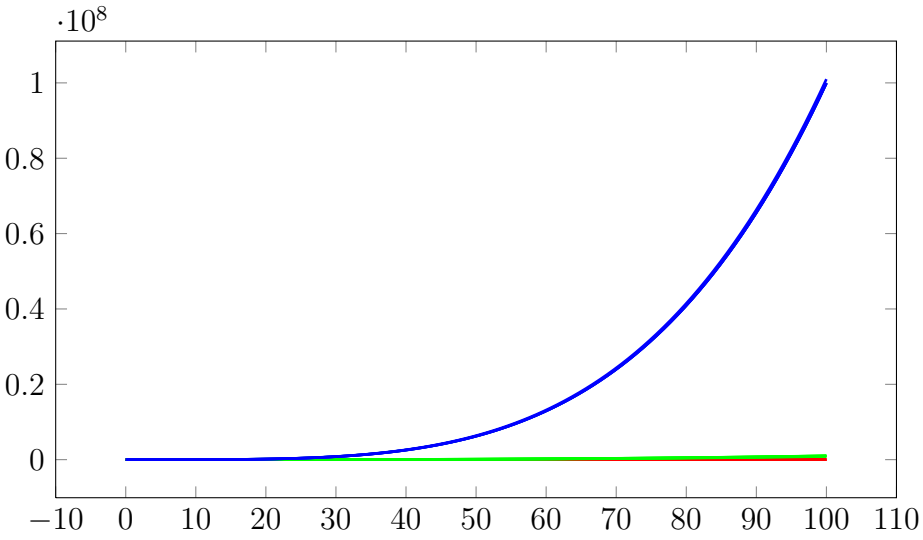
Look at this mess of 15 polynomail functions (I won't give you the polynomials just yet):



This looks like wires from a behind a rack of servers. Now if I increase the domain up to $n = 10$, we see:



Notice that the growth behavior of these functions are now clearer. Now if I increase the domain to $0 \leq n \leq 100$, we see:

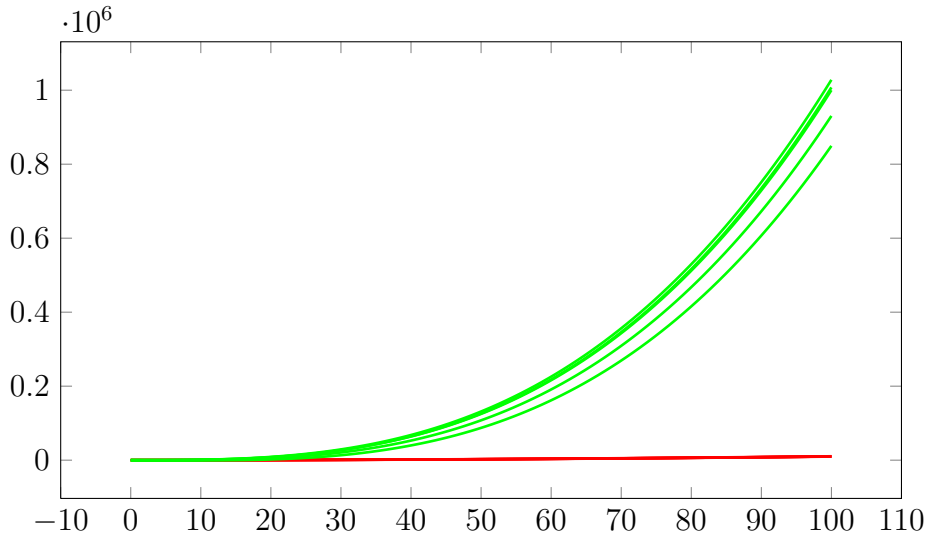


Five functions have separated away from the others. Now I reveal to you that

these five functions are:

$$
\begin{aligned}
&n^4\\
&n^4+n^2-20n-9\\
&n^4+n^3-25n-1\\
&n^4-15n^2-5n+1\\
&n^4-2n^2-7n-100
\end{aligned}
$$

Now I'm going to remove these 5 functions and plot the remaining 10:



The five functions which are higher up are:

$$
\begin{aligned}
&n^3\\
&n^3+3n^2-20n-5\\
&n^3+n^2-25n-15\\
&n^3-15n^2-5n+1\\
&n^3-7n^2+5n-100
\end{aligned}
$$

and the last group of five functions are:

$$n^2$$
$$n^2 + 2$$
$$n^2 + 1$$
$$n^2 + 5n - 5$$
$$n^2 - 3n - 8$$

As you can see, in terms of growth, for large values of n, the 15 functions

$$n^4$$
$$n^4 + n^2 - 20n - 9$$
$$n^4 + n^3 - 25n - 1$$
$$n^4 - 15x^2 - 5n + 1$$
$$n^4 - 2x^2 - 7m - 100$$
$$n^3$$
$$n^3 + 3n^2 - 20n - 5$$
$$n^3 + n^2 - 25n - 15$$
$$n^3 - 15n^2 - 5n + 1$$
$$n^3 - 7n^2 + 5n - 100$$
$$n^2$$
$$n^2 + 2$$
$$n^2 + 1$$
$$n^2 + 5n - 5$$
$$n^2 - 3n - 8$$

bunches themselves up into 3 groups determined by their degrees. You can think of the following as leaders in the three groups:

$$n^4$$
$$n^3$$
$$n^2$$

(because they are the simplest.)

In general *all* polynomial functions with 1 for the leading coefficient – such polynomials are sid to be **monic** polynomials – group themselves up into

bunches led by the following leaders:

$$1, \quad n, \quad n^2, \quad n^3, \quad n^4, \quad n^5, \quad n^6, \quad \ldots$$

The bunching up for large n is due to the fact that they grow (or climb) at the same rate for large n. This means that the function

$$n^2 - 42n + 691$$

has the same growth rate as n^2 for large n. Graphically, this means that when you zoom out (i.e., when you draw their graph for a large domain), the graphs collapse into one. Intuitively, you can think of it this way:

$$n^2 - 42n + 691 \text{ “roughly =” } n^2 \qquad \text{for large } n$$

Now let's get back to big-O. Whereas the above examples talked about

$$\ldots \text{ “roughly =” } \ldots \qquad \text{for large } n$$

big-O is more like

$$\ldots \text{ “roughly } \leq \text{” } \ldots \qquad \text{for large } n$$

Graphically, if the graph of $f(n)$ is *below* the graph to $g(n)$ *for large* n, then we can say

$$f(n) = O(g(n))$$

Now, one of the above 15 functions is this:

$$n^3 + 3n^2 - 20n - 5$$

We have already seen that the graph of $n^3 + 3n^2 - 20n - 5$ is the same as the graph of n^3 for large n. I can say

$$n^3 + 3n^2 - 20n - 5 = O(n^3)$$

But, there's more. The graph of $n^3 + 3n^2 - 20n - 5$ is roughly the graph of n^3 (for large n) and is of course the graph of n^3 is below the graph of n^4. Therefore the graph of $n^3 + 3n^2 - 20n - 5$ is roughly below the graph of n^4 (for large n). Therefore I can also say

$$n^3 + 3n^2 - 20n - 5 = O(n^4)$$

Altogether I have

$$n^3 + 3n^2 - 20n - 5 = O(n^3)$$
$$n^3 + 3n^2 - 20n - 5 = O(n^4)$$

It is also true that

$$n^3 + 3n^2 - 20n - 5 = O(n^3 + 1)$$
$$n^3 + 3n^2 - 20n - 5 = O(n^4 + 1)$$

OK, let's try another function. Here's another function from the 15:

$$n^2 + 5n - 5$$

Using the same reasoning we have all the following:

$$n^2 + 5n - 5 = O(n^2)$$
$$n^2 + 5n - 5 = O(n^3)$$
$$n^2 + 5n - 5 = O(n^4)$$

But there's a little bit more to big-O. What about multiples of the above functions? Recall that in the previous section, I said that you should ignore multiples by replacing constants with 1. It seems to mean that constants don't determine function growth rate. Is that true?

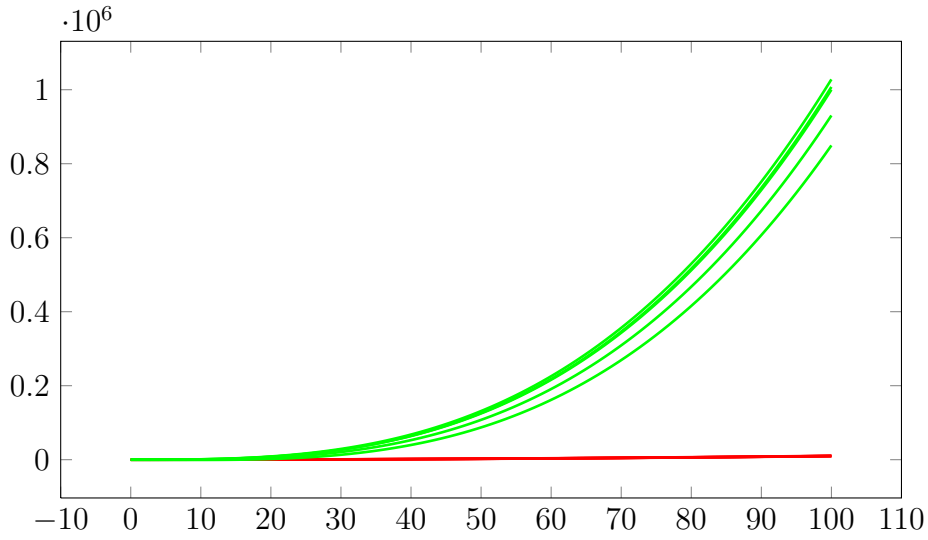
Here are the original 15 functions again:

$$
\begin{aligned}
&n^4\\
&n^4+n^2-20n-9\\
&n^4+n^3-25n-1\\
&n^4-15x^2-5n+1\\
&n^4-2x^2-7m-100\\
&n^3\\
&n^3+3n^2-20n-5\\
&n^3+n^2-25n-15\\
&n^3-15n^2-5n+1\\
&n^3-7n^2+5n-100\\
&n^2\\
&n^2+2\\
&n^2+1\\
&n^2+5n-5\\
&n^2-3n-8
\end{aligned}
$$

and now I'm going change the leading coefficients

$$
\begin{aligned}
&n^4\\
&2n^4+n^2-20n-9\\
&3n^4+n^3-25n-1\\
&4n^4-15x^2-5n+1\\
&5n^4-2x^2-7m-100\\
&6n^3\\
&7n^3+3n^2-20n-5\\
&8n^3+n^2-25n-15\\
&9n^3-15n^2-5n+1\\
&10n^3-7n^2+5n-100\\
&11n^2\\
&12n^2+2\\
&13n^2+1\\
&14n^2+5n-5\\
&15n^2-3n-8
\end{aligned}
$$

and then plot the new functions on the domain $0 \leq n \leq 100$:



The graphs have shifted vertically but the grouping is still somewhat visible. (Of course since the polynomials in each group differ by multiples you would expect their graphs to separate a little.) Regardless of the shifts, you would notice one crucial thing: If you plot a large enough domain, regardless of the multiple, a degree 3 polynomial will not beat a degree 4 polynomial.

Graphically, if you plot monic polynomials for a large enough domain, the polynomials bunches up and each bunch ultimately becomes a thin line. If you plot polynomials in general (not necessarily monic), then each group of polynomials occupy sort of a band. This means that a degree 3 polynomial cannot enter the band for the degree 4 polynomials for large n.

So here's the definition of big-O if I use graphs. In order to say

$$f(n) = O(g(n))$$

I have to show that the graph of $f(n)$ is below a multiple of $g(n)$ for large n. I'll give you more examples in the next section together with the formal definition of big-O that does not depends on graphs.

The following summarizes what I have just said. I will prove the statement later.

Theorem 102.7.1. *Let $f(n)$ be a polynomial of degree d and $g(n)$ is a polynomial of degree e If $d \leq e$, then*

$$f(n) = O(g(n))$$

In particular

$$f(n) = O(n^d)$$

If $d > e$, then

$$f(n) \neq O(g(n))$$

□

Example 102.7.1. Here are some examples that uses the above theorem.

(a) $3n^3 + n + 1 = O(n^3)$
(b) $3n^3 + n + 1 = O(n^4)$
(c) $3n^3 + n + 1 = O(n^{100})$
(d) $3n^3 + n + 1 = O(2n^3 + n^2 + n - 1)$
(e) $3n^3 + 1 \neq O(n^2)$
(f) $3n^3 + 1 \neq O(n + 10000)$

Usually we will pick the simplest and "smallest" $g(n)$ for our big-O statement in

$$f(n) = O(g(n))$$

For instance if it is true that

$$f(n) = O(n^3), \;\; f(n) = O(3n^3 - 10n + 1), \;\; f(n) = O(n^{1000})$$

then we prefer

$$f(n) = O(n^3)$$

Exercise 102.7.1. What is the big-O of the following functions. Always choose the simplest and smallest.

debug: exercises/big-O-of-functions/question.tex

(a) 100
(b) $5n^2 - 10$
(c) $42 - 3n$
(d) $5 - n + n^3 + 2n^2$
(e) $5 - n + 1000000n^3 + n^2$

(Go to solution, page 4064) □

Solutions

Solution to Exercise 102.7.1.

debug:
exercises/big-O-of-functions/answer.tex

(a) $100 = O(1)$

(b) $5n^2 - 10 = O(n^2)$

(c) $42 - 3n = O(n)$

(d) $5 - n + n^3 + 2n^2 = O(n^3)$

(e) $5 - n + 1000000n^3 + n^2 = O(n^3)$ □

102.8 Definition of big-O debug: definition-of-big-O.tex

You are now ready for the definition of big-O, at least graphically. A more mathematical methods will come later.

Suppose $f(n)$ and $g(n)$ are functions (of n of course). We say that $f(n)$ is the big-O of $g(n)$ and we write

$$f(n) = O(g(n))$$

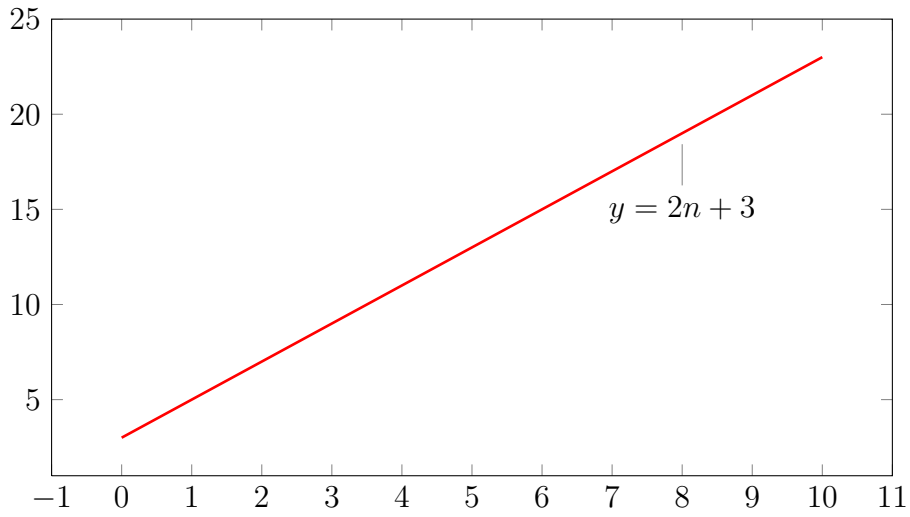
if a *multiple* of $|g(n)|$ beats (i.e., is $\geq$) $|f(n)|$, not necessarily for all n but for *all large values* of n. Of course, the absolutely value $|\cdot|$ is not necessary if the functions are positive. For this section, our $g(n)$ will be $n^0, n^1, n^2, ...$

This means that given $f(n)$ and $g(n)$ in order to say $f(n) = O(g(n))$, I need to find a C and an N such that $C|g(n)|$ beats $|f(n)|$ for n beyond N.

Let me give you some examples.

Suppose we're looking at this function

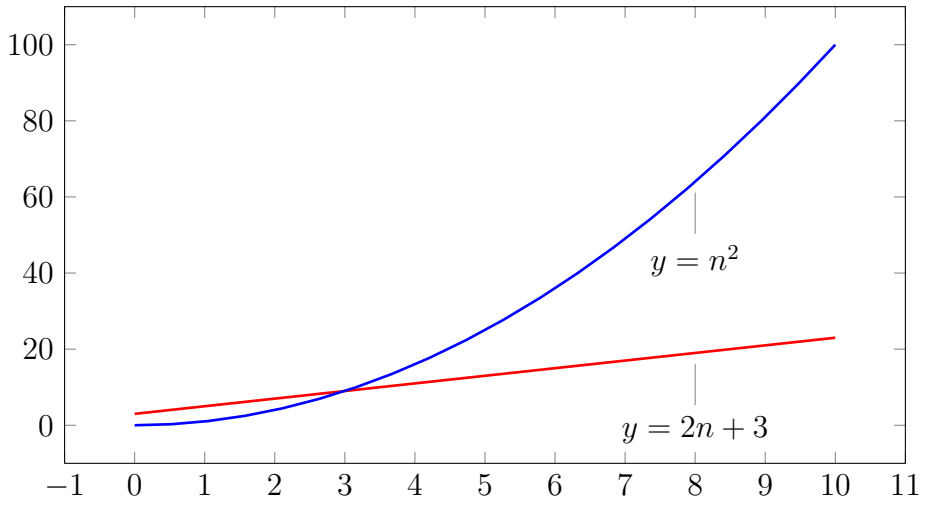
$$f(n) = 2n + 3$$



Let's compare that to

$$g(n) = n^2$$

Here they are in a plot:



You see that

$$f(n) \leq g(n) \text{ for } n \geq 3$$

If I choose $C = 1$ and $N = 3$, then for $n \geq N = 3$, we have (from the graph):

$$f(n) \leq Cg(n)$$

So we say that

$$f(n) = O(g(n))$$

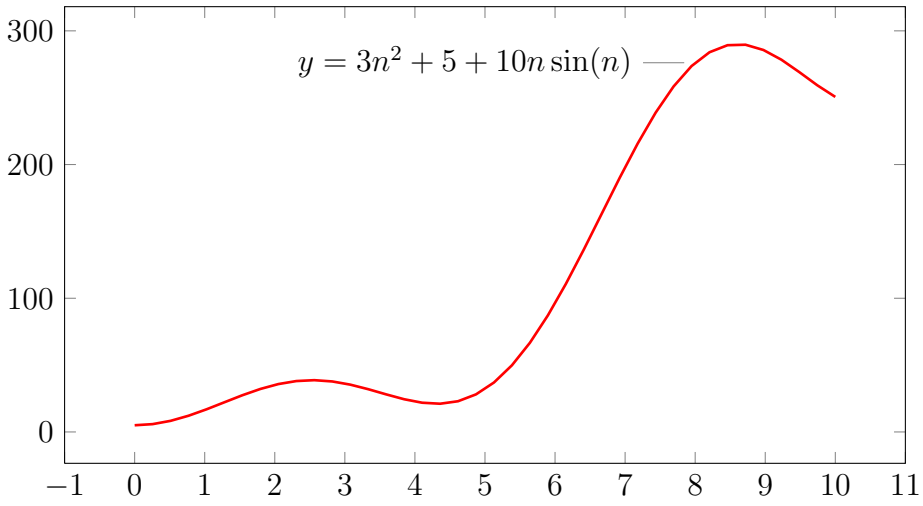
Note that the choice of C and N is not unique. You can also choose $C = 2, N = 10$.

Here's another example.

Suppose

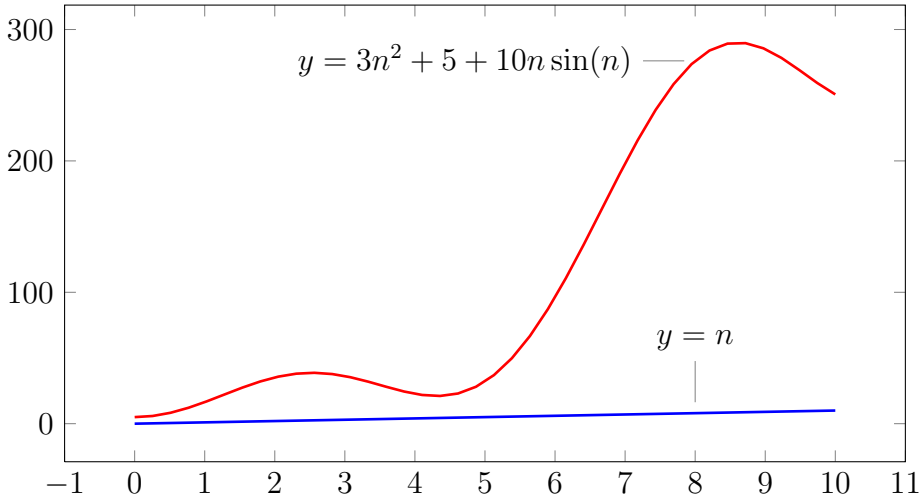
$$f(n) = 3n^2 + 5 + 10n \sin(n)$$

Here's the plot:



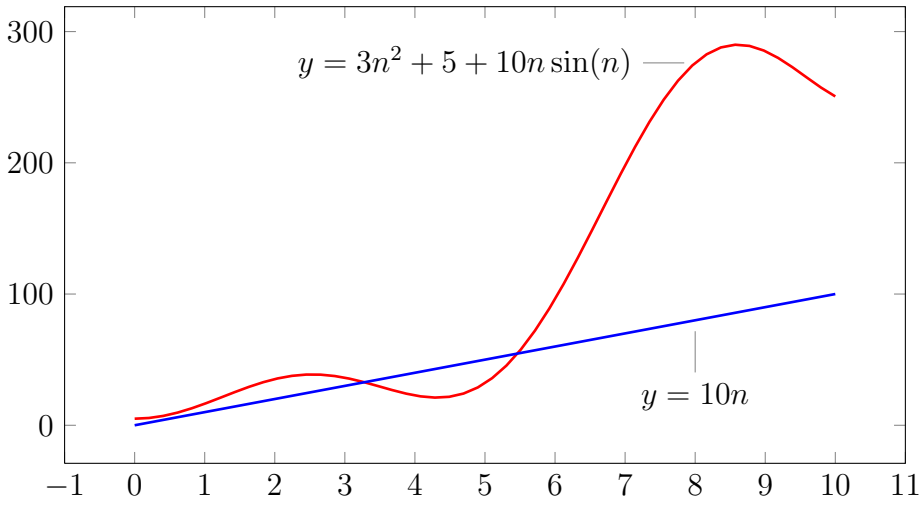
For this example $f(n)$ is positive so $|f(n)| = f(n)$. Let's see if we can *cap* it with

$$g(n) = n$$



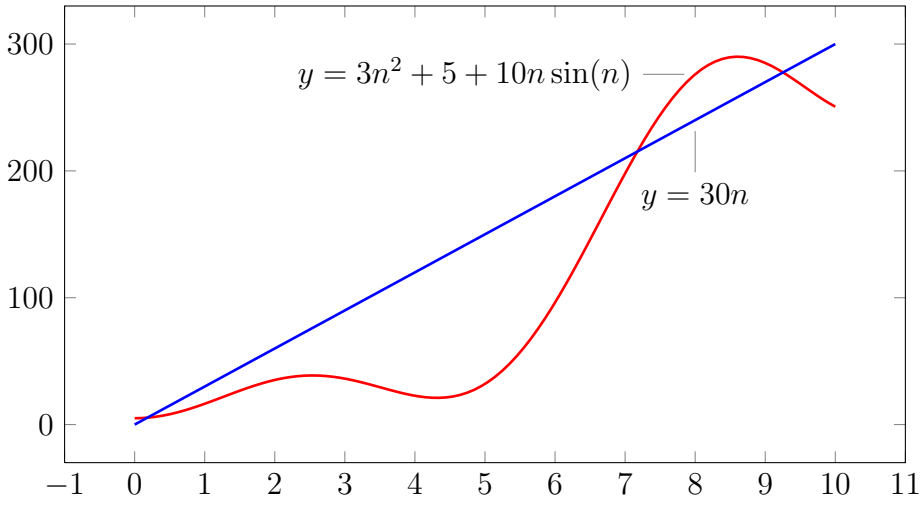
Not good. But don't forget that if we do want to say $f(n) = O(g(n))$, then we are allowed to use multiples of $g(n)$. So let's try

$$10g(n) = 10n$$



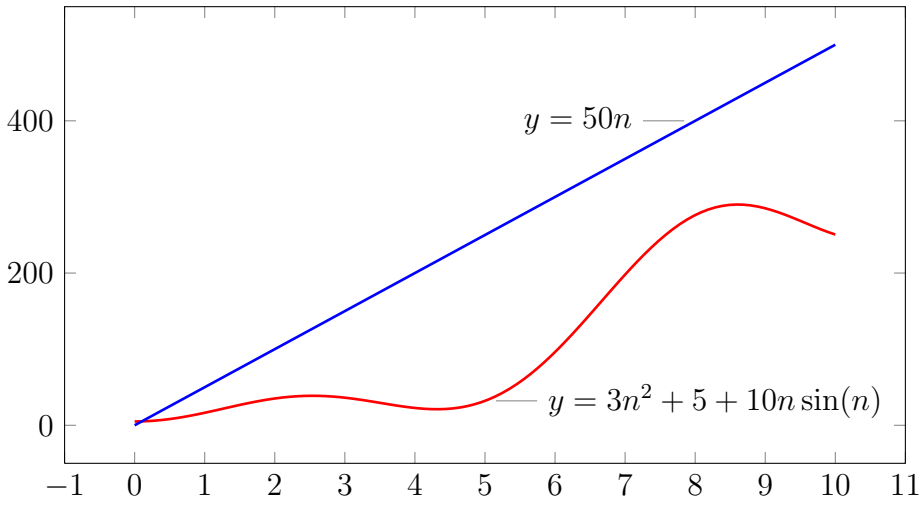
Better!!! But just by looking at the graph, my multiple of $g(n)$ must at least overcome the bump of $f(n)$ at around $n = 8.5$. Let's try

$$30g(n) = 30n$$

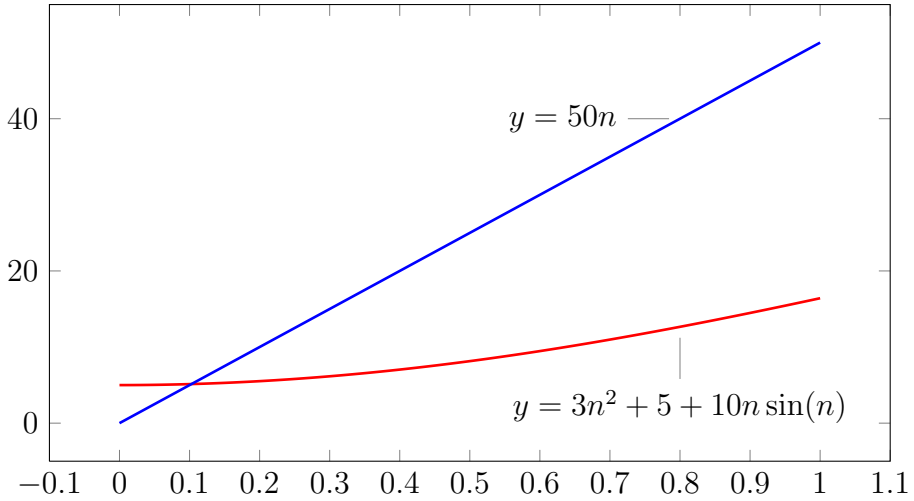


Finally let's try

$$50g(n) = 50n$$



The picture is not that clear for small n values. So let's zoom in near $n = 0$ and see how $50n$ performs against $f(n)$:



Clearly from the plot for $0 \leq n \leq 1$, we see that $50g(n)$ beats $f(n)$ after 0.1.

So from the previous two graphs can we say that for $n \geq 1$, $50g(n)$ beats $f(n)$? ... i.e., can we say

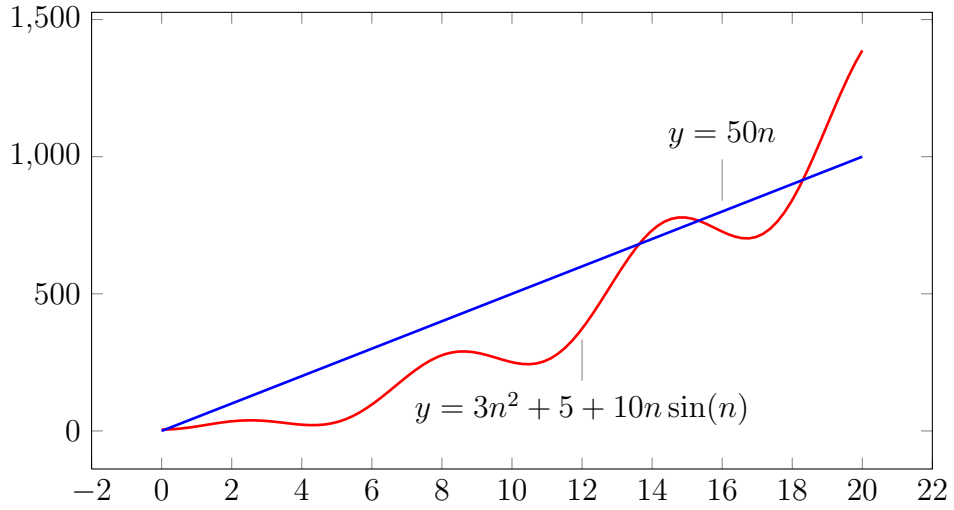
$$f(n) \leq 50g(n) \text{ for } n \geq 1$$

and conclude that $f(n) = O(g(n))$?

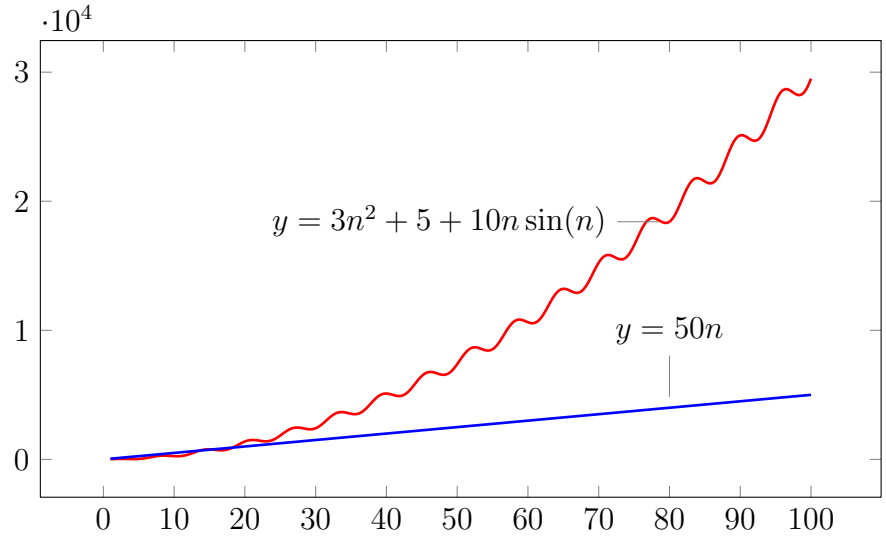
NO!!!

The problem is that our graphs cannot show *all* large values of n. In fact when

we plot for n up to 20, we see that trend changes:



It's even more revealing when I plot up to $n = 100$:



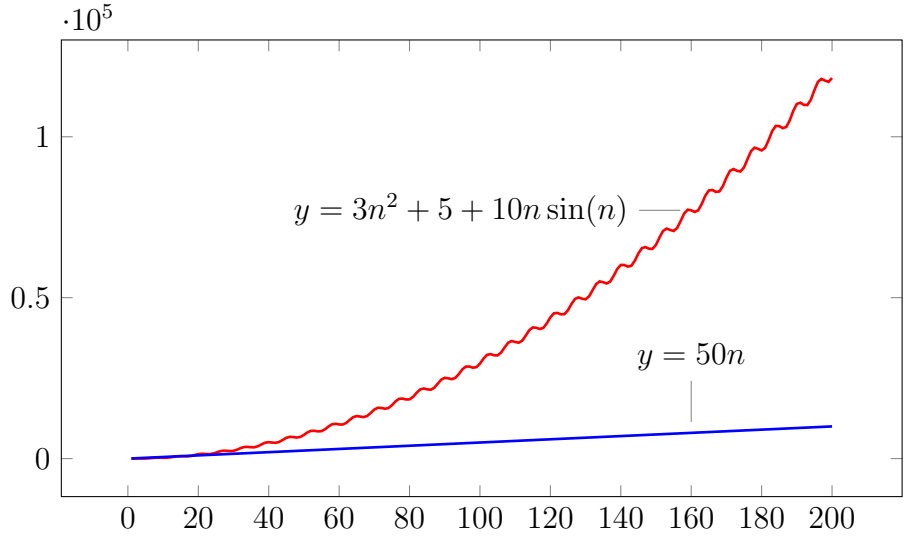
Clearly *no* straight line is going to dominate $f(n)$ because it seems that the graph of $f(n)$ *bends* up (with wiggles along the way).

Here's an important advice:

GRAPHS ARE USEFUL TOOLS BUT THEY CAN DECEIVE!!!

Let's see more of the graph to see if the pattern of bending upward with wiggles

persists. Let's plot up to $n = 200$:

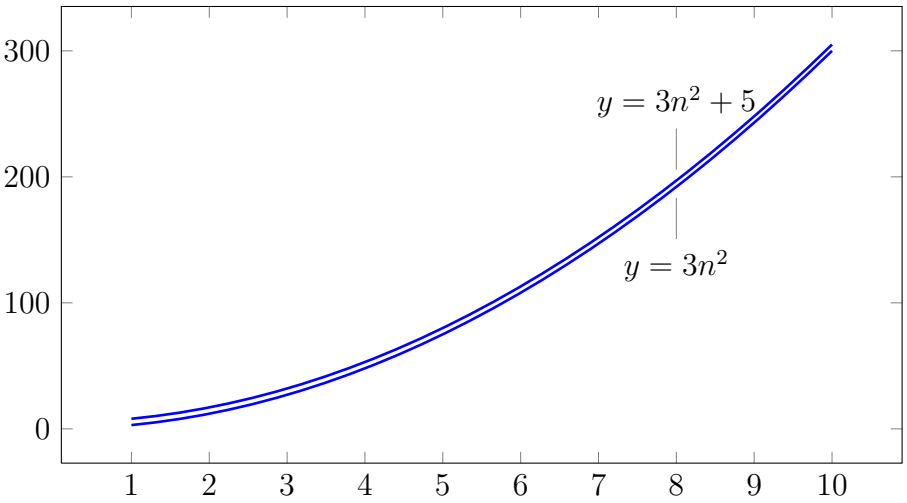


So let's abandon our $g(n) = n$ altogether.

What should we use to chase $f(n)$? Of course you know that $g(n) = n^2$ is a parabola and bends up. But does it increase (or bends) fast enough? If you look at

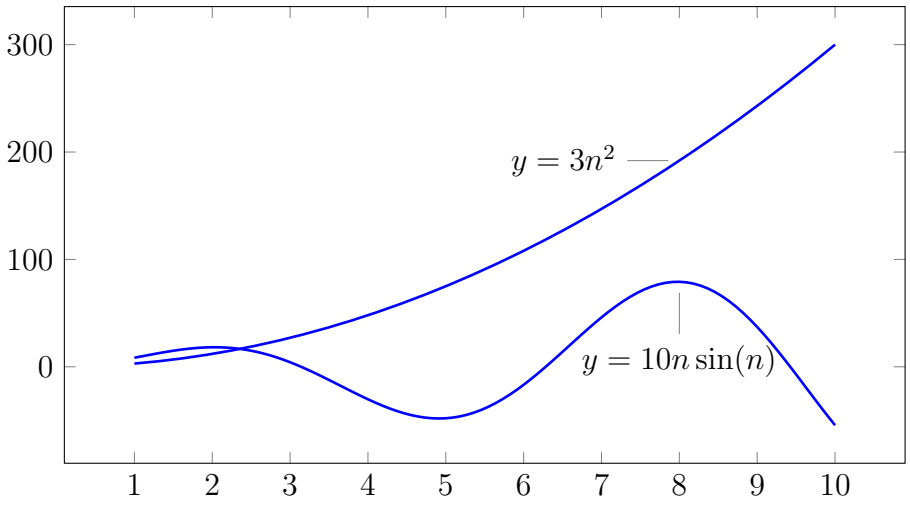
$$f(n) = 3n^2 + 5 + 10n\sin(n)$$

You see that it is made up of three functions: $3n^2$, 5, and $10n\sin n$. Of course $3n^2$ is going to beat 5. So the growth of $3n^2 + 5$ is primarily determined by $3n^2$. Let's look at them together in a graph:

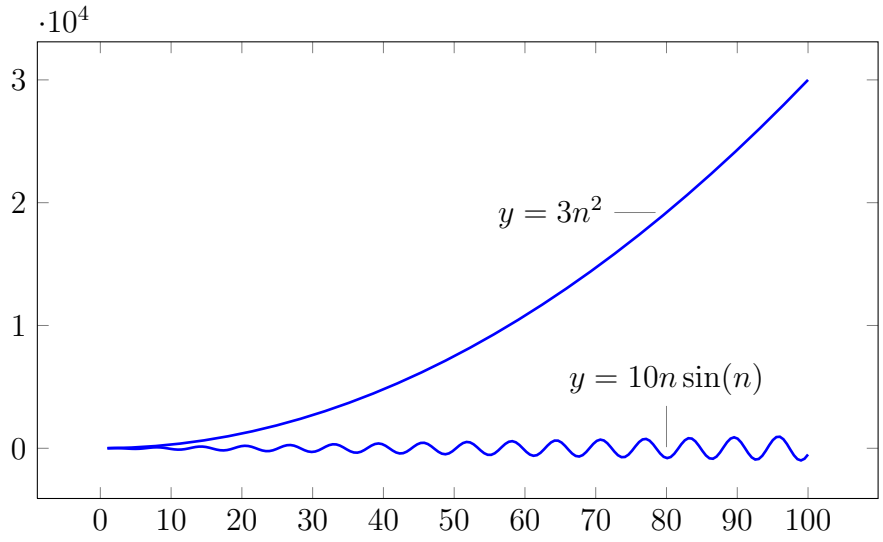


Of course since we can control $f(n)$ with multiples $g(n) = n^2$, later we just need to choose a huge multiple of n^2 to beat $3n^2+5$, for instance $1000000g(n) = 1000000n^2$.

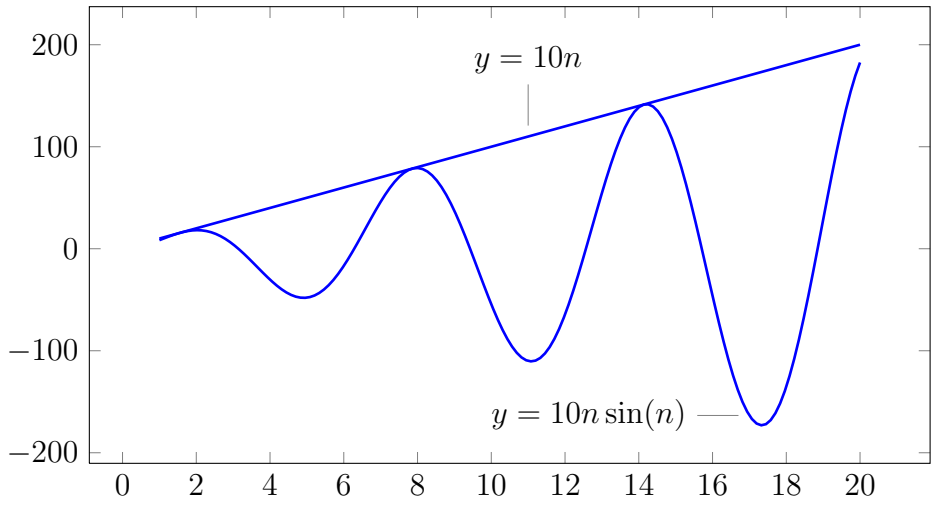
What about $10n \sin n$? If we plot that with $3g(n) = 3n^2$ we get:



To make sure that the pattern persists, I'm going to plot up to $n = 100$:



At this point, we suddenly recall that the sine function wobbles between the value of -1 and 1. Therefore $10n \sin n$ wobbles between $10n(-1)$ and $10n(+1)$, i.e., $10n \sin n$ can be at most $10n$. Let's check that with a plot for $n = 1$ to $n = 20$:



AHA!!!

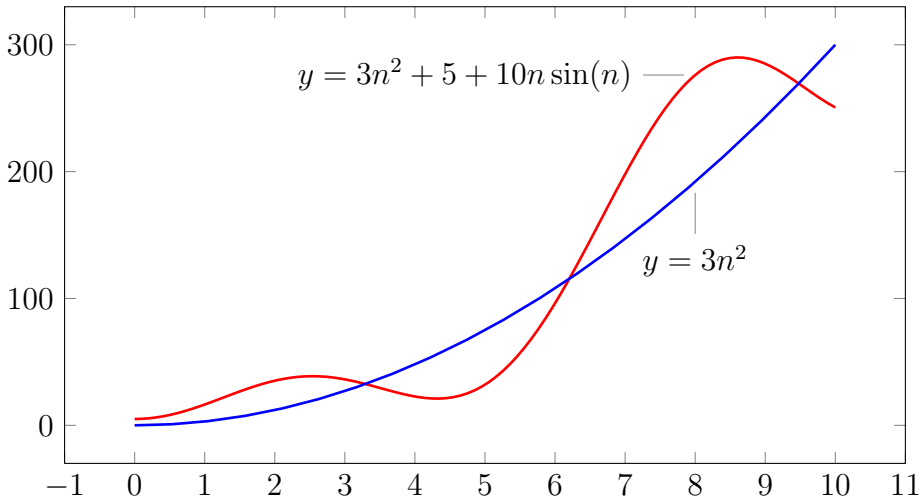
This means that $3g(n) = 3n^2$ will beat $10n \sin n$ for large values of n.

Altogether, this means that the growth of

$$f(n) = 3n^2 + 5 + 10n\sin(n)$$

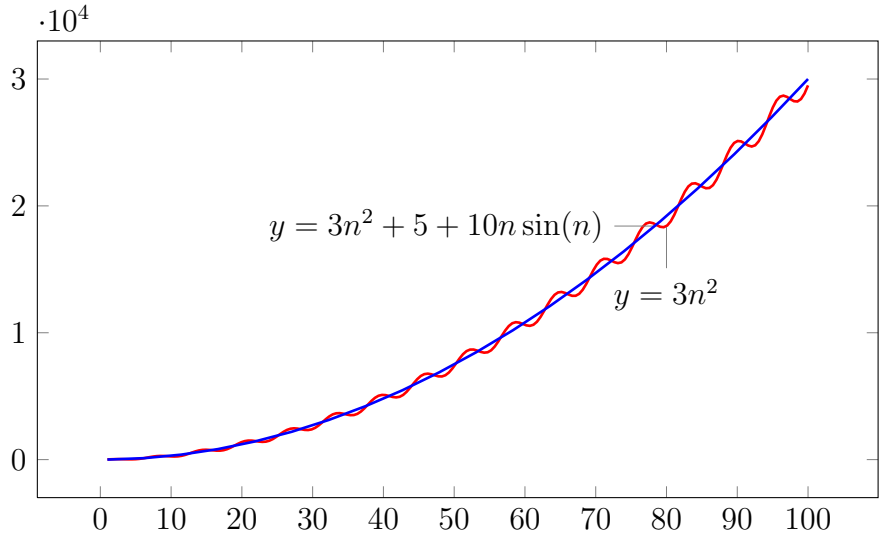
is roughly $3g(n) = 3n^2$ and therefore a higher multiple of $g(n) = n^2$ will beat $f(n) = 3n^2 + 5 + 10n\sin(n)$ for large values of n.

Here's the plot of $3g(n) = 3n^2$ and $f(n)$ for $0 \leq n \leq 10$:

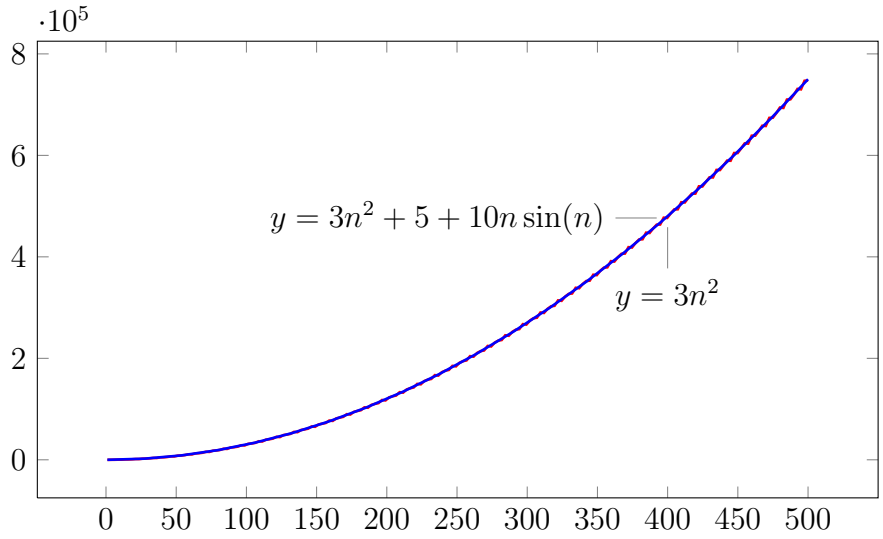


It does seem like $f(n)$ winds itself around $3g(n) = 3n^2$. To get a better feel

for the pattern of things, I'm going to plot up to $n = 100$:

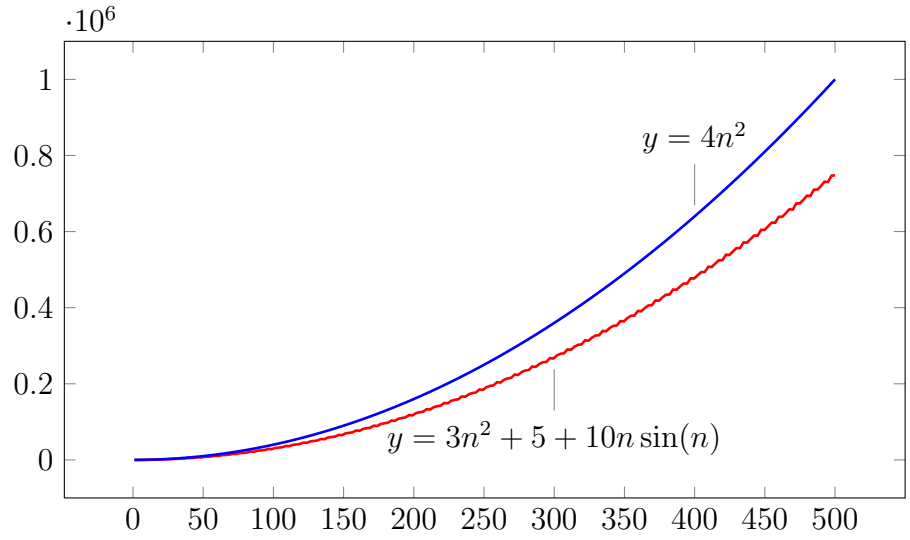


and then up to $n = 500$:

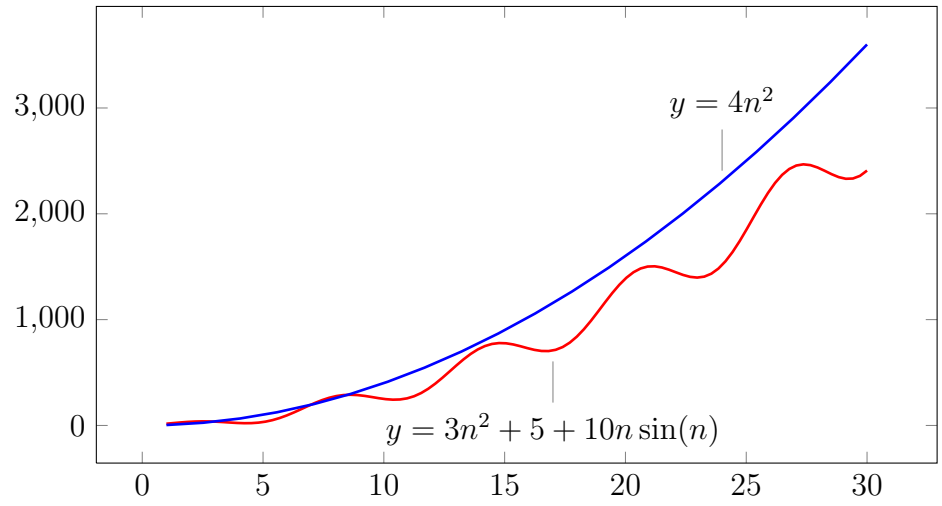


GRRREAT!!!

I see that $f(n) = 3n^2 + 5 + 10n\sin(n)$ follows $3g(n)$ very tightly. So to beat $f(n)$, let me try $4g(n) = 4n^2$ (... well ... I'm sure $(3.5)g(n)$ works too ... but I'll stick to $4g(n)$):



To see roughly when $4g(n) = 4n^2$ beats $f(n) = 3n^2 + 5 + 10n \sin n$, I zoom in and plot the graphs for $0 \leq n \leq 30$:



It looks like $4g(n)$ definitely beats $f(n)$ for *all* n such that $n \geq 10$. (Actually I can zoom in further and see if "$n \geq 9$" works as well ... but I'll just use "$n \geq 10$".)

That's it!!!

We can now say that

$$3n^2 + 5 + 10n\sin(n) \leq 4n^2 \text{ for } n \geq 10$$

Therefore, if I choose $C = 4$ and $N = 10$, I can say that

$$|f(n)| \leq C|g(n)| \text{ for } n \geq N$$

(Don't forget that $f(n)$ is positive for $n \geq 0$.) This means that I can now say

$$f(n) = O(g(n))$$

Now we are ready for the formal definition of big-O:

Definition 102.8.1. Let $f(n)$ and $g(n)$ be functions. We write

$$f(n) = O(g(n))$$

O

and say that "$f(n)$ is **big-O** of $g(n)$" if we can find a C and an N such that for all $n \geq N$, we have

big-O

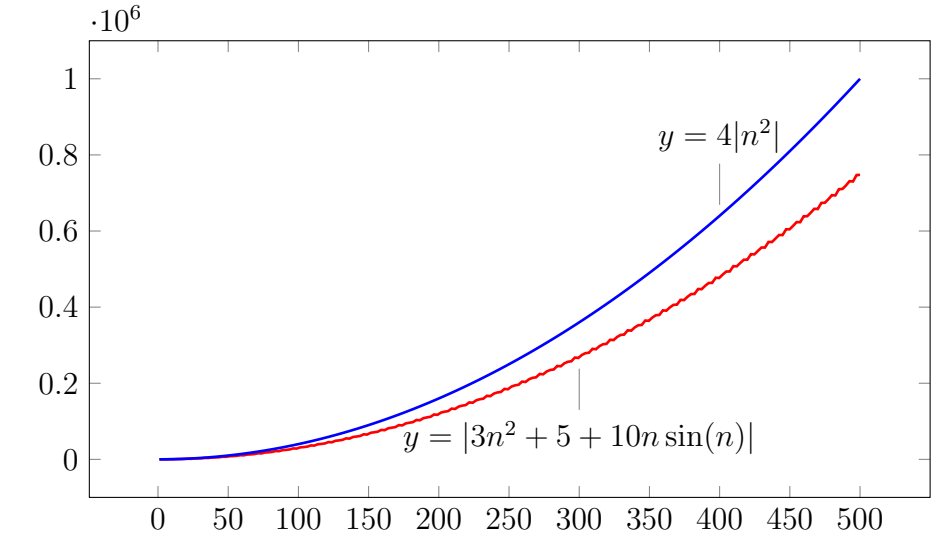
$$|f(n)| \leq C|g(n)|$$

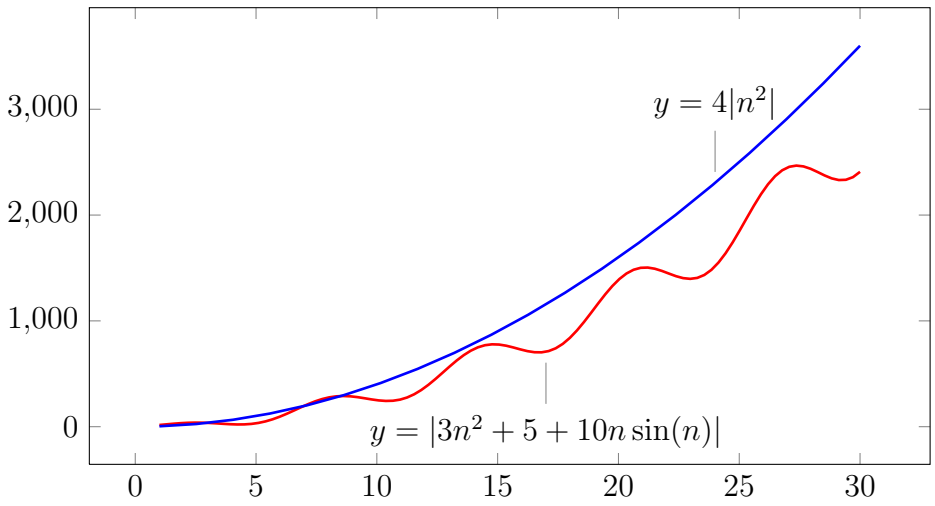
So now I'm ready to give a proper presentation of my solution to the previous problem:

Example 102.8.1. Show graphically that if $f(n) = 3n^2 + 5 + 10n\sin(n)$ and $g(n) = n^2$, then

$$f(n) = O(g(n))$$

Solution. Let $N = 10$ and $C = 4$. From the following graphs:





we see that, for $n \geq N$, we have:

$$|3n^2 + 5 + 10n\sin(n)| \leq C|n^2|$$

i.e.,

$$|f(n)| \leq C|g(n)|$$

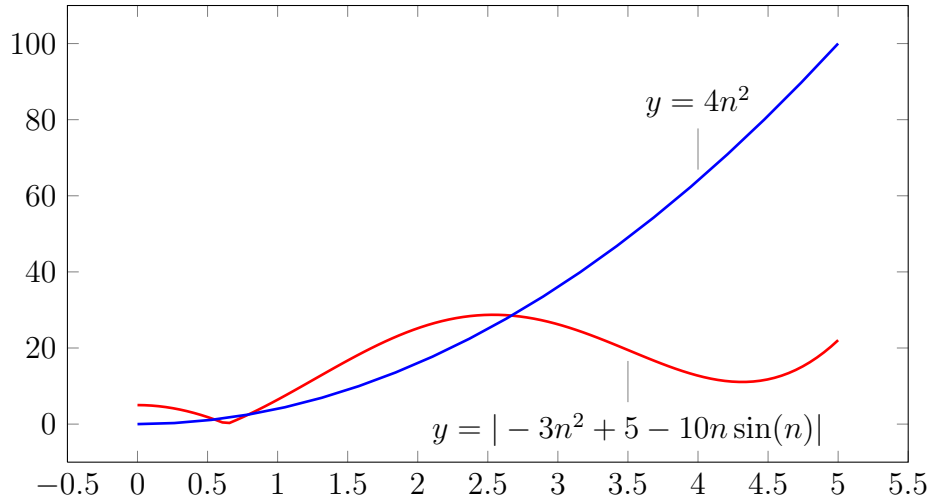
Hence $f(n) = O(g(n))$. $\square$

Remember that technically speaking you cannot prove a mathematical fact like $f(n) = O(g(n))$ using graphs because graph cannot possible show you that a multiple of $|g(n)|$ beats $|f(n)|$ for *all* sufficiently large n. A graph can only show you the graphs up to some *finite* n. How would you know that the graph of $|f(n)|$ won't suddenly shoot up and overtake $4|g(n)|$ at $n = 1000000000000$? Later, I'll show you how to prove big-O statements without a shadow of doubt.

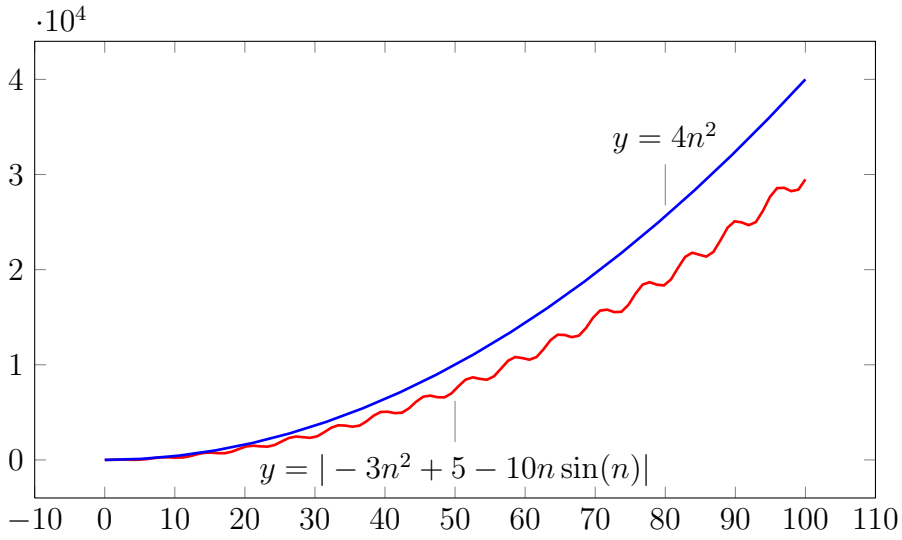
Now, let me give you another example. Suppose I change my $f(n)$ to *this*:

$$f(n) = -3n^2 + 5 - 10n\sin(n)$$

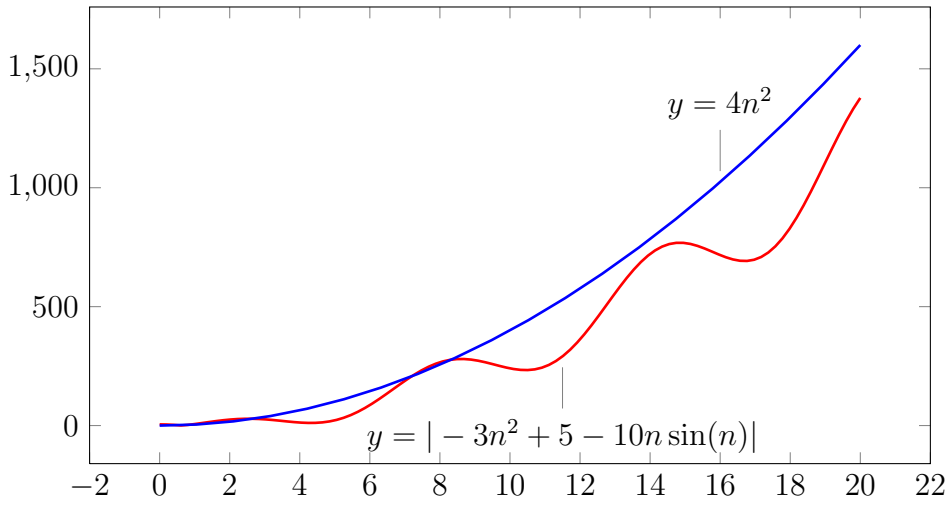
then when I plot $|-3n^2 + 5 - 10n\sin(n)|$ and $4|n^2|$ for $0 \leq n \leq 5$, I get



and then for $0 \leq n \leq 100$:



In this case n^2 still works, i.e., $-3n^2 + 5 - 10n\sin(n) = O(n^2)$. Also, it seems that $4n^2$ breaks away from $|-3n^2 + 5 - 10n\sin(n)|$ around $n = 20$. So let's plot the graphs for $0 \leq n \leq 20$:



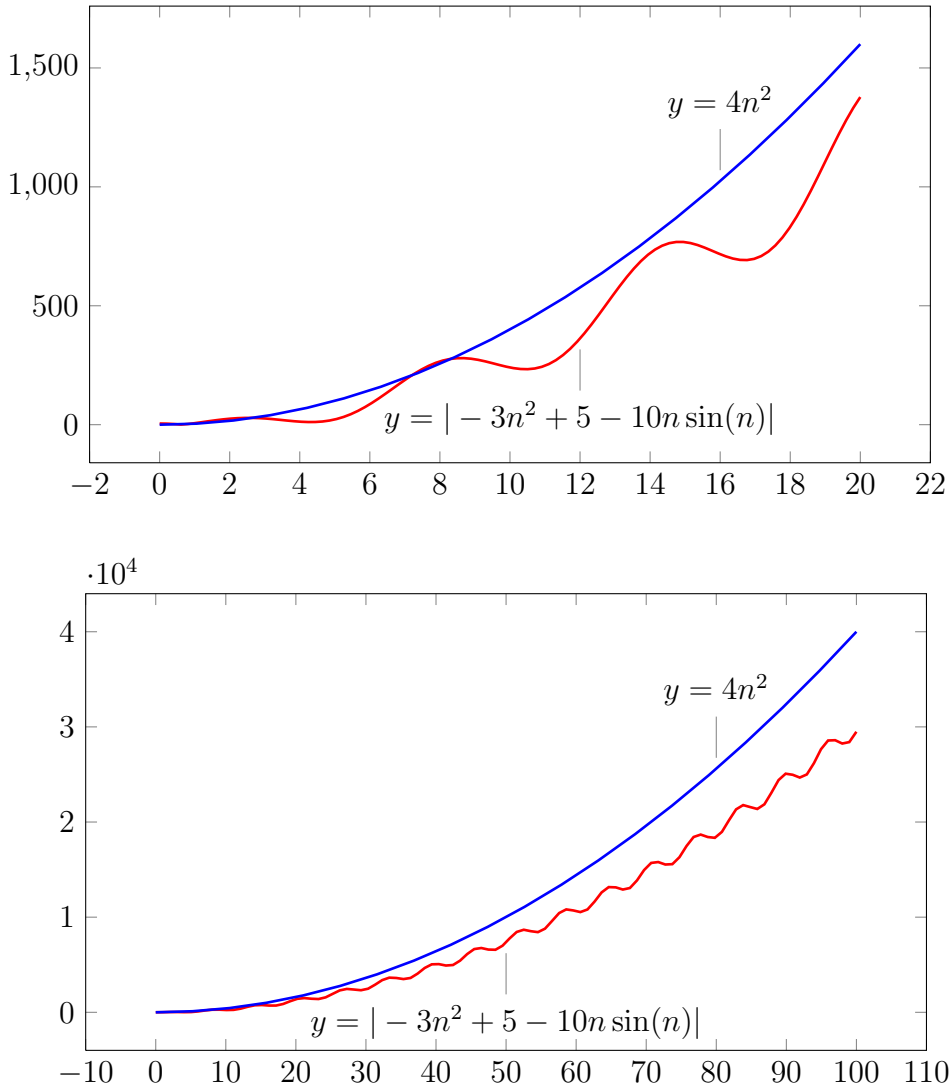
Clearly for $n \geq 10$, $4|g(n)|$ beats $|f(n)|$. So I'm going to pick $N = 10$.

Now I'm ready to present this ...

Example 102.8.2. Let $f(n) = -3n^2 + 5 - 10n\sin(n)$ and $g(n) = n^2$. Show graphically that

$$f(n) = O(g(n))$$

Solution. From the following graphs:



we see that if we choose $C = 4$ and $N = 20$, then for $n \geq N$, we have

$$|-3n^2 + 5 - 10n\sin(n)| \leq 4|n^2|$$

i.e.,

$$|f(n)| \leq C|g(n)|$$

Hence $f(n) = O(g(n))$. □

It's not surprising that if

$$3n^2 + 5 + 10n\sin(n) = O(n^2)$$

then it is also true that

$$3n^2 + 5 + 10n\sin(n) = O(n^3)$$

since n^3 grows faster than n^2. So there are many possible functions to "control" $3n^2 + 5 + 10n\sin(n)$ from above. Also, if

$$|3n^2 + 5 + 10n\sin(n)| \leq C|n^2|$$

then of course if you replace C by a large value, say $C + 1423$, then of course

$$|3n^2 + 5 + 10n\sin(n)| \leq (C + 1423)|n^2|$$

is still true for $n \geq N$. Furthermore, this is also true if you replace N by a larger value such as $N + 15313$. Therefore the choice of C and N is not unique.

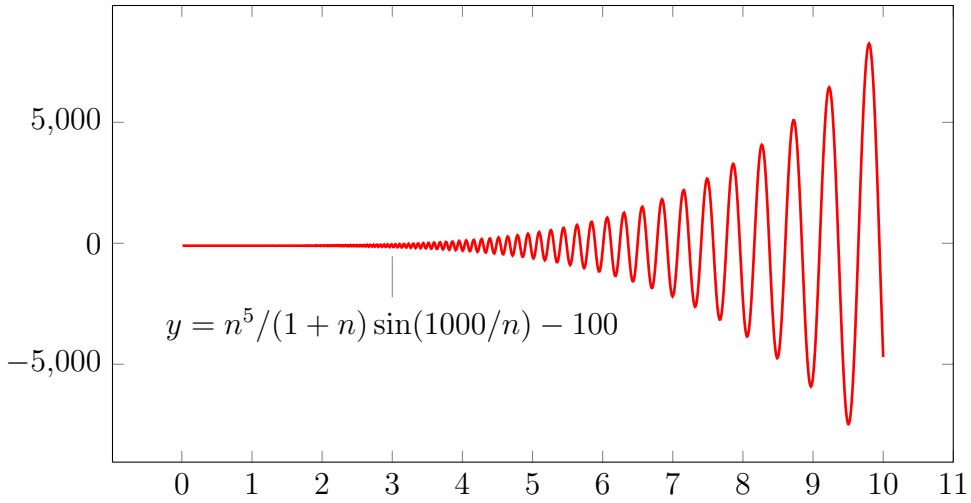
Here's another example.

Let

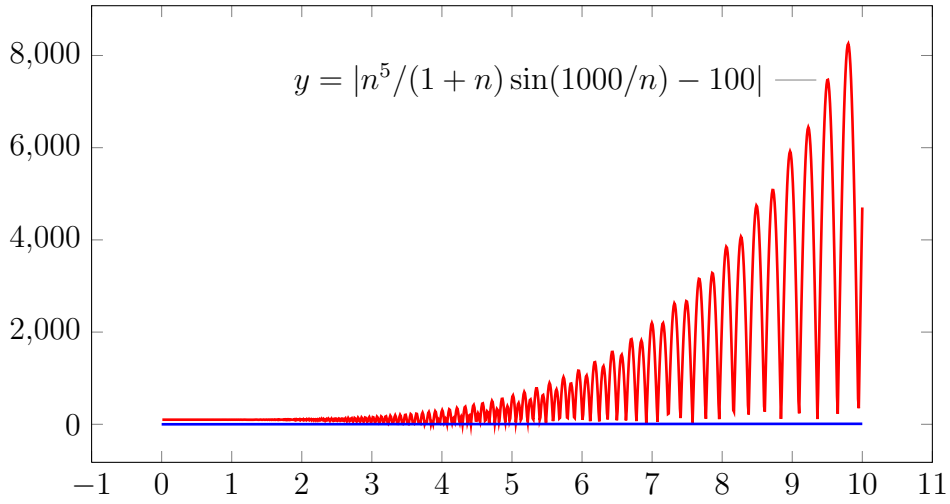
$$f(n) = \frac{2n^5}{1+n}\sin\frac{1000}{n} - 100$$

Let's find some function $g(n)$ of the form n^0, n^1, n^2, n^3 or ... such that $f(n) = O(g(n))$. Let's have a few plots to get a feel for the function.

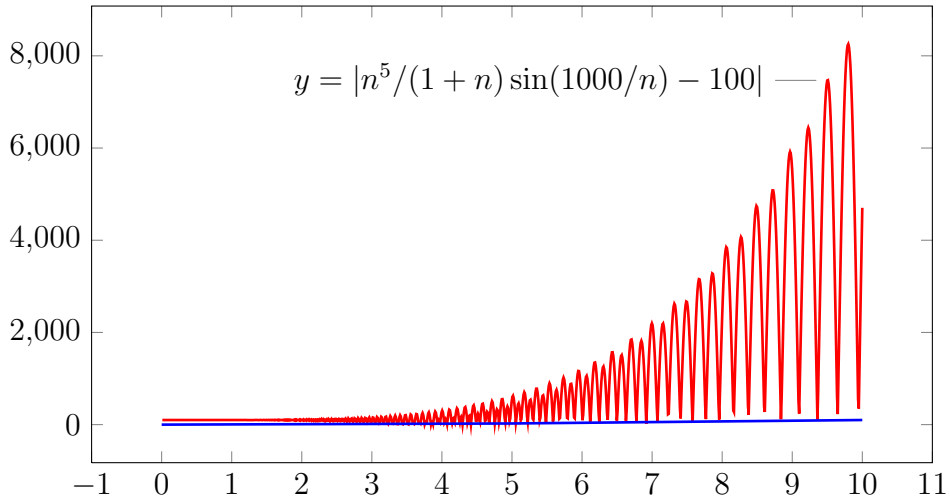
Here's the plot of $f(n)$ for $0 \leq n \leq 10$:



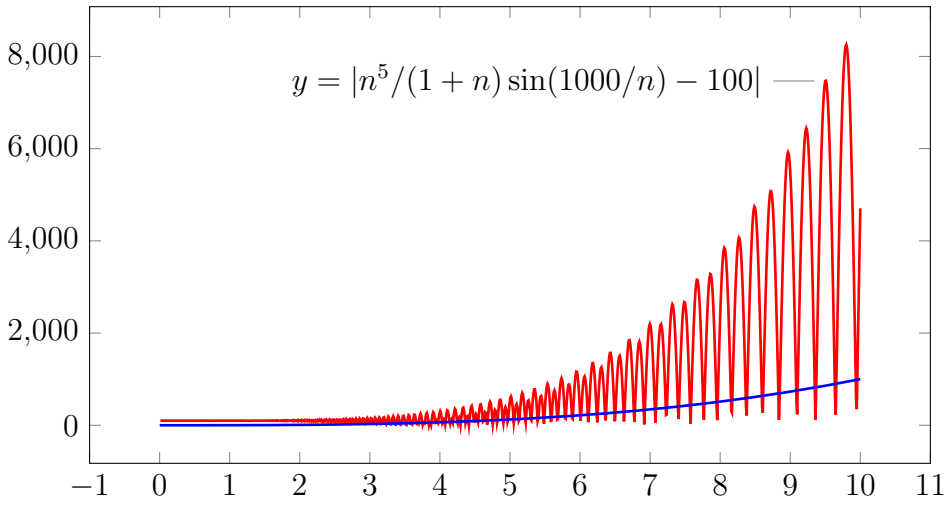
Aha ... $f(n)$ can be negative. I'd better plot $|f(n)|$ instead of $f(n)$. Also, clearly $g(n)$ can't be $n^0 = 1$ (i.e., no multiple of $g(n) = 1$ is going to beat $|f(n)|$ ... right?) so I'm going to skip that. I'll try higher powers. With $g(n) = n$:



With $g(n) = n^2$:

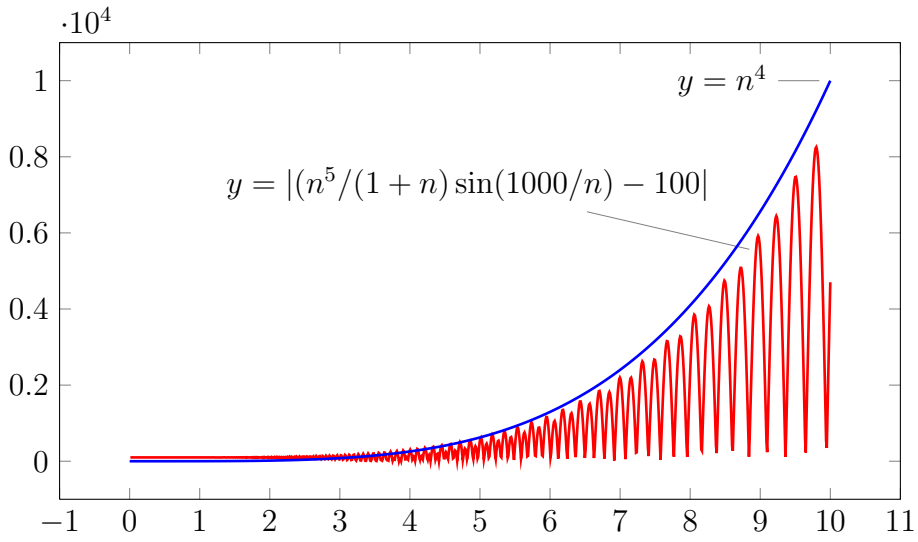


Wow. $f(n)$ is exploding so fast that I can't even see n^2 bending up at all! With $g(n) = n^3$:



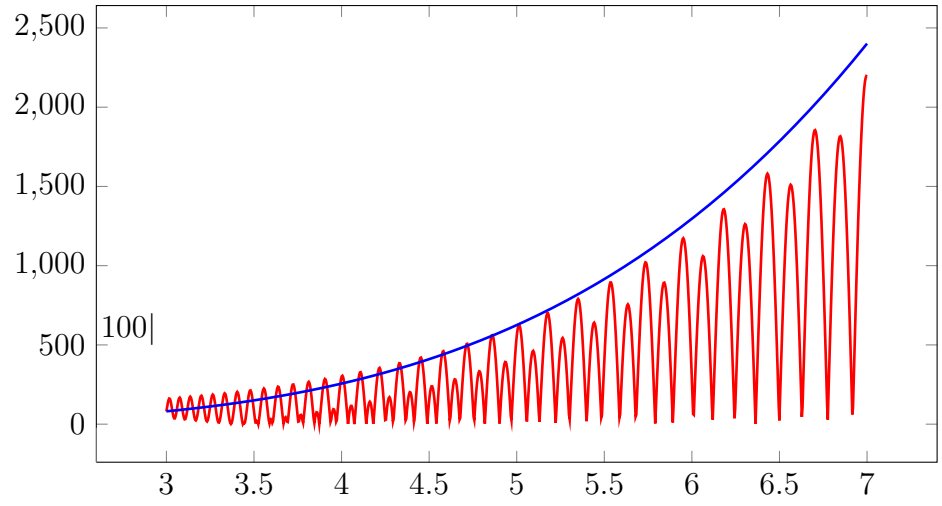
OK ... at least I can see n^3 bending up a little. But it's still not high enough to bound $f(n)$.

With $g(n) = n^4$:



Aha! n^4 beats $f(n)$!!!

You notice that $g(n)$ beats $|f(n)|$ around $5 \leq n \leq 6.5$. Let me zoom in. Here's the plot for $3 \leq n \leq 7$:



From the graphs we see that for $n \geq 6$

$$\left| \frac{n^5}{1+n} \sin \frac{1000}{n} - 100 \right| \leq |n^4|$$

So if I choose $C = 1$ and $N = 6$, then for $n \geq N$,

$$\left| \frac{n^5}{1+n} \sin \frac{1000}{n} - 100 \right| \leq C\,|n^4|$$

i.e.,

$$|f(n)| \leq C|n^4|$$

Hence

$$f(n) = O(n^4)$$

So here's the solution ...

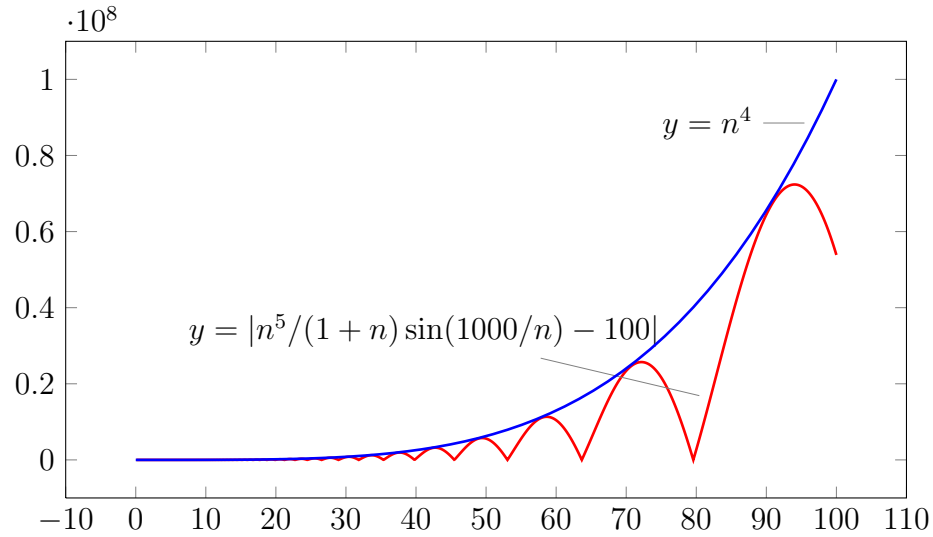
Example 102.8.3. Let

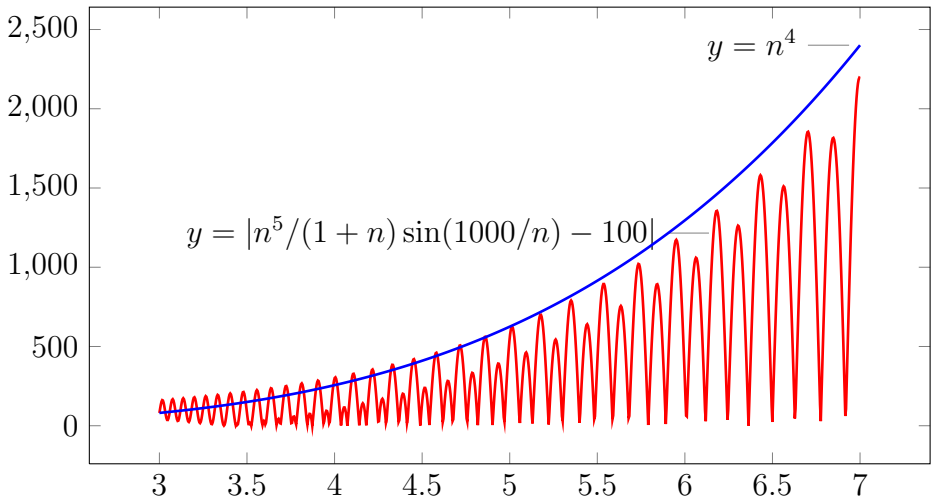
$$f(n) = \frac{n^5}{1+n} \sin \frac{1000}{n} - 100$$

Find the smallest integer k such that for $g(n) = n^k$, we have

$$f(n) = O(g(n))$$

Solution. The following are plots of $|f(n)|$ and $g(n) = n^4$:





If we choose $N = 6$ and $C = 1$, we see that for $n \geq N$,

$$\left| \frac{n^5}{1+n} \sin \frac{1000}{n} - 100 \right| \leq C \left| n^4 \right|$$

i.e.,

$$|f(n)| \leq C|g(n)|$$

Hence

$$f(n) = O(n^4)$$

□

Exercise 102.8.1. Let

debug: exercises/7n4-20/question.tex

$$f(n) = 7n^4 + 20$$

Using plots, find the smallest integer k such that $f(n) = O(n^k)$. Supply a C and an N such that

$$|f(n)| \leq C|g(n)|$$

for $n \geq N$.

(Go to solution, page 4097) □

Exercise 102.8.2. Let

debug:
exercises/0-31415n4-1000n3/question.tex

$$f(n) = 0.31415n^4 + 1000n^3$$

Using plots, find the smallest integer k such that $f(n) = O(n^k)$. Supply a C and an N such that

$$|f(n)| \leq C|g(n)|$$

for $n \geq N$.

(Go to solution, page 4098) □

Exercise 102.8.3. Let

debug: exercises/n4-1234n3/question.tex

$$f(n) = n^4 - 1234n^3$$

Using plots, find the smallest integer k such that $f(n) = O(n^k)$. Supply a C and an N such that

$$|f(n)| \leq C|g(n)|$$

for $n \geq N$.

(Go to solution, page 4099) □

Exercise 102.8.4. Let

debug: exercises/3n3-5-42-n5-n2-1/main.tex

$$f(n) = -3n^{3.5} + 42\frac{n^5}{n^2+1}$$

Using plots, find the smallest *real number* k (not necessarily an integer) such that $f(n) = O(n^k)$. Supply a C and an N such that

$$|f(n)| \leq C|g(n)|$$

for $n \geq N$.

(Go to solution, page 4100) □

debug: exercises/20-n5-n3-1-5n3-cos-n-7-8/question.tex

Exercise 102.8.5. It's a fact that if

$$f_0(n) = O(g(n))$$
$$f_1(n) = O(g(n))$$
$$f_2(n) = O(g(n))$$

then

$$f_0(n) + f_1(n) + f_2(n) = O(g(n))$$

Let

$$f(n) = 20\frac{n^5}{n^3+1} + 5n^3 \cos n + 7^8$$

Find the smallest integer k such that $f(n) = O(n^k)$ using plots. Supply a C and an N such that

$$|f(n)| \leq C|g(n)|$$

for $n \geq N$. Do it in the following steps:

(a) Graphically, find k_0, C_0, N_0 such that $20\frac{n^5}{n^3+1} \leq C_0 n^{k_0}$ for $n \geq N_0$. Choose k_0 to be minimal.
(b) Graphically, find k_1, C_1, N_1 such that $|5n^3 \cos n| \leq C_1 n^k$ for $n \geq N_1$. Choose k_1 to be minimal.
(c) Graphically, find k_2, C_2, N_2 such that $7^8 \leq C_2 n^k$ for some C_2 for $n \geq N_2$. Choose k_2 to be minimal.
(d) Using (a)-(c), let k be the maximum of k_0, k_1, k_2, C be the maximum of C_0, C_1, C_2, and N be the maximum of N_0, N_1, N_2. Graphically show that $|f(n)| \leq C|n^k|$ for $n \geq N$.

(Go to solution, page 4101) □

Exercise 102.8.6. Let

debug:
exercises/123456789–
1n-n3/question.tex

$$f(n) = 123456789 + (-1)^n n^3$$

Using plots, find the smallest *real number* k (not necessarily an integer) such that $f(n) = O(n^k)$. Supply a C and an N such that

$$|f(n)| \leq C|g(n)|$$

for $n \geq N$.

(Go to solution, page 4105) □

Exercise 102.8.7. Let

debug: exercises/7n2-7-lg-n-20-n5-n-1/question.tex

$$f(n) = 7n^{2.7} \lg n + \frac{n^5}{n^2+1}$$

Using plots, find the smallest integer k such that $f(n) = O(n^k)$. Supply a C and an N such that

$$|f(n)| \leq C|g(n)|$$

for $n \geq N$. (Note: $\lg = \log_2$. For the study of algorithms log-base-2 is more common than base 10 and base e.)

(Go to solution, page 4106) □

Exercise 102.8.8. What is the big-O of

debug: exercises/big-O-0/question.tex

$$\frac{1}{1000}n + 1000 \ln n$$

[Hint: Plot the two terms and see which one grows faster. Make sure you use a large domain for n.] (Go to solution, page 4107) □

Exercise 102.8.9. What is the big-O of

debug: exercises/big-O-1/question.tex

$$1000n \ln n + n^{1.001}$$

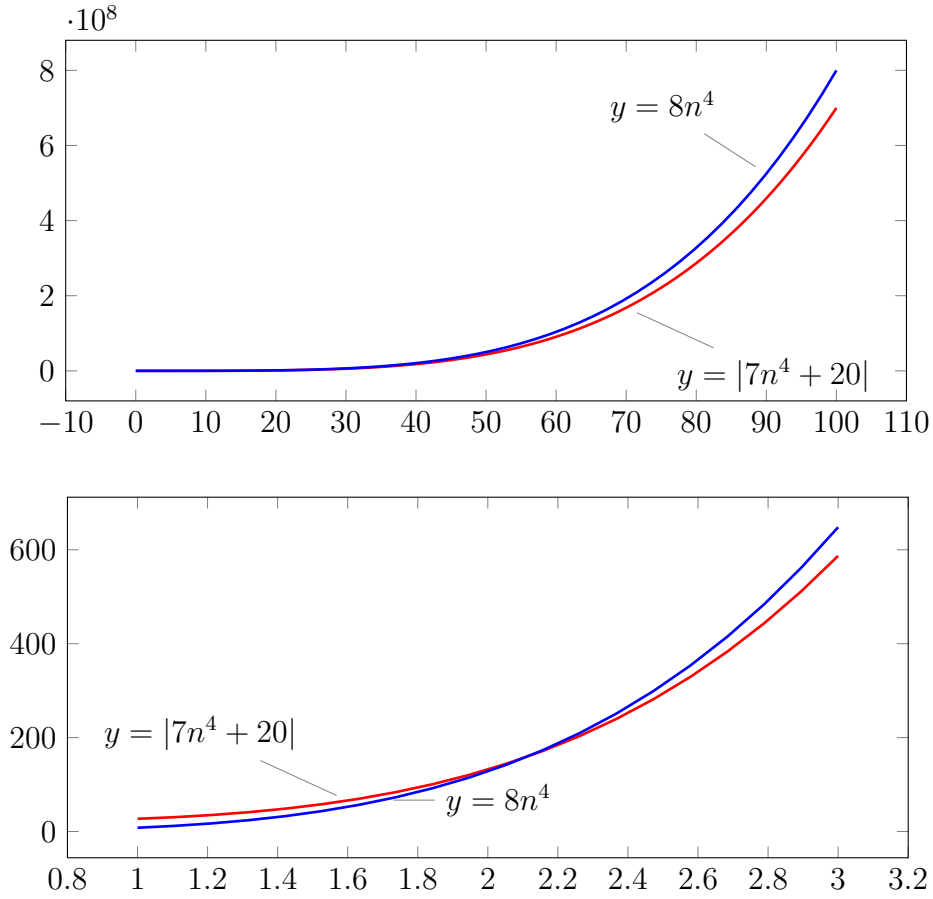
[Hint: Plot the two terms and see which one grows faster. Make sure you use a large domain for n.] (Go to solution, page 4108) □

Solutions

Solution to Exercise 102.8.1.

debug: exercises/7n4-20/answer.tex

The following are plots of $|f(n)|$ and $g(n) = n^4$:



If we choose $C = 8$ and $N = 3$, we see that for $n \geq N$,

$$\left|7n^4 + 20\right| \leq C \left|n^4\right|$$

i.e.,

$$|f(n)| \leq C|g(n)|$$

Hence

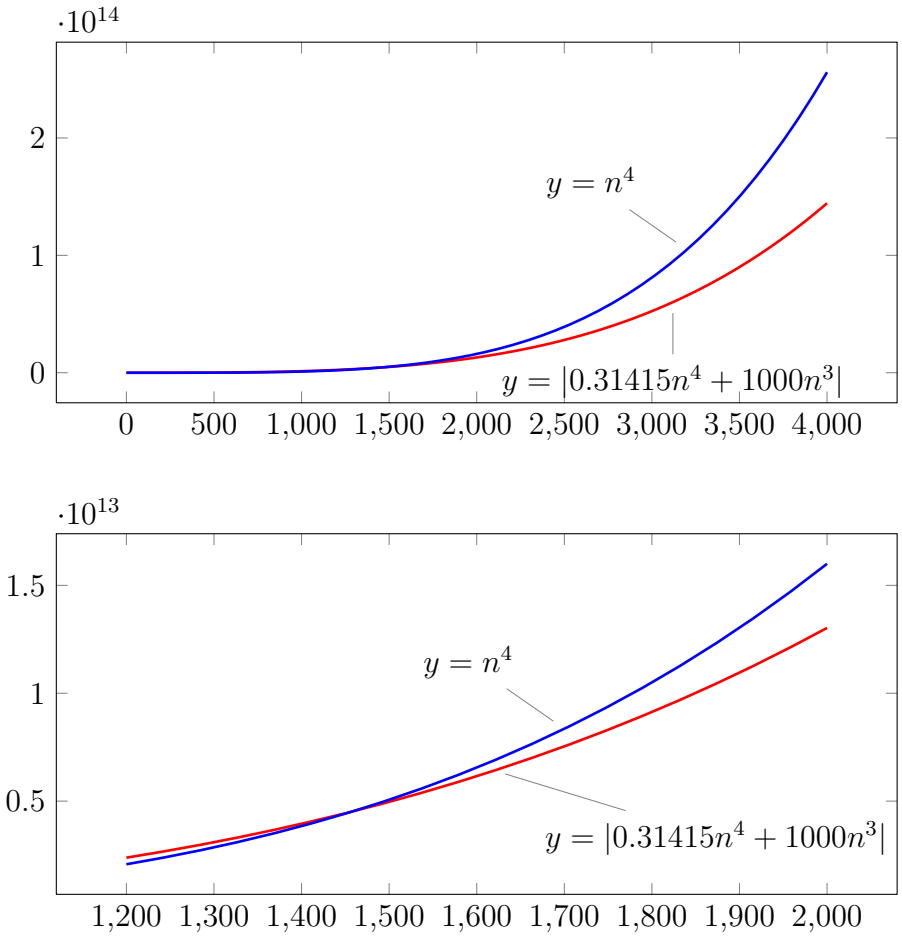
$$f(n) = O(n^4)$$

□

Solution to Exercise 102.8.2.

debug:
exercises/0-31415n4-1000n3/answer.tex

The following are plots of $|f(n)|$ and $g(n) = n^4$:



If we choose $C = 1$ and $N = 1600$, we see that for $n \geq N$,

$$\left|0.31415n^4 + 1000n^3\right| \leq C \left|n^4\right|$$

i.e.,

$$|f(n)| \leq C|g(n)|$$

Hence

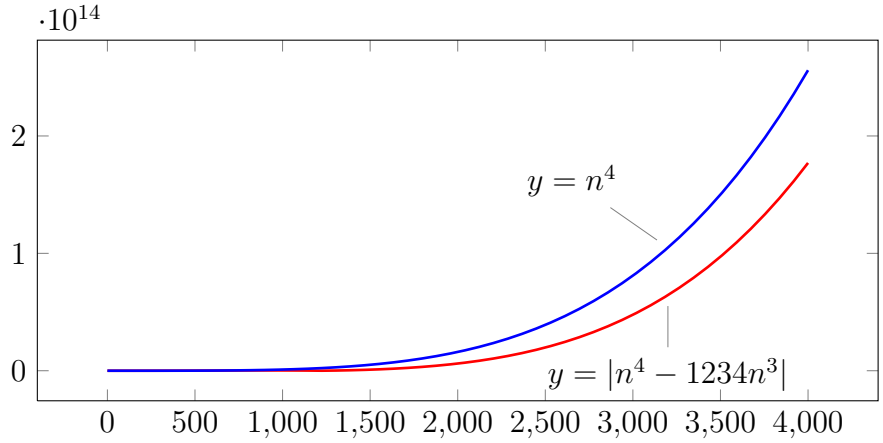
$$f(n) = O(n^4)$$

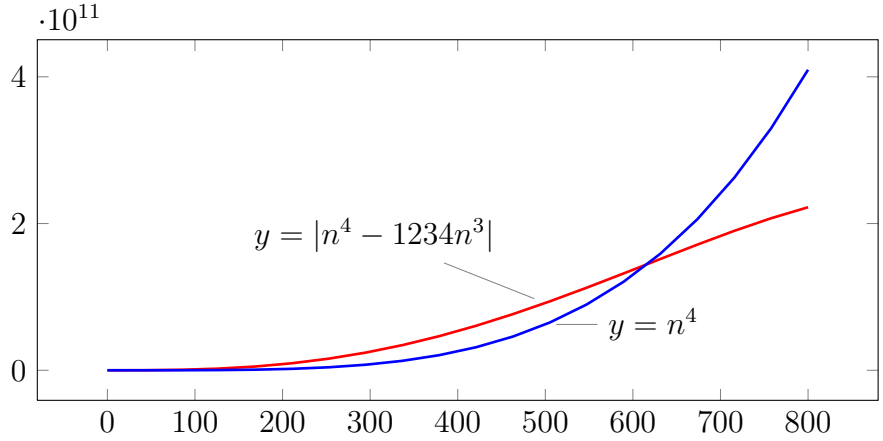
□

Solution to Exercise 102.8.3.

debug: exercises/n4-1234n3/answer.tex

The following are plots of $|f(n)|$ and $g(n) = n^4$:





If we choose $C = 1$ and $N = 700$, we see that for $n \geq N$,

$$\left|n^4 - 1234n^3\right| \leq C \left|n^4\right|$$

i.e.,

$$|f(n)| \leq C|g(n)|$$

Hence

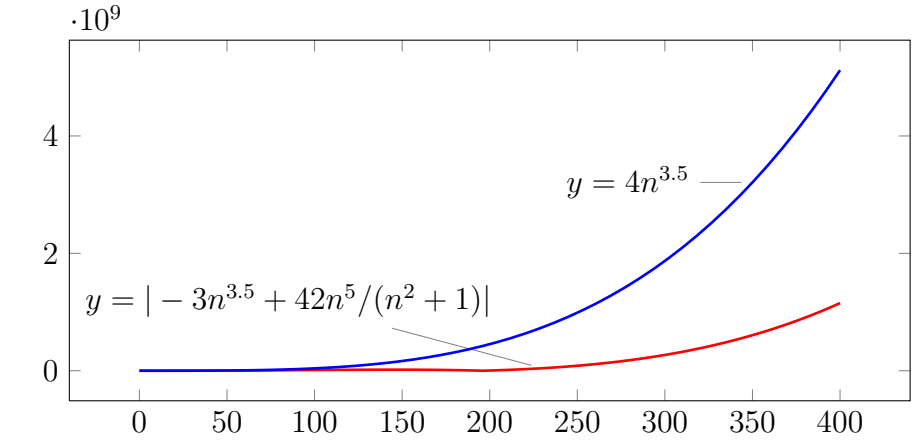
$$f(n) = O(n^4)$$

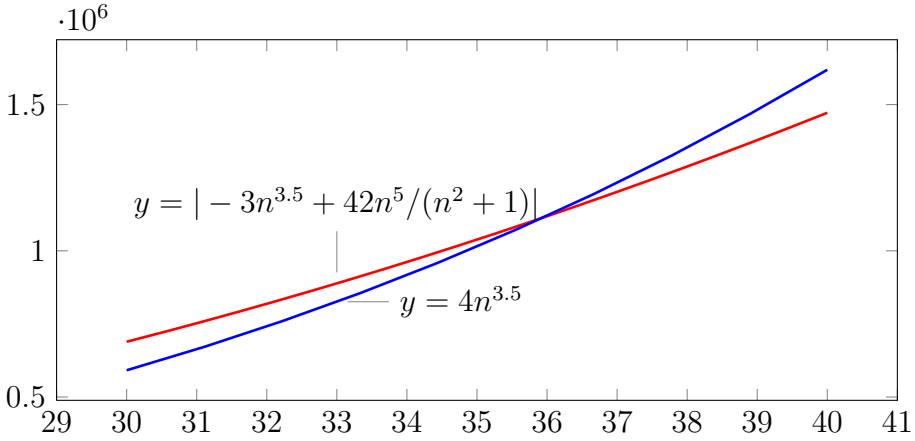
□

Solution to Exercise 102.8.4.

debug: exercises/3n3-5-42-n5-n2-1/main.tex

The following are plots of $|f(n)|$ and $g(n) = n^4$:





If we choose $C = 4$ and $N = 37$, we see that for $n \geq N$,

$$\left| -3n^{3.5} + 42\frac{n^5}{n^2+1} \right| \leq C \left| n^4 \right|$$

i.e.,

$$|f(n)| \leq C|g(n)|$$

Hence

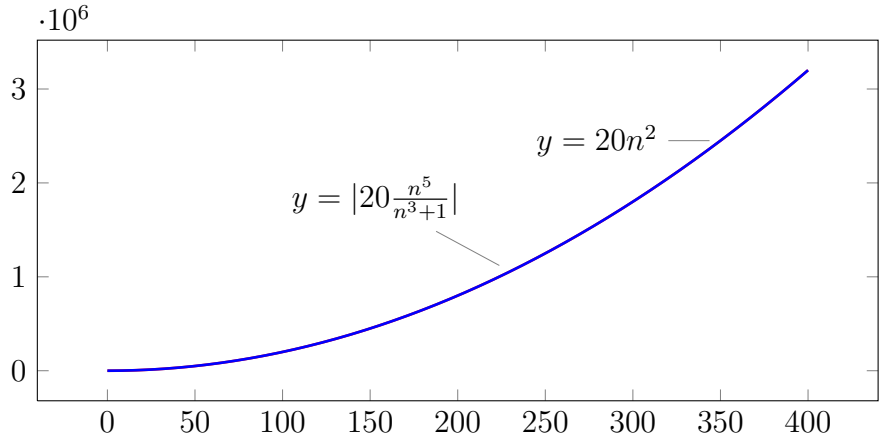
$$f(n) = O(n^4)$$

□

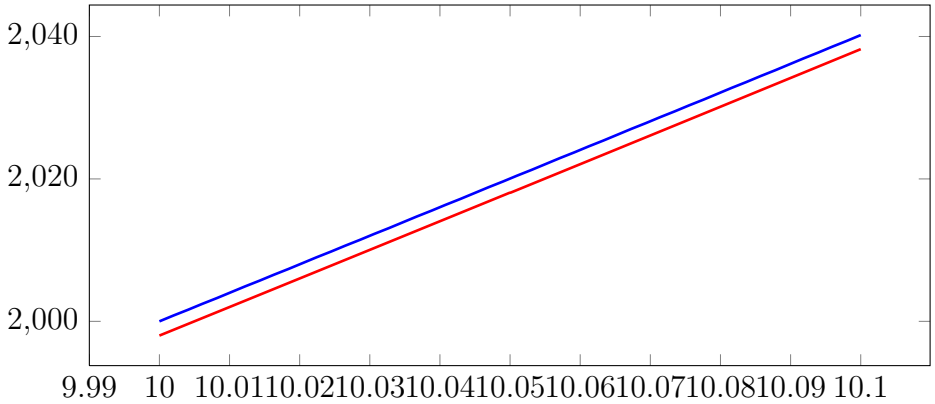
Solution to Exercise 102.8.5.

debug: exercises/20-n5-n3-1-5n3-cos-n-7-8/answer.tex

(a) Let $f_0(n) = 20n^5/(n^3+1)$. The following is a plot of $|f_0(n)|$ and $20n^2$:



We can't really tell that $|f_0(n)| \leq 20n^2$. But if we zoom in, we see that in the interval $[10, 10.1]$:



he blue graph of $20n^2$ is above the red graph of $f(n)$ on the interval $[10, 10.1]$. By zooming in at different places on the positive part of the real axis, we see that the blue graph is always above the red graph. In fact

$$|f_0(n)| \leq 20n^2$$

for all $n > 0$. This can be proven mathematically: If $n > 0$, then

$$n^3 < n^3 + 1$$

Therefore

$$\frac{1}{n^3+1} < \frac{1}{n^3}$$

and hence

$$20\frac{n^5}{n^3+1} < 20\frac{n^5}{n^3} = 20n^2$$

Therefore if we choose $C_0 = 20$ and $N_0 = 1$, we see that for $n \geq N_0$,

$$\left|20\frac{n^5}{n^3+1}\right| \leq C_0 \left|n^2\right|$$

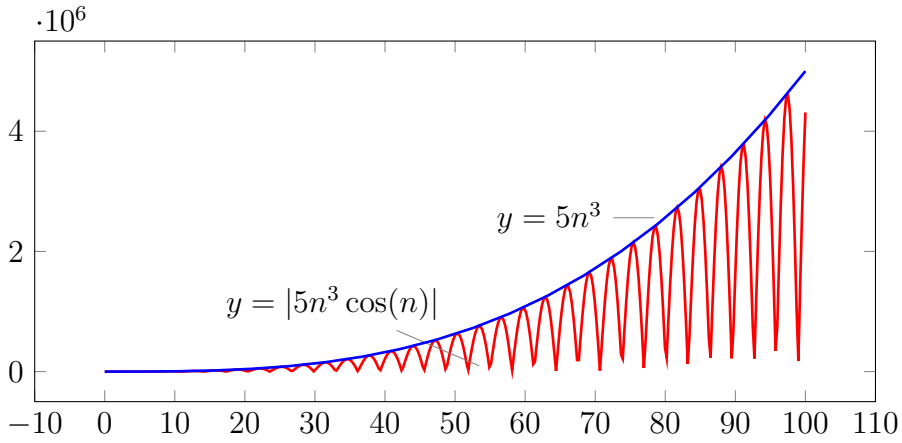
i.e.,

$$|f_0(n)| \leq C_0|g(n)|$$

Hence

$$f_0(n) = O(n^2)$$

(b) Let $f_1(n) = 5n^3\cos(n)$. The following are plots of $|f_1(n)|$ (in red) and $5n^3$:



Since

$$-1 \leq \cos(n) \leq 1$$

for $n \geq 0$, we have

$$-5n^3 \leq 5n^3\cos(n) \leq 5n^3$$

Therefore

$$|5n^3\cos(n)| \leq 5|n^3|$$

Hence if I can choose $C_1 = 5$, $N_1 = 0$, then for $n \geq N_1$,

$$|5n^3\cos(n)| \leq 5|n^3|$$

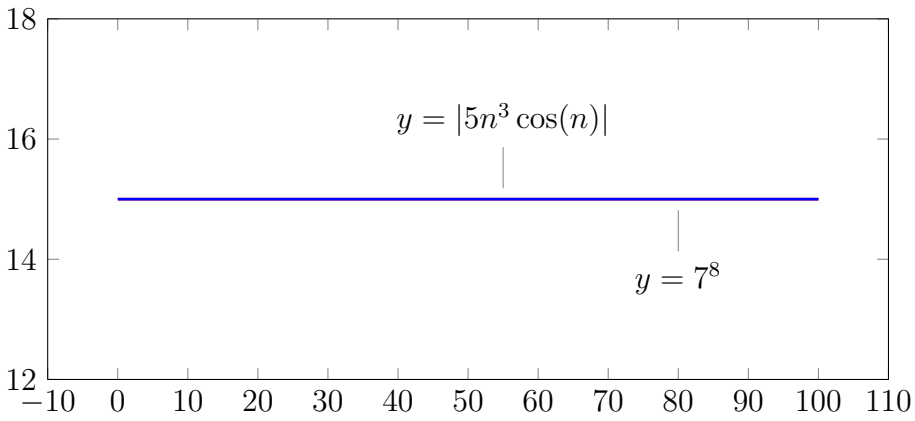
i.e.,

$$|f_1(n)| \leq 5|n^3|$$

Therefore

$$f_1(n) = O(5n^3)$$

(c) Let $f_2(n) = 7^8$. The following are plots of $|f_2(n)|$ (in red) and $7^8 n^0$:



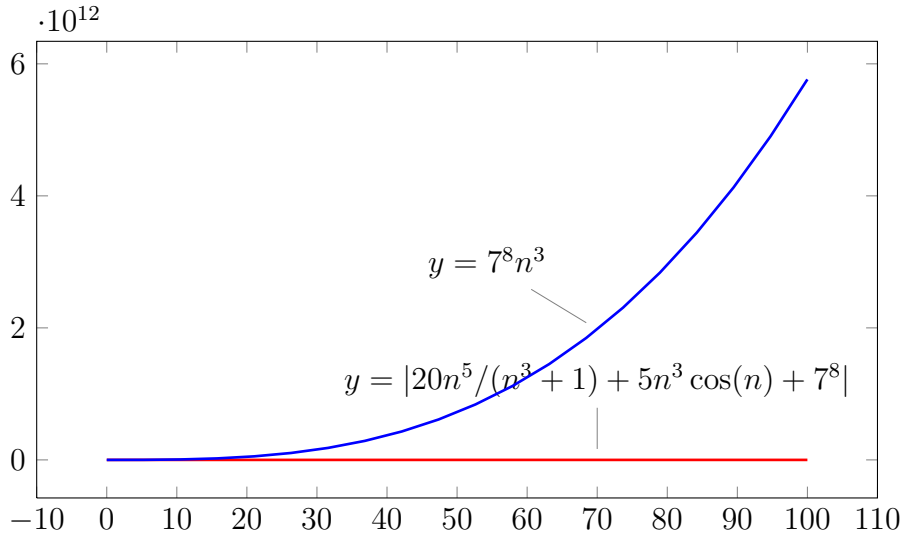
The two graphs are the same. If I set $C_2 = 7^8$ and $N_2 = 0$, then for $n \geq N_2$,

$$|f_2(n)| \leq C_2|n^0|$$

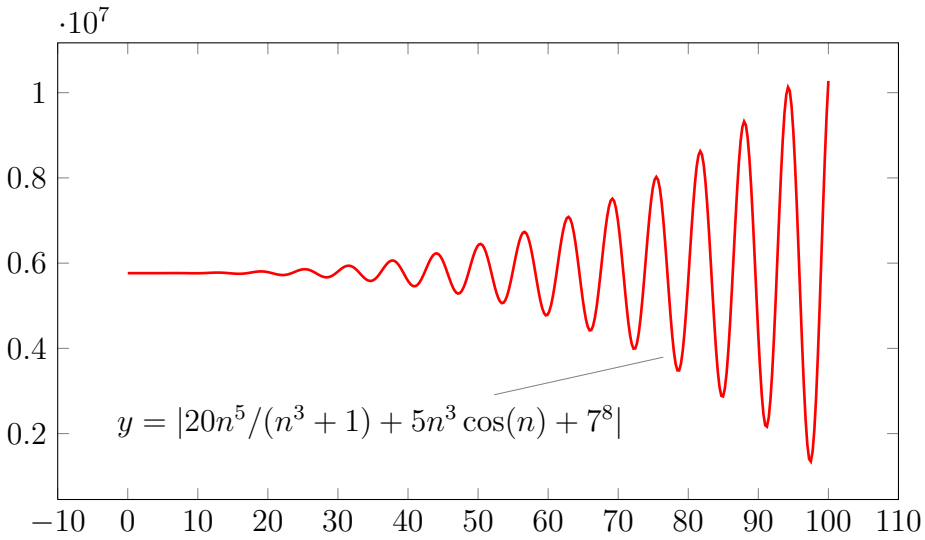
Therefore

$$f_2(n) = O(n^0)$$

(d) Let $C = \max(C_0, C_1, C_2) = \max(20, 5, 7^8) = 7^8$, $N = \max(N_0, N_1, N_2) = \max(1, 0, 0) = 1$, $k = \max(2, 3, 0) = 3$.



No. $f(n)$ is not flat. It appears to be flat only because $7^8 n^3$ is climbing too fast. Here's the plot of $f(n)$ without $7^8 n^3$



Note that the upper bound $7^8 n^3$ is not tight. A tigher upper bound is $5n^3$. Of course either way

$$f(n) = O(n^3)$$

□

Solution to Exercise 102.8.6.

Solution not provided. □

debug:
exercises/123456789–
1n-n3/answer.tex

Solution to Exercise 102.8.7.

Solution not provided. □

debug:
exercises/7n2-7-lg-n-20-n5-n-1/answer.tex

Solution to Exercise 102.8.8.

debug: exercises/big-O-0/answer.tex

$\frac{1}{1000}n + 1000 \ln n = O(n)$

Solution to Exercise 102.8.9.

debug: exercises/big-O-1/answer.tex

Solution not provided. See answer below.

$1000n \ln n + n^{1.001} = O(n^{1.001})$

102.9 Summation debug: summation.tex

I'm going to compute the runtime of the bubblesort very soon. Before I do that I'm going to give you the formula for the arithmetic sum which will be helpful in runtime computations:

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

In fact sums are common in this game. So I'm going to use the summation notation to simplify the computation.

Let a_i be a formula in i. The notation

$$\sum_{i=3}^{7} a_i$$

is a shorthand notation for

$$\sum_{i=3}^{7} a_i = a_3 + a_4 + a_5 + a_6 + a_7$$

For instance

$$\sum_{i=3}^{7} i^2 = 3^2 + 4^2 + 5^2 + 6^2 + 7^2$$

Exercise 102.9.1. The following is a practice on the summation notation. Compute the value of the following: debug: exercises/summation-0/question.tex

(a) $\sum_{i=2}^{5} i^2$
(b) $\sum_{j=0}^{4} (2j)$
(c) $\sum_{k=0}^{4} (2k+1)$
(d) $\sum_{\ell=2}^{5} 3\ell^2$
(e) $\sum_{m=2}^{10} 1$

(Go to solution, page 4113) □

Exercise 102.9.2. debug: exercises/summation-1/question.tex

(a) Compute $\sum_{i=2}^{5}(5i^2)$ and $5\sum_{i=2}^{5} i^2$ The two are the same. In general

$$\sum_{i=1}^{n} ca_i = c\sum_{i=1}^{n} a_i$$

(b) Compute $\sum_{i=2}^{5}(i^2+\sqrt{i})$ and $\sum_{i=2}^{5} i^2 + \sum_{i=2}^{5}\sqrt{i}$ The two are the same. In general

$$\sum_{i=1}^{n}(a_i+b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$$

(c) In general

$$\sum_{i=1}^{n}(ca_i+db_i) = c\sum_{i=1}^{n} a_i + d\sum_{i=1}^{n} b_i$$

(Go to solution, page 4114) □

Let me summarize the basic summmation formulas here so that you can refer to them:

Proposition 102.9.1.

(a)
$$\sum_{i=1}^{n} ca_i = c\sum_{i=1}^{n} a_i$$

(b)
$$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$$

(c)
$$\sum_{i=1}^{n} (ca_i + db_i) = c\sum_{i=1}^{n} a_i + d\sum_{i=1}^{n} b_i$$

Of course the above formulas hold when the lower limit of the summation are all changed to another value.

(a)
$$\sum_{i=1}^{n} ca_i = c\sum_{i=1}^{n} a_i$$

(b)
$$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$$

(c)
$$\sum_{i=1}^{n} (ca_i + db_i) = c\sum_{i=1}^{n} a_i + d\sum_{i=1}^{n} b_i$$

The following "splitting out terms from the bottom" is obviously true:

$$\sum_{i=0}^{10} a_i = a_0 + a_1 + a_2 + \sum_{i=2}^{10} a_i$$

So is "splitting out terms from the top":

$$\sum_{i=0}^{10} a_i = \sum_{i=0}^{8} a_i + a_9 + a_{10}$$

Likewise you can split a summation into two like this:

$$\sum_{i=0}^{10} a_i = \sum_{i=0}^{2} a_i + \sum_{i=3}^{10} a_i$$

Exercise 102.9.3. You are given

$$\sum_{i=1}^{n} a_i = 42 \quad \text{and} \quad \sum_{i=1}^{n} b_i = 60$$

Compute

$$\sum_{i=1}^{n} (2a_i + 3b_i)$$

□

Exercise 102.9.4. You are given $\sum_{i=1}^{11} a_i = 120$, $\sum_{i=11}^{20} a_i = 42$, and $\sum_{i=1}^{20} a_i = 691$. What can you tell me about a_{11}? □

Solutions

Solution to Exercise 102.9.1.

debug:
exercises/summation-0/answer.tex

(a) $\sum_{i=2}^{5} i^2 = 2^2 + 3^2 + 4^2 + 5^2 = 54$

(b) $\sum_{j=0}^{4} (2j) = 2 \cdot 0 + 2 \cdot 1 + 2 \cdot 2 + 2 \cdot 3 + 2 \cdot 4 = 20$

(c) $\sum_{k=0}^{4} (2k+1) = (2 \cdot 0 + 1) + (2 \cdot 1 + 1) + (2 \cdot 2 + 1) + (2 \cdot 3 + 1) + (2 \cdot 4 + 1) = 25$

(d) $\sum_{\ell=2}^{5} 3\ell^2 = 3 \cdot 2^2 + 3 \cdot 3^2 + 3 \cdot 4^2 + 3 \cdot 5^2 = 162$

(e) $\sum_{m=2}^{10} 1 = 9$

Solution to Exercise 102.9.2.

debug:
exercises/summation-1/answer.tex

Solution not provided.

102.10 Sums of Powers debug: formulas-for-sum-of-powers.tex

I'm going to give you two formulas which are extremely useful in runtime computations.

The first formula you have probably seen in quite a few math classes. Here's the arithmetic sum formula:

Proposition 102.10.1.

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

□

I won't prove the above formula.

Exercise 102.10.1. Check by hand that the arithmetic sum formula is correct for $n = 1, 2, 3, 4$. □

Using the summation notation you can write the above as

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Writing down the formula when the lower or upper limit of the sum is slightly altered is pretty easy. For instance suppose I want to compute $2 + 3 + \cdots + n$ as a polynomial expression in decreasing powers of n. Then from

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

I just subtract 1 to get

$$2 + \cdots + n = \frac{n(n+1)}{2} - 1$$

I then do some algebra to get this:

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

Here's the solution to this problem:

Example 102.10.1. Express $\sum_{i=2}^{n} i$ as a polynomial in descending powers of n.

Solution.

$$\begin{aligned}\sum_{i=2}^{n} i &= \sum_{i=1}^{n} i - 1\\ &= \frac{n(n+1)}{2} - 1\\ &= \frac{1}{2}n^2 + \frac{1}{2}n + \frac{1}{2} - 1\\ &= \frac{1}{2}n^2 + \frac{1}{2}n - \frac{1}{2}\end{aligned}$$

□

Now suppose I want

$$\sum_{i=1}^{n-1} i$$

Note that the upper limit of this summation is $n-1$ and not n. In this case, I just replace the n in the arithmetic sum formula by $n-1$ to get:

$$\sum_{i=1}^{n-1} i = \frac{(n-1)((n-1)+1)}{2} = \frac{(n-1)n}{2}$$

Example 102.10.2. Write

$$\sum_{i=1}^{n-1} i$$

as a polynomial in descending powers of n.

Solution.

$$\begin{aligned}\sum_{i=1}^{n-1} i &= \frac{(n-1)((n-1)+1)}{2}\\ &= \frac{(n-1)n}{2}\\ &= \frac{1}{2}n^2 - \frac{1}{2}n\end{aligned}$$

□

Exercise 102.10.2. Compute the sum $1+2+3+\cdots+1000$ by first rewriting it as a summation and then using the arithmetic sum formula. □

Exercise 102.10.3. Compute

$$\sum_{i=1}^{100} 2i$$

using the arithmetic sum formula. (Write down the first 3 terms of the summation and the last 3 just to make sure you know what you're adding.)

Exercise 102.10.4. Compute

$$\sum_{i=1}^{100}(2i+1)$$

using the arithmetic sum formula. (Write down the first 3 terms of the summation and the last 3 just to make sure you know what you're adding.)

Exercise 102.10.5. Compute the sum

$$1 + 4 + 7 + 10 + \cdots + 100$$

First write it as a summation. Next attempt to rewrite it so that you can see $\sum_{i=1}^{n} i$ (for some n) so that you can use the arithmetic sum formula.

Exercise 102.10.6. Write the following as a polynomial in decreasing power of n:

$$\sum_{i=2}^{n} i$$

□

Exercise 102.10.7. Write the following as a polynomial in decreasing power of n:

$$\sum_{i=2}^{n-1} i$$

□

Exercise 102.10.8. Write the following as a polynomial in decreasing power of n:

$$\sum_{i=0}^{n-2} i$$

□

Besides the arithmetic sum formula, there is also a sum of squares formula:

Proposition 102.10.2.

$$1^2 + 2^2 + \cdots + n^2 = \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

I won't prove the above formula.

Exercise 102.10.9. Check by hand that the sum of squares formula is correct for $n = 1, 2, 3, 4$.

Exercise 102.10.10. Compute the sum $1^2 + 2^2 + 3^2 + \cdots + 1000^2$. □

Exercise 102.10.11. Write the following as a polynomial in decreasing power of n:

$$\sum_{i=1}^{n+1} i^2$$

□

Exercise 102.10.12. Write the following as a polynomial in decreasing power of n:

$$\sum_{i=0}^{n-1} i^2$$

□

Actually there are also formulas for

$$\sum_{i=1}^{n} i^3, \quad \sum_{i=1}^{n} i^4, \quad \sum_{i=1}^{n} i^5, \ldots$$

Google for their formulas and their stories.

102.11 Bubblesort: double for-loops debug: bubblesort.tex

Here's bubblesort. Suppose `a[i]` ($i = 0,\ldots,n-1$) is an array of numbers (integers or floats or doubles, we don't care). The following sorts `a[i]` ($i = 0,\ldots,n-1$) in ascending order:

```
for i = n - 2, n - 3, ...,  0:
    for j = 0, 1, 2, ..., i:
        if a[j] > a[j + 1]:
            t = a[j]
            a[j] = a[j + 1]
            a[j + 1] = t
```

Let's analyze the time complexity of the above algorithm.

```
          i = n - 2
LOOP1:    if i == -1:
              goto ENDLOOP1

          j = 0
LOOP2:    if j > i:
              goto ENDLOOP2
          if a[j] > a[j + 1]
              goto ENDIF
          t = a[j]
          a[j] = a[j + 1]
          a[j + 1] = t
ENDIF     j = j + 1
          goto LOOP2

ENDLOOP2: i = i - 1
          goto LOOP2

ENDLOOP1:
```

With time for each statement:

```
          i = n - 2              t1
LOOP1:    if i == -1:            t2
              goto ENDLOOP1      t3

          j = 0                  t4
LOOP2:    if j > i:              t5
              goto ENDLOOP2      t6
          if a[i] > a[j + 1]:    t7
              goto ENDIF         t8
          t = a[i]               t9
          a[i] = a[j]            t10
```

```
          a[j] = t                t11
ENDIF:    j = j + 1               t12
          goto LOOP2              t13

ENDLOOP2: i = i - 1               t14
          goto LOOP1              t15

ENDLOOP1:
```

Including the number of times each statement is executed, the worst case runtime is:

```
          i = n - 2               t1  1
LOOP1:    if i == -1:             t2  n
              goto ENDLOOP1       t3  1

          j = 0                   t4  1+...+1     = n-1
LOOP2:    if j > i:               t5  n+...+2     = (n-1)(n+2)/2
              goto ENDLOOP2       t6  1+...+1     = n-1
          if a[i] > a[j + 1]:     t7  (n-1)+...+1 = (n-1)n/2
              goto ENDIF          t8  0
          t = a[i]                t9  (n-1)+...+1 = (n-1)n/2
          a[i] = a[j]             t10 (n-1)+...+1 = (n-1)n/2
          a[j] = t                t11 (n-1)+...+1 = (n-1)n/2
          j = j + 1               t12 (n-1)+...+1 = (n-1)n/2
          goto LOOP2              t13 (n-1)+...+1 = (n-1)n/2

ENDLOOP2: i = i - 1               t14 n-1
          goto LOOP1              t15 n-1

ENDLOOP1:
```

For the case when $i = n - 2$, the inner loop will have j running from 0 to $i + 1 = n - 1$, with the body running for $j = 0, ..., i = n - 2$ ($n - 1$ times) and exiting when $j = i + 1 = n - 1$ (once):

```
              j = 0               t4   1
LOOP2:        if j > i:           t5   n
                  goto ENDLOOP2   t6   1
              if a[i] > a[j + 1]: t7   (n-1)
                  goto ENDIF      t8   0
              t = a[i]            t9   (n-1)
              a[i] = a[j]         t10  (n-1)
              a[j] = t            t11  (n-1)
              j = j + 1           t12  (n-1)
              goto LOOP2          t13  (n-1)

ENDLOOPS:     ...
```

All other cases of i values are similar.

So the worst case runtime is

$$\begin{aligned} f(n) = {} & (t_1 + t_3) \\ & + n \cdot t_2 \\ & + (n-1) \cdot (t_4 + t_6 + t_{14} + t_{15}) \\ & + \frac{(n-1)n}{2} \cdot (t_7 + t_9 + t_{10} + t_{11} + t_{12} + t_{13}) \\ & + \frac{(n-1)(n+2)}{2} \cdot t_5 \end{aligned}$$

Rewriting this as a polynomial in n, we get

$$\begin{aligned} T(n) = {} & \frac{t_5 + t_7 + t_9 + t_{10} + t_{11} + t_{12} + t_{13}}{2} \cdot n^2 \\ & + \left(t_2 + t_4 + t_6 + t_{14} + t_{15} - \frac{t_7 + t_9 + t_{10} + t_{11} + t_{12} + t_{13} + t_5}{2} \right) \cdot n \\ & + (t_1 + t_3 - t_4 - t_6 - t_{14} - t_{15} - t_5) \end{aligned}$$

In other words the worst case runtime is of the form

$$f(n) = An^2 + Bn + C$$

where A, B, C are constants. And of course in this case

$$f(n) = O(n^2)$$

For the best case:

```
          i = n - 2               t1   1
LOOP1:    if i == -1:             t2   n
              goto ENDLOOP1       t3   1

          j = 0                   t4   1+...+1     = n-1
LOOP2:    if j > i:               t5   n+...+2     = (n-1)(n+2)/2
              goto ENDLOOP2       t6   1+...+1     = n-1
          if a[i] < a[j + 1]:     t7   (n-1)+...+1 = (n-1)n/2
              goto ENDIF          t8   (n-1)+...+1 = (n-1)n/2
          t = a[i]                t9   0
          a[i] = a[j]             t10  0
          a[j] = t                t11  0
          j = j + 1               t12  (n-1)+...+1 = (n-1)n/2
          goto LOOP2              t13  (n-1)+...+1 = (n-1)n/2
```

```
ENDLOOP2: i = i - 1              t14  n-1
          goto LOOP1             t15  n-1

ENDLOOP1:
```

I'll leave it to you to check that the best case runtime is also $O(n^2)$.

Exercise 102.11.1. The following is a variant of the bubblesort:

debug: exercises/bubblesort-3/question.tex

```
for i = n - 2, ..., 0:
    swap = FALSE
    for j = 0 to i
        if a[i] < a[i + 1]:
            swap = true
            t = a[i]
            a[i] = a[j]
            a[j] = t
    if !swap:
        break
```

The point is that if there are no swaps in a pass, then the array is already sorted. The boolean variable `swap` is used to remember if swaps occurred during a pass. Therefore if `swap` is FALSE after a pass, the algorithm stops.

(a) Compute the big-O of the best runtime of the above algorithm. [Obviously the best case occurs when `swap` is FALSE at the end of the first pass.]
(b) Compute the big-O of the worst runtime of the above algorithm.

(Go to solution, page 4146) □

Exercise 102.11.2. The following algorithm computes the sum of values in a 2D array `x` of size n-by-n

debug: exercises/double-for-loop-1/question.tex

```
s = 0
for i = 0, 1, 2, ..., n - 1:
    for j = 0, 1, 2, ..., n - 1:
        s = s + x[i][j]
```

Compute the big-O of the runtime of the above algorithm. [NOTE: The best and worst runtimes are the same. Correct?] (Go to solution, page 4147) □

Exercise 102.11.3. The following algorithm computes the sum of upper triangular values in a 2D array x of size n-by-n:

debug: exercises/double-for-loop-2/question.tex

```
s = 0
for i = 0, 1, 2, ..., n - 1:
    for j = i, i + 1, i + 2, ..., n - 1:
        s = s + x[i][j]
```

Compute the big-O of the runtime of the above algorithm. (Go to solution, page 4148) □

Exercise 102.11.4. Here's another algorithm:

debug: exercises/double-for-loop-3/question.tex

```
s = 0
for i = 0, 1, 2, ..., n - 1:
    for j = 0, 1, 2:
        s = s + x[i][j]
```

Compute the big-O of the runtime of this algorithm. (Go to solution, page 4149) □

Exercise 102.11.5. Here's another algorithm:

debug: exercises/double-for-loop-4/question.tex

```
s = 0
for i = 0, 1, 2, ..., n - 1:
    for j = 0, 1, 2, ..., n - 1:
        for k = 0, 1, 2, ..., n - 1:
            s = s + x[i][j] + k
```

Compute the big-O of the runtime of this algorithm. (Go to solution, page 4150) □

Exercise 102.11.6. Here's another algorithm:

debug: exercises/double-for-loop-5/question.tex

```
s = 0
for i = 0, 1, 2, ..., n - 1:
    for j = 0, 1, 2, ..., i:
        for k = j, j + 1, ..., n - 1:
            s = s + x[i][j] + k
```

Compute the big-O of the runtime of this algorithm. (Go to solution, page 4151) □

Exercise 102.11.7. Yet another algorithm:

debug: exercises/double-for-loop-6/question.tex

```
s = 0
for i = 0, 1, 2, ..., n - 1:
    for j = 0, 1, 2, 3:
        for k = i, i + 1, ..., n - 1:
            s = s + x[i][k] * j
```

Compute the big-O of the runtime of this algorithm. (Go to solution, page 4152) □

Exercise 102.11.8. Yet another algorithm:

debug: exercises/double-for-loop-7/question.tex

```
s = 0
for i = 0, 1, 2, ..., n - 1:
    for j = n/10, ..., n/2:
        for k = i, i + 1, ..., n - 1:
            s = s + x[i][k] * j
```

Compute the big-O of the runtime of this algorithm. (Go to solution, page 4153) □

Exercise 102.11.9. Yet another algorithm:

debug: exercises/double-for-loop-8/question.tex

```
s = 0
for i = 0, 1, 2, ..., (n - 1)/2:
    for j = 0, 1, 2, ..., n/4:
        for k = i, i + 1, ..., n - 1:
            s = s + x[i][k] * j
```

Compute the big-O of the runtime of this algorithm. (Go to solution, page 4154) □

Exercise 102.11.10. Given an n–by–n 2D array, the main (or first) diagonal are the entries at index $(0,0),(1,1),(2,2),...(n-1,n-1)$. The second diagonal are the entries at $(0,1),(1,2),(2,3),...(n-2,n-1)$. The third diagonal are the entries at $(0,2),(1,3),(2,4),...(n-3,n-1)$. Etc. The above are n diagonals. Write a program that stores the sum of first diagonal, sum of second diagonal, sum of third diagonal, etc. of array `x` into another array `y`. What is the runtime? (Go to solution, page 4155) □

debug: exercises/double-for-loop-9/question.tex

Exercise 102.11.11. Given two n–by–n matrices (i.e., 2D arrays), A and B, the matrix product or matrix multiplication AB is the n–by–n matrix C where

debug: exercises/matmult/question.tex

$$C[r][c] = \sum_{k=0}^{n-1} A[r][k] \cdot B[k][c]$$

What is the big-O runtime of matrix multiplication? (Go to solution, page 4156) □

Exercise 102.11.12. * (The doors problem) Remember this problem from CISS245? Suppose you are in a mansion with n rooms, numbered $0, 1, 2, ..., n-1$.

debug: exercises/doors/question.tex

- You run from door 0 to door $n-1$ and open all of them.
- Next, you run from door $n-1$ to door 0 and close every other door.
- Next, you run from door 0 to door $n-1$ opening every third door.
- Next, you run from door $n-1$ to 0, closing every fourth door.
- Etc. You stop if a run would only open or close one door.

The question is this: When the above process ends, how many doors are open? Here's an example when $n = 10$.

- You run from door 0 to door $n - 1 = 9$ and open all of them, i.e., you open doors $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.
- Then you close doors $9, 7, 5, 3, 1$.
- Then you open doors $0, 3, 6, 9$.
- Then you close doors $9, 5, 1$.
- Then you open doors $0, 5$.
- Then you close doors $9, 3$.
- Then you open doors $0, 7$.
- Then you close doors $9, 1$.
- Then you open doors $0, 9$.
- And you stop because at this stage if you want to close doors, you can only close door 9.

Write a function to do the above where you use a boolean array `open[0..n-1]` so that door i is open iff `open[i]` is `true`. The function returns the number of doors which are open when the above process ends.

(a) What is the runtime of your code? Besides using n^k for your big-O, you can also use lg (log base 2). The sharper the bound the better.
(b) Can you design an algorithm that is faster and uses less memory?

(Go to solution, page 4157) □

Solutions

Solution to Exercise 102.11.1.

debug:
exercises/bubblesort-3/answer.tex

Solution not provided. See answers below.

(a) $O(n)$

(a) $O(n^2)$

Solution to Exercise 102.11.2.

debug:
exercises/double-for-loop-1/answer.tex

Solution not provided. See answer below.

$O(n^2)$

Solution to Exercise 102.11.3.

debug:
exercises/double-for-loop-2/answer.tex

Solution not provided. See answer below.

$O(n^2)$

Solution to Exercise 102.11.4.

debug:
exercises/double-for-loop-3/answer.tex

Solution not provided. See answer below.

$O(n)$

Solution to Exercise 102.11.5.

debug:
exercises/double-for-loop-4/answer.tex

Solution not provided. See answer below..

$O(n^3)$

Solution to Exercise 102.11.6.

debug:
exercises/double-for-loop-5/answer.tex

Solution not provided. See answer below.

$O(n^3)$

Solution to Exercise 102.11.7.

debug:
exercises/double-for-loop-6/answer.tex

Solution not provided. See answer below.

$O(n^2)$

Solution to Exercise 102.11.8.

debug:
exercises/double-for-loop-7/answer.tex

Solution not provided. See answer below.

$O(n^2)$

Solution to Exercise 102.11.9.

debug:
exercises/double-for-loop-8/answer.tex

Solution not provided. See answer below.

$O(n^3)$

Solution to Exercise 102.11.10.

debug:
exercises/double-for-loop-9/answer.tex

Solution not provided. See answer below.

$O(n^2)$

Solution to Exercise 102.11.11.

debug: exercises/matmult/answer.tex

Solution not provided. See answer below.

$O(n^3)$

Solution to Exercise 102.11.12.

debug: exercises/doors/answer.tex

Answer and solution not shown.

Index