Computer Science

Y. Liow (May 31, 2025)

Contents

0.1 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 0.1.1. 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".

0.2 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 difficult to compute mathematically. For such cases, measuring runtime experimentally is helpful.

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

0.2.1 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>
#include <unistd.h>

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.

0.2.2 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 0.2.1. 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 **??**) □

0.3 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.

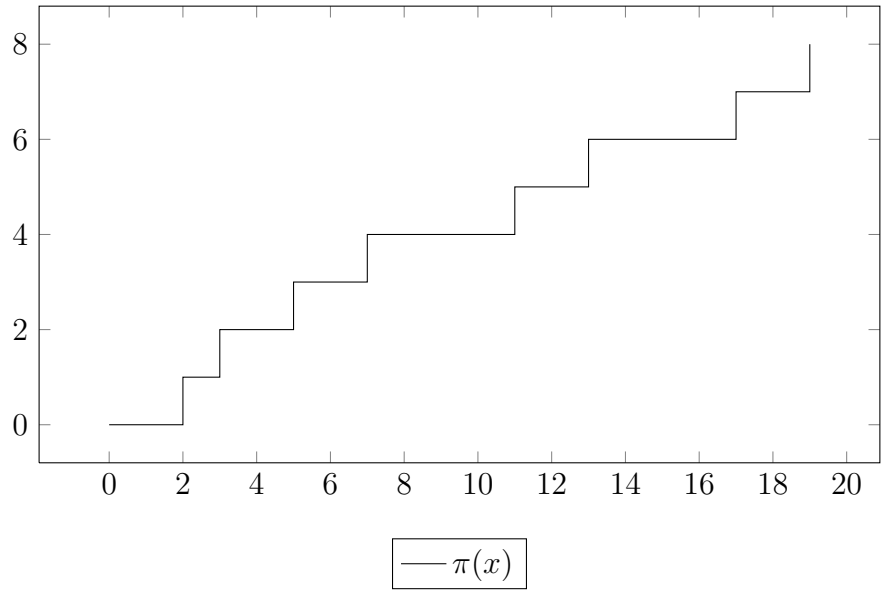
0.4 Roots of Asymptotics (DIY) debug: roots-of-asymptotics.tex

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 originally number theory involved the study of whole numbers, later on techniques involving numbers which are not integers are also 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 theorists, especially analytic number theorists, are also interested 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 $\pi(x)$:

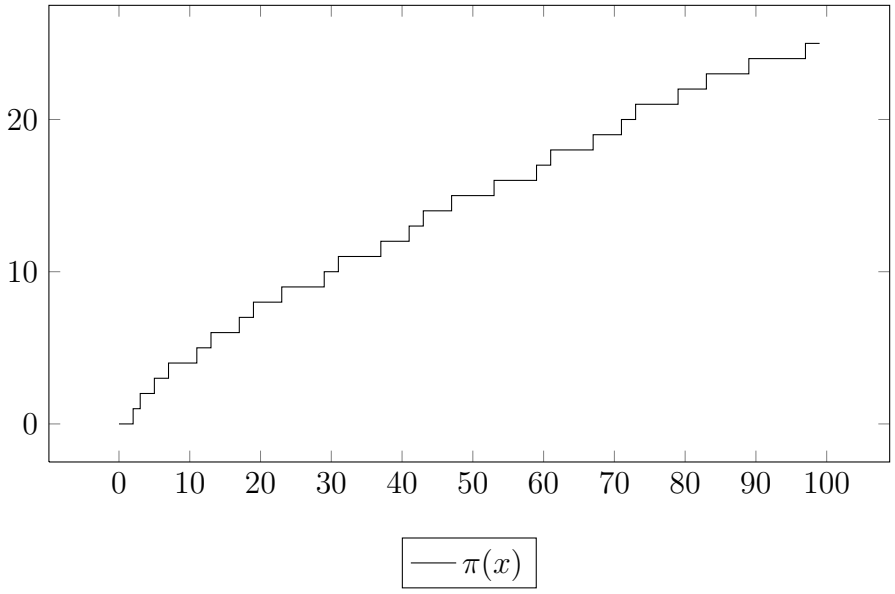
$$\pi(x) = \ldots \text{ formula in } x \ \ldots$$

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

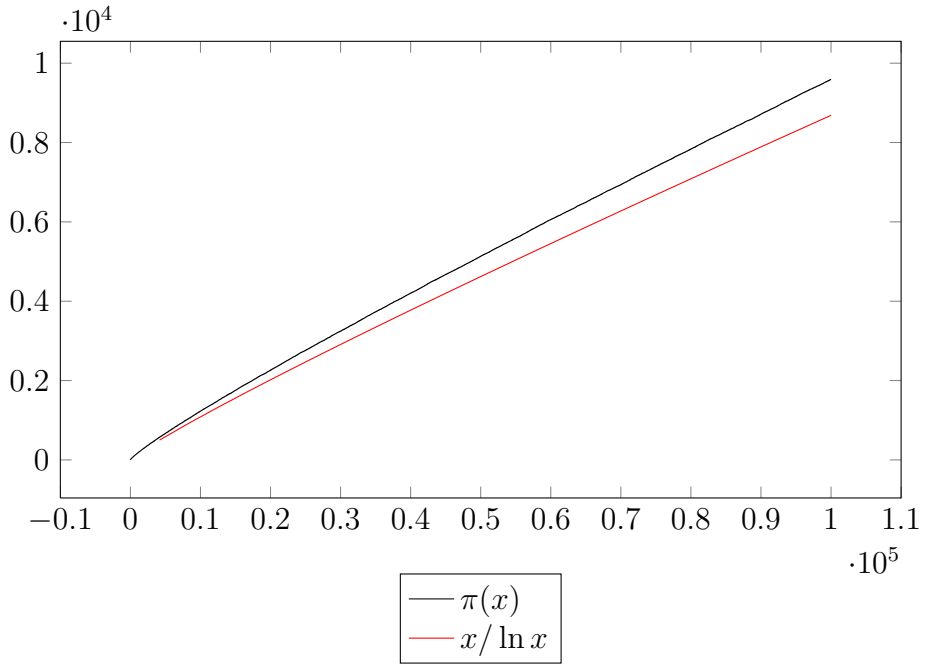
$$n = \pi(p_n) = \ldots \text{ formula in } p_n \ \ldots$$

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 prove 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 proved 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 functions when the x is huge, which informally is the same as looking at the graphs of the functions *when you zoom out*.

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

0.5 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 algorithm 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 mean 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 implemented

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 is 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:

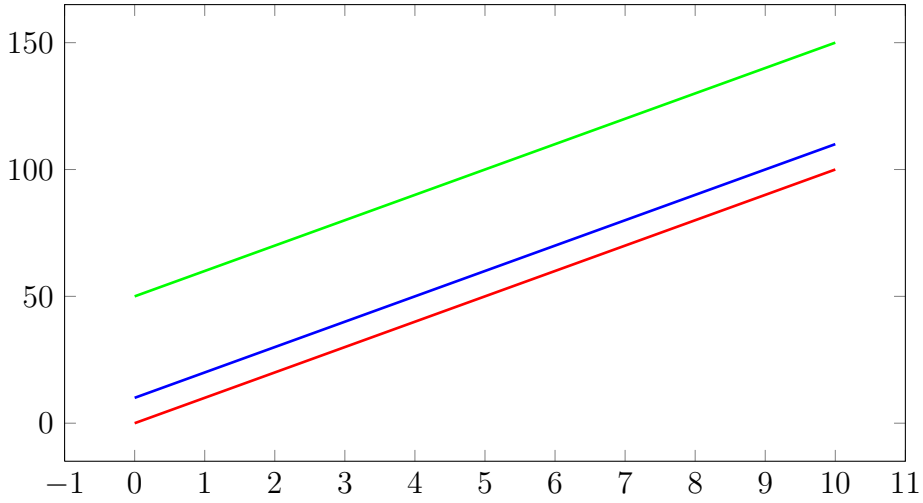
$$T(n) = O(n)$$

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

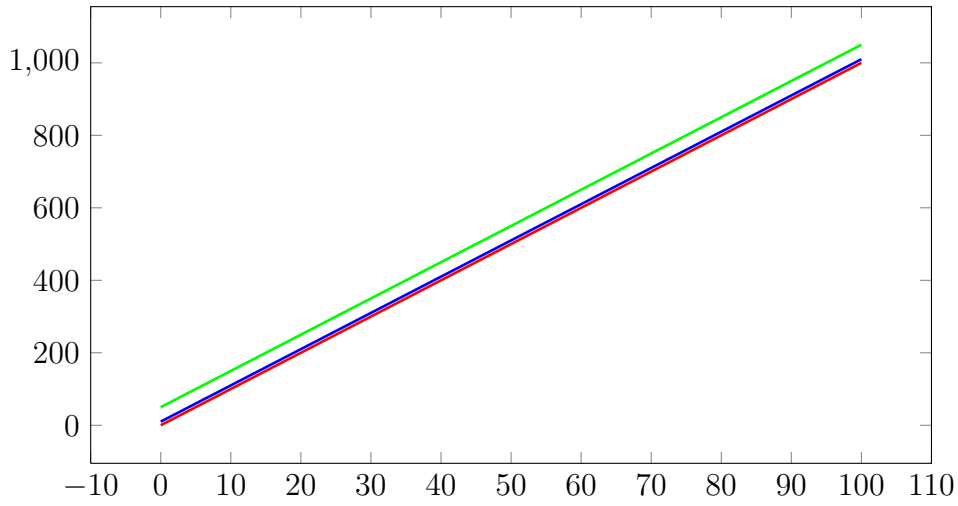
- You only keep the term that is the largest in the long run (i.e. for large n). The growth of $f(n)$ is determined by this term. Typically, your runtime function might have more than 2 terms.
- You replace constant(s) by 1 because different constants indicates different hardware being used.

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

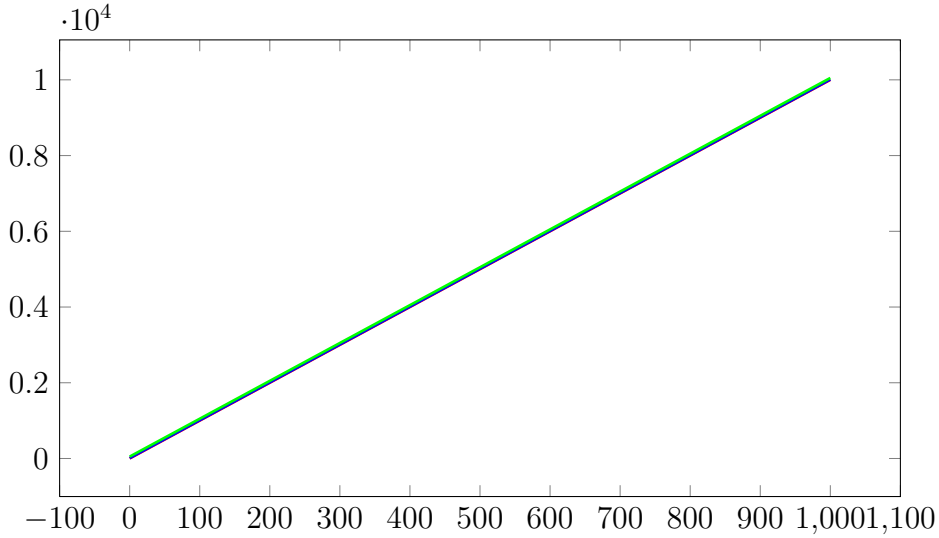
OK, let me show you *graphically* what the fudging does to a function. Here are the plots of $y = 10n, y = 10n + 10, y = 10n + 50$:



(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 complexity 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

$$\text{SPACE}(n) = O(n^0) = O(1)$$

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

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

Exercise 0.5.1. 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 ??) □

Exercise 0.5.2. 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 ??) □

Exercise 0.5.3. 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 **??**) □

Exercise 0.5.4.

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

Let

$$f(n) = 300 + 3n^2 + \sin(n)$$

(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 **??**) □

Exercise 0.5.5. Repeat the previous problem with this function:

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

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

Use a sufficiently large domain for n.

(Go to solution, page **??**) □

Exercise 0.5.6. 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 **??**) □

0.6 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 assumes 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 requires a little more probability theory. For now we will only handle very simplistic average cases.

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

(1) state what cases you're averaging 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 computed 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 0.6.1. 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 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 **??**) □

Exercise 0.6.2. 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 **??**) □

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

Exercise 0.6.3. 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 **??**) □

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

Exercise 0.6.4. 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 **??**) □

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

Exercise 0.6.5. 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 **??**) □

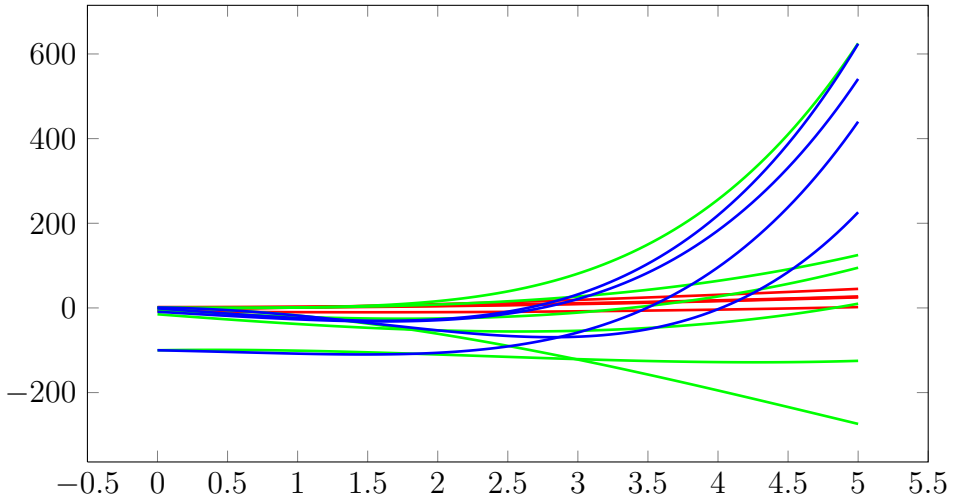
0.7 Separating polynomial functions debug:

separating-polynomial-functions.tex

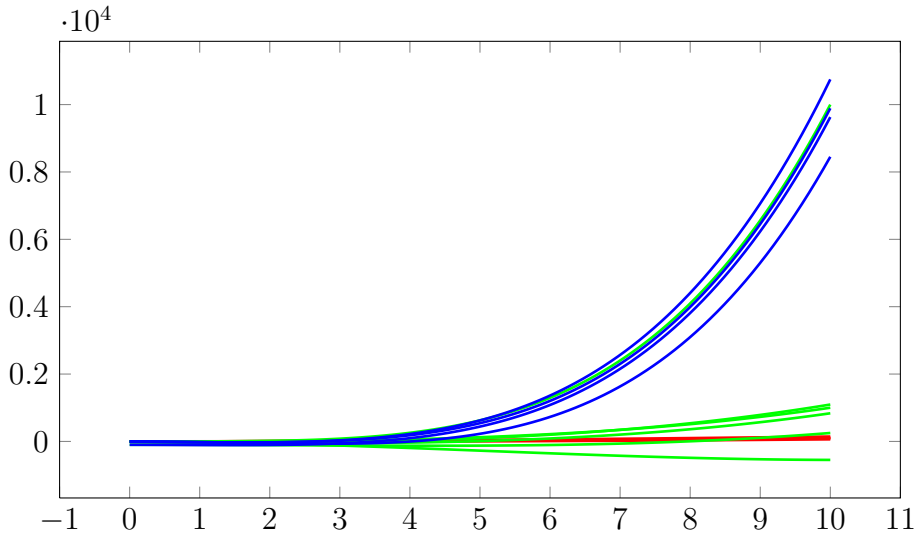
I'll be giving you the formal definition of big-O soon. But before that, I'm going to motivate the (formal) definition of big-O by talking about the way the graphs of polynomial climb. The rate at which they climb essentially tells you the story of big-O among polynomials. This will give you the intuitive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

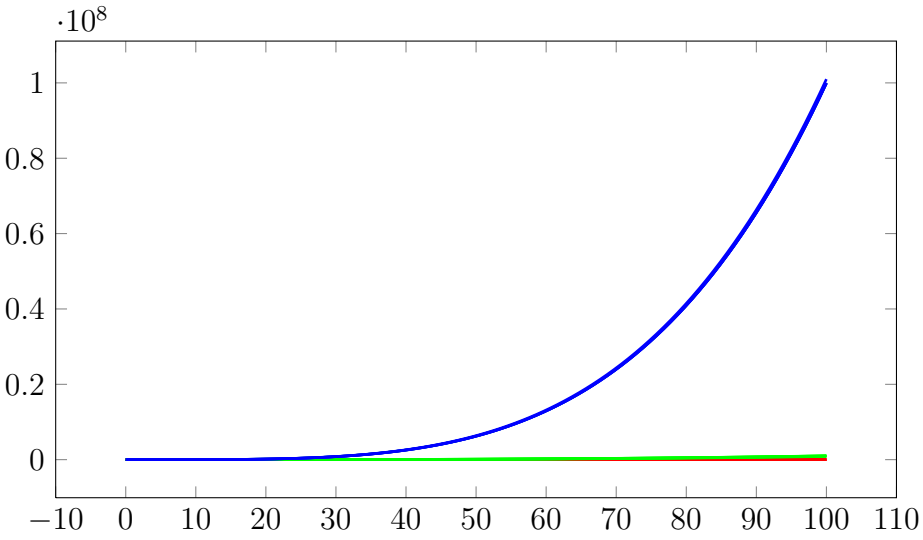
Look at this mess of 15 polynomial functions (I won't give you the polynomials just yet):



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:

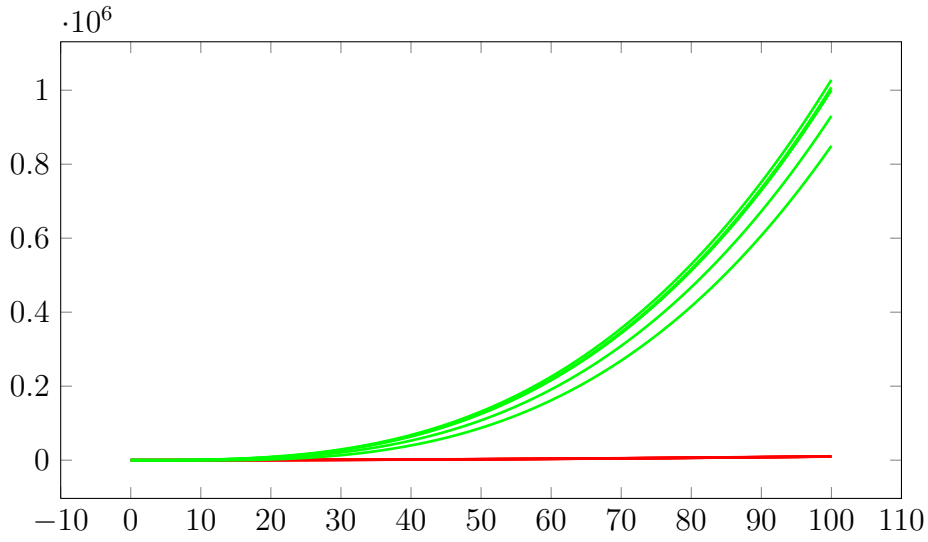


Five functions have separated away from the others. Now I reveal to you that

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:

$$
\begin{aligned}
&n^2\\
&n^2+2\\
&n^2+1\\
&n^2+5n-5\\
&n^2-3n-8
\end{aligned}
$$

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

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

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

$$
\begin{aligned}
&n^4\\
&n^3\\
&n^2
\end{aligned}
$$

(because they are the simplest.)

In general *all* polynomial functions with 1 for the leading coefficient – such polynomials are said 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:

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

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

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

big-O is more like

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

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?

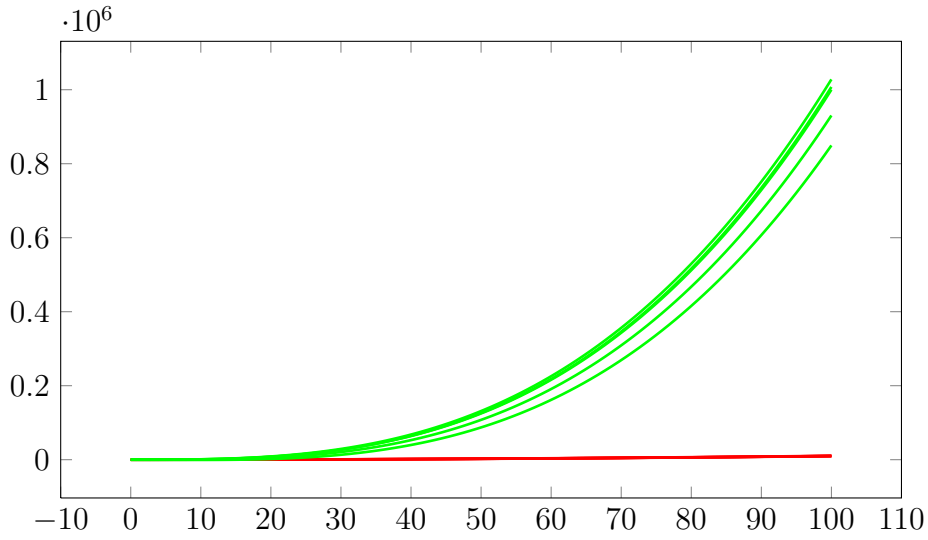
Here are the original 15 functions again:

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

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 0.7.1. *Let $f(n)$ be a polynomial of degree d and $g(n)$ be 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 0.7.1. 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 0.7.1. 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 **??**) □

0.8 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

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

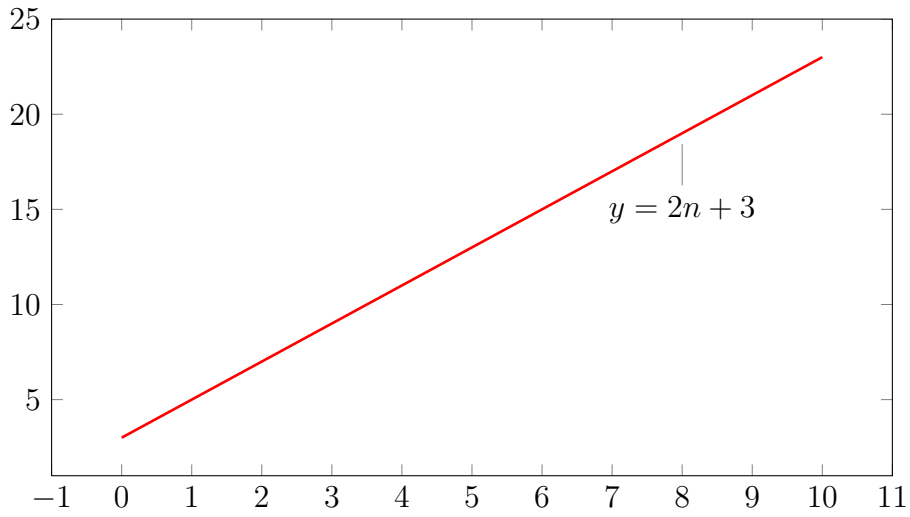
if a *multiple* of $|g(n)|$ beats (i.e., is $\geq$) $|f(n)|$, not necessarily for all n but for *all large values* of n. Of course, the absolutely value $|\cdot|$ is not necessary if the functions are positive. For this section, our $g(n)$ will be $n^0, n^1, n^2, ...$

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

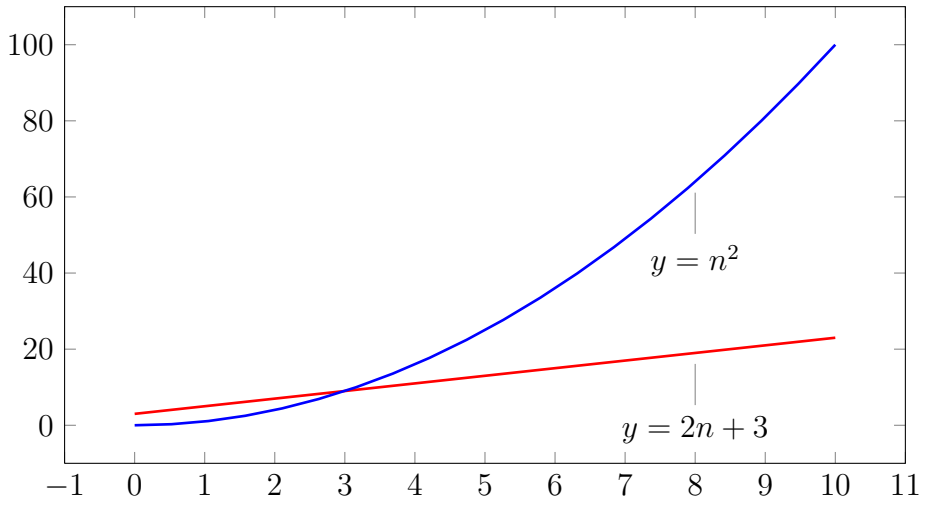
$$f(n) = 2n + 3$$



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

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

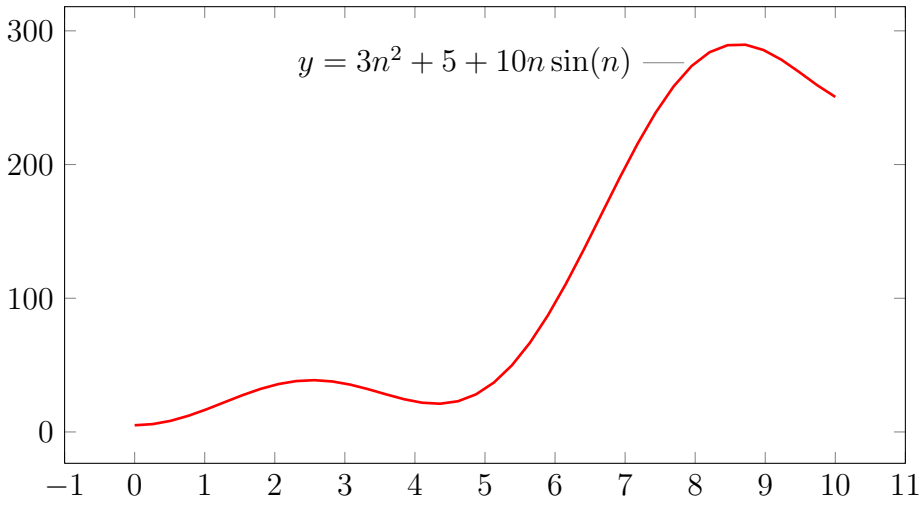
Note that the choice of C and N is not unique. You can also choose $C = 2, N = 10$.

Here's another example.

Suppose

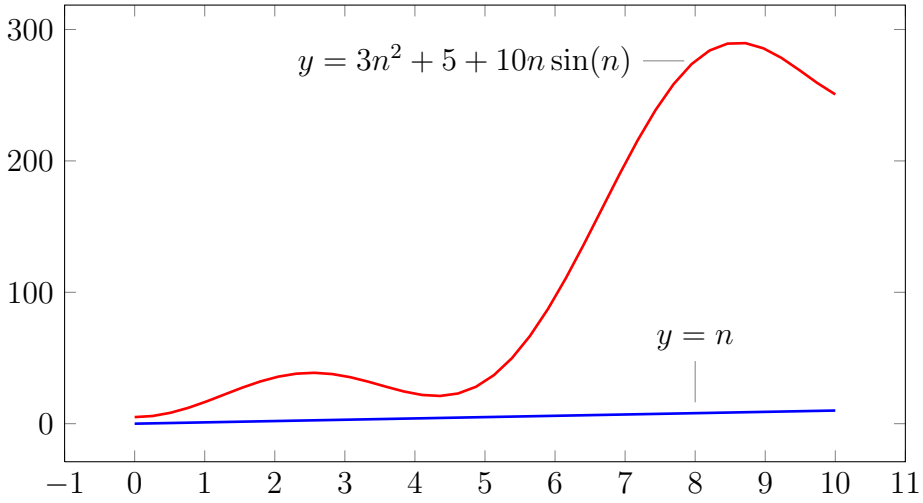
$$f(n) = 3n^2 + 5 + 10n \sin(n)$$

Here's the plot:



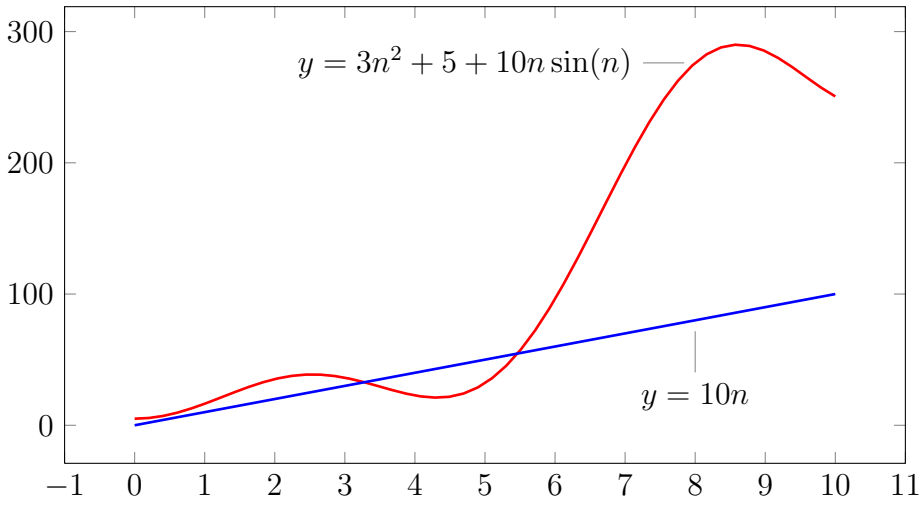
For this example $f(n)$ is positive so $|f(n)| = f(n)$. Let's see if we can *cap* it with

$$g(n) = n$$



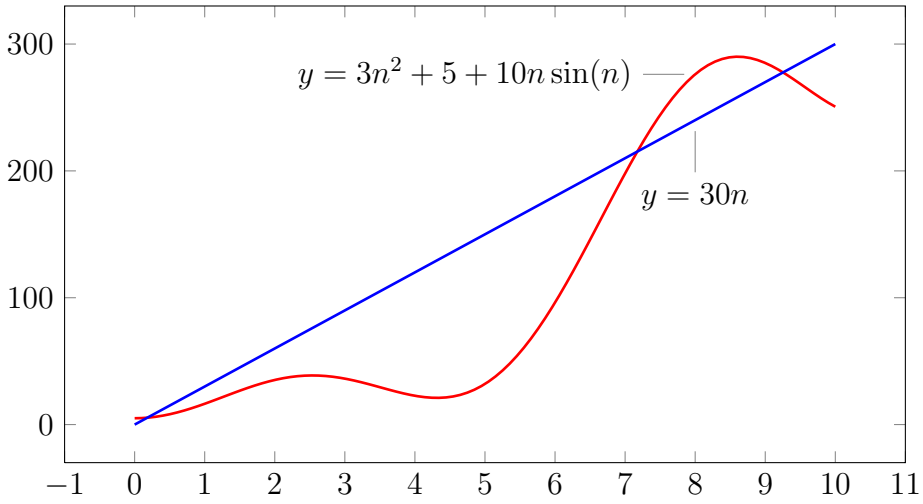
Not good. But don't forget that if we do want to say $f(n) = O(g(n))$, then we are allowed to use multiples of $g(n)$. So let's try

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



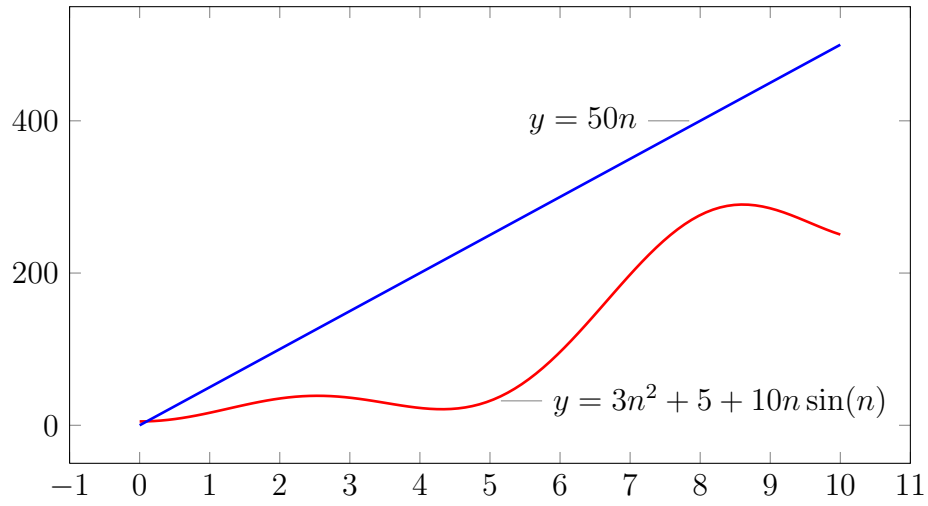
Better!!! But just by looking at the graph, my multiple of $g(n)$ must at least overcome the bump of $f(n)$ at around $n = 8.5$. Let's try

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

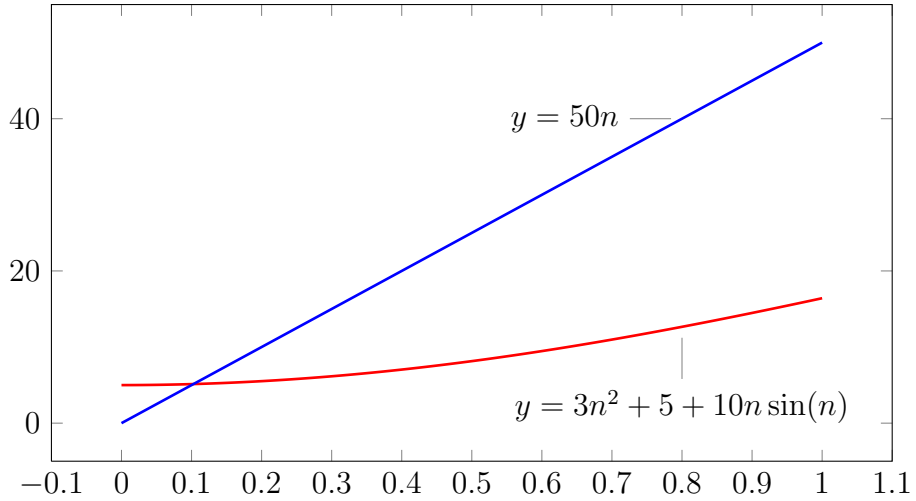


Finally let's try

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



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

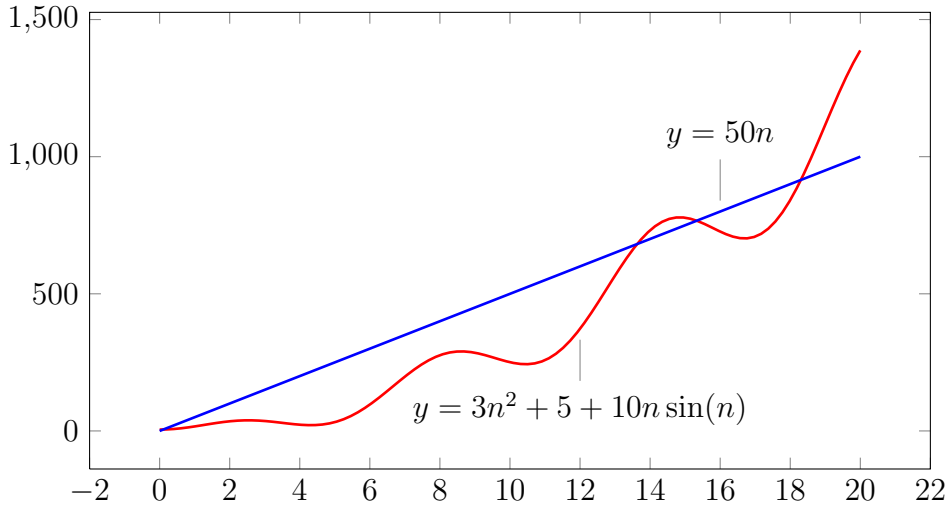
$$f(n) \leq 50g(n) \text{ for } n \geq 1$$

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

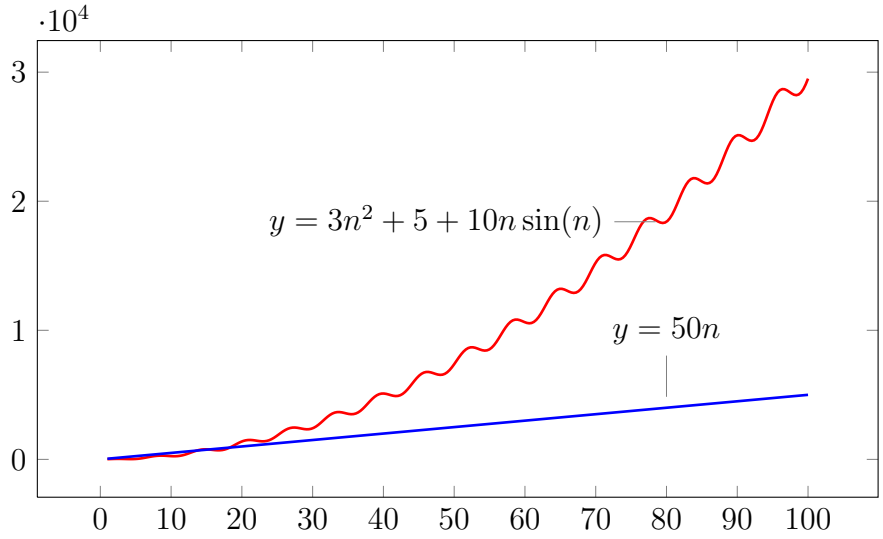
NO!!!

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 $n = 100$:



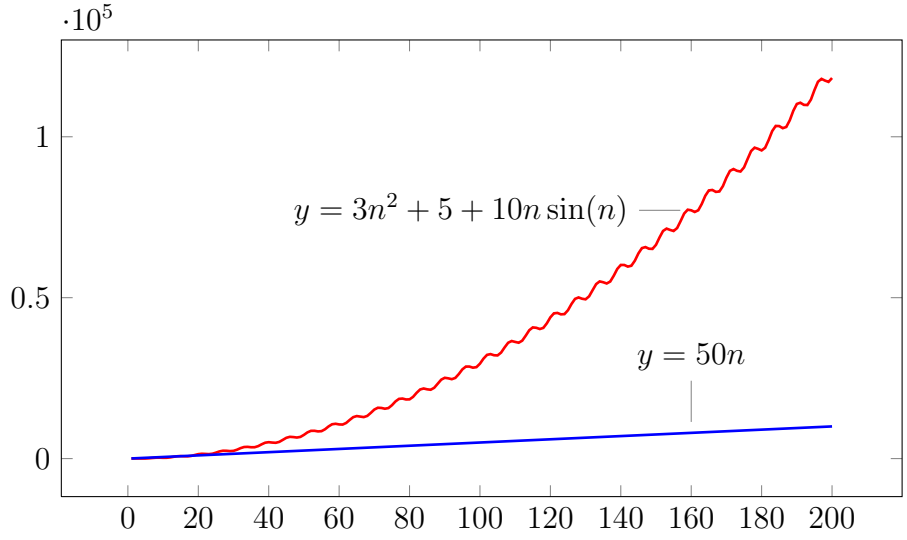
Clearly *no* straight line is going to dominate $f(n)$ because it seems that the graph of $f(n)$ *bends* up (with wiggles along the way).

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 $n = 200$:

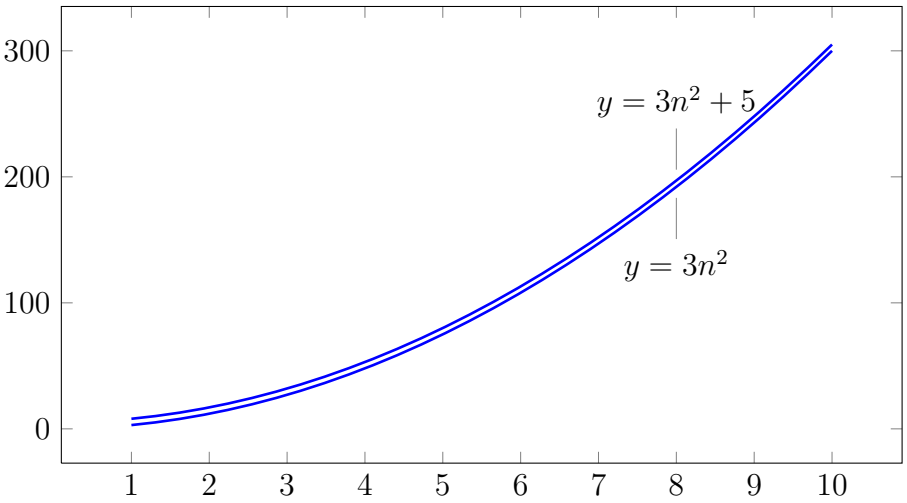


So let's abandon our $g(n) = n$ altogether.

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

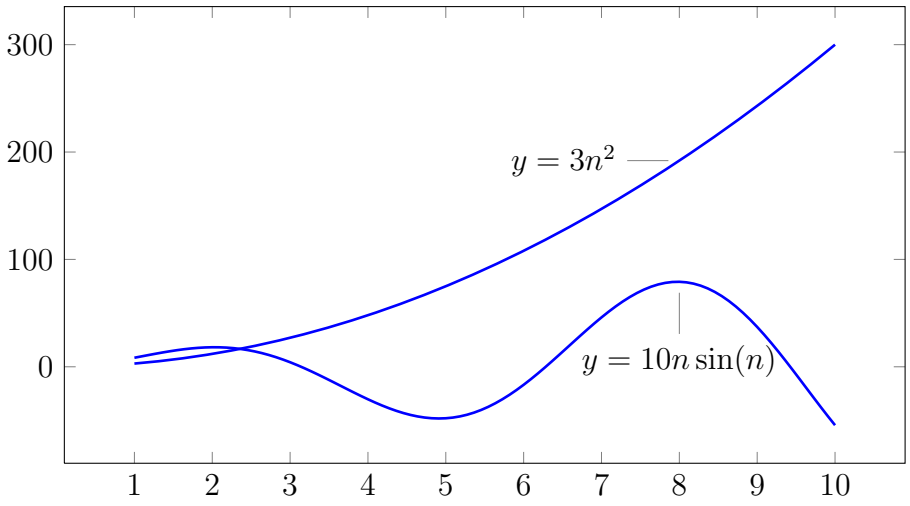
$$f(n) = 3n^2 + 5 + 10n\sin(n)$$

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:

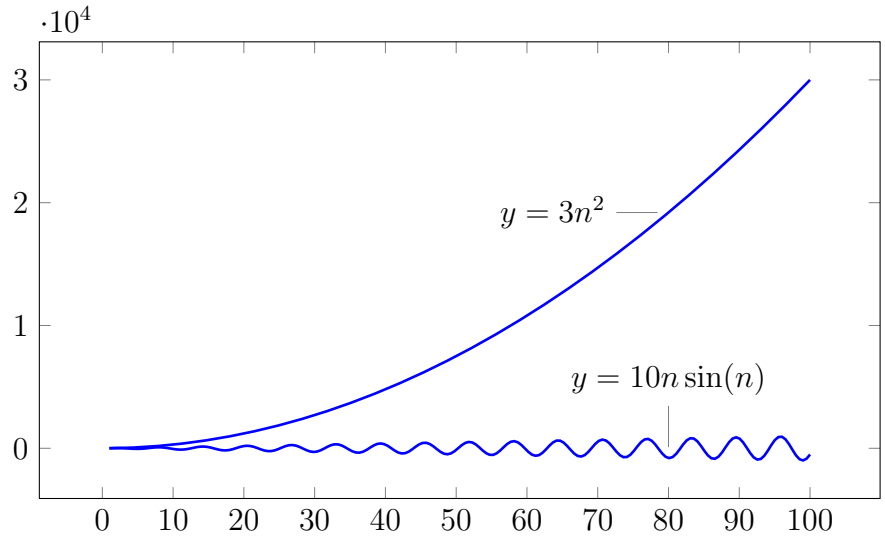


Of course since we can control $f(n)$ with multiples $g(n) = n^2$, later we just need to choose a huge multiple of n^2 to beat $3n^2+5$, for instance $1000000g(n) = 1000000n^2$.

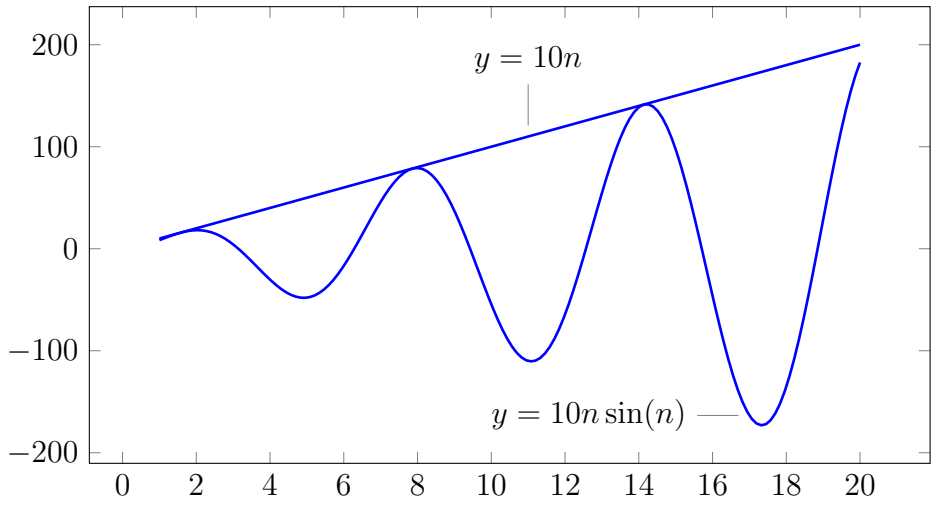
What about $10n \sin n$? If we plot that with $3g(n) = 3n^2$ we get:



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!!!

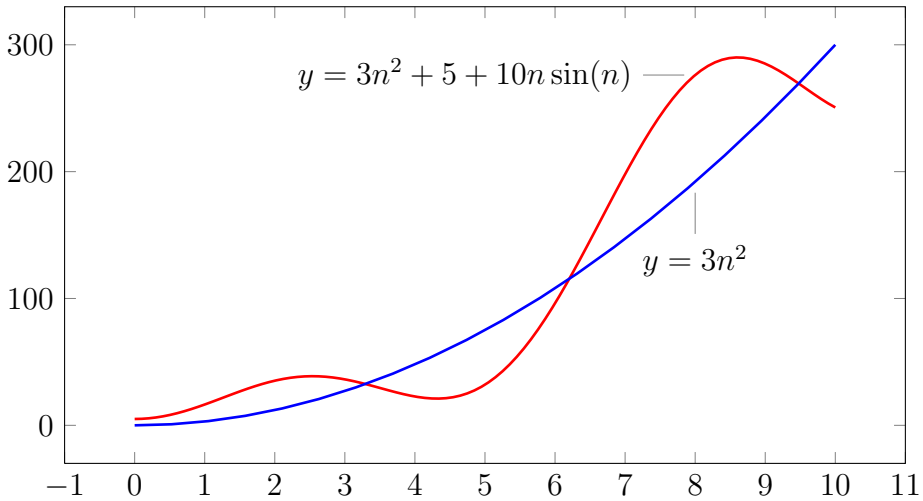
This means that $3g(n) = 3n^2$ will beat $10n \sin n$ for large values of n.

Altogether, this means that the growth of

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

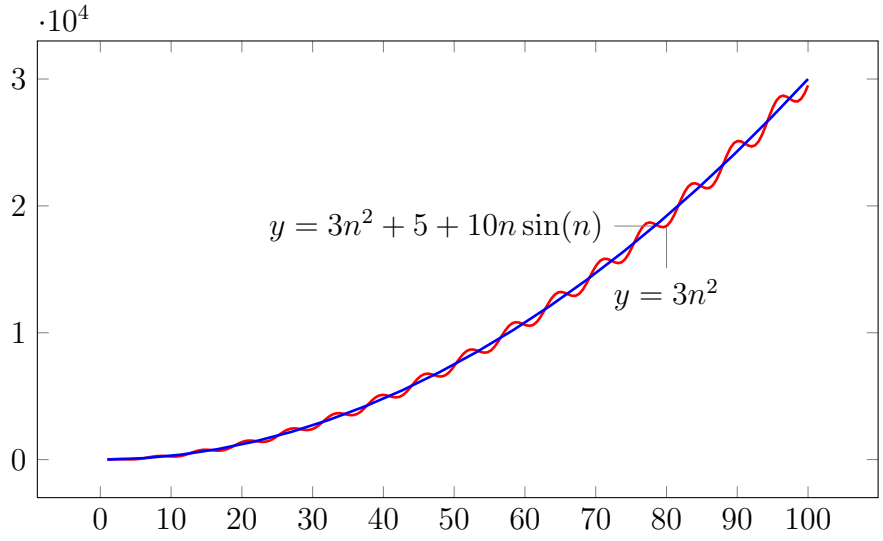
is roughly $3g(n) = 3n^2$ and therefore a higher multiple of $g(n) = n^2$ will beat $f(n) = 3n^2 + 5 + 10n\sin(n)$ for large values of n.

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

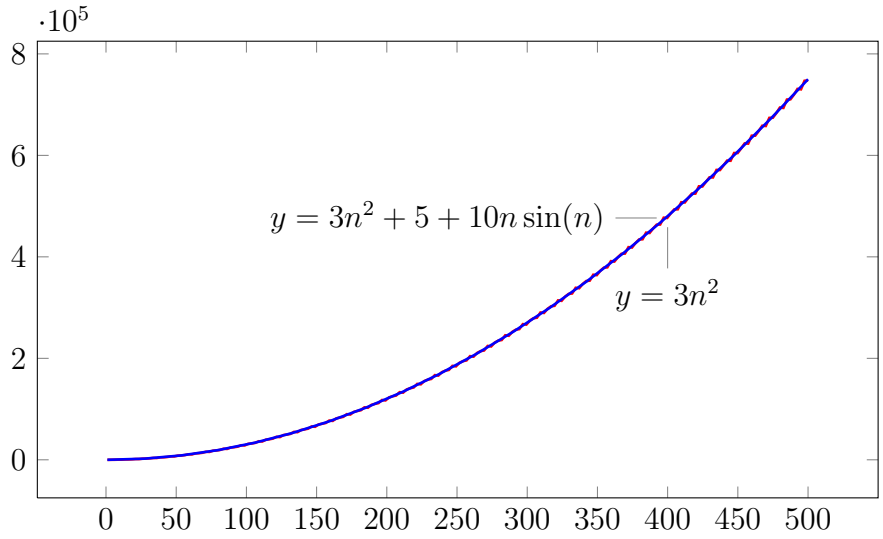


It does seem like $f(n)$ winds itself around $3g(n) = 3n^2$. To get a better feel

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

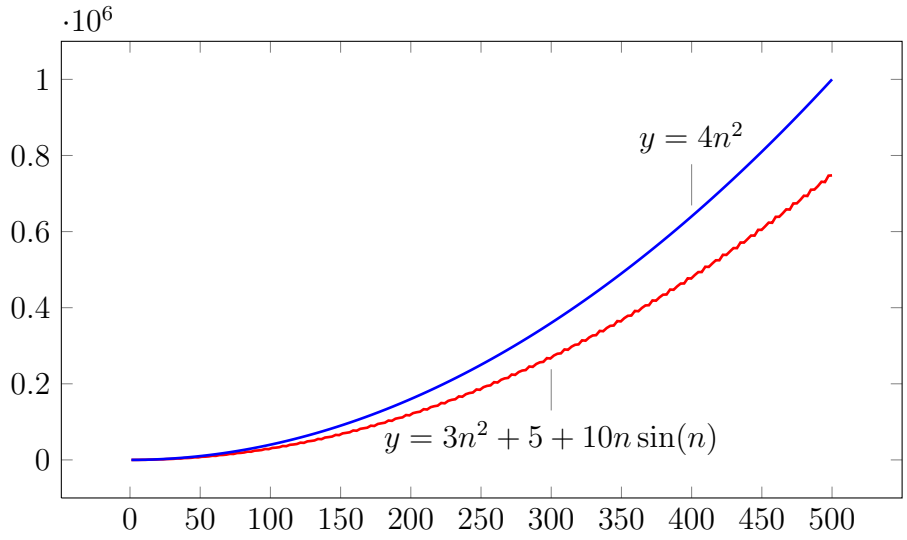


and then up to $n = 500$:

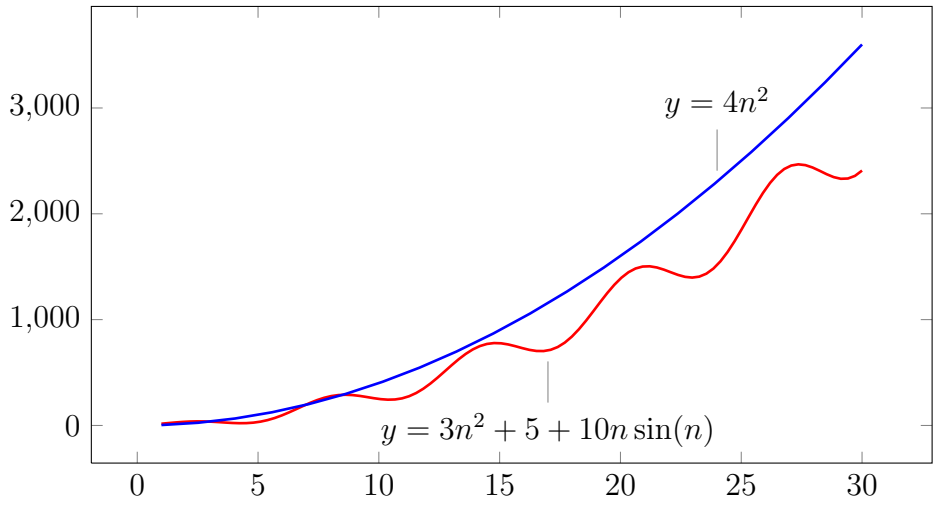


GRRREAT!!!

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 0.8.1. 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

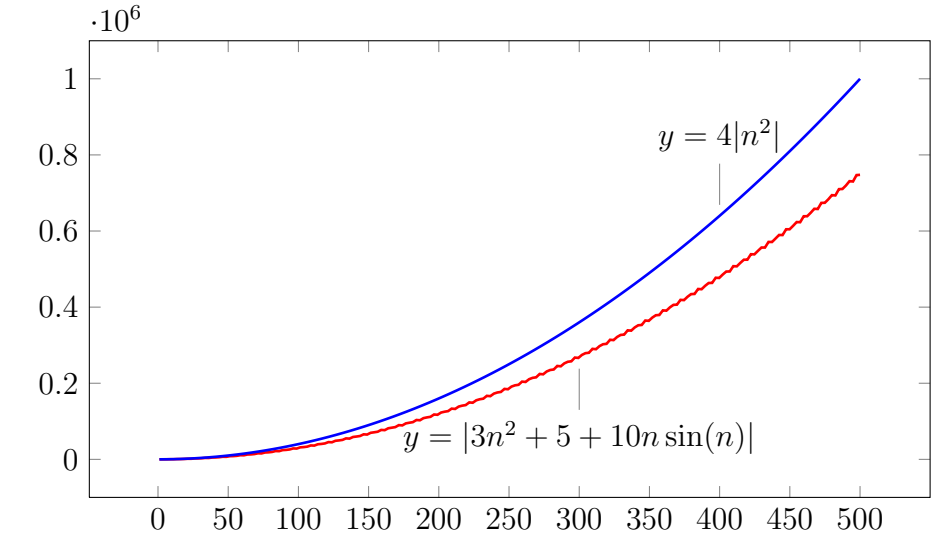
$$|f(n)| \leq C|g(n)|$$

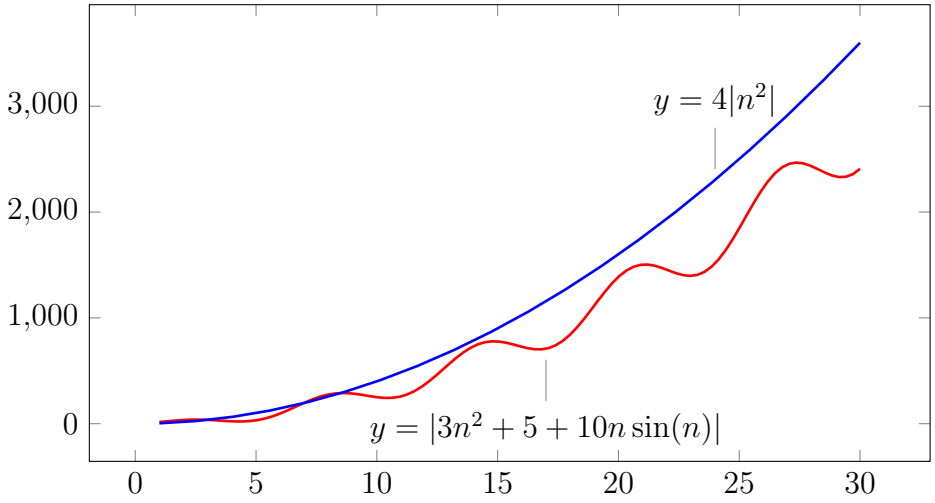
So now I'm ready to give a proper presentation of my solution to the previous problem:

Example 0.8.1. 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.,

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

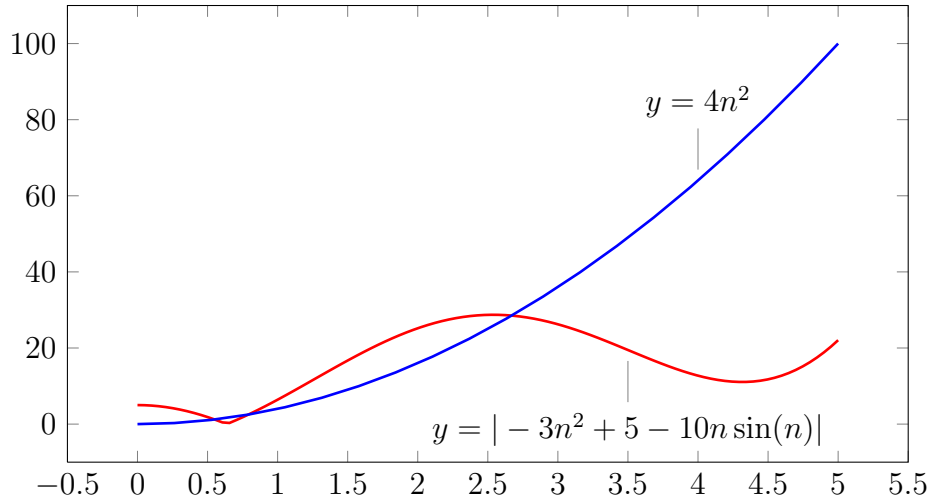
Hence $f(n) = O(g(n))$. $\square$

Remember that technically speaking you cannot prove a mathematical fact like $f(n) = O(g(n))$ using graphs because a graph cannot possibly 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.

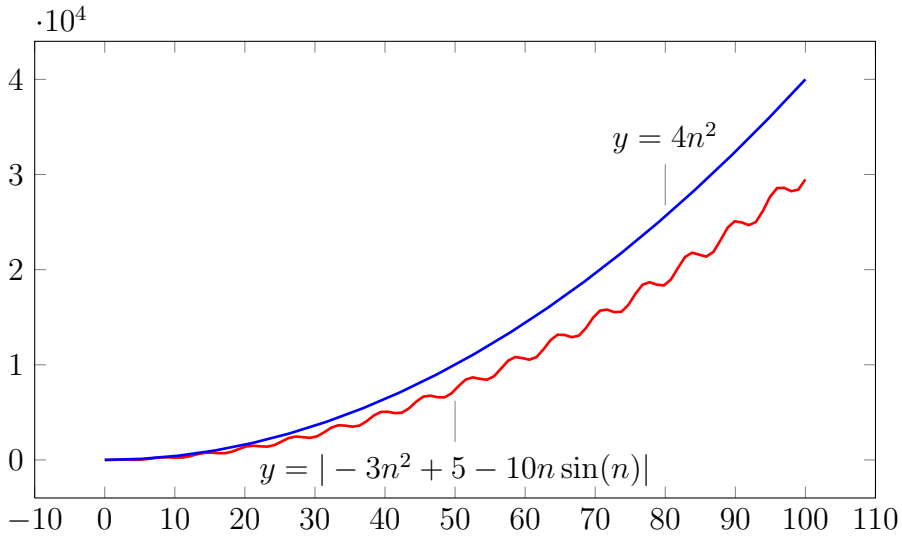
Now, let me give you another example. Suppose I change my $f(n)$ to *this*:

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

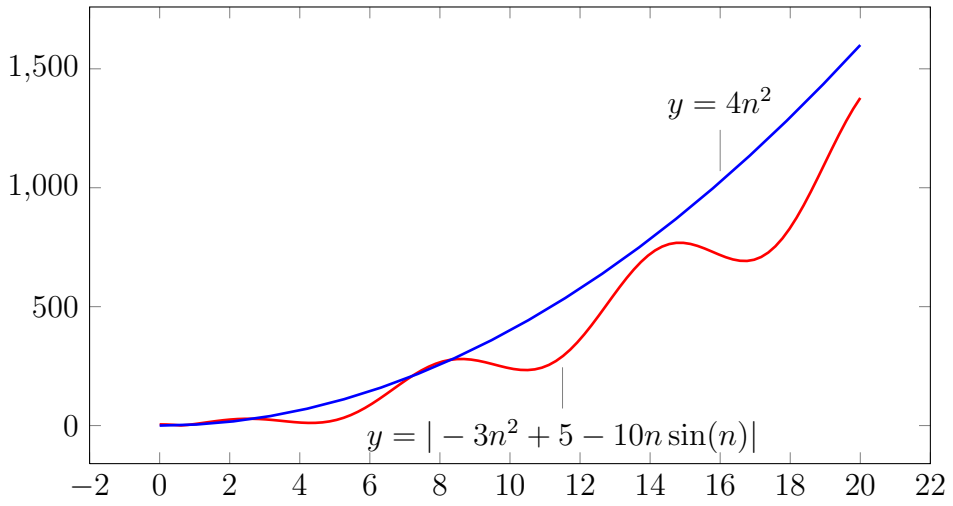
then when I plot $|-3n^2 + 5 - 10n\sin(n)|$ and $4|n^2|$ for $0 \leq n \leq 5$, I get



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 $0 \leq n \leq 20$:



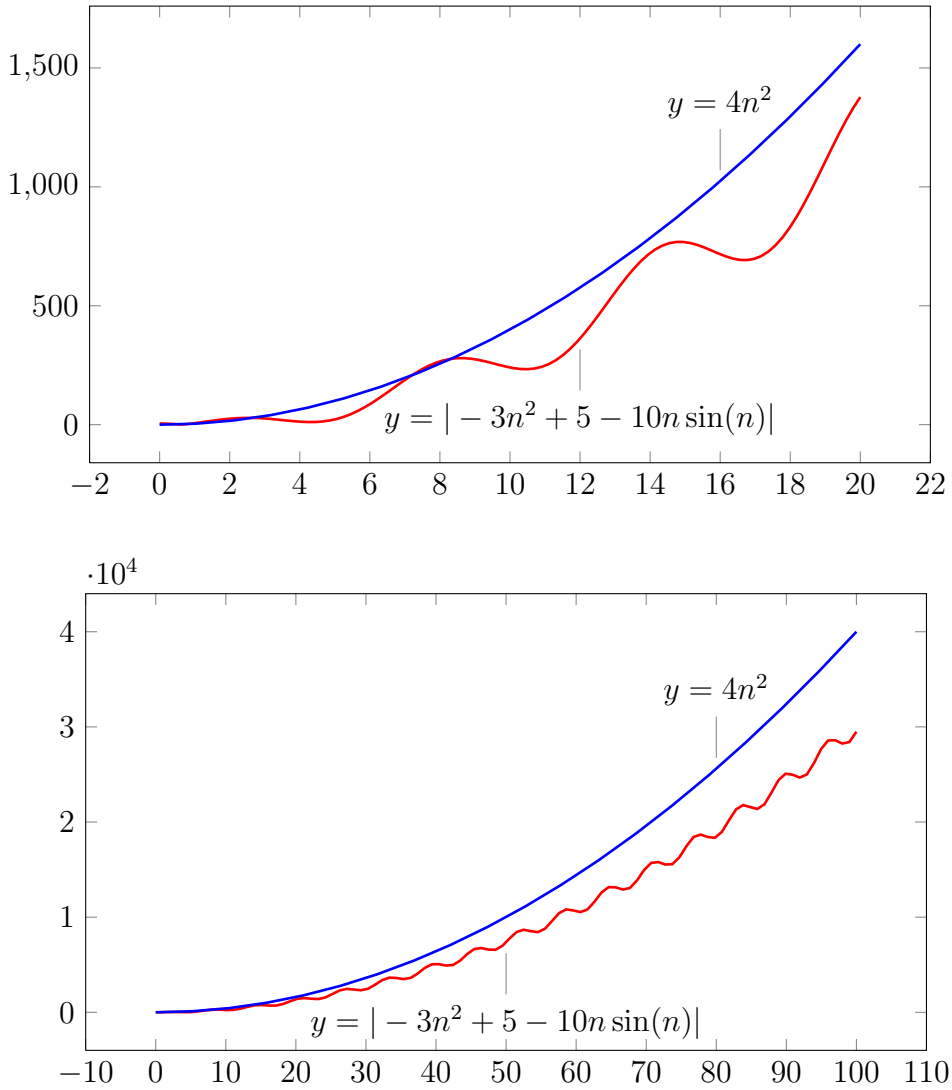
Clearly for $n \geq 10$, $4|g(n)|$ beats $|f(n)|$. So I'm going to pick $N = 10$.

Now I'm ready to present this ...

Example 0.8.2. 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.

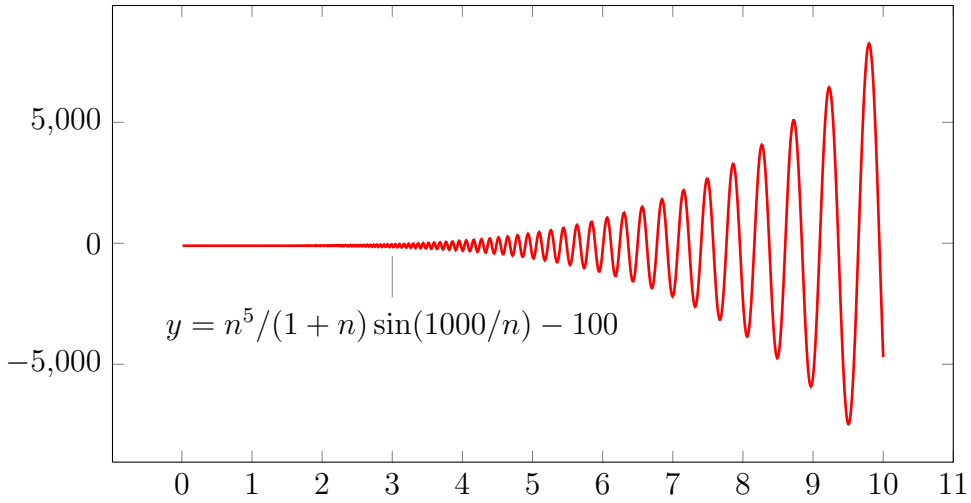
Here's another example.

Let

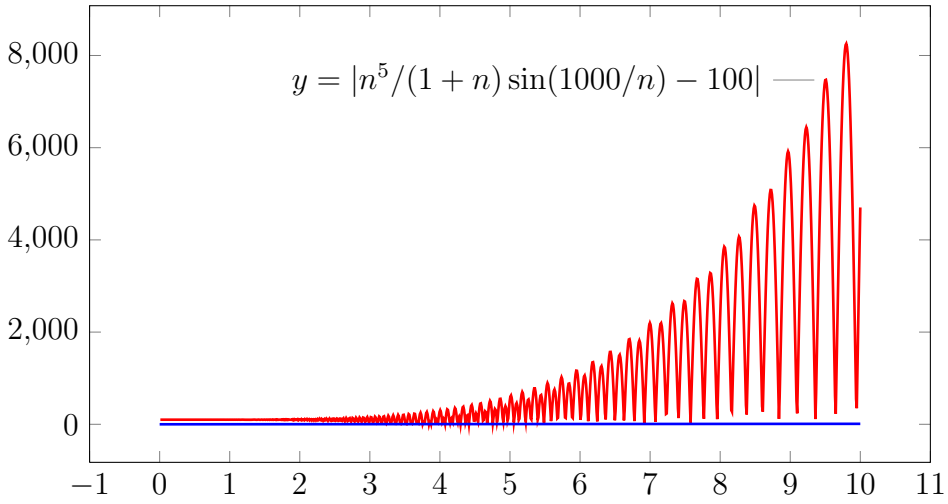
$$f(n) = \frac{2n^5}{1+n}\sin\frac{1000}{n} - 100$$

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.

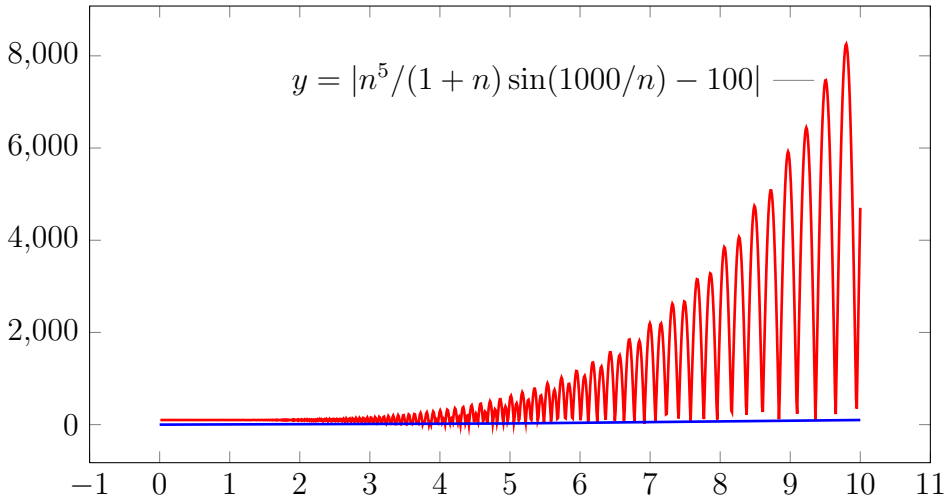
Here's the plot of $f(n)$ for $0 \leq n \leq 10$:



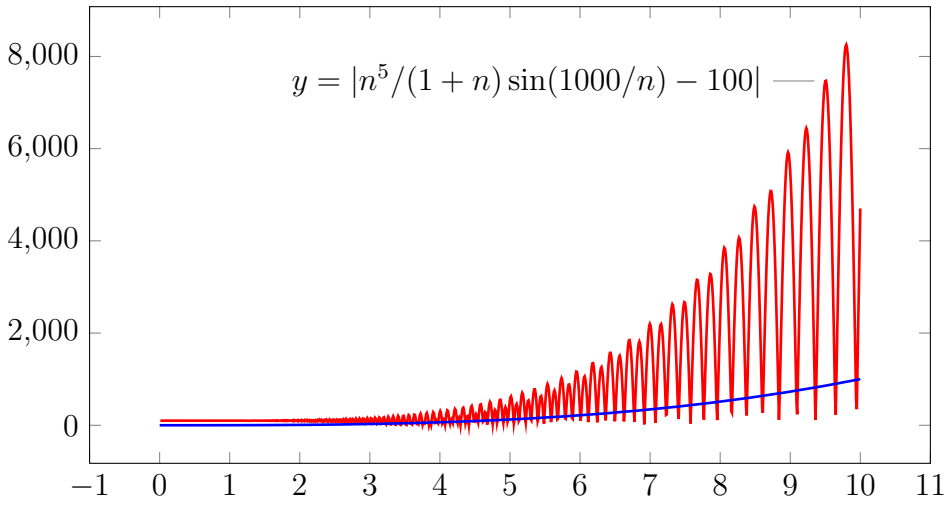
Aha ... $f(n)$ can be negative. I'd better plot $|f(n)|$ instead of $f(n)$. Also, clearly $g(n)$ can't be $n^0 = 1$ (i.e., no multiple of $g(n) = 1$ is going to beat $|f(n)|$ ... right?) so I'm going to skip that. I'll try higher powers. With $g(n) = n$:



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

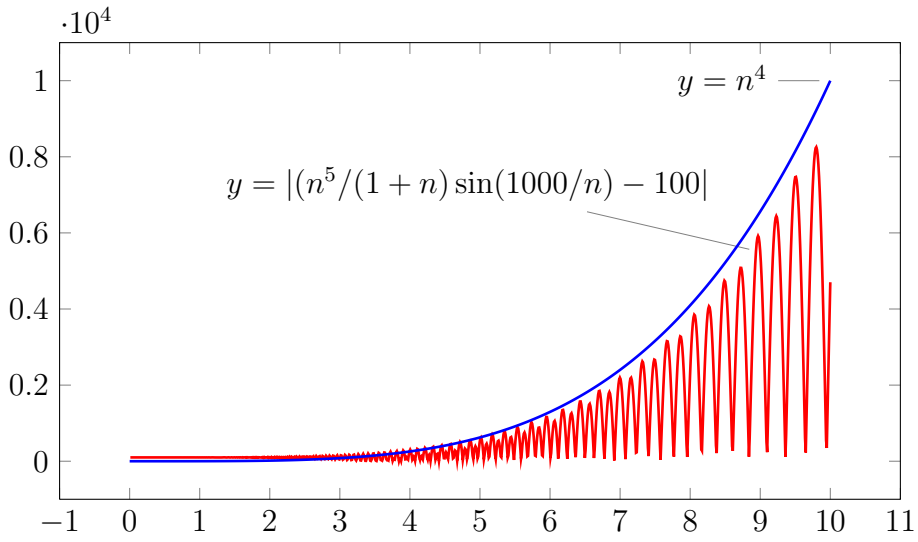


Wow. $f(n)$ is exploding so fast that I can't even see n^2 bending up at all! With $g(n) = n^3$:



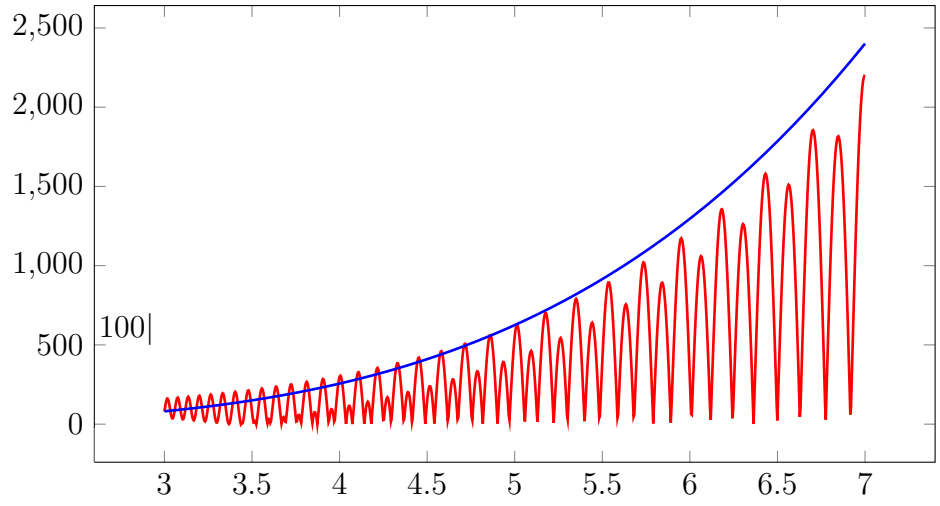
OK ... at least I can see n^3 bending up a little. But it's still not high enough to bound $f(n)$.

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 ...

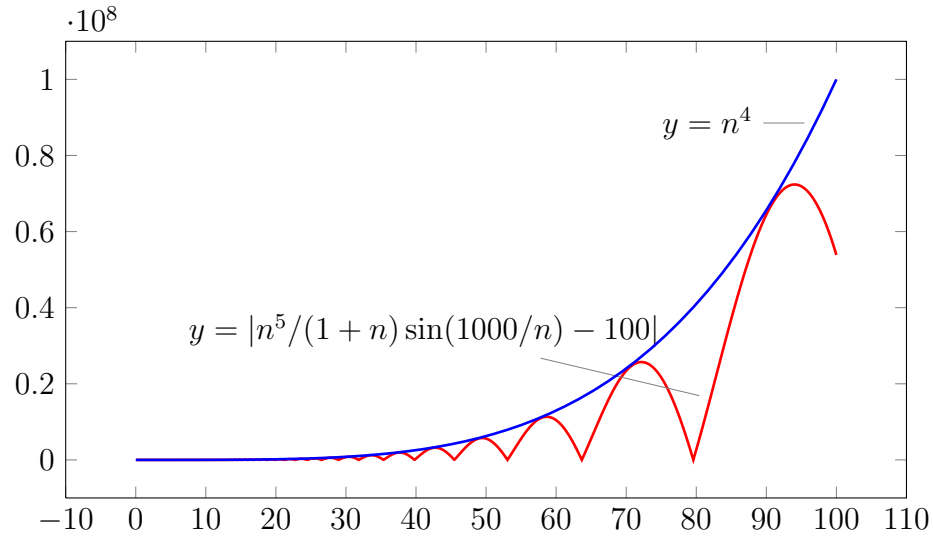
Example 0.8.3. Let

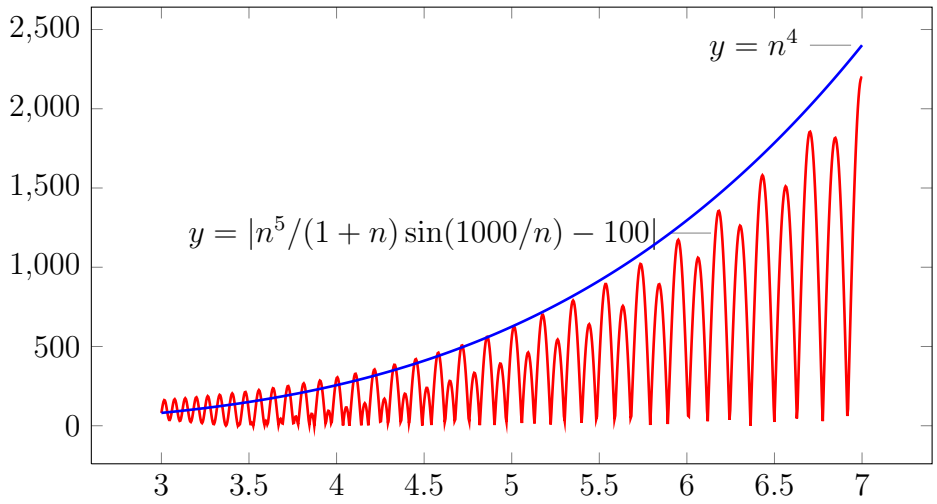
$$f(n) = \frac{n^5}{1+n} \sin \frac{1000}{n} - 100$$

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 0.8.1. 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 **??**) □

Exercise 0.8.2. 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 **??**) □

Exercise 0.8.3. 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 **??**) □

Exercise 0.8.4. 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 **??**) □

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

Exercise 0.8.5. 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 **??**) □

Exercise 0.8.6. 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 **??**) □

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

Exercise 0.8.7. Let

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

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 **??**) □

Exercise 0.8.8. 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 **??**) □

Exercise 0.8.9. 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 **??**) □

0.9 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 0.9.1. 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 **??**) □

Exercise 0.9.2. 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 **??**) □

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

Proposition 0.9.1.

(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 0.9.3. 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 0.9.4. 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}? □

0.10 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 0.10.1.

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

□

I won't prove the above formula.

Exercise 0.10.1. 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 0.10.1. 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 0.10.2. 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 0.10.2. Compute the sum $1 + 2 + 3 + \cdots + 1000$ by first rewriting it as a summation and then using the arithmetic sum formula. □

Exercise 0.10.3. 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 0.10.4. 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 0.10.5. 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 0.10.6. Write the following as a polynomial in decreasing power of n:

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

□

Exercise 0.10.7. Write the following as a polynomial in decreasing power of n:

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

□

Exercise 0.10.8. 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 0.10.2.

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

I won't prove the above formula.

Exercise 0.10.9. Check by hand that the sum of squares formula is correct for $n = 1, 2, 3, 4$.

Exercise 0.10.10. Compute the sum $1^2 + 2^2 + 3^2 + \cdots + 1000^2$. □

Exercise 0.10.11. Write the following as a polynomial in decreasing power of n:

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

□

Exercise 0.10.12. 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.

0.11 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 LOOP1

ENDLOOP1:
```

Next include 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[j] <= a[j + 1]:    t7
              goto ENDIF          t8
          t = a[j]                t9
          a[j] = a[j + 1]         t10
```

```
          a[j + 1] = 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[j] <= a[j + 1]:    t7  (n-1)+...+1 = (n-1)n/2
              goto ENDIF          t8  0
          t = a[j]                t9  (n-1)+...+1 = (n-1)n/2
          a[j] = a[j + 1]         t10 (n-1)+...+1 = (n-1)n/2
          a[j + 1] = 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[j] <= a[j + 1]:    t7  (n-1)
              goto ENDIF          t8  0
          t = a[j]                t9  (n-1)
          a[j] = a[j + 1]         t10 (n-1)
          a[j + 1] = 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[j] <= a[j + 1]:   t7  (n-1)+...+1 = (n-1)n/2
              goto ENDIF         t8  (n-1)+...+1 = (n-1)n/2
          t = a[j]               t9  0
          a[j] = a[j + 1]        t10 0
          a[j + 1] = 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 0.11.1. 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[j] > a[j + 1]:
            swap = true
            t = a[j]
            a[j] = a[j + 1]
            a[j + 1] = 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 **??**) □

Exercise 0.11.2. 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 **??**) □

Exercise 0.11.3. 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 ??) □

Exercise 0.11.4. 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 ??) □

Exercise 0.11.5. 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 ??) □

Exercise 0.11.6. 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 ??) □

Exercise 0.11.7. 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 ??) □

Exercise 0.11.8. 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 ??) □

Exercise 0.11.9. 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 ??) □

Exercise 0.11.10. 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

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

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 **??**) □

Exercise 0.11.11. 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 **??**) □

Exercise 0.11.12. * (The doors problem) Remember this problem from CISS245? debug: exercises/doors/question.tex Suppose you are in a mansion with n rooms, numbered $0,1,2,...,n-1$.

- 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 **??**) □

0.12 Other types of asymptotic bounds

debug:

other-types-of-asymptotic-bounds.tex

In this section, all functions are functions of $n \in \mathbb{N}$. I'll write f instead of $f(n)$, etc. I'll assume that all functions f are asymptotically nonzero, i.e., there is some N such that for all $n \geq N$, I will assume $f(n) \neq 0$.

Recall that $f = O(g)$ is informally some kind of "$\leq$" inequality: a multiple of g bounds f from above asymptotically. We also say that f is **asymptotically bounded above** by g.

asymptotically bounded above

It's also useful to have the concept of asymptotic *lower* bound:

Definition 0.12.1. We say f is the **big-Ω** of g and write $f = \Omega(g)$ if there exist some $C > 0$ and some N such that for $n \geq N$,

big-Ω

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

In other words a multiple of g (asymptotically) bounds f from *below*. I will say that f is **asymptotically bounded below** by g.

asymptotically bounded below

You can put the two definitions together and get this definition:

Definition 0.12.2. $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$, i.e., there exist $C_1 > 0, N_1$ and $C_2 > 0, N_2$ such that for $n \geq N_1$

$$C_1|g(n)| \leq |f(n)|$$

and for $n \geq N_2$,

$$|f(n)| \leq C_2|g(n)|$$

If $f(n) = \Theta(g(n))$, I would say that $f(n)$ has an **asymptotic upper and lower bound** of $g(n)$. Note that you don't need two cutoff points N_1 and N_2 for n since if you set $N = \max\{N_1, N_2\}$, then for $n \geq N$,

asymptotic upper and lower bound

$$C_1|g(n)| \leq |f(n)| \leq C_2|g(n)|$$

So the above definition is equivalent to this: $f = \Theta(g)$ if there are constants $C_1 > 0, C_2 > 0$ and N such that for $n \geq N$,

$$C_1|g(n)| \leq |f(n)| \leq C_2|g(n)|$$

Informally, you want to think about O, Ω, Θ roughly as:

$$\begin{array}{lll} f = O(g) & \text{roughly} & f \text{“}\leq\text{”} g \\ f = \Omega(g) & \text{roughly} & f \text{“}\geq\text{”} g \\ f = \Theta(g) & \text{roughly} & f \text{“}=\text{”} g \end{array}$$

Recall that you want your asymptotic upper and lower bound be as tight as possible. For instance it is true that

$$n^2 + n + 1 = O(n^{100})$$

But this is a tighter upper bound:

$$n^2 + n + 1 = O(n^2)$$

This is the same for Ω:

$$n^2 + n + 1 = \Omega(n^1)$$

So O and Ω can be tight.

Here are some basic facts you should know.

Proposition 0.12.1. *$f = O(g)$ iff $g = \Omega(f)$.*

Proposition 0.12.2. *Let f, g, h, f_i, g_i be functions of $n \in \mathbb{N}$ and $c \neq 0$ be a constant.*

(a) Reflexive:

$$\begin{aligned} f &= O(f) \\ f &= \Omega(f) \\ f &= \Theta(f) \end{aligned}$$

(b) Symmetric: *If f, g are asyptotically nonzero, then*

$$f = \Theta(g) \implies g = \Theta(f)$$

(c) Transitive:

$$\begin{aligned} f = O(g),\ g = O(h) &\implies f = O(h) \\ f = \Omega(g),\ g = \Omega(h) &\implies f = \Omega(h) \\ f = \Theta(g),\ g = \Theta(h) &\implies f = \Theta(h) \end{aligned}$$

(d) Antisymmetric:

$$f = O(g),\ g = O(f) \implies f = \Theta(g)$$

□

Proposition 0.12.3.

(a) Summation:

$$\begin{aligned} f_1 = O(g_1),\ f_2 = O(g_2) &\implies f_1 + f_2 = O\big(\max(|g_1|, |g_2|)\big) \\ f_1 = \Omega(g_1),\ f_2 = \Omega(g_2) &\implies f_1 + f_2 = \Omega\big(\min(|g_1|, |g_2|)\big) \end{aligned}$$

(b) Product:

$$\begin{aligned} f_1 = O(g_1),\ f_2 = O(g_2) &\implies f_1 f_2 = O(g_1 g_2) \\ f_1 = \Omega(g_1),\ f_2 = \Omega(g_2) &\implies f_1 f_2 = \Omega(g_1 g_2) \\ f_1 = \Theta(g_1),\ f_2 = \Theta(g_2) &\implies f_1 f_2 = \Theta(g_1 g_2) \end{aligned}$$

Corollary 0.12.1.

$$\begin{aligned} f_1 = O(g),\ f_2 = O(g) &\implies f_1 + f_2 = O(g) \\ f_1 = \Omega(g),\ f_2 = \Omega(g) &\implies f_1 + f_2 = \Omega(g) \\ f_1 = \Theta(g),\ f_2 = \Theta(g) &\implies f_1 + f_2 = \Theta(g) \end{aligned}$$

Corollary 0.12.2. Constant cancellation:

$$\begin{aligned} f = O(cg) &\iff f = O(g) \quad \textit{and} \quad cf = O(g) \iff f = O(g) \\ f = \Omega(cg) &\iff f = \Omega(g) \quad \textit{and} \quad cf = \Omega(g) \iff f = \Omega(g) \\ f = \Theta(cg) &\iff f = \Theta(g) \quad \textit{and} \quad cf = \Theta(g) \iff f = \Theta(g) \end{aligned}$$

Now I mentioned earlier

$$n^2 + n + 1 = O(n^2)$$
$$n^2 + n + 1 = O(n^3)$$
$$n^2 + n + 1 = O(n^{100})$$

There are times when I want to say "f is asymptotically bounded above by g but g is *not* a tight upper bound". That's where the "little" notations come in: the little-o and little-ω. (ω is the lowercase of Greek Ω.)

Definition 0.12.3. Let f, g be functions of n.

(a) $f = o(g)$ if for *every* constant $C > 0$, there is some $N(C)$ such that for $n \geq N(C)$, o

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

(b) $f = \omega(g)$ if for *every* constant $C > 0$, there is some $N(C)$ such that for $n \geq N(C)$, ω

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

Compare the definitions of little-o and big-O and then little-ω and big-Ω very carefully.

What is the difference between little-o and big-O? Just from the logic behind the two definitions, it's clear that

$$f = o(g) \implies f = O(g)$$

But what is the intuition behind the definition of little-o? For now think of C as a fixed number. Then if $f = o(g)$, there is some $N(C)$ such that for

$$n \geq N(C) \implies |f(n)| \leq C|g(n)|$$

If I change C to a larger number C', I can use the same $N(C)$ for the above implication. So no big deal. But what if I replace C by a smaller C'? Then there is some $N(C')$ such that

$$n \geq N(C') \implies |f(n)| \leq C'|g(n)|$$

This basically says that no matter how small I "shrink" $|g(n)|$ by a very tiny factor C', the $C'|g(n)|$ still bound $|f(n)|$. In other words, if $f = o(g)$, then it means that g is an upper bound of f in such a way that no matter how I try to

"shrink g by a constant multiplier, f can never go beyond g (asymptotically).

Using a concrete example,

$$100n^2 + n + 1 = O(n^2) \text{ and } 100n^2 + n + 1 = O(n^3)$$

but

$$100n^2 + n + 1 \neq o(n^2) \text{ and } 100n^2 + n + 1 = o(n^3)$$

Therefore, informally, while you think of "$f = O(g)$" as roughly "$f \leq g$, you think of "$f = o(g)$" as roughly "$f < g$".

This is the same for little-ω. Informally, you want to think about O, o, Ω, ω, Θ roughly as:

$f = O(g)$	roughly	f"$\leq$"g
$f = o(g)$	roughly	f"$<$"g
$f = \Omega(g)$	roughly	f"$\geq$"g
$f = \omega(g)$	roughly	f"$>$"g
$f = \Theta(g)$	roughly	f"$=$"g

Exercise 0.12.1. Show that

(a) $1 \neq o(1)$ and $1 = o(n)$.
(b) $5n \neq o(12n)$ and $5n = o(2n^2)$.
(c) $100n^2 + n + 1 \neq o(n^2)$ and $100n^2 + n + 1 = o(n^3)$.

Proposition 0.12.4.

(a) *If $f = o(g)$, then $f = O(g)$.*
(b) *If $f = \omega(g)$, then $f = \Omega(g)$.*

Proposition 0.12.5.

(a) Transitivity:

$$f = o(g),\ g = o(h) \implies f = o(h)$$
$$f = \omega(g),\ g = \omega(h) \implies f = \omega(h)$$

(b) Summation:

$$f_1 = o(g_1),\ f_2 = o(g_2) \implies f_1 + f_2 = o\big(\max(|g_1|, |g_2|)\big)$$
$$f_1 = \omega(g_1),\ f_2 = \omega(g_2) \implies f_1 + f_2 = \omega\big(\min(|g_1|, |g_2|)\big)$$

(c) Product:

$$f_1 = o(g_1),\ f_2 = o(g_2) \implies f_1 f_2 = o(g_1 g_2)$$
$$f_1 = \omega(g_1),\ f_2 = \omega(g_2) \implies f_1 f_2 = \omega(g_1 g_2)$$

Corollary 0.12.3.

$$f_1 = o(g),\ f_2 = o(g) \implies f_1 + f_2 = o(g)$$
$$f_1 = \omega(g),\ f_2 = \omega(g) \implies f_1 + f_2 = \omega(g)$$

Corollary 0.12.4. Constant cancellation:

$$f = o(cg) \iff f = o(g) \quad \textit{and} \quad cf = o(g) \iff f = o(g)$$
$$f = \omega(cg) \iff f = \omega(g) \quad \textit{and} \quad cf = \omega(g) \iff f = \omega(g)$$

Besides *inequality* types asymptotic bounds, there are also *limit* types asymptotic properties. Here's one:

Definition 0.12.4. f and g are **asymptotically equivalent**, and we write

asymptotically equivalent

$$f(n) \sim g(n)$$

if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

For instance

$$\lim_{n\to\infty} \frac{n^2 + 1000 \ln n}{n^2 + 10000 n \sin(n)} = 1$$

This does *not* mean that they actually meet, i.e., it does not mean that there is some N such that for $n \geq N$, $f(n) = g(n)$.

Note that

$$f \sim g$$

is the not the same as

$$f = \Theta(g)$$

$f \sim g$ is clearly a lot tighter than $f = \Theta(g)$. For instance

$$\sin(n) = \Theta(1)$$

But

$$\sin(n) \not\sim (1)$$

Proposition 0.12.6.

(a) REFLEXIVE: $f \sim f$
(b) SYMMETRIC: $f \sim g \implies g \sim f$
(c) TRANSITIVE: $f \sim g, g \sim h \implies f \sim h$
(d) SUMMATION: $f_1 \sim g_1, f_2 \sim g_2 \implies f_1 + f_2 \sim g_1 + g_2$
(e) PRODUCT: $f_1 \sim f_2, g_1 \sim g_2 \implies f_1 g_1 \sim f_2 g_2$
(f) QUOTIENT: *If f_2, g_2 are asymptotically nonzero, then*

$$f_1 \sim g_1, f_2 \sim g_2 \implies f_1/f_2 \sim g_1/g_2$$

(a)-(c) says that asymptotic equivalence is an equivalence relation.

Recall that you can view big-Θ as a asymptotic rough version of "=". Specifically $C_1|g(n)| \leq f(n) \leq C_2|g(n)|$ for large n. There C_1, C_2 are some fixed constants. Asymptotic equivalence is tighter than big-Θ:

Proposition 0.12.7. *If $f \sim g$, then $f = \Theta(g)$.*

Therefore if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists, one can use $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ as the definition of $f = o(g)$. This is in fact done in some books. But my definition is more general.

Proposition 0.12.8. *Assume f, g are asymptotically > 0. Suppose $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists (including the case of ∞). Let $L = \lim_{n\to\infty} \frac{f(n)}{g(n)}$. Note that $L \in [0, \infty)$ or $L = \infty$. For simplicity, I'll write, "$L \in (0, \infty]$" as a shorthand for "$L \in (0, \infty)$ or $L = \infty$".*

(a) $L \in [0, \infty) \iff f = O(g)$ (d) $L = 0 \iff f = o(g)$

(b) $L \in (0, \infty] \iff f = \Omega(g)$
(c) $L \in (0, \infty) \iff f = \Theta(g)$
(e) $L = \infty \iff f = \omega(g)$
(f) $L = 1 \iff f \sim g$

Proof. (d) Let

1 $\lim_{n\to\infty} f(n)/g(n) = 0$ means: For all $\epsilon > 0$, there is some $N(\epsilon)$ such that if $n \geq N(\epsilon)$, then $|f(n)/g(n) - 0| < \epsilon$. The inequality is equivalent to $|f(n)| < \epsilon|g(n)|$.
2 $f = o(g)$ means: For all $C > 0$, there is some $N(C)$ such that $|f(n)| \leq C|g(n)|$.

Therefore I need to show (1)$\iff$(2).

(1)$\implies$(2): Let $C > 0$. By (a), there is some $N(C)$ such that for $n > N(C)$, $|f(n)| < C|g(n)|$ and hence $|f(n)| \leq C|g(n)|$.

(2)$\implies$(1): Let $\epsilon > 0$. By (b), there is some $N(\epsilon/2)$ such that for $n \geq N(\epsilon/2)$, $|f(n)| \leq (\epsilon/2)|g(n)|$, and hence $|f(n)| \leq (\epsilon/2)|g(n)| < \epsilon|g(n)|$. □

Therefore, provided $\lim\limits_{n\to\infty} f(n)/g(n)$ exists, the above provides an alternative definition for $O, \Omega, \Theta, o, \omega$. For instance if $\lim\limits_{n\to\infty} f(n)/g(n)$ exists, then

$$f = o(g) \iff \lim_{n\to} \frac{f(n)}{g(n)} = 0$$

Therefore when $\lim\limits_{n\to\infty} f(n)/g(n)$ exists, we can use Calculus to compute the asymptotic relation between f and g. The following are some results that uses this technique.

Proposition 0.12.9.

(a) $\ln^a n = o(n^b)$ *where* $b/a > 0$ *or* $a = 0, b > 0$
(b) $n^b = o(c^n)$ *where* $c > 1$

Proof. (a) Suppose $b/a > 0$.

$$\begin{aligned}
\lim_{n\to\infty} \frac{\ln^a n}{n^b} &= \lim_{n\to\infty} \left(\frac{\ln n}{n^{b/a}}\right)^a \\
&= \lim_{n\to\infty} \left(\frac{1/n}{(b/a)n^{b/a-1}}\right)^a && \text{by l'Hôpital's rule} \\
&= \lim_{n\to\infty} \left(\frac{1}{(b/a)n^{b/a}}\right)^a \\
&= 0
\end{aligned}$$

For the case $a = 0, b > 0$, $\ln^a n = 1$ and $\lim\limits_{n\to\infty} 1/n^b = 0$. Hence in both cases $\ln^a n = o(n^b)$.

(b) Suppose $b > 0$.

$$\begin{aligned}
\lim_{n\to\infty} \frac{n^b}{c^n} &= \lim_{n\to\infty} \left(\frac{n}{c^{n/b}}\right)^b \\
&= \left(\lim_{n\to\infty} \frac{n}{c^{n/b}}\right)^b \\
&= \left(\lim_{n\to\infty} \frac{1}{c^{n/b}(\ln c)(1/b)}\right)^b && \text{by l'Hôpital's rule} \\
&= \left(\frac{b}{\ln c} \cdot \lim_{n\to\infty} \frac{1}{c^{n/b}}\right)^b \\
&= 0
\end{aligned}$$

If $b \le 0$, $\lim\limits_{n\to\infty} \dfrac{n^b}{c^n} = \lim\limits_{n\to\infty} \dfrac{1}{n^{|b|}c^n} = 0$. Hence in both cases $n^b = o(c^n)$. □

Proposition 0.12.10. *If $c < d$, then $c^n = o(d^n)$.*

The follow says that in any asymptotic relation, a function can be replaced by another asymptotic equivalent function:

Proposition 0.12.11. *Suppose $f_1 \sim f_2, g_1 \sim g_2$.*

(a) $f_1 = O(g_1), \implies f_2 = O(g_2)$
(b) $f_1 = \Omega(g_1), \implies f_2 = \Omega(g_2)$
(c) $f_1 = \Theta(g_1), \implies f_2 = \Theta(g_2)$
(d) $f_1 = o(g_1), \implies f_2 = o(g_2)$

(e) $f_1 = \omega(g_1), \implies f_2 = \omega(g_2)$

0.13 Function call debug: function-call.tex

The runtime analysis for function call is similar when it comes to computing the runtime of the *body* of the function. The only other things you need to be careful about are

1. the cost of parameter passing
2. the cost of entering the function
3. the cost of returning from the function
4. the cost of return value

(See CISS360 for details. Also, see CISS240 note on functions, especially the function call stack.) Here, cost means both time and memory. You'll definitely need to know how to compute the cost for the case of recursive functions.

Suppose you have a function `f()` that takes an array and returns the first value:

```
int f(int x[])
{
    return x[0];
}
```

and `main()` calls `f()` like this:

```
int main()
{
    int x[] = {1, 2, 3};
    // A. start timing
    int y = f(x);
    // B. end timing
    return 0;
}
```

What is the time taken (from point A to point B)? Here's roughly what happens:

1. The arguments are stored somewhere (for instance in the function call stack). For this example, the address `x` is stored.
2. The point of return from `f()` is stored somewhere (for instance in the function call stack). This means the address of the machine code to be executed after returning is stored.
3. The **program counter** is set to the address of `f()`. The program counter is a device in the CPU that basically remembers the address of the machine code of your program to execute next.
4. Function `f()` begins execution including creating parameters using the value(s) stored in the function call stack.
5. If there is a return value, the function has to store the return value somewhere.
6. The program counter is set to the point of return from function call `f()`.
7. `main()` continues execution.

program counter

(I said roughly because it's an over-simplification, but sufficient for us to compute the time and space complexity.)

The time taken to set the program counter in order to enter `f()` or return to `main()` is constant. The time taken to store the point of return to `main()` is also constant. The space usage for address of the function and point of return is also constant. So we don't really care too much about the above, time-wise or space-wise. The main cost would be the time and space usage for

- storing the arguments
- retrieving the arguments and constructing the parameters
- storing return value

In the case of our example

```
int f(int x[])
{
    return x[0];
}
```

the argument is the address of an array. The amount of memory used is constant. So the time to store the address, retrieving the address, setting the parameter `x` to this address are all constant. `x` is a pointer since, from CISS245, the above code is the same as

```
int f(int * const x)
{
    return x[0];
}
```

so `x` is most likely a 64-bit pointer (or 32). For sure it does not depend on the size of the array. So that space complexity is constant too. No matter how large the array `x` points to, `x` consumes 64 bits. That's why the time taken to set up `x` and the space usage of `x` are all constant. (Don't remember pointers? You had better re-study your pointer notes from CISS245.)

Therefore the total runtime to execute `f()` is $O(1)$ and the space complexity if also $O(1)$.

Now suppose I want to use a `std::vector< int >` instead of an integer array. What will happen then?

If I do this:

```
int f(const std::vector< int > & x)
{
    return x[0];
}
```

then all timing are the same as before except that we need to think about the time for setting up `x`. In this case, `x` is a reference, which (recall) is just like a pointer. So the total runtime is still $O(1)$ and the space complexity is still $O(1)$.

Now come the trick question ... what about this:

```
int f(const std::vector< int > x)
{
    return x[0];
}
```

or

```
int f(std::vector< int > x)
{
    return x[0];
}
```

In this case, `x` is a `std::vector< int >` *object* – it is not a reference and not a pointer. If in `main()`, I have

```
int main()
{
    std::vector< int > y;
    ...
    std::cout << f(y) << '\n';
    ...
}
```

then the `x` of `f()` is a *copy* of `y` of `main()`, created through the copy constructor of `std::vector`. If `y` is a vector of 1000 values, then it takes a certain amount

of time to set up `x`, which includes requesting memory for `x` and *copying* the 1000 values of `y` to `x`. And if `y` is a vector of 1000000000 values, then it takes even more time to set up `x`. Therefore the set up time for `x` is actually $O(n)$ where n is the size of `y`. Hence this algorithm has a total runtime of $O(n)$ and the space complexity is also $O(n)$.

In the same way if you *return* a value that is large in size, then the time due to returning might not be $O(1)$. For instance look at this:

```
std::vector< int > f(std::vector< int > & v)
{
    return v;
}
```

In this case, the parameter set up time is $O(1)$ because `v` is a reference. But the return type is a `std::vector< int >` value (not reference). So the copy constructor is used to create a `std::vector< int >` value for returning from the value `v` references. So hidden in the above code is a $O(n)$ runtime where n is the size of the vector that `v` references and the space complexity is also $O(n)$.

This explains why in CISS245, for "large" variables such as `struct` variables or objects, we always pass by pointer or pass by reference. This is the same reason that an array parameter of a function is actually a pointer:

```
void f(int x[]) // same as void f(int * const x)
{
     ...
     return;
}
```

Exercise 0.13.1.

debug: exercises/function-call-1/question.tex

Assume the size of parameter `x` is n. What is the asymptotic runtime and space complexity of calling the following functions?

(a)
```
std::vector< int > & g(std::vector< int > & x)
{
    return x;
}
```

(b)
```
int g(const std::vector< int > x)
{
    return x[0];
```

```
}
```

(c)

```
int g(const std::vector< int > x)
{
    int s = 0;
    for (int i = 0; i < x.size(); ++i)
    {
        s += x[i];
    }
    return s
}
```

(d)

```
std::vector< int > g(const std::vector< int > & x)
{
    return x
}
```

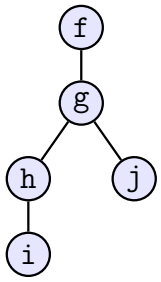
(Go to solution, page **??**) □

Now suppose you have a sequence of function calls where the first function called is `f()`:

```
1  void j(double y)
2  {
3       int c[30];
4  }
5  void i(int x[])
6  {
7       int c[30];
8  }
9  void h(int x[], double y)
10 {
11      int c[30];
12      i(x);
13 }
14 double g(int x[])
15 {
16     double y;
17     int b[20];
18     h(x, y);
19     int d[40];
20     j(y);
21 }
22 int f(int x[])
23 {
24     int a[10];
25     g(x)
26     return 42;
```

```
27 }
```

The following is the function call graph:



The following lists memory usage (by line numbers) at various points during the program execution:

end of line 24 : 10 integers
end of line 17 : 30 integers, 1 pointer, 1 double
end of line 11 : 60 integers, 2 pointer, 2 double
end of line 26 : 10 integers

Recall (from CISS245) that this structure is called a tree where `f` is called the root of the tree. We'll be studying trees in detail.

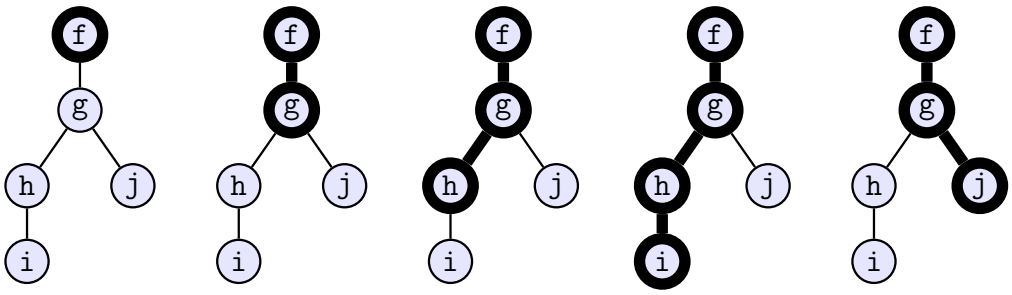
Note that the space usage grows *and shrinks* with program execution since

- memory used by a local variable is reclaimed when this local variable goes out of scope
- memory used by a function is reclaimed when you exit a function
- memory in the heap is reclaimed when you deallocate the memory

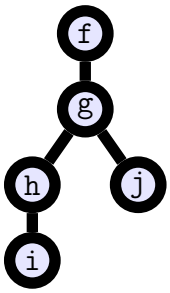
However time usage only increases with program execution. You can reuse memory, but you cannot reuse time!!! To compute the worst case scenario of memory usage, i.e., maximum memory usage, you would need to consider the memory usage by

- `f()`
- `f()` , `g()`,
- `f()` , `g()`, `h()`
- `f()`, `g()`, `h()`, `i()` and
- `f()`, `g()`, `j()`

in the function call graph:



For the case of computing runtime, you have to look at the time used for *all* functions:



and if `g()` calls either `h()` or `j()` (but not both), then you have to consider

- `f()`, `g()`, `h()`, `i()` and
- `f()`, `g()`, `j()`

Remember: you can reuse space, but you cannot reuse time!!!

Exercise 0.13.2. Give me a simple example of a function `f()` that called function `g()` and the space complexity is highest when the program execution is at a point in `f()` instead of in `g()`. The point: it does *not* mean that you use more memory if the stack of function calls is larger. (Go to solution, page ??) □

debug: exercises/function-call-3/question.tex

Exercise 0.13.3. What is the runtime of executing the following function `f` (including function calling and returning) and what is the space complexity? Assume the size of all vectors involved is n:

debug: exercises/function-call-2/question.tex

```
int h(std::vector< int > x)
{
    ++x[0];
```

```
    return x[0];
}

int g(std::vector< int > x)
{
    --x[0];
    return x[0];
}

void f(std::vector< int > x)
{
    if (x[0] < 0)
    {
        std::cout << g(x) << '\n';
    }
    else
    {
        std::cout << h(x) << '\n';
    }
}
```

Speed up the code by modifying the functions – do not change anything in the body of `f()`. What's the new runtime and space complexity after your improvement?

(Go to solution, page **??**) □

Exercise 0.13.4.

debug: exercises/function-bad-bubblesort/question.tex

(a) What is the runtime of this bubblesort, assuming all `std::vector< int >` have size n. Explain every step of your reasoning very clearly.

```
std::vector< int > swap(std::vector< int > x, int i, int j)
{
    int = x[i];
    x[i] = x[j];
    x[j] = x[i];
    return x;
}

std::vector< int > bubblesort(std::vector< int > x)
{
    int n = x.size();
    for (int i = n - 2; i >= 0; --i)
```

```
    {
        for (int j = 0; j <= i; ++j)
        {
            if (x[j] > x[j + 1])
            {
                x = swap(x, j, j + 1);
            }
        }
    }
    return x;
}
```

(b) What is the space complexity?

(c) Rewrite the above to speed it up and reduce memory consumption. Note: Both functions must return something. What is the new runtime and memory used?

(Go to solution, page **??**) □

0.14 Recursive function call

debug: recursive-function-call.tex

Now let me consider the case of recursive functions. Recursive functions can take many forms. I'll focus on three typical cases.

Of course we have to look at Fibonacci:

```
int fib(int n)
{
    if (n == 0 || n == 1)
    {
        return 1;
    }
    else
    {
        return fib(n - 1) + fib(n - 2);
    }
}
```

In this case

`fib(n)` calls `fib(n - 1)`, `fib(n - 2)` and performs some computation

In general a recurrence that looks like

`f(n)` calls `f(n - 1)` and performs some computation

or

`f(n)` calls `f(n - 1)`, `f(n - 2)` and performs some computation

or

`f(n)` calls `f(n - 1)`, `f(n - 2)`, `f(n - 3)` and performs some computation

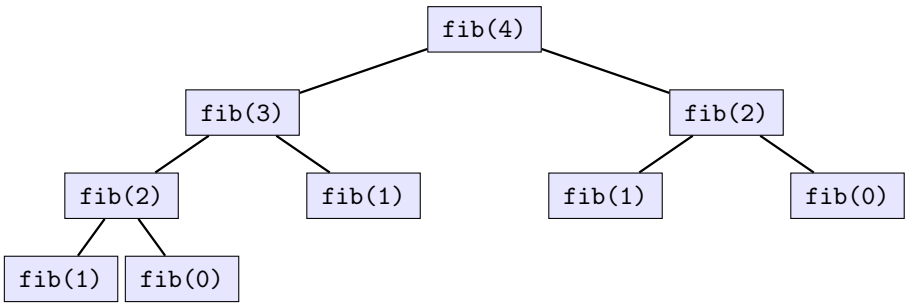
or

`f(n)` calls `f(n - 2)`, ..., `f(n - 5)` and performs some computation

etc. are called **linear recursion**.

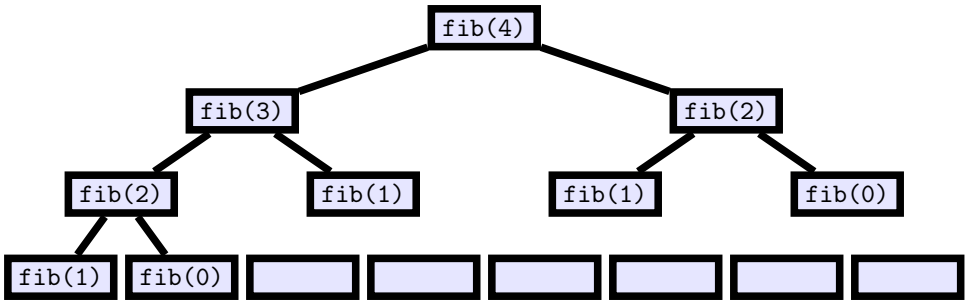
linear recursion

Here's the function call graph for `fib(4)`:



First let's think about the runtime. Watchout: for this section, the "runtime of `f(n)`" refers to the time spent in `f()` and *does not* include the runtime of the recursive `f(n - 1)` and `f(n - 2)`. For the *next* section, the "runtime of `f(n)`" *will* include the runtime of the recursive function calls `f(n - 1)` and `f(n - 2)`. The second method is more common since there are mathematical techniques to handle runtime computations for the second method.

Each `f(n)` takes constant time, whether it's the recursive case or the base cases. Remember I'm not including the time used in recursive function calls. Let A be the maximum time of `f(n)` (i.e., maximum of the time for recursive case and the base cases). Therefore the total time taken of `f(4)` is (the number of function calls) $\times A$. We're tempted to simplify the counting by adding some rectangles like this:



In this case, the time taken is $\leq (1 + 2 + 2^2 + 2^3) \times A = (2^4 - 1)A$. For `f(n)`, the time taken is $\leq (2^n - 1)A$, i.e., the runtime of the above algorithm is

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

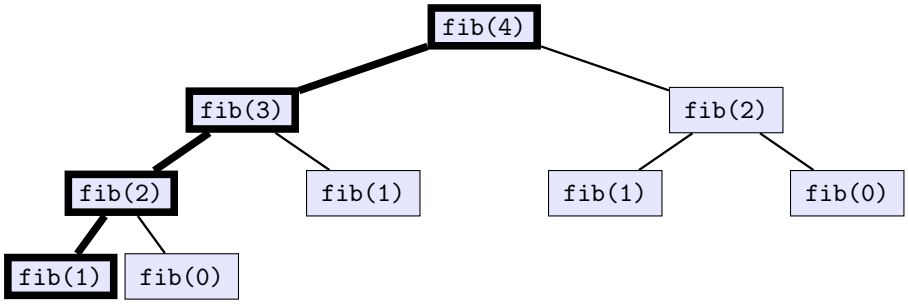
But is this a tight asymptotic upper bound? In the next section I'll show you more precisely that the runtime is

$$T(n) = O(\phi^n)$$

where $\phi = (1+\sqrt{5})/2 = 1.6180...$ is the so–called **golden ratio**. Note that if $c < 2$, then c^n is asymptotically strictly smaller than 2^n, i.e., $c^n = O(2^n)$ but $2^n \neq O(c^n)$.

golden ratio

As for the space complexity, each `f(n)` takes constant time, whether the recursive case or base case was executed. Let A be the maximum space usage of `f(n)`. Note that A is constant and does not depend on `n`. It's then clear that the space complexity is due to the *longest* path in the tree starting from the root:



i.e., it's $4A$. In general, the space complexity of `f(n)` is

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

Exercise 0.14.1. What is the runtime of this bubblesort, assuming all `std::vector< int >` have size n. This one uses recursion. Explain every step of your reasoning very clearly. (More on recursion later ...)

debug: exercises/bubblesort-2/question.tex

```
std::vector< int > swap(std::vector< int > x, int i, int j)
{
    int t = x[i];
    x[i] = x[j];
    x[j] = x[i];
    return x;
}

std::vector< int > bubblesort(std::vector< int > x,
                              int last_index = -1)
{
    if (last_index == -1)
    {
        last_index = x.size() - 1;
    }
```

```
    if (last_index == 0)
    {
        return x;
    }
    else
    {
        // Do one pass:
        for (int i = 0; i < last_index; ++i)
        {
            if (x[i] > x[i + 1])
            {
                x = swap(x, i, i + 1);
            }
        }
        // Recurse:
        x = bubblesort(x, last_index - 1);
        return x;
    }
}
```

What is the space complexity?

(Go to solution, page ??) □

0.15 Linear recursion debug: linear-recursion.tex

Recall that a recursive function is a function that calls itself. For instance here's Fibonacci again:

```
def fib(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Or in C/C++:

```
int fib(int n)
{
    if (n == 0 || n == 1):
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

Of course this is your standard first Fibonacci implementation. (Review your recursion notes from CISS245.)

The form of this function is derived from math. In your math classes, one would write this function as

$$f(n) = \begin{cases} 1 & \text{if } n = 0, 1 \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases}$$

In math, one would also say that this definition is piecewise ... because there are two pieces in the definition, right?

Each of the following cases

$$f(0) = 1, \quad f(1) = 1$$

is usually called a **base case**. The case that has recursion

base case

$$f(n) = f(n-1) + f(n-2), \quad n > 1$$

is called the **recursive case** (duh).

recursive case

How do you analyze the runtime of the above recursive function? There are

many types of recursion. The above recursion

$$f(n) = f(n-1) + f(n-2)$$

is an example of a **linear recursion** with fixed degree. In the Fibonacci case, the **degree** is 2 because the parameters in the above are

linear recursion

degree

$$n, \quad n-1, \quad n-2$$

and the largest difference between n and the other parameters is 2. In general a degree 2 linear recurrrence $g(n)$ looks like this:

$$g(n) = a \cdot g(n-1) + b \cdot g(n-2)$$

where a, b are constants. A degree 3 linear recurrrence $h(n)$ looks like this:

$$h(n) = a \cdot h(n-1) + b \cdot h(n-2) + c \cdot h(n-3)$$

where a, b, c are constants.

Here's what you do. First you let

$$T(n)$$

denote the total time to execute `f(n)` *including* the time due to the recursive calls. That includes the function call, the execution of the body (which can include the time to execute the base case code or the time due to the execution of the recursive calls) and the return.

```
def f(n):                                   t1
    if n == 0 or n == 1:                    t2
        return 1                            t3
    else:                                   t4
        return f(n - 1) + f(n - 2)          t5
```

If the value of `n` is `0`, the above function returns 1. Therefore

$$T(0) = t_1 + t_2 + t_3$$
$$T(1) = t_1 + t_2 + t_3$$

Since we're going to compute big-O, we're just going to write

$$T(0) = T(1) = A$$

for some constant A. Now for the recursive case ...

What if $n > 1$? Notice that your function call of `f(n)` will have to make two calls: a call to `f(n - 1)` and then another call to `f(n - 2)`. Of course the time taken to execute `f(n - 1)` is $T(n-1)$. And the time to execute `f(n - 2)` is $T(n-2)$. When you get the results of `f(n - 1)` and `f(n - 2)`, you have to add them together (which takes constant time) and return the result (which also takes constant time). Note the ... *VERY IMPORTANT POINT* ... here:

Time t_5 depends on `n` and therefore is *not* constant (with respect to `n`).

Let me do this carefully (and slowly) and include everything. You'll see later that most of the extra things/details to consider are actually $O(1)$, i.e. constant time, and therefore not that crucial ... but you have to see it to see what's happening. Here's the timing for t_5:

$$\begin{aligned}
t_5 &= [\text{time to compute } n-1] + [\text{time to execute } f(n-1)] \\
&\quad + [\text{time to compute } n-2] + [\text{time to execute } f(n-2)] \\
&\quad + [\text{time to add the return values of above}] \\
&\quad + [\text{time to return the sum}] \\
&= [\text{some constant}] + T(n-1) \\
&\quad + [\text{some constant}] + T(n-2) \\
&\quad + [\text{some constant}] \\
&\quad + [\text{some constant}] \\
&= T(n-1) + T(n-2) + B
\end{aligned}$$

where B is a constant. (There are other details. For instance time might be taken to store the return value of the $f(n-1)$ function call temporarily before making the next function call of $f(n-2)$. The time to store an integer value is constant and does not change with n. So again this takes constant time.)

So for the case of $n > 1$,

$$\begin{aligned}
T(n) &= t_1 + t_2 + t_4 + t_5 \\
&= t_1 + t_2 + t_4 + T(n-1) + T(n-2) + B \\
&= T(n-1) + T(n-2) + C
\end{aligned}$$

for some constant C.

So, voilà, all together we have

$$T(n) = \begin{cases} A & \text{if } n = 0, 1 \\ T(n-1) + T(n-2) + C & \text{otherwise} \end{cases}$$

Hey! That does not give a formula for $T(n)$ in terms of n ... but in terms of $T(n-1)$ and $T(n-2)$!!!

The above *recursive algorithm* gives a *recursive runtime formula*.

Or we say $T(n)$ satisfies a **recurrence relation**. Of course in analyzing runtimes, you prefer to put this $T(n)$ into one of the standard runtime big-O classes such as

recurrence relation

$$O(1), \quad O(n^a), \quad O(n^b \ln^c n), \quad O(c^n), ...$$

so that you can tell how fast or slow it runs. So I need to rewrite a recursive runtime formula into one that is not recursive (yes, it's possible). Such a formula is said to be a **closed form formula**.

closed form formula

0.16 Linear recursive runtime

debug: linear-recursive-runtime.tex

So you see that when you measure the runtime of an algorithm like our Fibonacci function implemented with linear recursion will give you a runtime function that looks rather similar. So the next step is to convert the runtime function to a *non-recursive* closed form.

Before I show you the closed form for the runtime of Fibonacci, let's just get a feel for the runtime functions by calculating one. It turns out that it's pretty bad.

Suppose I need to solve the problem of computing this function (it's somewhat like my Fibonacci). Note that $g(n)$ is defined by linear recursion.

$$g(n) = \begin{cases} 1 & \text{if } n = 0 \\ 3 & \text{if } n = 1 \\ 2g(n-1) + 3g(n-2) & \text{if } n \geq 2 \end{cases}$$

Suppose I choose to write my algorithm like this:

```
def g(n):
    if n == 0:
        return 1
    elif n == 1:
        return 3
    else:
        return 2 * g(n - 1) + 3 * g(n - 2)
```

You know that the runtime function would be like this:

$$T(n) = \begin{cases} A & \text{if } n = 0, 1 \\ T(n-1) + T(n-2) + B & \text{if } n > 1 \end{cases}$$

for some constants A and B. Later I'll show you how to compute an approximation (in the big-O sense) for $T(n)$. For now, suppose we compute $g(5)$ using the above program, compute like what a program would, executing one statement or one operation at a time. This is what you would see:

$$
\begin{aligned}
g(5) &= 2g(4) + 3g(3) \\
&= 2(2g(3) + 3g(2)) + 3g(3) \\
&= 2(2(2g(2) + 3g(1)) + 3g(2)) + 3g(3) \\
&= 2(2(2(2g(1) + 3g(0)) + 3g(1)) + 3g(2)) + 3g(3) \\
&= 2(2(2(2(3) + 3g(0)) + 3g(1)) + 3g(2)) + 3g(3) \\
&= 2(2(2(6 + 3g(0)) + 3g(1)) + 3g(2)) + 3g(3) \\
&= 2(2(2(6 + 3(1)) + 3g(1)) + 3g(2)) + 3g(3) \\
&= 2(2(2(6 + 3) + 3g(1)) + 3g(2)) + 3g(3) \\
&= 2(2(2(9) + 3g(1)) + 3g(2)) + 3g(3) \\
&= 2(2(18 + 3g(1)) + 3g(2)) + 3g(3) \\
&= 2(2(18 + 3(3)) + 3g(2)) + 3g(3) \\
&= 2(2(18 + 9) + 3g(2)) + 3g(3) \\
&= 2(2(27) + 3g(2)) + 3g(3) \\
&= 2(54 + 3g(2)) + 3g(3) && \text{will this ever end?!?} \\
&= 2(54 + 3(2g(1) + 3g(0))) + 3g(3) \\
&= 2(54 + 3(2(3) + 3g(0))) + 3g(3) \\
&= 2(54 + 3(6 + 3g(0))) + 3g(3) \\
&= 2(54 + 3(6 + 3(1))) + 3g(3) \\
&= 2(54 + 3(6 + 3)) + 3g(3) \\
&= 2(54 + 3(9)) + 3g(3) \\
&= 2(54 + 27) + 3g(3) \\
&= 2(81) + 3g(3) \\
&= 162 + 3g(3) \\
&= 162 + 3(2g(2) + 3g(1)) && \text{... oh no ... here we go again ...} \\
&= 162 + 3(2(2g(1) + 3g(0)) + 3g(1)) \\
&= 162 + 3(2(2(3) + 3g(0)) + 3g(1)) \\
&= 162 + 3(2(6 + 3g(0)) + 3g(1)) \\
&= 162 + 3(2(6 + 3(1)) + 3g(1)) \\
&= 162 + 3(2(6 + 3) + 3g(1)) \\
&= 162 + 3(2(9) + 3g(1)) \\
&= 162 + 3(18 + 3g(1)) \\
&= 162 + 3(18 + 3(3)) \\
&= 162 + 3(18 + 9) \\
&= 162 + 3(27) \\
&= 162 + 81 \\
&= 243 && \text{PHEW!!!}
\end{aligned}
$$

Exercise 0.16.1. Show that if `g(n)` is the following

```
def g(n):
    if n == 0:
        return 1
    elif n == 1:
        return 3
    else:
        return 2 * g(n - 1) + 3 * g(n - 2)
```

Then the runtime is $T(n) = O(2^n)$. (This is not the best big-O.) What is the space complexity?

Exercise 0.16.2. Let

```
def h(n):
    if n == 0:
        return 1
    elif n == 1:
        return 2
    elif n == 2:
        return 42
    else:
        return 7 * h(n - 1) - 3 * h(n - 2) + 4 * h(n - 3)
```

What is a big-O of $T(n)$ for `h(n)`? What is the space complexity?

So the execution of $g(5)$ using the above algorithm seems to indicate that $T(5)$ is huge. You can see even from this simple simulation that the main problem is that many function calls were actually repeats. For instance go ahead and count the number of times $g(2)$ was executed in the above.

It turns out that except for recursion like this:

```
def h(n):
    if n == 0:
        return 42
    else:
        return 7 * h(n - 1) + n
```

where in the recursive part only *one* recursive function call was made, most cases will usually be extremely bad, as in exponential bad.

It turns out that in general if you have a numeric recursive function like

$$T(n) = aT(n-1) + bT(n-2) + c$$

where a, b, and c are constants, you can always compute a nice formula for $T(n)$. A "nice" formula for $T(n)$ that does *not* depend on $T(i)$ for smaller i but only on n, i.e., you can compute a closed form formula for $T(n)$.

In fact more generally there are extremely powerful tools to compute an *exact* closed form for more complicated recursion such as

$$T(n) = n^2T(n-1) + (1+n)T(n-2) + \frac{2}{3}n^2 - 1$$

But I'll stick to the simple case of

$$T(n) = aT(n-1) + bT(n-2) + c$$

where a, b, c are constants.

Although there are techniques to compute exact closed forms for $T(n)$, remember that we only need the big-O of $T(n)$.

For the case of a recursive function such as

```
def g(n):
    if n == 0:
        return 1
    elif n == 1:
        return 3
    else:
        return 2 * g(n - 1) + 3 * g(n - 2)
```

which has a runtime $T(n)$ satisfying

$$T(n) = T(n-1) + T(n-2) + A$$

where A is a constant, a rough approximation (using the method in an earlier section) the runtime is

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

i.e., exponential. Which means the runtime is bad. How bad? What exactly is "exponential bad"? You can use the fibonacci function as an example:

```
def fib(n):
    if n == 0:
        return 1
```

```
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

and print `fib(n)` for `n = 0, 1, 2, ..., 100`. Remember we did this before in CISS245. It will grind to a halt before hitting `60`.

In the above, instead of $O(2^n)$, we can be more precise.

First, you ignore the constant c in

$$T(n) = aT(n-1) + bT(n-2) + c \tag{1}$$

and look at this recursion instead:

$$T^{(h)}(n) = aT^{(h)}(n-1) + bT^{(h)}(n-2) \tag{2}$$

homogeneous part

This is called the **homogeneous part** of $T(n)$. You need to know that $T^{(h)}(n)$ must have a solution roughly of the form

$$T^{(h)}(n) = r^n$$

where r is a constant. (Take this on faith for now – the justification will be provided in MATH325.) The immediate goal now is to compute r. You substitute this into (2) to get

$$r^n = ar^{n-1} + br^{n-2}$$

You cross out r^{n-2} to get

$$r^2 = ar + b$$

i.e., a quadratic equation. This then allows you to solve for r.

Let's try this technique on our

$$T(n) = T(n-1) + T(n-2) + c$$

The homogeneous part of this equation is

$$T^{(h)}(n) = T^{(h)}(n-1) + T^{(h)}(n-2)$$

Let $T^{(h)}(n) = r^n$ and substitute it into

$$T^{(h)}(n) = T^{(h)}(n-1) + T^{(h)}(n-2)$$

to get

$$r^n = r^{n-1} + r^{n-2}$$

Cross out r^{n-2} to get

$$r^2 = r + 1$$

which gives us the quadratic

$$r^2 - r - 1 = 0$$

Using the quadratic polynomial root formula you get

$$r = \frac{-(-1) \pm \sqrt{(-1)^2 - 4(1)(-1)}}{2}$$

which gives us

$$r = \frac{1 \pm \sqrt{5}}{2}$$

Let

$$r_1 = \frac{1 + \sqrt{5}}{2}$$

(... this is the one that is about 1.618) and

$$r_2 = \frac{1 - \sqrt{5}}{2}$$

Then you know that if the recurrence relation on $T^{(h)}(n)$ is

$$T^{(h)}(n) = T^{(h)}(n-1) + T^{(h)}(n-2)$$

then $T^{(h)}(n)$ must have the form

$$T^{(h)}(n) = \alpha \cdot r_1^n + \beta \cdot r_2^n$$

where α, β are constants. This is the closed form for $T^{(h)}(n)$, not $T(n)$. Now, $|r_2| = 0.618...$ whereas $r_1 = 1.618....$ Since $r_1 > |r_2|$, I get

$$T^{(h)}(n) = \alpha \cdot r_1^n + \beta \cdot r_2^n = O(r_1^n)$$

But what about $T(n)$? Remember that

$$T(n) = T^{(h)}(n) + c$$

Since $T^{(h)}(n) = O(r_1^n)$ and $r_1 > 1$, $T^{(h)}(n)$ is going to climb way faster than the constant c. Therefore the big-O of $T(n)$ is governed by the big-O of $T^{(h)}(n)$

and not c. Hence

$$T(n) = O\left(T^{(h)}(n) + c\right) = O\left(T^{(h)}(n)\right) = O(r_1^n)$$

which means that the algorithm has exponential runtime with base $r_1 > 1$. Note that base of the exponential runtime is 1.618... instead of 2. 2^n climbs much faster than 1.618^n. That's because

$$\frac{2^n}{1.618^n} = 1.236...^n$$

and $1.236...^n$ grows toward infinity as n grows unboundedly since $1.236 > 1$. So the asymptotic

$$T(n) = O(r_1^n)$$

is way more accurate than

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

Note that at the stage where you solve for r in the quadratic equation in r:

$$r^2 = ar + b \qquad (*)$$

in the general case, you will have *three* cases: $(*)$ has

(A) Two distinct real roots
(B) Two distinct complex (and nonreal) roots
(C) Two roots with the same real value

The Fibonacci case of

$$r^2 = r + 1$$

gives us two distinct real roots.

For Case (A) and Case (B) above, the $T(n)$ is just

$$T(n) = \alpha r_1^n + \beta r_2^n$$

(For Case (B) you would have to use complex numbers – obviously you would need to know complex numbers for such cases.) For Case (C) where there is one value r_1 for both roots, the closed form becomes

$$T(n) = \alpha r_1^n + \beta n r_1^n$$

so in this case

$$T(n) = O(nr_1^n)$$

Of course as for the specific big-O class, you have to compute the exact root values.

[EXERCISES]

0.17 Speeding up linear recursion debug: fast-linear-recursion.tex

Recall that if you have a runtime that is like this:

$$T(n) = \begin{cases} A & \text{if } n = 0, 1 \\ T(n-1) + T(n-2) + B & \text{otherwise} \end{cases}$$

which is the case for the Fibonacci function:

```
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

then the runtime is exponential:

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

When you look at the computation of the degree 2 linear recurrence computation-by-hand in a previous section, you see that the problem is the repeated computations of `fib(i)` (`i = n - 2, ...`) when you compute `fib(n)`.

One way to prevent re-computation is to ... save your work!!! So let's say we have an array `table` of size 1000 to store `fib(0)`, `fib(1)`, ..., `fib(999)`. Initially, the table is empty. To indicate that `table[i]` is "empty", I'll set it to -1 since the Fibonacci numbers are positive.

Then, whenever I compute `fib(n)`, I first check if it's already in the `table`. If it is, I'll use the number. If it's not, I'll compute `fib(n)`, save it in the table, then return it. Now the function looks like this:

```
for i = 0, ..., 999:
    table[i] = -1

def fib(n):
    if n < 2:
        return 1
    else:
        if table[n] is -1:
            table[n] = fib(n - 1) + fib(n - 2)
        return table[n]
```

This is not new. You have already seen this technique in CISS245. The

concept of storing computations in a table to avoid re-computation is extremely important and appears in a huge number of algorithms (not just fibonacci computations).

Exercise 0.17.1. Assuming your `table` is large enough to store all computation of `fib(0)`, `fib(1)`, `fib(2)`, `fib(3)`, ..., `fib(n)`, compute the runtime of the above implementation. □

Of course it's possible that you want $f(10000000)$. You can either increase the size of your `table` or simply use the `table` only when `n` is less than 1000.

Exercise 0.17.2. Modify the above problem to use the `table` only when `n` is less than 1000.

Notice that the above method requires storage to save computations in order to prevent re-computation. There's another way to compute the n–th Fibonacci without too much storage space. Here's the idea: Instead of compute "top-to-bottom" (i.e., going from n to 1 and 0) you start at 0 and 1 and compute up to n. Here's an execution of $f(6)$. First of all, the Fibonacci numbers up to the 6–th is this:

```
1, 1, 2, 3, 5, 8, 13
```

You need only 3 variables. I'll call them a, b, c. First you set a, b, c to these values:

```
1, 1, 2, 3, 5, 8, 13
a  b  c
```

In other words $c = a + b$ after initializing a and b to 1. Next you want to perform some computation to get this:

```
1, 1, 2, 3, 5, 8, 13
   a  b  c
```

Next you want to perform some computation to get this:

```
1, 1, 2, 3, 5, 8, 13
      a  b  c
```

and then this:

```
1, 1, 2, 3, 5, 8, 13
         a  b  c
```

and finally this:

```
1, 1, 2, 3, 5, 8, 13
            a  b  c
```

Basically `a` plays the role of `f(i - 2)`, `b` plays the role of `f(i - 1)`, and `c` plays the role of `f(i)`. You run your `i` from `i = 2` to `i = n`.

I'm going to call this the **bottom-up** technique. You can use a for-loop or recursion to implement a bottom-up algorithm.

bottom-up

Exercise 0.17.3. Implement this new Fibonacci function (a) using a for-loop and (b) using recursion. Test them. Compute the runtime. □

Exercise 0.17.4. The following is a linear recursion:

```
def f(n):
    if n == 0:
        return 1
    elif n == 1:
        return 2
    else:
        return f(n - 1) + 3 * f(n - 2) + 1
```

Run this and compute `f(0)`, `f(1)`, ..., `f(50)`. Notice that it slows down tremendously. Compute the runtime of this implementation. Re-implement this using the bottom-to-top technique (first using a for-loop and next using recursion.) □

Exercise 0.17.5. The following is a linear recursion:

```
def f(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    elif n == 2:
        return 2
    else:
        return f(n - 1) + 2*f(n - 2) + 3*f(n - 3)
```

Run this and compute $f(0)$, $f(1)$, ..., $f(50)$. Notice that it slows down tremendously. Compute the runtime of this implementation. Re-implement this using the bottom-to-top technique (first using a for-loop and next using recursion.) □

0.18 Divide-and-conquer algorithms

debug: divide-and-conquer.tex

Another type of recursive runtime function looks like this:

$$T(n) = T(n/2) + A$$

Such a recursion is called a **divide-and-conquer recursion**. These are very different from linear recursion. An algorithm whose runtime is a divide-and-conquer recursive runtime function is called ... drumroll ... **divide-and-conquer algorithm** ... (duh).

divide-and-conquer recursion

The binary search algorithm (see CISS240 notes) is an algorithm with a runtime of

$$T(n) = \begin{cases} A & \text{if } n = 0 \\ T(n/2) + B & \text{otherwise} \end{cases}$$

More generally a divide-and-conquer runtime function looks like this:

$$T(n) = \begin{cases} A & \text{for } n = 0, 1, 2, ..., B \\ aT(n/b) + c(n) & \text{otherwise} \end{cases}$$

for constants A, B, a, b. Here $c(n)$ is a function of n (it can be a constant of course).

Mergesort (a sorting algorithm) has a runtime of

$$T(n) = \begin{cases} A & \text{if } n = 0, 1 \\ 2T(n/2) + Bn + C & \text{otherwise} \end{cases}$$

Before analyzing some divide-and-conquer algorithms, let me give a high-level overview of such animals. Divide-and-conquer algorithms are easy to spot.

Let's start with the binary search. (Again see CISS240 notes.) Suppose you're given a sorted (ascending) array of $n = 10$ integers:

$$\texttt{x} = \{\texttt{1, 3, 4, 6, 7, 8, 10, 12, 13, 17}\}$$

The goal is to find `target` in x, in the sense that the algorithm should compute the index where the value of `target` occurs in x. If that value is not found, then the algorithm returns -1.

Suppose `target` = 7. The idea behind the binary search is to look at `target`

(or rather the value of `target`) in the middle of x. The middle index is

$$\texttt{mid} = n/2 = 10/2 = 5$$

Since

$$\texttt{x[mid]} = 8 > \texttt{target}$$

and since `x` is sorted, `target` must be in the left half of x (if at all), before index `mid`. Therefore I'm going to hunt for `target` in

$$\texttt{x[0..4]} = \{\texttt{1, 3, 4, 6, 7}\}$$

Note that the size of this new array is $n = 5$, half of the original. When I repeat the above I get

$$\texttt{mid} = n/2 = 5/2 = 2$$

Since

$$\texttt{x[mid]} = \texttt{x[2]} = 4 < \texttt{target}$$

the `target` must be in the right half of x, i.e., past `mid`. The new array is

$$\texttt{x[3..4]} = \{\texttt{6, 7}\}$$

which is about half of `x[0..4]`.

Note that the size of the array looks like this:

$$10 \rightarrow 5 \rightarrow 2$$

Each time, after looking at `x[mid]`, you cut down the array by half: you work on the left half or the right half of `x`.

Stepping back, you see that the size of work (approximately) halves each time.

The above is the main idea behind the binary search. However note that it's a waste of time to create new arrays each time we make a recursive call since the array x is *not modified*. The correct thing to do is to use the same x, i.e., pass x by reference and also pass two index variable `left` and `right` to tell the function where's the starting index and ending index of the current search. Initially of course `left` $= 0$ and `right` $= n - 1$ where n is the size of the array x.

See the section on the runtime analysis of the binary search.

Exercise 0.18.1. You are given the following recursive function: debug: exercises/recursion-0/question.tex

$$f(n) = \begin{cases} 2 & \text{if n} = 0 \\ 3f(n-1)+5 & \text{otherwise} \end{cases}$$

Compute $f(7)$ performing your computation like a computer, evaluating one operation of performing only one substitution at a time. (Go to solution, page **??**) □

Exercise 0.18.2. You are given the following recursive function: debug: exercises/recursion-1/question.tex

$$f(n) = \begin{cases} 2 & \text{if n} = 0 \\ 3f(n/2)+5 & \text{otherwise} \end{cases}$$

Note that "$n/2$ " really meant the floor of $n/2$ so technically I should really write

$$f(n) = \begin{cases} 2 & \text{if n} = 0 \\ 3f(\lfloor n/2 \rfloor)+5 & \text{otherwise} \end{cases}$$

Compute $f(7)$ performing your computation like a computer, evaluating one operation of performing only one substitution at a time. (Recall that floor $\lfloor x \rfloor$ means the integer before x and ceiling $\lceil x \rceil$ means the integer after x. For instance $\lfloor 3.4 \rfloor = 3$ and $\lceil 3.4 \rceil = 4$.) (Go to solution, page **??**) □

Now let's look at a high level overview of mergesort. It's easier to do an example by hand. The main idea when you call mergesort with an array x, is that the function splits x into two arrays of about equal size, say we call them y and z. Next, we call mergesort on y and save the result in say a; this a will be sorted. Third, we call mergesort on z and save the result in say b; this b is sorted. Fourth, we merge the sorted a and b into c and return c. I'll talk about the merge operation in a bit. In mergesort, if the list x has size 0 or 1, you just return x; this is the base case.

Now for the merge operation. If you have two sorted arrays a and b and you want to merge them into c, you just scan a and b left-to-right, picking the smallest to be placed into another array. Briefly, you put one finger at index 0 of a and another at index 0 of b. You then look at what you're pointing to, pick the smaller and dump it to array c, moving your finger past the smaller value. That's it.

For instance if $a = [2, 4, 5]$ and $b = [1, 2, 3]$, you look at $a[0]$ and $b[0]$. Since

$b[0]$ is smaller, you put $b[0]$ into $c[0]$. So $c[0] = 1$. $b[0]$ is done and you look at $b[1]$. We now compare $a[0]$ and $b[1]$. They are the same. I'll just choose $a[0]$ and put that into $c[1]$. So now $c[0] = 1, c[1] = 2$. We now move now to the next element in a, i.e., $a[1]$. We compare $a[1]$ and $b[1]$. Since $b[1]$ is smaller, I put that into $c[2]$. I now compare $a[1]$ and $b[2]$. Since $b[2]$ is smaller, I put that into $c[3]$. Once an array is finished (in this case b), I just put the rest of what remains in a into c.

Exercise 0.18.3. Write a merge function. For instance `merge(u, v, w)` where `u`, `v`, `w` are `std::vector< int >` objects, and `u,v` are sorted, and you want to merge `u,v` into `w`. This means that `w` has all the values in `u` and `v` and `w` is sorted. (Go to solution, page **??**) □

debug: exercises/merge/question.tex

Suppose you're given this array:

$$x = [3, 6, 4, 2, 7, 8, 12, 15, 1, 5]$$

Let me call mergesort:

```
STEP1. mergesort(x = [3, 6, 4, 2, 7, 8, 12, 15, 1, 5])
First I divide x into two halves y and z and call mergesort(y).
y = [3, 6, 4, 2, 7]
z = [8, 12, 15, 1, 5]
a = mergesort(y) ... WAITING FOR RETURN VALUE FROM STEP2
b = ?
c = ?
```

```
STEP2. mergesort(x = [3, 6, 4, 2, 7])
I repeat the process on $x = [3, 6, 4, 2, 7]$.
y = [3, 6]
z = [4, 2, 7]
a = mergesort(y) ... WAITING FOR RETURN VALUE FROM STEP3
b = ?
c = ?
```

```
STEP3. mergesort(x = [3, 6])
I repeat the process on $[3, 6]$.
y = [3]
z = [6]
a = mergesort(y) ... WAITING FOR RETURN VALUE FROM STEP4
b = ?
c = ?
```

```
STEP4. mergesort(x = [3])
```

```
Since the size of x is 1, we are in the base case.
So I return [3] back to STEP3.
```

```
STEP3. mergesort(x=[3,6]) ... returning from STEP 4 ...
The result of mergesort(y) is stored in a and we call mergesort(z)
y = [3]
z = [6]
a = [3]
b = mergesort(z) ... WAITING FOR RETURN FROM STEP5 ...
c = ?
```

```
STEP5. mergesort(x = [6])
Since the size of x is 1, we are in the base case.
So I return [6] back to STEP3.
```

```
STEP3. mergesort(x=[3,6]) ... returning from STEP 5 ...
The result of mergesort(z) is stored in b and we merge a, b and store in c
y = [3]
z = [6]
a = [3]
b = [6]
c = [3, 6]
Return c to STEP2
```

```
STEP2. mergesort(x = [3, 6, 4, 2, 7])
The return value from STEP3 is stored in a. Call mergesort(z)
y = [3, 6]
z = [4, 2, 7]
a = [3, 6]
b = mergesort(z) ... WAITING FOR RESULT FROM STEP6 ...
c = ?
```

```
STEP6. mergesort(x = [4, 2, 7])
y = [4]
z = [2, 7]
a = mergesort(y) ... WAITING FOR RESULT FROM STEP7
b = ?
c = ?
```

```
STEP7. mergesort([4])
Return [4] back to STEP6.
```

```
STEP6. mergesort(x = [4, 2, 7])
y = [4]
z = [2, 7]
a = [4]
b = mergesort(z) ... WAITING FOR RESULT FROM STEP8 ...
```

```
c = ?
```

```
STEP8. mergesort(x = [2, 7])
y = [2]
z = [7]
a = mergesort(y) ... WAITING FOR RESULT FROM STEP9
b = ?
c = ?
```

```
STEP9. mergesort(x = [2])
Return [2] back to STEP8.
```

```
STEP8. mergesort(x = [2, 7])
y = [2]
z = [7]
a = [2]
b = mergesort(z) ... WAITING FOR RESULT FROM STEP10 ...
c = ?
```

```
STEP10. mergesort(x = [7])
Return [7] back to STEP8.
```

```
STEP8. mergesort(x = [2, 7])
y = [2]
z = [7]
a = [2]
b = [7]
c = [2, 7]
Return c to STEP6
```

```
STEP6. mergesort(x = [4, 2, 7])
y = [4]
z = [2, 7]
a = [4]
b = [2, 7]
c = [2, 4, 7]
Return c to STEP2
```

```
STEP2. mergesort(x = [3, 6, 4, 2, 7])
The return value from STEP3 is stored in a. Call mergesort(z)
y = [3, 6]
z = [4, 2, 7]
a = [3, 6]
b = [2, 4, 7]
c = [2, 3, 4, 6, 7]
Return c to STEP1
```

```
STEP1. mergesort(x = [3, 6, 4, 2, 7, 8, 12, 15, 1, 5])
First I divide x into two halves y and z and call mergesort(y).
y = [3, 6, 4, 2, 7]
z = [8, 12, 15, 1, 5]
a = [2, 3, 4, 6, 7]
b = mergesort(z) ... WAITING FOR RESULT FROM STEP11 ...
c = ?
```

Exercise 0.18.4. Finish the above execution by hand. □

Informally, you see that if $T(n)$ is the time to execute mergesort with an array of size n, the function call will split the array into two pieces (this requires linear time) then call mergesort with these two halves (this requires $2T(n/2)$) and then merge the two return results (this requires linear time). This results in a runtime of the form

$$T(n) = 2T(n/2) + An + B$$

for some constants A, B.

Again, just like binary search, there are ways to prevent the over-creation of new arrays. Therefore memory usage can be improved. But the above is the main idea behind mergesort.

See later section on mergesort.

As you can see, both binary search and mergesort involves splitting an array into two halves.

- In the case of binary search, the algorithm determines which half to continue with the search (the other is then thrown away), resulting in a runtime that looks like $T(n) = T(n/2) + A$.
- In the case of mergesort, the mergesort then recursively work on each half (both halves are used), resulting in a runtime that looks like $T(n) = 2T(n/2) + An + B$.

For binary search, of course if you have an array of size n, then in the next iteration of binary search, excluding the value at the `mid` index, there are `n - 1` values to analyze and the left and right subarray would have sizes of either $\lfloor (n-1)/2 \rfloor$ or $\lceil (n-1)/2 \rceil$. But asymptotically when `n` is large, it's

enough to replace these by $n/2$. So instead of saying

$$T(n) = T(\lfloor (n-1)/2 \rfloor) + A$$

or

$$T(n) = T(\lceil (n-1)/2 \rceil) + A$$

we just write the recurrence relation of $T(n)$ as

$$T(n) = T(n/2) + A$$

This is the same for the recurrence relation for mergesort.

In general divide-and-conquer algorithms involve splitting up the work into roughly equal smaller pieces, recursively performing the algorithm on the smaller pieces and frequently combining the work done on the smaller pieces. The work done is either performed on data that is sent into the functions using pass-by-reference or through the function return values.

Exercise 0.18.5. If instead of splitting up an array to 2 pieces in mergesort, what if I split up the array into 3, call mergesort on the 3 pieces and then merge the three sorted pieces. What do you think the runtime is going to look like? (Go to solution, page **??**) □

debug: exercises/mergesort-3/question.tex

Exercise 0.18.6. Here's our simple sum to n algorithm:

debug: exercises/dac-sum/question.tex

```
def sum(n):
    s = 0
    for i = 1, 2, 3, ..., n:
        s = s + i
    return s
```

Let's do it in a different way. Here's another function that sums from one point to another:

```
def sum2(m, n):
    s = 0
    for i = m, m + 1, m + 2, ..., n:
        s = s + i
    return s
```

Design and write a sum to n function that works recursively like this: Instead

of summing 1 to `n` in a loop, the function calls `sum2` to sum from `1` to `n/2`, calls `sum2` to sum from `n/2 + 1` to `n`, and finally returns the sum of the two return values. The base case is when `n` is `1`. Test your code. What do you think is the runtime? (Go to solution, page **??**) □

Exercise 0.18.7. Using the same idea as above, write a `sum` function that accepts an array (of integers, say) and recursively splits the array into two, calls the sum function, and return the sum of the return values. The base case is when the array is empty in which case you return 0 or when the array has one value in which case you return that value. Test your code. What do you think is the runtime? (Go to solution, page **??**) □

debug: exercises/dac-sum-array/question.tex

Exercise 0.18.8. Write a recursive `max` that takes an array, splits the array into two pieces, calls max on the pieces, and returns the max of the two return value. I'll let you figure out the base case scenario and what to do. (Go to solution, page **??**) □

debug: exercises/dac-max/question.tex

Exercise 0.18.9. Write a `count` function that accepts an array `x` and a value `v` and returns the numbers of times the value of `v` occurs in `x`. Use the divide-and-conquer strategy. (Go to solution, page **??**) □

debug: exercises/dac-count/question.tex

0.19 Master theorem debug: master-theorem.tex

So far I have been talking about recurrences like

$$T(n) = c_1 T(n-1) + c_2 T(n-2) + f(n)$$

where c_i are constants and $f(n)$ is an expressions in n. More generally,

$$T(n) = c_1(n) T(n-1) + c_2(n) T(n-2) + f(n)$$

where $c_i(n)$ are also expressions in n. Then there are the divide-and-conquer type recurrence relations. Here's one:

$$T(n) = T(n/2) + A$$

and here's another

$$T(n) = 2T(n/2) + An + B$$

There are also more complex ones like

$$T(n) = f(n) + \frac{1}{n} \sum_{i=0}^{n-1} T(i)$$

For the divide-and-conquer runtime recurrence, more generally, we are interested in recurrences like this:

$$T(n) = a \cdot T(n/b) + f(n)$$

where a and b are constants. For CISS350, the two cases you must be familiar with are

Theorem 0.19.1.

(a) *If*

$$T(n) = T(n/2) + A$$

then

$$T(n) = O(\lg n)$$

(b) *If*

$$T(n) = 2T(n/2) + An + B$$

then

$$T(n) = O(n \lg n)$$

For a more general theorem, see the Master Theorem.

Example 0.19.1. For instance let's consider the time complexity of the binary search.

```
def binarysearch(x, lower, upper, target):
    if lower > upper: return None
    else:
        mid = (lower + upper) / 2
        if x[mid] == target:
            return i
        elif x[mid] < target:
            return binarysearch(x, mid + 1, upper, target)
        else:
            return binarysearch(x, lower, mid - 1, target)
```

If $T(n)$ is the runtime to perform binarysearch on an array of size n (sorted of course), then

$$T(n) = T(n/2) + A$$

Note that to be really precise, the array `x[0..n-1]` is cut up into *three* parts: `x[0..mid - 1]`, `x[mid]`, `x[m + 1..n - 1]`. But note that `mid` is roughly `n/2` and so `mid` ± 1 is roughly $n/2$. □

Example 0.19.2. What about mergesort? The list continually split into two equal parts to be processed by mergesort. On return from mergesort, the two sublists are combined.

```
MERGESORT
INPUT: x: an array
       start, end: index values

if start >= end:
    return
else:
    mid = (start + end) / 2
    MERGESORT(x, start, mid)
    MERGESORT(x, mid + 1, end)
    MERGE(x, start, mid, mid+1, end)
```

The runtime of MERGE is $O(n)$. The runtime for the computation of `mid` is constant. So the runtime for MERGE and the computation of `mid` is of the

form $An + B$. Therefore

$$T(n) = 2T(n/2) + An + B$$

Here's the Master Theorem which covers the two cases mentioned above:

Theorem 0.19.2. (Master theorem) *Let $T : \mathbb{N} \to \mathbb{R}$ be a function such that*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $a \geq 1$ and $b > 1$.

(a) If $f(n) = O(n^c)$ where $c < \log_b a$, then

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

(b) If $f(n) = \Theta(n^c \log^k n)$ where $c = \log_b a$, then

$$T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$$

(c) If $f(n) = \Omega(n^c)$ where $c > \log_b a$ and there is a constant $k < 1$ such that

$$af(n/b) \leq kf(n)$$

for sufficiently large n, then

$$T(n) = \Theta(f(n))$$

Example 0.19.3. For binary search, if we let $T(n)$ be the runtime, then

$$T(n) = T(n/2) + A$$

$$a = 1, \;\; b = 2, \;\; f(n) = O(1) = O(n^0), \;\; c = 0$$

Note that

$$\log_b a = \log_2 1 = 0$$

and

$$c \not< \log_b a$$

So case 1 of the master theorem does not apply. Yikes. Now what?

If you look at the second case, you see that

$$0 = c = \log_b a$$

Now note that $f(n)$ is in fact $\Theta(1)$, i.e., $f(n) = \Theta(n^c \log^k n)$ for $c = 0$ and $k = 0$. Therefore,using case (b),

$$T(n) = \Theta(n^0 \log^{0+1} n) = \Theta(\lg n)$$

□

□

Example 0.19.4. For mergesort

$$a = 2, \quad b = 2, \quad f(n) = \Theta(n) = \Theta(n^c), c = 1$$

Note that

$$\log_b a = \log_2 2 = 1$$

and

$$c = \log_b a$$

So case (b) of Master Theorem applies with $k = 0$. This gives us

$$T(n) = \Theta(n \log n)$$

□

Example 0.19.5. Let $T(n) = 2T(n/2) + \Theta(n^2)$. Then $a = 2 = b$ and $\log_b a = 1$. Using (c) of Master Theorem, with $c = 2 > 1 = \log_b a$. With $f(n) = n^2$,

$$af(n/b) = 2(n/2)^2 = n^2/2 \leq kn^2 = kf(n)$$

for $k = 1/2 < 1$. Then $T(n) = \Theta(n^2)$. □

Exercise 0.19.1. Let

debug: exercises/master-theorem-0/question.tex

$$T(n) = 2T(n/2) + \Theta(1)$$

Find k such that $T(n) = \Theta(n^k)$. (Go to solution, page **??**) □

Exercise 0.19.2. We have big-O concept for $a(n)$. Suppose $a(n) = O(b(n))$. Let

$$f(x) = \sum_{n=0}^{\infty} a(n)x^n$$

and

$$g(x) = \sum_{n=0}^{\infty} b(n)x^n$$

What can we say about the relationship between $f(x)$ and $g(x)$ and vice versa? Is it true that

$$a(n) = O(b(n)) \iff f(x) = O(g(x))$$

□

Index