Computer Science

Dr. Y. Liow (August 1, 2020)

Contents

Chapter 102

Asymptotics and Algorithmic Runtime Analysis

File: chap.tex

File: introduction.tex

102.1 Introduction

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

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

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

Exercise 102.1.1. Use the above functions and compute `fib(n)` and `factorial(n)` and for `n` = 0, 1, 2, 3, 4, 5, 6. □

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".

File: timing.tex

102.2 Timing

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

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

102.2.1 First method

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

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

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

    return 0;
}
```

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

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

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

    return 0;
}
```

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

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

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

102.2.2 Second method

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

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


int main()
{
    rusage start, end;

    getrusage(RUSAGE_SELF, &start);

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

    getrusage(RUSAGE_SELF, &end);

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

    return 0;
}
```

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

```
#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;
}
```

File: what-is-asymptotic-analysis.tex

102.3 What is Asymptotic Analysis?

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 say

$$f(n) \text{ is roughly } \leq \text{ to } g(n) \text{ for large values of } n$$

or

$$f(n) \text{ is roughly } \geq \text{ to } g(n) \text{ for large values of } n$$

or

$$f(n) \text{ is roughly } = \text{ to } g(n) \text{ for large values of } n$$

I'll get more specific later. The big picture to keep in mind is that it's a way to compare functions.

I'm going to use this tool to measure various things about algorithms, including their runtime and resource usage such as memory usage.

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

102.4 Algorithmic Analysis: How Fast is an Algorithm?

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

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

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

This algorithms solves the following problem:

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

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

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

and

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

Our algorithm:

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

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

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

Different people design different algorithms to solve the same problem.

We are interested in measuring how fast an algorithm runs so that we can pick the best. It's clear that the crucial thing to focus on is the performance of an algorithm when n is large. After all, the sum from 1 to 3 is easy – in fact we can do $1 + 2 + 3$ in our heads and not bother with running a program at all, right? It takes more time just to let your computer boot up or wake up!!!

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

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:
```

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

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

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

```
          s = 0                       time 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 the header and body):

```
LOOP:     if i > n:                   time t3
              goto ENDLOOP            time 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 that times taken to execute each of the above statement, t_1, t_2, ... are *constants with respect to* n. What this mumbo-jumbo meant was, whether

I run the above algorithm with $n = 10$ or $n = 1000000$, the time taken this:

```
        s = 0                       time t1
```

is still t_1. Make sense, right?

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

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

For instance

```
        s = 0                       time t1 (once)
```

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

```
LOOP:   if i > n:                   time t3 (n + 1 times)
            goto ENDLOOP            time t4 (once)
```

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

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

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:

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

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 the time taken to carry out the algorithm is roughly the function 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. 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).

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:

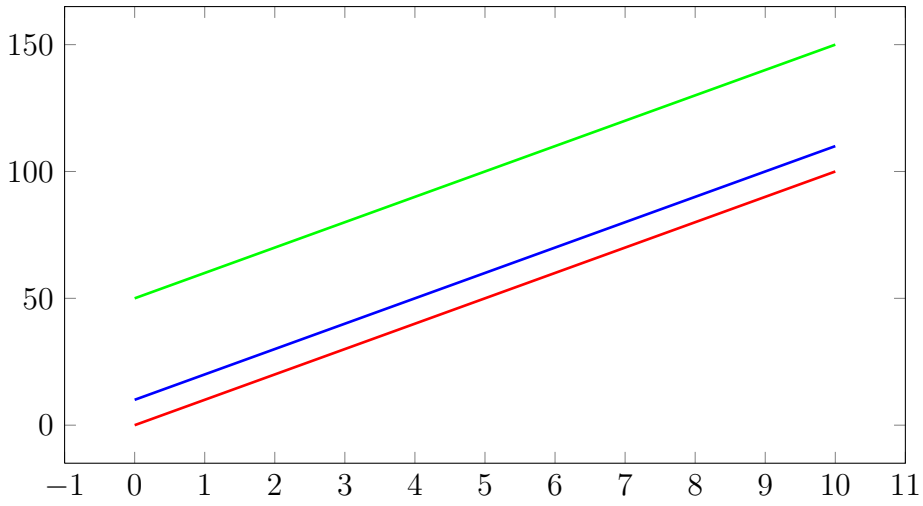
$$T(n) = O(n)$$

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

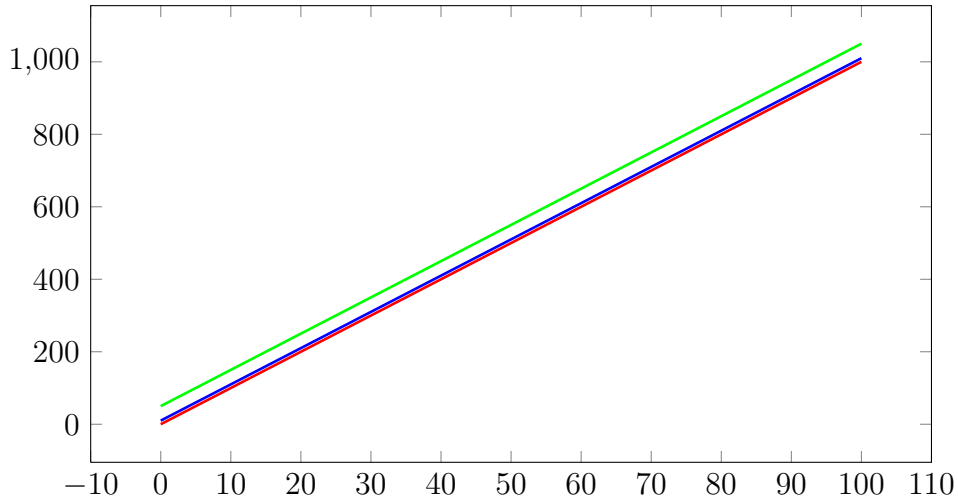
- You only keep the term that grows the fastest because in the long run (i.e. for large n), the growth of $f(n)$ is determined by this term. Typically, your runtime function might have more than 2 terms.
- You replace constant(s) by 1 because different constants indicates different hardware being used.

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

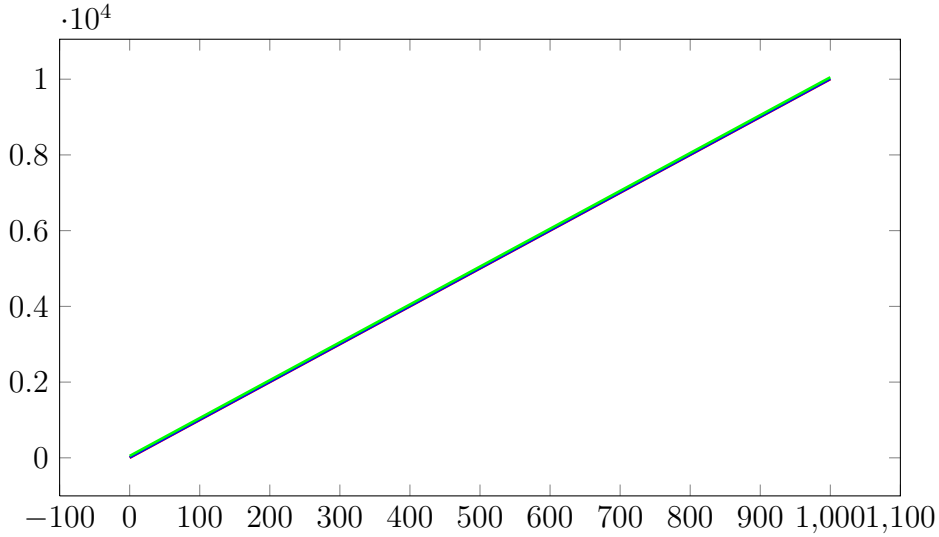
OK, let me show you *graphically* what the fudging does to a function. Here are the plots of $y = 10n, y = 10n + 10, y = 10n + 50$:



(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 $10n$.

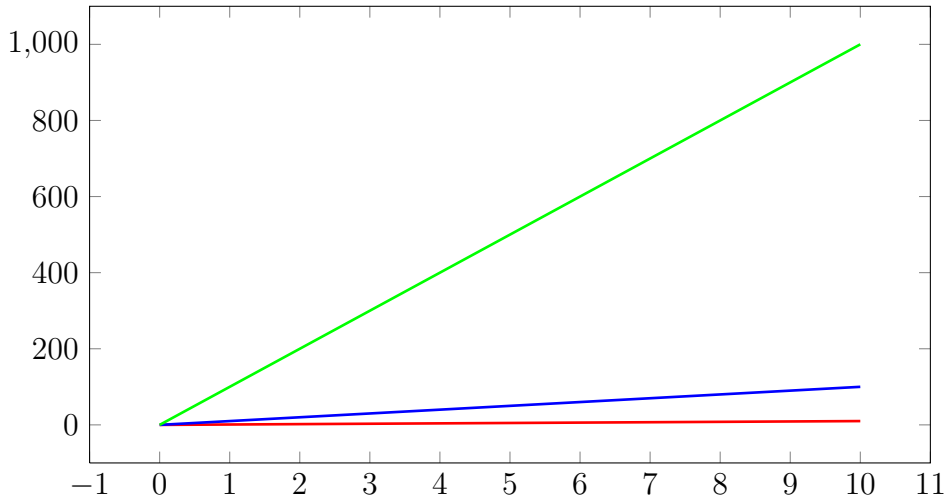
The second fudging step is when I throw away the A in the function

$$An$$

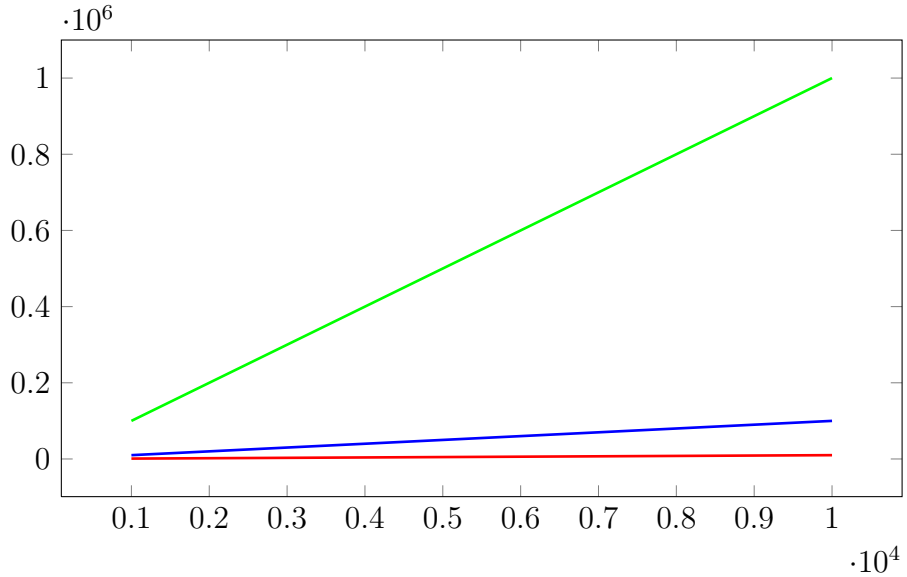
to get the function

$$n$$

because the constant A is hardware dependent. Here's a plot of $y = n$, $y = 10n$, $y = 100n$ for $0 \leq n \leq 10$:



For $1000 \leq n \leq 10000$:



They are constant multiples of each other. And remember this important fact: I don't care about multiple differences since in the case of functions measuring resource use (time in our case) it just mean that there's a difference in the hardware used.

Also, although they look like they are separated (because of their multiples), 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 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.

Exercise 102.4.1. The following computes the sum of squares from 1^2 to n^2:

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

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

```
         i = 1              time t0
         s = 0              time t1
LOOP:    if i > n:          time t2
             goto ENDLOOP   time t3
         term = i * i       time t4
         s = s + term       time t5
         i = i + 1          time t6
         goto LOOP          time t7
ENDLOOP:
```

(a) Compute the time taken as a function of n, t_0, ..., t_7.

(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 ... □

Exercise 102.4.2. The following sums up all the values in array x of size n and sets the values of the array to 0:

```
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_0, ...

(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 ... □

Exercise 102.4.3. The following pseoducode is similar to the above.

```
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 ... □

Exercise 102.4.4. Let

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

(a) Plot the graphs of

$$y = 100$$
$$y = 3n^2$$
$$y = \sin(n)$$

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 $f(n)$?

(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.] □

Exercise 102.4.5. Repeat the previous problem with this function:

$$f(n) = 10000 + \frac{1}{10}n^2 + \sin(n)$$

Use a sufficiently large domain for n. □

Exercise 102.4.6. Repeat the previous problem with this function:

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

Use a sufficiently large domain for n. □

File: best-average-and-worse-time.tex

102.5 Best, Average, and Worse Runtime

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

Example. 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:

(For non-programmers: when I say array `x` 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}.)

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

(For non-programmers: `break` means to get out of the current loop.) 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:
```

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 worse case scenarios.

Here's the timing calculation for the best scenario:

```
                                time    no. 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 worse case where the `target` is the last element of the array, i.e., at index `n - 1`, we have the following:

```
                                time    no. 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

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

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

```
                                time   no. 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 worse 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 worse 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, worse, 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 worse 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, worse, and average runtime, I will write $T_b(n)$, $T_w(n)$, and $T_a(n)$. (Technically speaking there are two different worse 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 worse case $T_w(n)$.

I have shown you above that for the linear search

$$\begin{aligned} T_b(n) &= O(1) \\ T_w(n) &= O(n) \end{aligned}$$

The average case is a little more complicated. In the above linear search algorithm, we looked at three cases (one best and two worse). 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, ..., T_n where the algorithm has $n+1$ cases (T_0 corresponding to the time where the `target` is at index 0, ... and T_n corresponding to the time where the `target` is not in the array at all) and then take the average:

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

But that assume something: That the cases corresponding to times T_0, ..., T_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

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

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

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

(1) state what cases you're average over and

(2) what is the likelihood of each case.

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

For many algorithms and many average scenarios, the average runtimes tend to be the same as the worse 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    no. 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

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

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

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

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:

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

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 $T_0, \ldots, T_{n-1}$ ... do *not* include T_n.

This average runtime is

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

That's it!

Exercise 102.5.1. The linear search algorithm searches from index value 0 to the last index value. The *reverse* linear search is pretty much the same as the linear search except that it starts with the last index value and moves toward the 0 index value. For the following, assume as before the the array is `x` and the size of the array is `n`.

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

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

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

```
                                time
         index = -1             t1
         i = n - 1              t2
LOOP:    if i == -1:            t3
             goto ENDLOOP       t4
         if x[i] is target:     t5
             index = i          t6
             goto ENDLOOP       t7
         i = i - 1              t8
         goto LOOP              t9
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 worse 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 worse 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.]

Exercise 102.5.2. The following algorithm computes the maximum 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 worse runtime (where the body if the `if` statement is always executed for each iteration of the loop. □

Exercise 102.5.3. 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 worse runtime (where the body of the `if` statement is always executed for each iteration of the loop. □

Exercise 102.5.4. 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.

(b) Assume that each line requires constant time. Assign times to each statement.

(c) Compute the best runtime.

(d) Compute the big-O of the best runtime.

(e) Compute the worse runtime.

(f) Compute the big-O of the worse runtime.

(g) 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.

(h) 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.

Exercise 102.5.5. 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 is 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] is -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 worse runtime of the above and the big-O. (Be careful now!!! What exactly is the worse case???)

(c) Compare the best and worse runtimes of this shuffling algorithm with the previous.

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

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

into goto statements:

```
...
LOOP3:      if y[j] is not -1:
                goto ENDLOOP3
            j = (j + 1) % n
```

```
            goto LOOP3

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

[MORE EXERCISES]

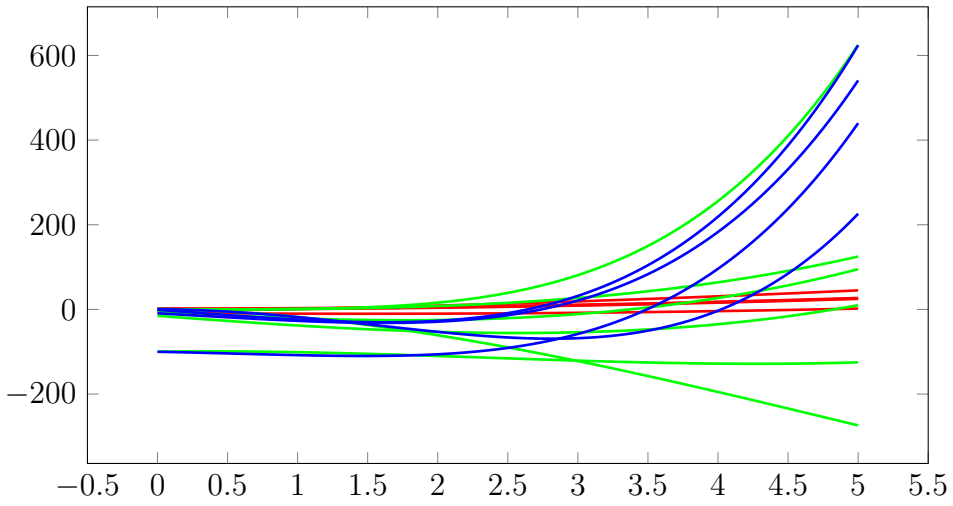
File: separating-polynomial-functions.tex

102.6 Separating Polynomial Functions

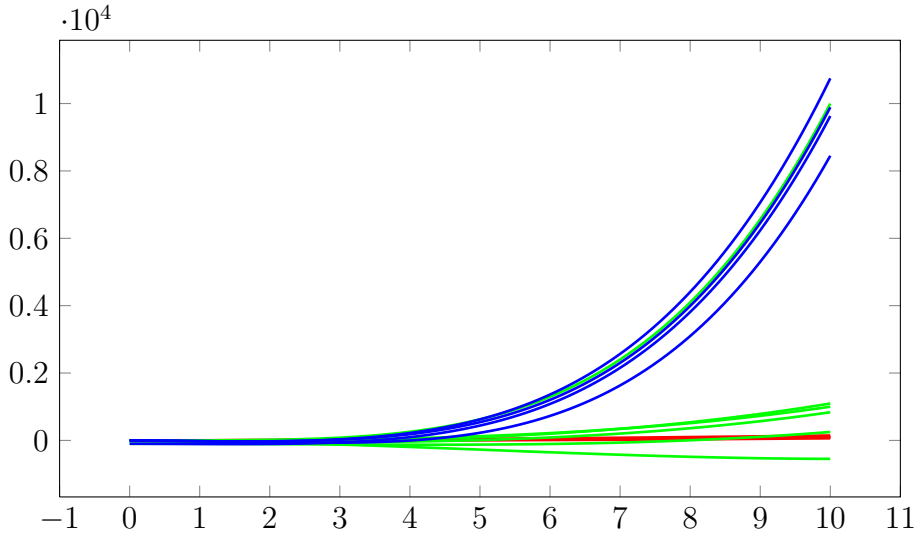
I'll be giving you the formal definition of big-O soon. But before that, I'm going to motivate the (formal) definition of big-O by talking about the way the graphs of polynomial climb. The rate at which they climb essential tells you the story of big-O among polynomials. This will give you the intuitive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

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

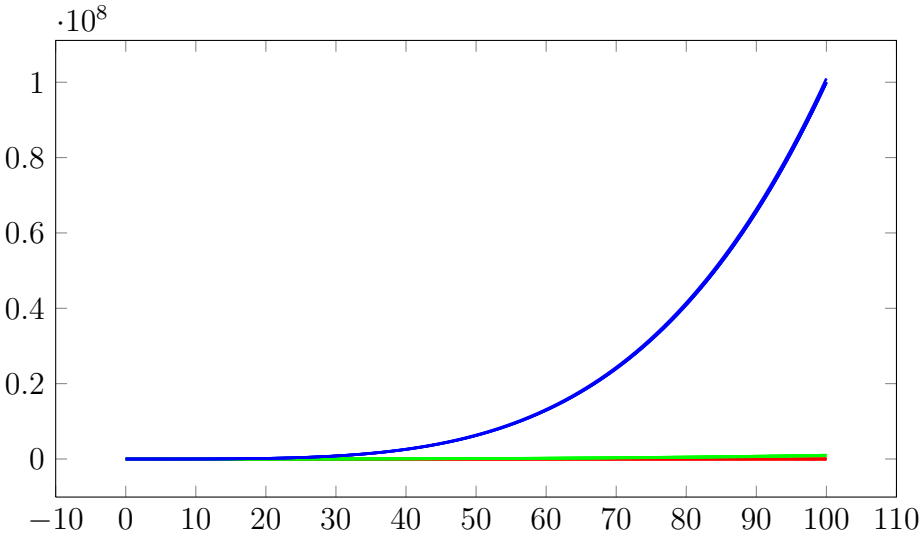
Look at this mess of 15 polynomail functions (I won't give you the polynomials just yet):



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



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

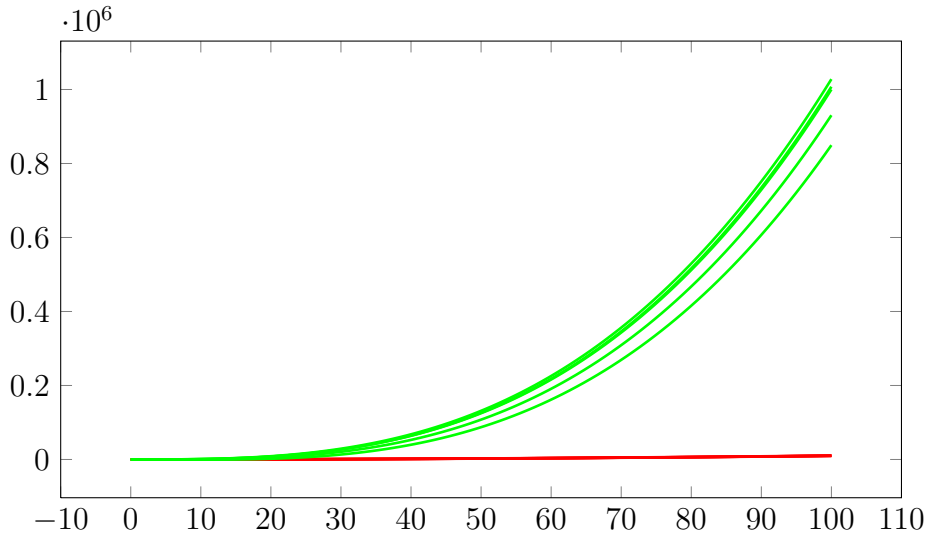


Five functions have separated away from the others. Now I reveal to you that

these five functions are:

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

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



The five functions which are higher up are:

$$n^3$$
$$n^3 + 3n^2 - 20n - 5$$
$$n^3 + n^2 - 25n - 15$$
$$n^3 - 15n^2 - 5n + 1$$
$$n^3 - 7n^2 + 5n - 100$$

and the last group of five functions are:

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

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

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

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

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

(because they are the simplest.)

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

bunches led by the following leaders:

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

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

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

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

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

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?

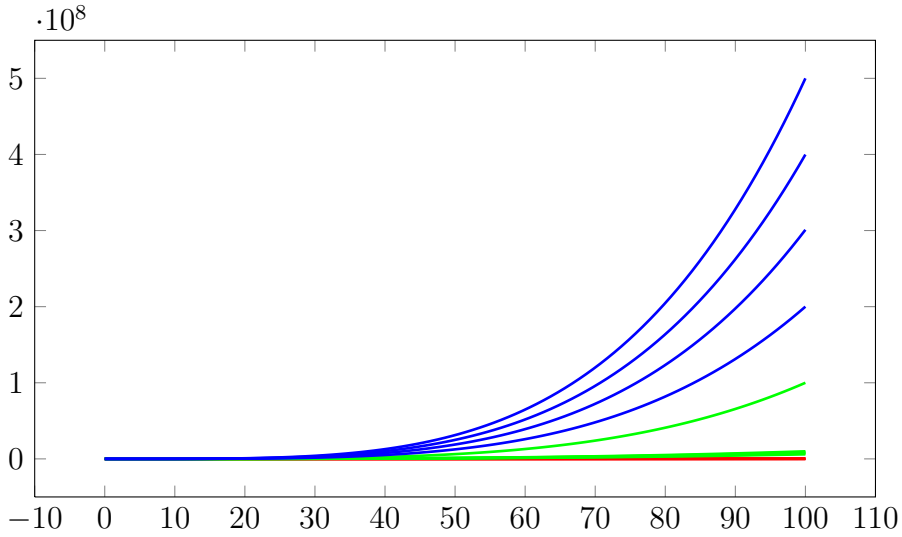
Here are the original 15 functions again:

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

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 102.6.1. *Let $f(n)$ be a polynomial function of degree d:*

$$f(n) = c_0 + c_1 n + c_2 n^2 + \cdots + c_d n^d$$

Then

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

Furthermore if $e > d$*, then we also have*

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

In general, if $f(n)$ *is a polynomial of degree* d*,* $g(n)$ *is a polynomial of degree* e*, and* $d \leq e$*, then*

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

□

Usually we will pick the simplest $g(n)$ for our big-O statement:

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

Also we pick the "smallest" function $g(n)$. So if you know that $f(n) = O(n^3)$ and $f(n) = O(n^5)$, you will usually use $f(n) = O(n^3)$.

Exercise 102.6.1. What is the big-O of 100? □

Exercise 102.6.2. What is the big-O of $5n^2 - 10$? □

Exercise 102.6.3. What is the big-O of $42 - 3n$? □

Exercise 102.6.4. What is the big-O of $5 - n + n^3 + 2n^2$? □

Exercise 102.6.5. What is the big-O of $5 - n + 1000000n^3 + n^2$? □

Exercise 102.6.6. What is the big-O of

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

[HINT: Plot the two terms and see which one grows faster. Make sure you use a large domain for n.] □

Exercise 102.6.7. What is the big-O of

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

[HINT: Plot the two terms and see which one grows faster. Make sure you use a large domain for n.] □

File: definition-of-big-O.tex

102.7 Definition of big-O

You are now ready for the definition of big-O, at least graphically.

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

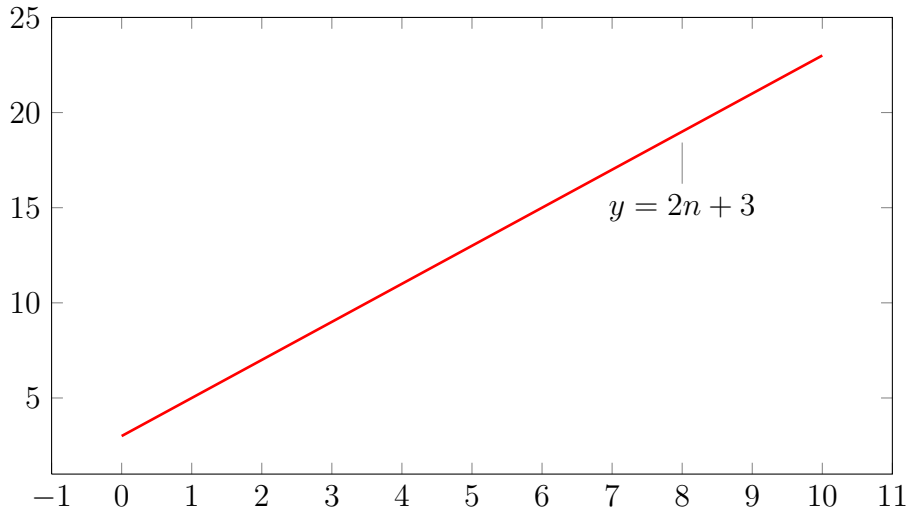
$$f(n) = O(g(n))$$

if a *multiple* of $g(n)$ beats $f(n)$ not necessarily for all n but for *all large values* of n. (I have to make a small adjustment later but this is ood enough for now.)

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

Let me give you some examples. Suppose we're looking at this function

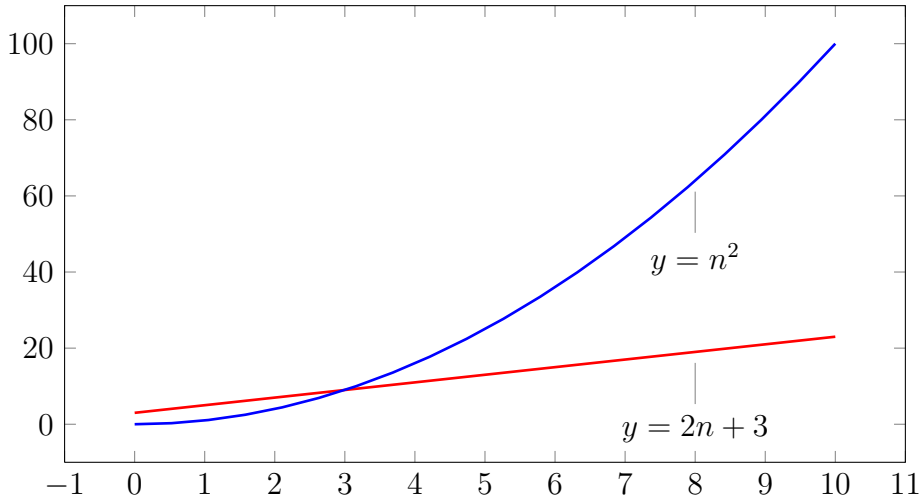
$$f(n) = 2n + 3$$



Let's compare that to

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

Let's put them together in a plot:



You see that $f(n) = 2n + 3$ is *at worse* $g(n) = n^2$ for large values of n, i.e.,

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

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

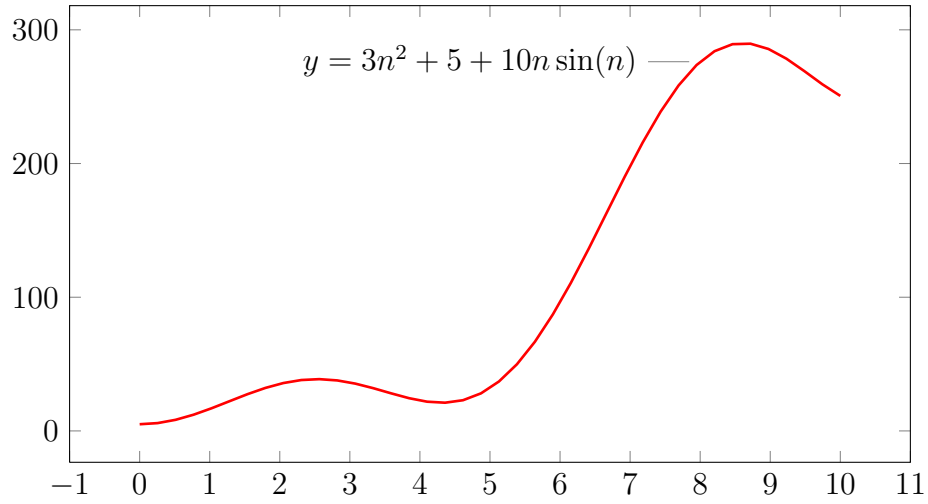
$$f(n) = O(g(n))$$

Here's another example.

Suppose

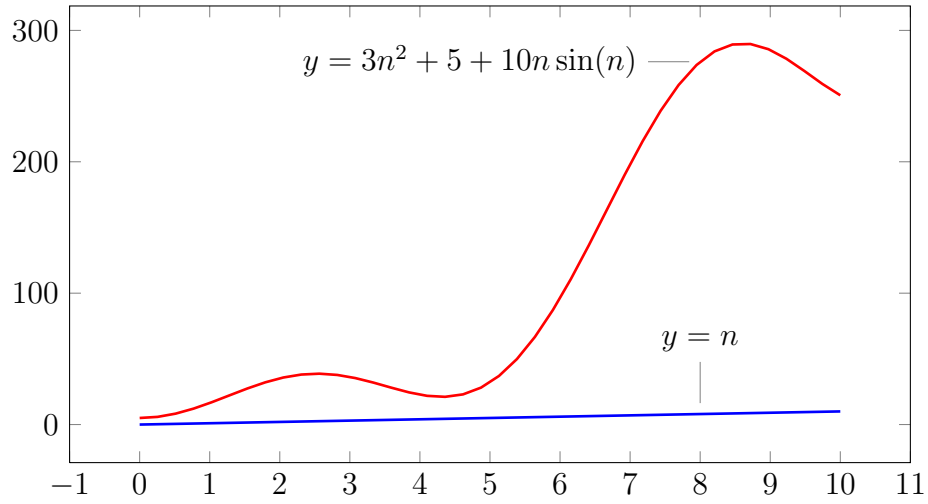
$$f(n) = 3n^2 + 5 + 10n \sin(n)$$

Here's the plot:



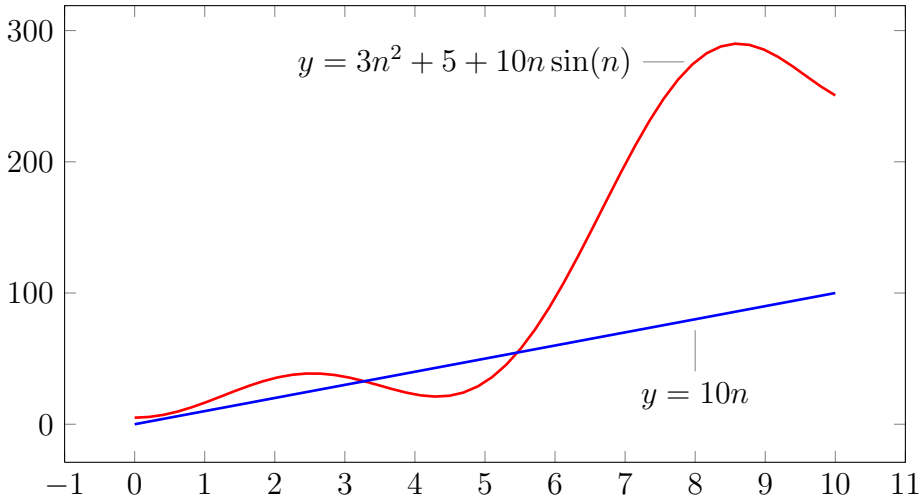
Let's see if we can *cap* it with

$$g(n) = n$$



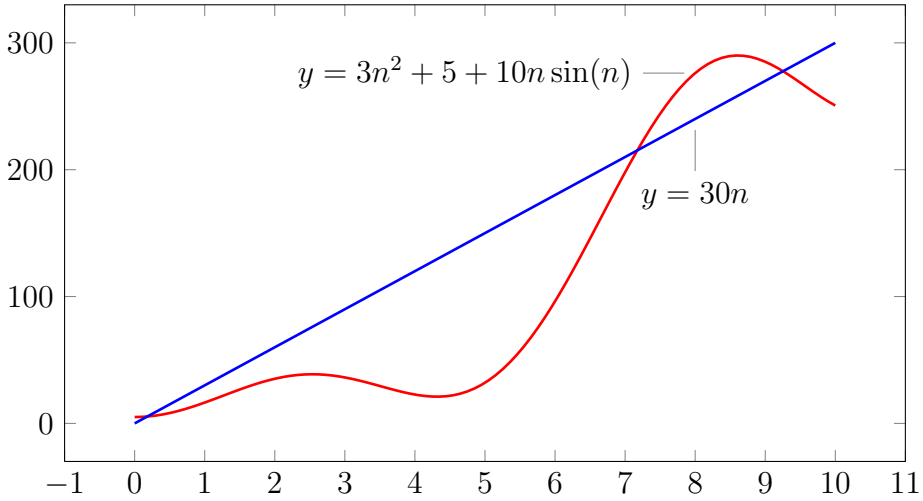
Not good. But don't forget that if we do want to say $f(n) = O(g(n))$, then we are allowed to use multiples of $g(n)$. So let's try

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



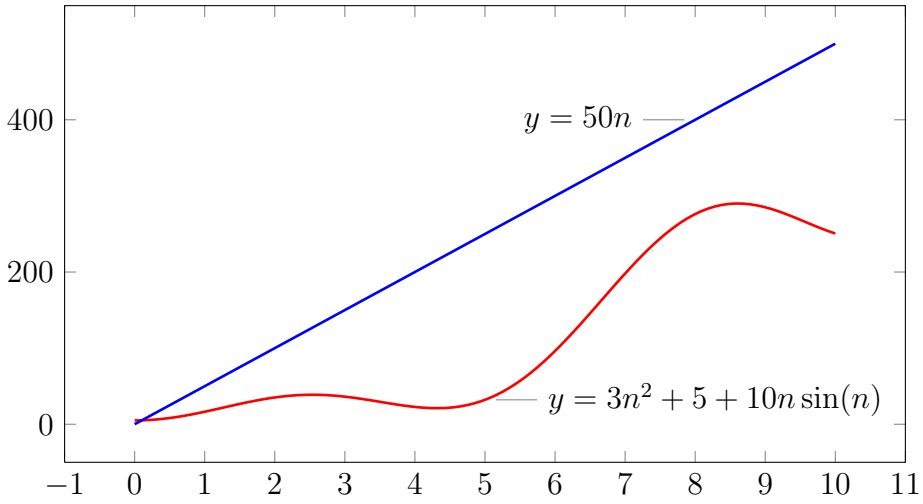
Better!!! But just by looking at the graph, my multiple of $g(n)$ must at least overcome the bump of $f(n)$ at around $n = 8.5$. Let's try

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

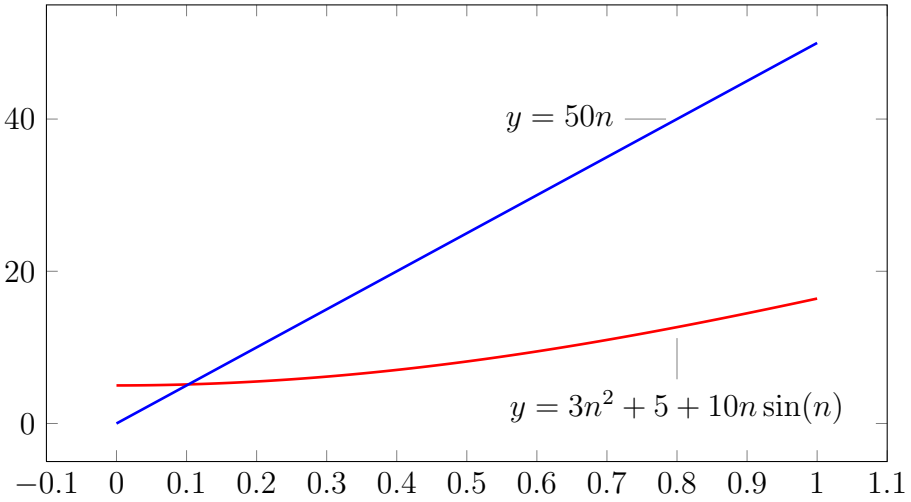


Finally let's try

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



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.

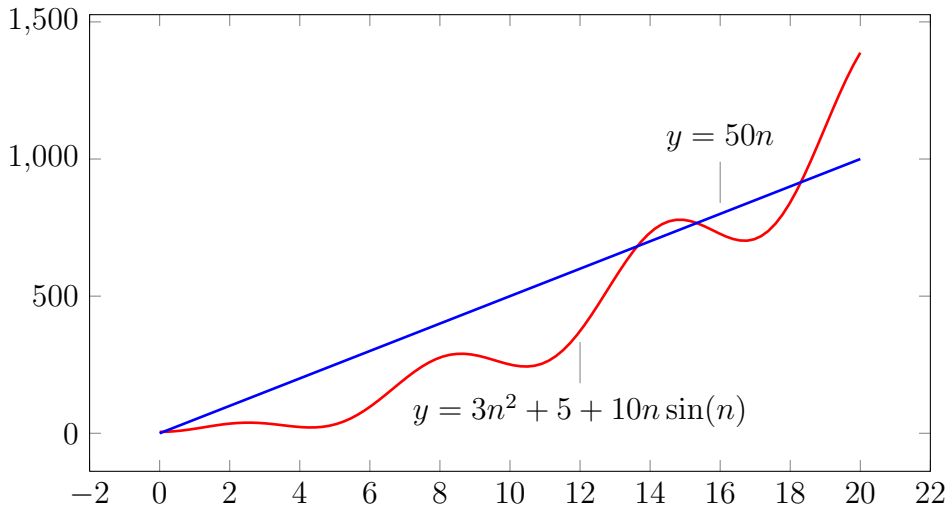
So from the previous two graphs can we say that for $n \geq 1$, $50g(n)$ beats $f(n)$? ... i.e., can we say

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

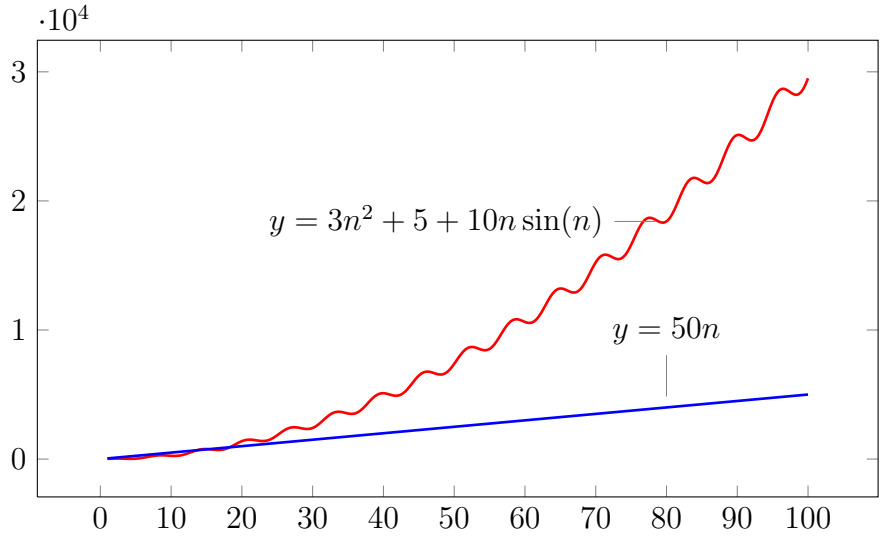
and conclude that $f(n) = O(g(n))$?

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 $n = 100$:

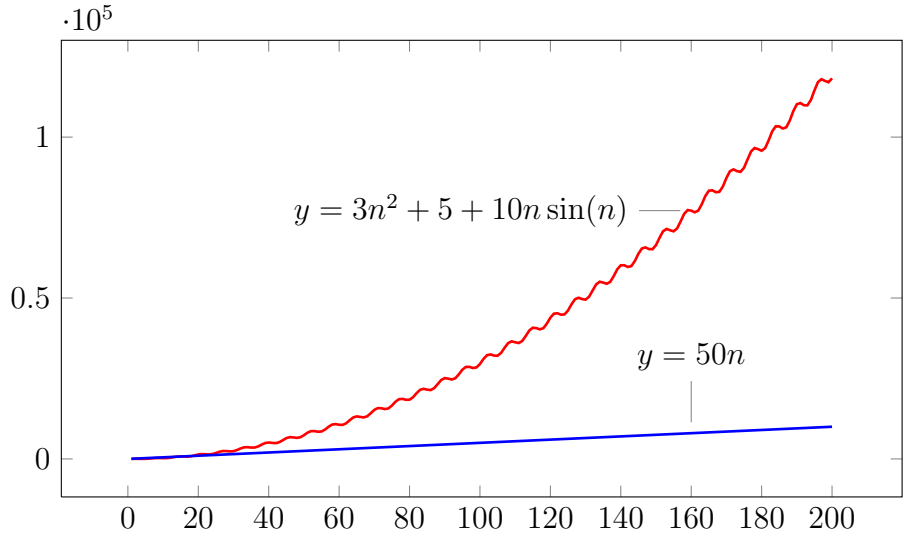


Clearly *no* straight line is going to dominate $f(n)$ because it seems that the graph of $f(n)$ *bends* up (with wiggles along the way).

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 $n = 200$:

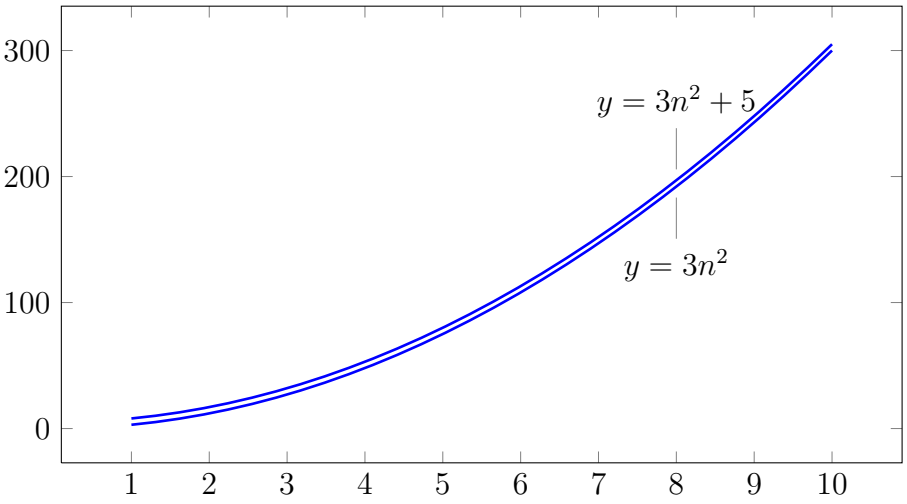


So let's abandon our $g(n) = n$ altogether.

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

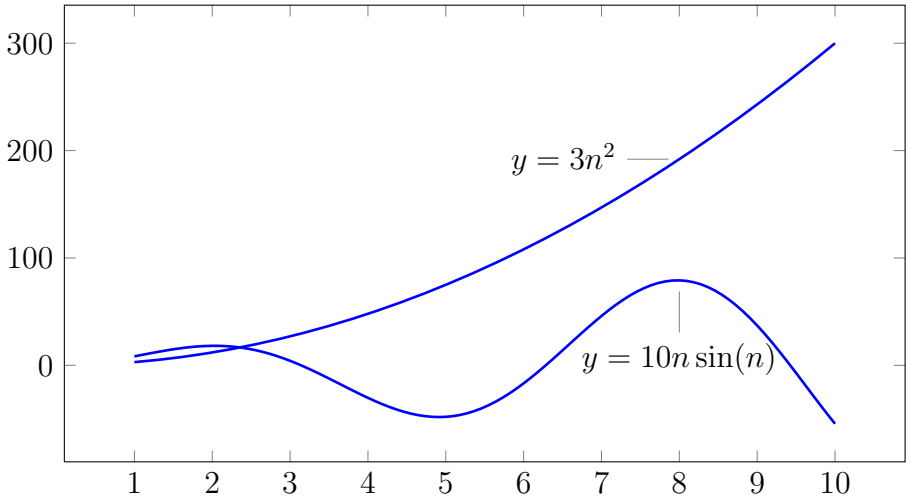
$$f(n) = 3n^2 + 5 + 10n\sin(n)$$

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:

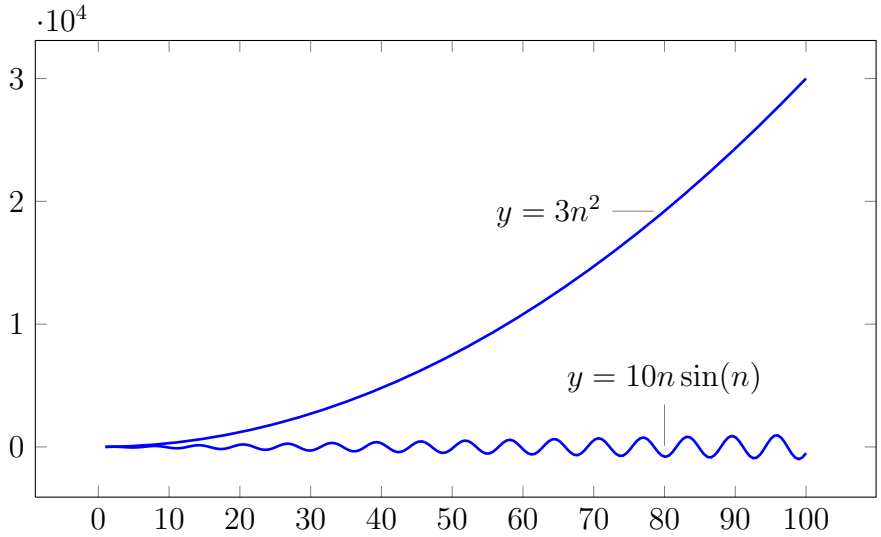


Of course since we can control $f(n)$ with multiples later we just need to choose a huge multiple of $3n^2$ to beat $3n^2 + 5$, for instance $1000000(3n^2)$.

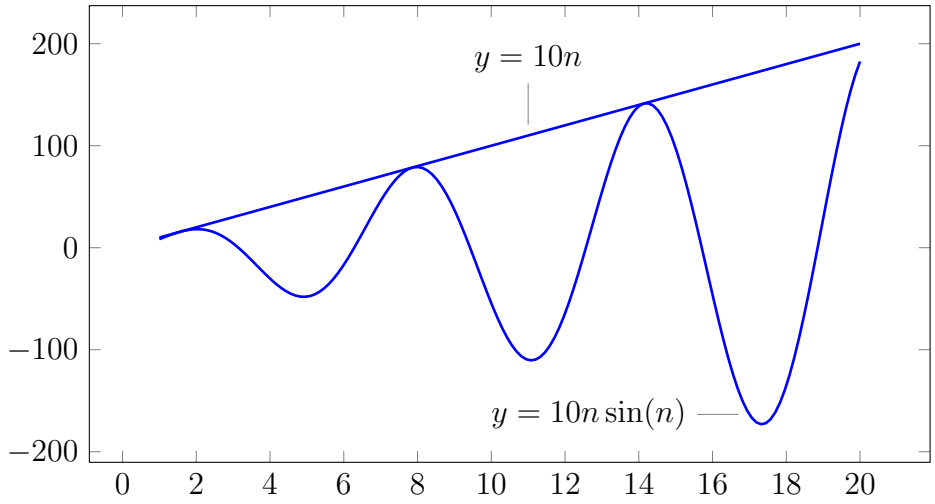
What about $10n \sin n$? If we plot that with $3g(n) = 3n^2$ we get:



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!!!

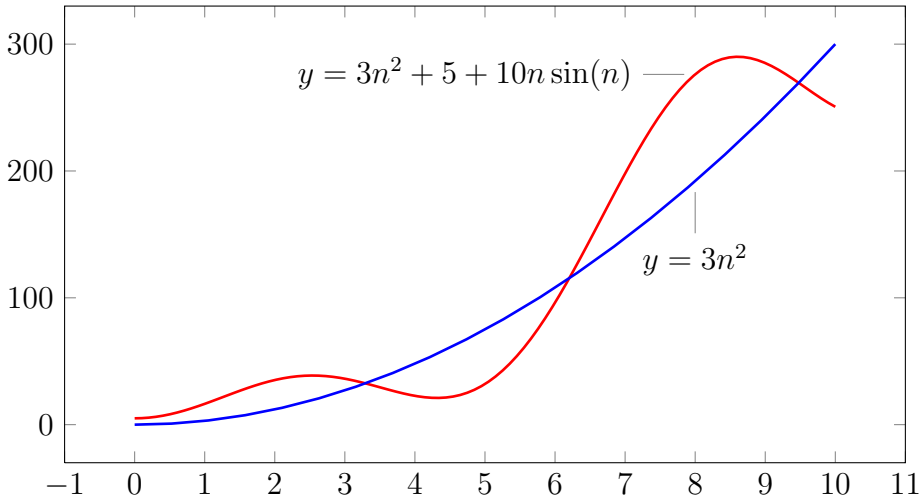
This means that $3n^2$ will beat $10n \sin n$ for large values of n.

Altogether, this means that the growth of

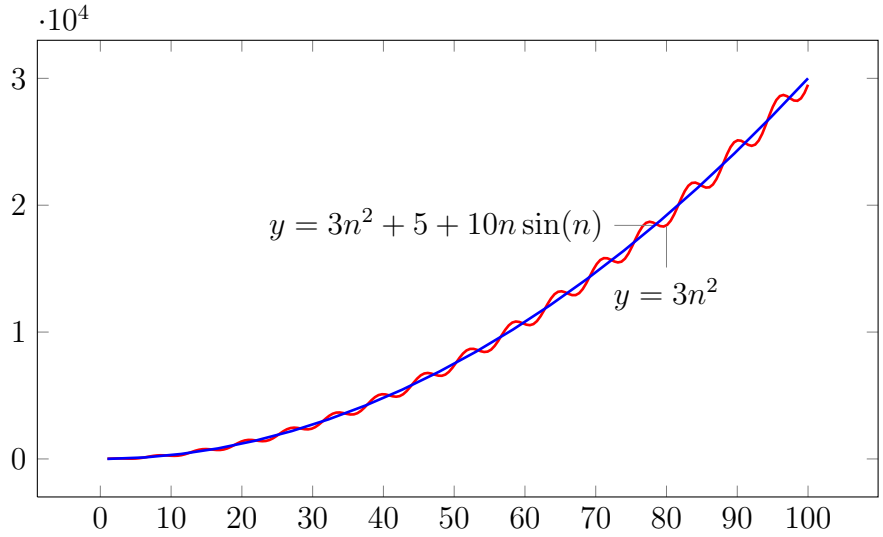
$$f(n) = 3n^2 + 5 + 10n\sin(n)$$

is ultimately determined by $3g(n) = 3n^2$ 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.

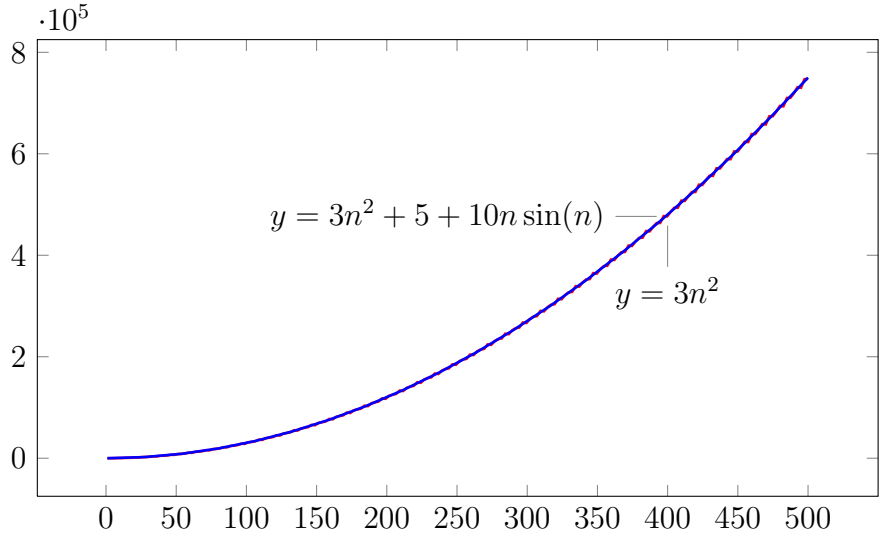
Here's the plot of $3g(n) = 3n^2$ and $f(n)$ for $0 \leq n \leq 10$:



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 $n = 100$:

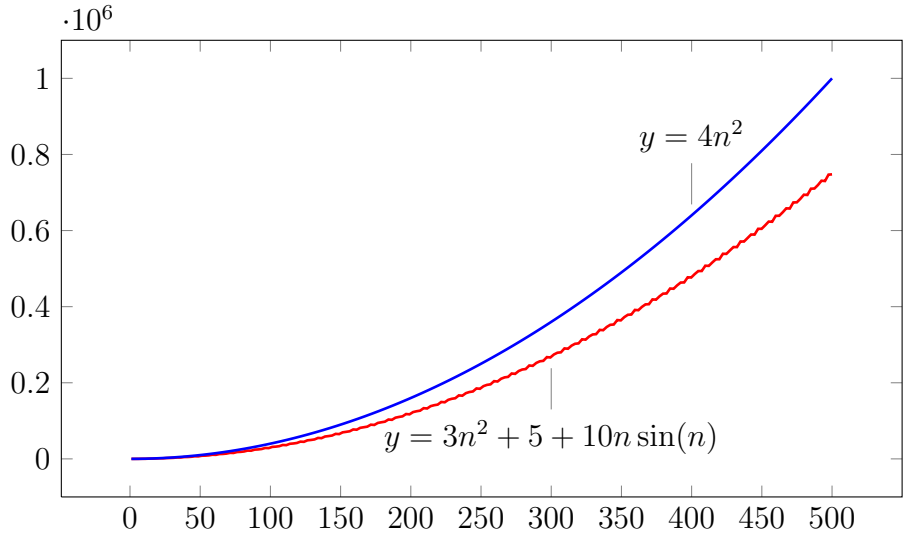


and then up to $n = 500$:

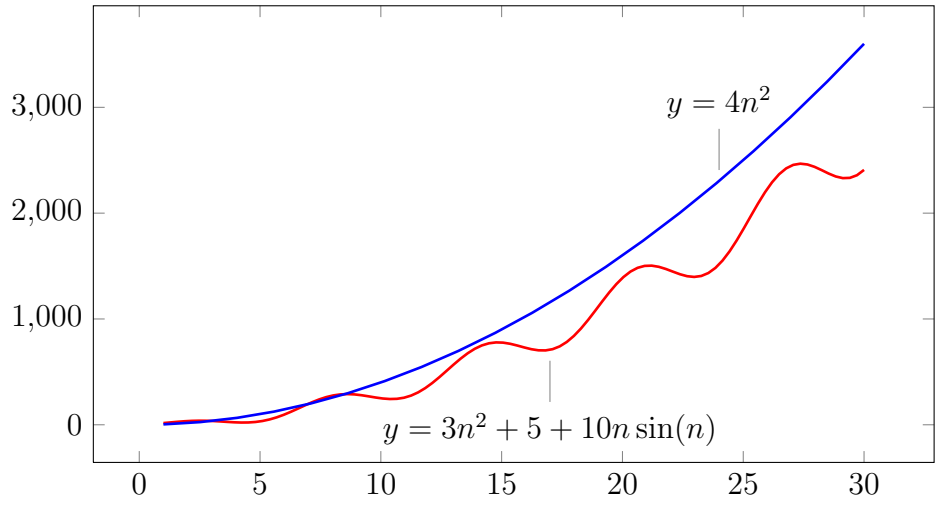


GRRREAT!!!

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

$$f(n) \leq Cg(n) \text{ for } n \geq N$$

This means that I can now say

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

This is more or less the formal idea of big-O. There is one detail to add: I need to add that you should add the *absolute value signs* to both sides of the above inequality. So (finally!) here's the formal definition of big-O:

Definition 102.7.1. Let $f(n)$ and $g(n)$ be functions. We write

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

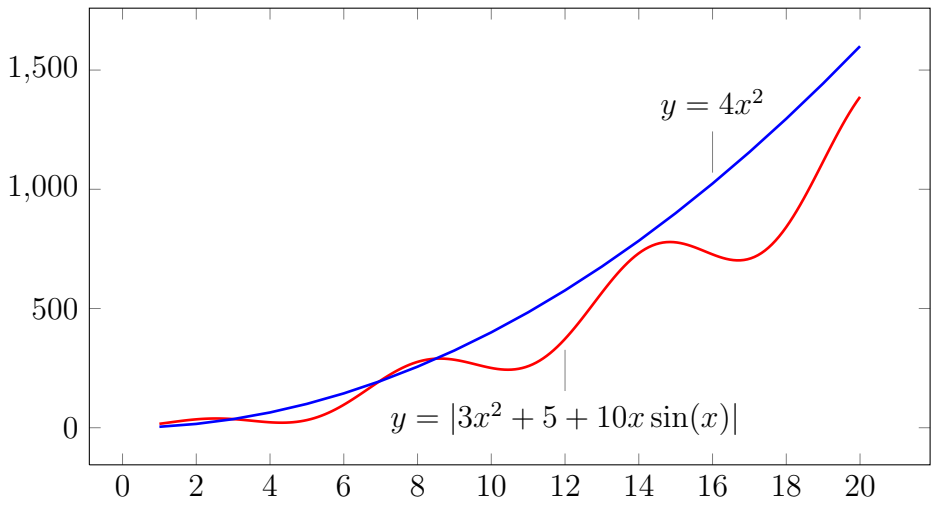
and say that

$$\text{“}f(n) \text{ is } \textbf{big-O} \text{ of } g(n)\text{”}$$

if we can find a C and an N such that for all $n \geq N$, we have

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

Here's our original problem but this time I'm plotting $|3n^2 + 5 + 10n\sin(n)|$ and $4|n^2|$:

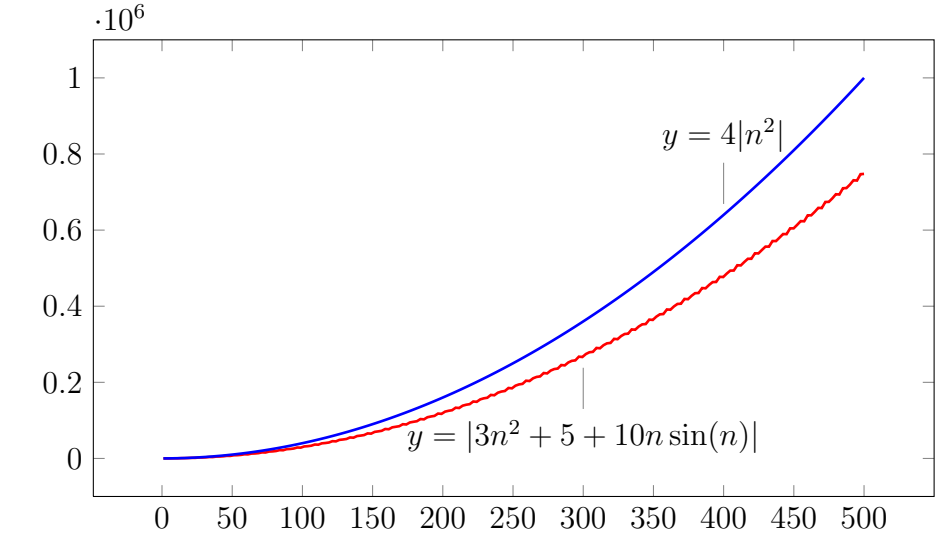


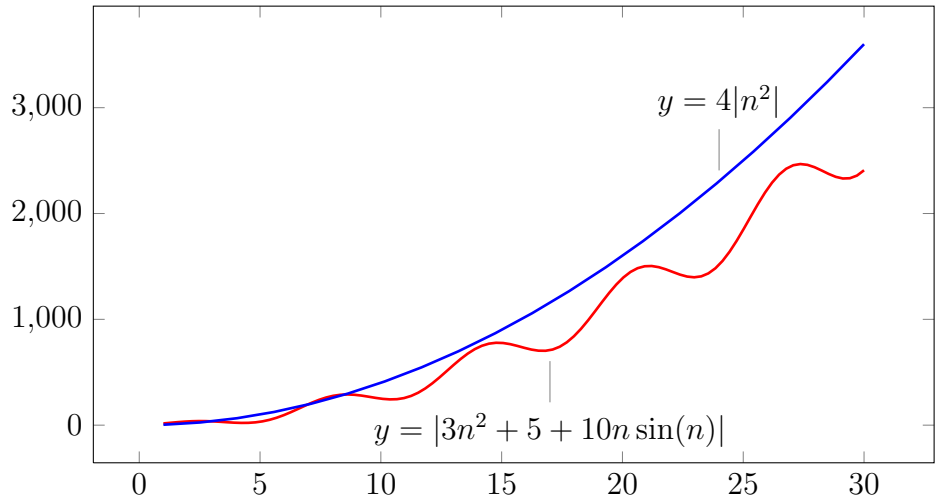
Well ... what do you know ... in this case it seems like both functions are positive anyway. (Yes, they *are* positive for all $n \geq 0$. Do you see why?) Therefore the graph looks the same and so I don't have to change my choice of C and N. Phew!!! So now I'm ready to present my solution to this problem ...

Example 102.7.1. 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.,

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

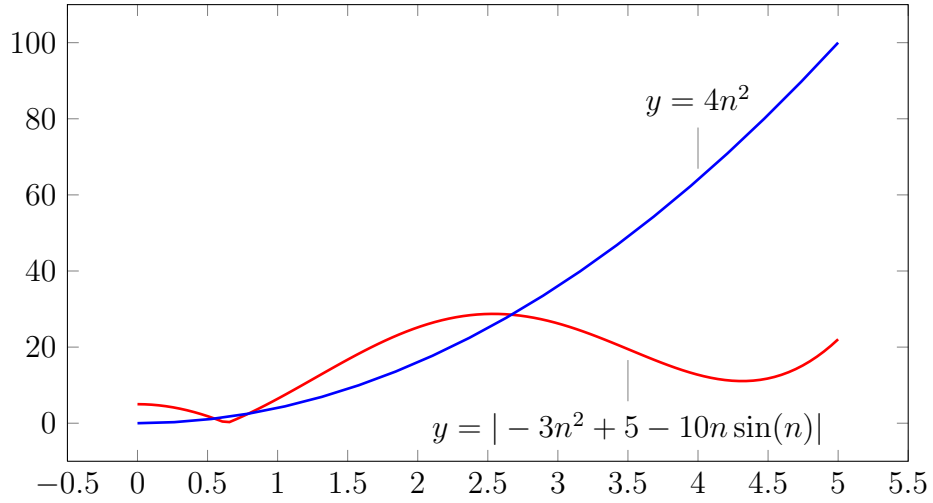
Hence $f(n) = O(g(n)$. □

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

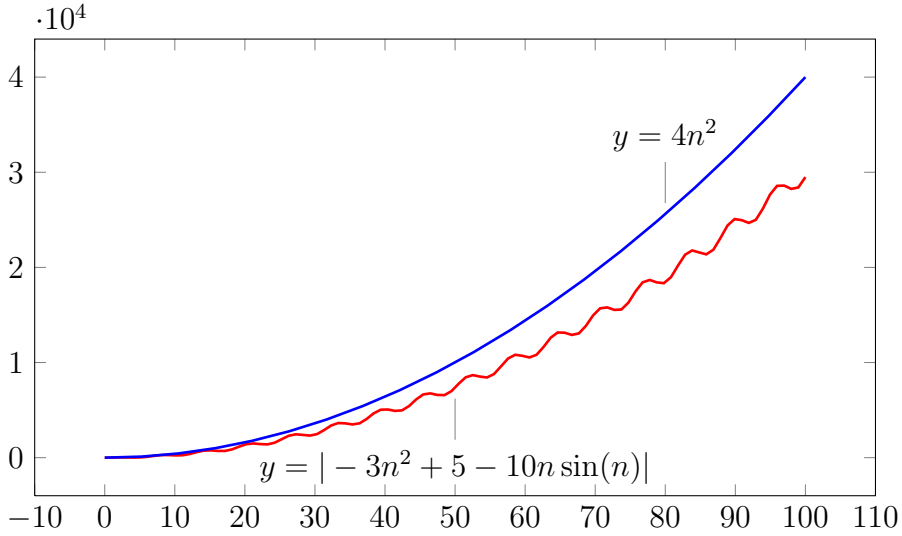
Now, let me give you another example. Suppose I change my $f(n)$ to *this*:

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

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

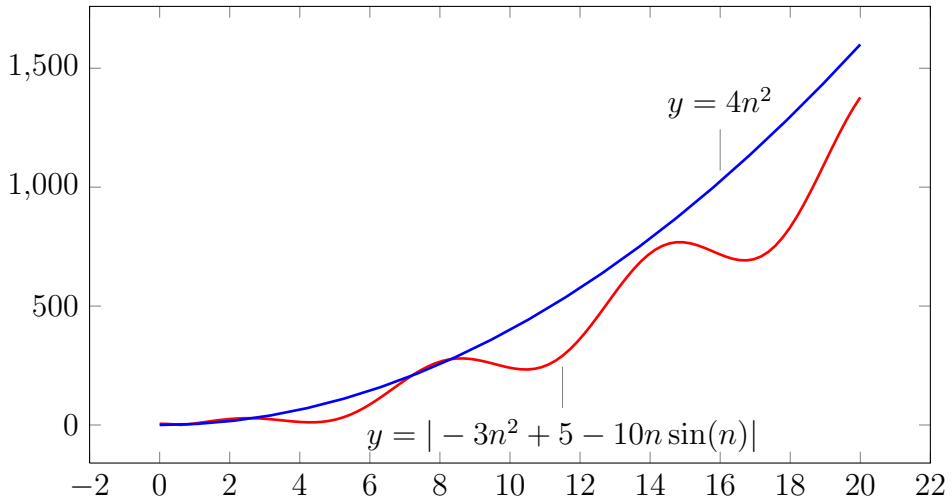


and then for $0 \leq n \leq 100$:



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 $0 \leq n \leq 20$:



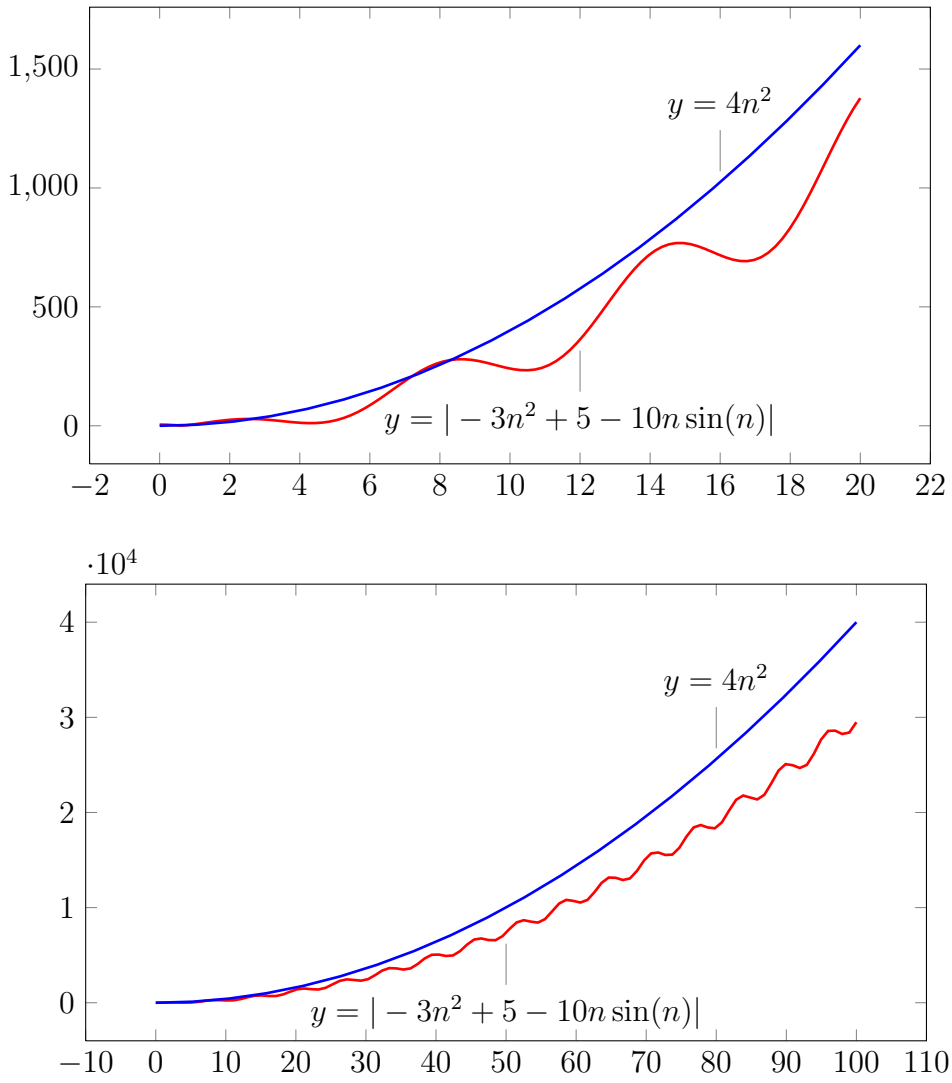
Clearly for $n \geq 10$, $4g(n)$ beats $|f(n)$. So I'm going to pick $N = 10$.

Now I'm ready to present this ...

Example 102.7.2. 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.

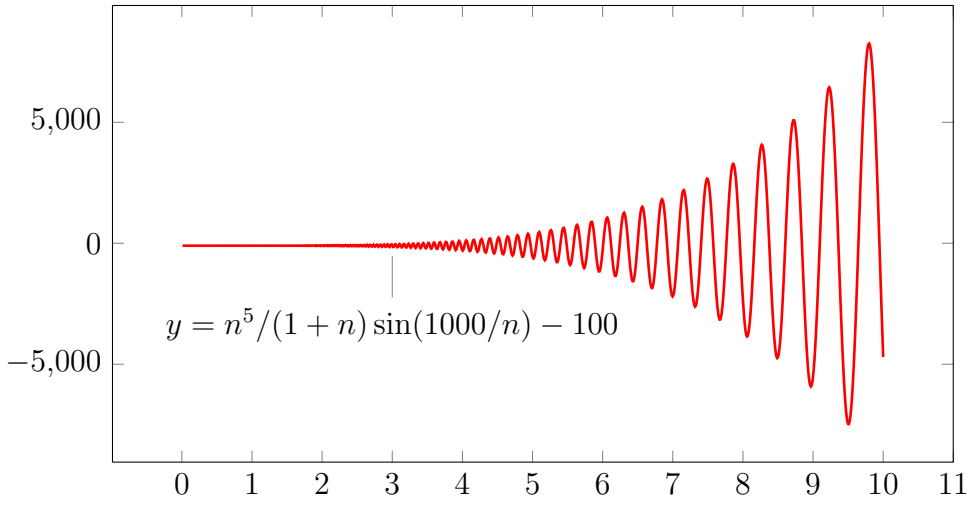
Here's another example.

Let

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

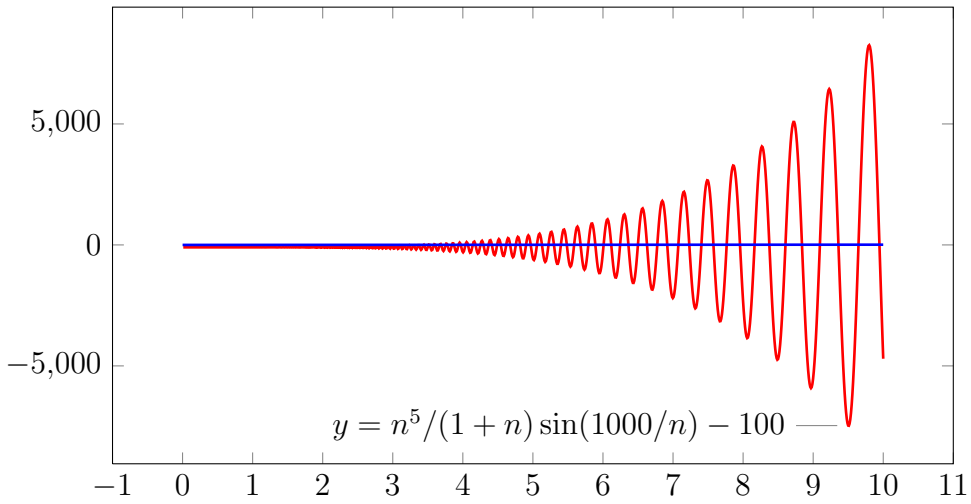
Let's find some function $g(n)$ of the form n or n^2 or n^3 or ... such that $f(n) = O(g(n))$.

Let's have a few plots to get a feel for the function. Here's the plot for $0 \leq n \leq 10$:

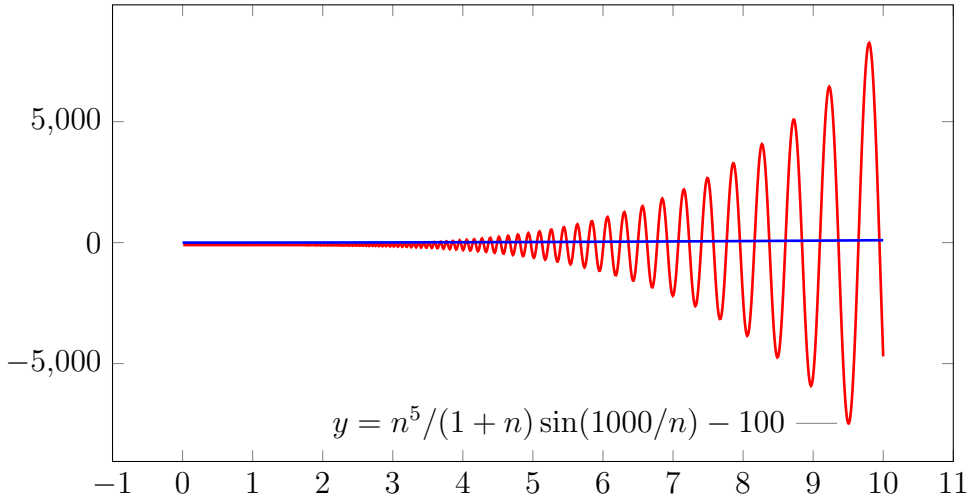


Clearly $g(n)$ can't be $n^0 = 1$ (i.e., no multiple of $g(n) = 1$ is going to beat $f(n)$ ... right?) so I'm going to skip that. I'll try higher powers.

With $g(n) = n$:

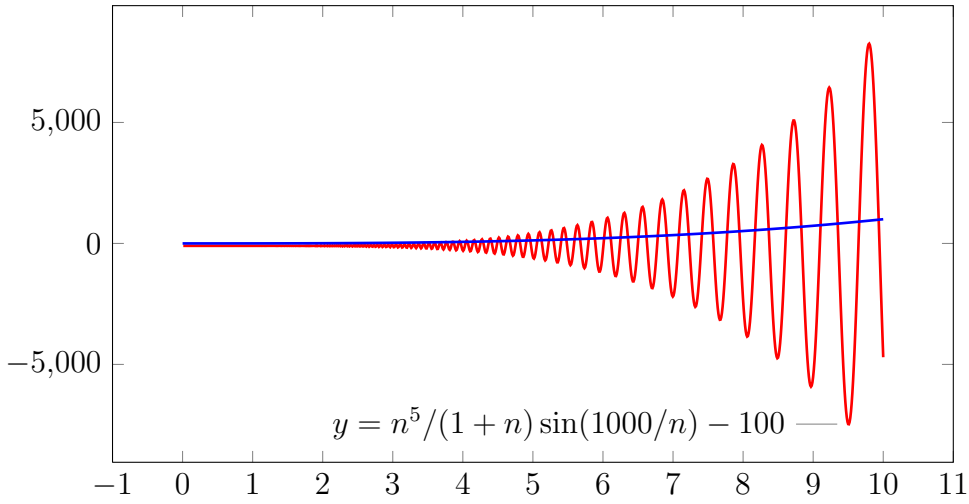


With $g(n) = n^2$:



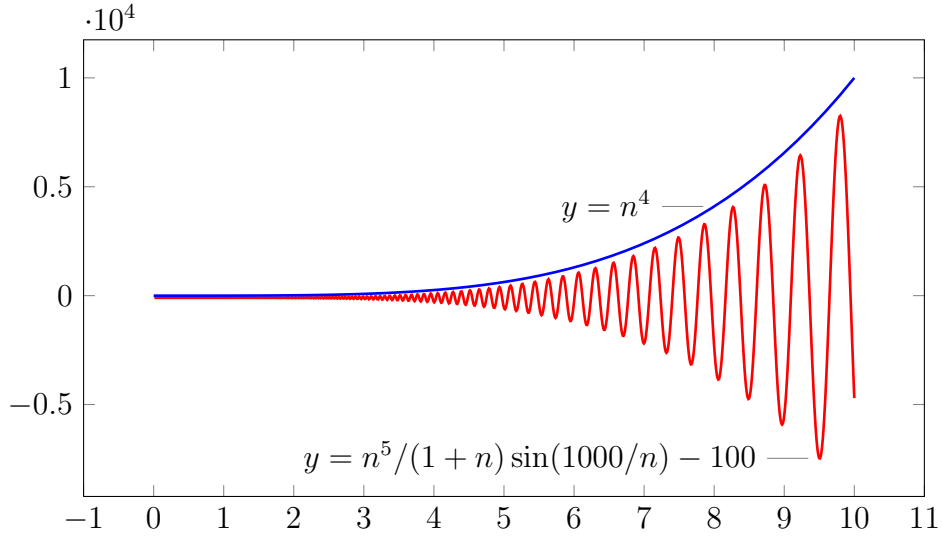
Wow. $f(n)$ is exploding so fast that I can't even see n^2 bending up at all ...

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 $f(n)$.

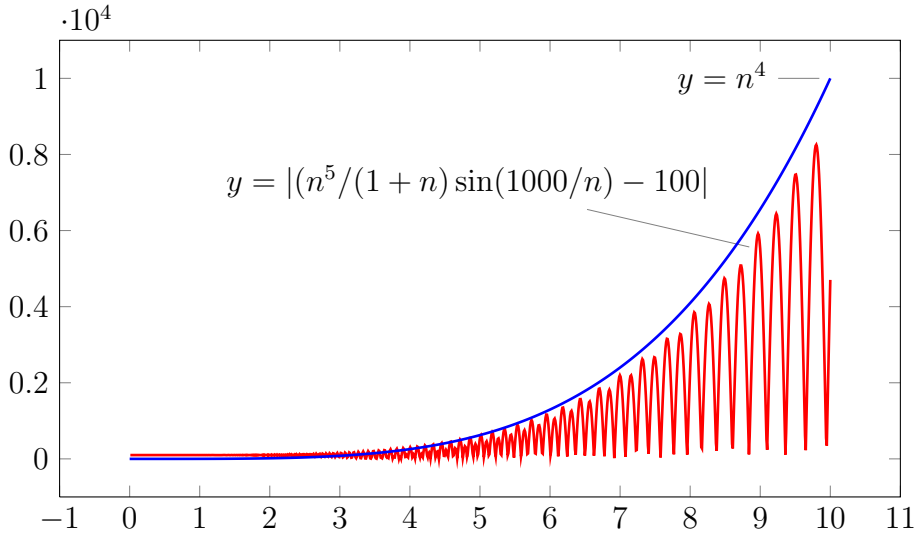
With $g(n) = n^4$:



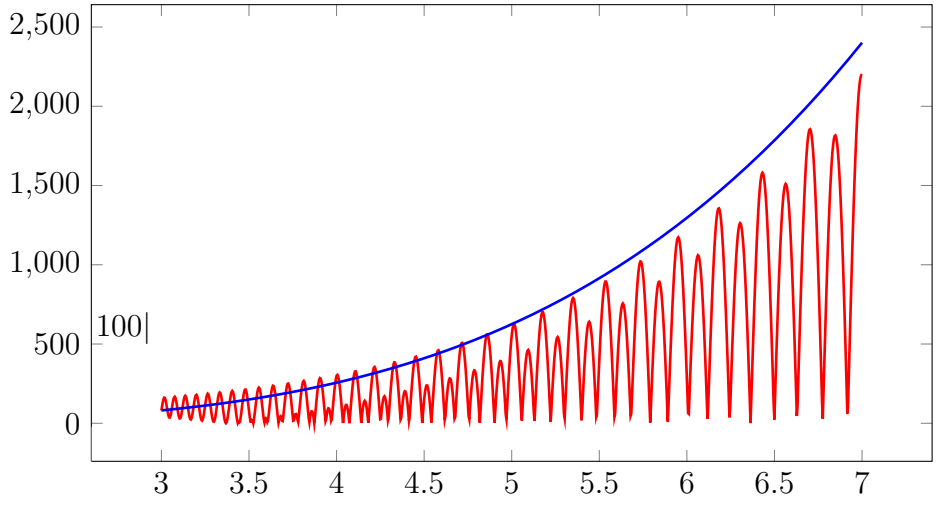
Aha! n^4 beats $f(n)$!!!

But wait ... we recall that we should plot the *absolute value* of $|f(n)|$ and $|g(n)|$! Duh!!! Note that it won't change $|g(n)|$ since $g(n) = n^4$ is positive. But $|f(n)|$ will look different from $f(n)$.

Here's the plot for $|f(n)|$ and $g(n) = n^4$ for $0 \leq n \leq 10$:



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 $N = 6$ and $C = 1$, then for $n \geq N$,

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

i.e.,

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

Hence

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

So here's the solution ...

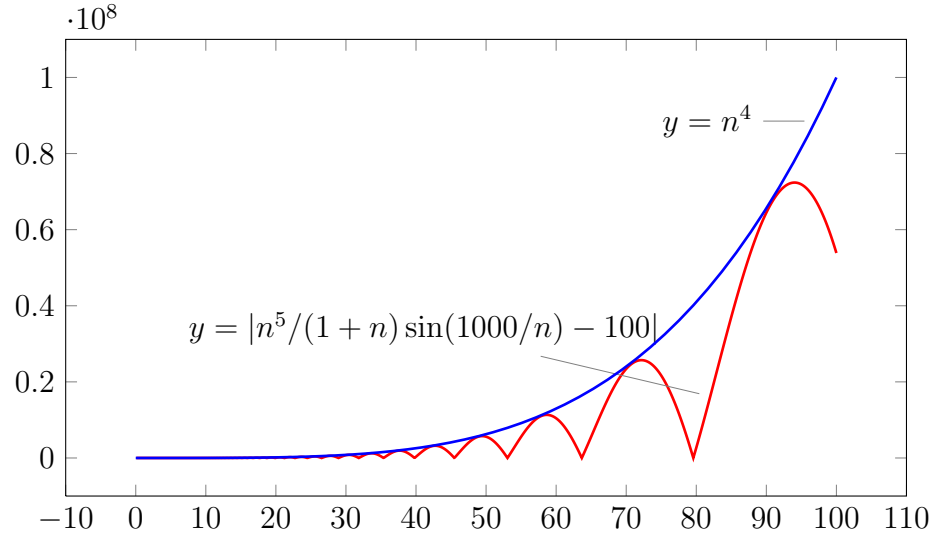
Example 102.7.3. Let

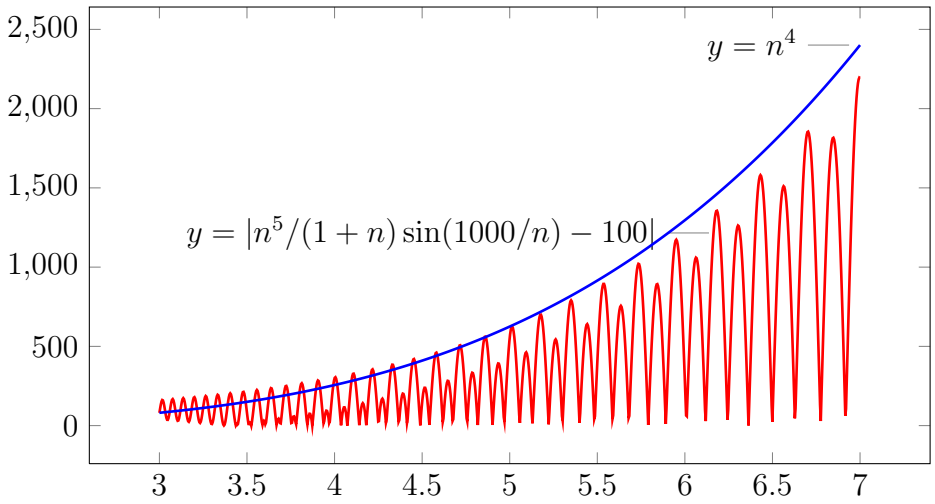
$$f(n) = \frac{n^5}{1+n} \sin \frac{1000}{n} - 100$$

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

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

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





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

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

i.e.,

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

Hence

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

□

Exercise 102.7.1. 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) Using a plot, find the smallest positive integer k such that $20\frac{n^5}{n^3+1}$ is $\leq Cn^k$ for some C and for large values of n.

(b) Using a plot, find the smallest positive integer k such that $|5n^3 \cos n|$ is $\leq Cn^k$ for some C and for large values of n.

(c) Using a plot, find the smallest positive integer k such that 7^8 is $\leq Cn^k$ for some C and for large values of n.

(d) Let k be the largest k values from (a)-(c). Plot n^k and $f(n)$ from $n = 0$ to a large value.

(e) Find a suitable C if necessary to move n^k beyond $f(n)$ for large values of n.

(f) Zoom in on small values of n and find a suitable N such that for $n \geq N$, $|f(n)| \leq C|g(n)|$ for $n \geq N$.

□

Exercise 102.7.2. Let

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

for $n \geq 1$. 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 $n \geq N$. □

Exercise 102.7.3. Let

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

for $n \geq 1$. 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 $n \geq N$. □

Exercise 102.7.4. Let

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

for $n \geq 1$. 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 $n \geq N$. □

Exercise 102.7.5. Let

$$f(n) = 7n^{2.7} \ln n + 20\frac{n^5}{n+1}$$

for $n \geq 1$. 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: $\ln = \log_e$.] □

Exercise 102.7.6. Let

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

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 $n \geq N$. □

Exercise 102.7.7. Let

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

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 $n \geq N$. □

Exercise 102.7.8. Let

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

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 $n \geq N$. □

File: summation.tex

102.8 Summation

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

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

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

Let a_i be a formula in i. The notation

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

is a shorthand notation for

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

For instance

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

Exercise 102.8.1. Compute the value of $\sum_{i=2}^{5} i^2$ □

Exercise 102.8.2. Compute the value of $\sum_{i=0}^{4} (2i)$ □

Exercise 102.8.3. Compute the value of $\sum_{i=0}^{4} (2i+1)$ □

Exercise 102.8.4. Compute $\sum_{i=2}^{5} i^2$ □

Exercise 102.8.5. Compute $\sum_{i=2}^{10} 1$ □

Exercise 102.8.6.

(a) Write down the following as a sum of terms without simplifying:

$$\sum_{i=2}^{5}(5i^2)$$

(b) Now write down the following by writing down the product of 5 and a sum of terms. Do not simplify.

$$5\sum_{i=2}^{5} i^2$$

Check that the expressions in (a) and (b) are the same. □

Do you see that in general

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

Exercise 102.8.7.

(a) Write down the following as a sum of terms without simplifying:

$$\sum_{i=2}^{5}(i^2+\sqrt{i})$$

(b) Now write down the following by expanding each summation into two sums of terms. Do not simplify.

$$\sum_{i=2}^{5} i^2 + \sum_{i=2}^{5} \sqrt{i}$$

Check that the expressions in (a) and (b) are the same. □

It's not too surprising that in general

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

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

Theorem 102.8.1.

(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. For instance

$$\sum_{i=5}^{n} ca_i = c\sum_{i=5}^{n} a_i$$

In general:

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

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

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

Exercise 102.8.8. Compute $\sum_{i=0}^{5}\left(\sum_{j=0}^{7} i(i-1)j\right)$ □

Exercise 102.8.9. Compute $\sum_{i=0}^{5}\left(\sum_{j=i}^{7} i(i-1)j\right)$ □

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

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

So is "splitting out terms from the top":

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

Likewise you can split a summation into two like this:

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

Exercise 102.8.10. You are given

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

Compute

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

□

Exercise 102.8.11. 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}? □

[EXERCISES]

File: formulas-for-sum-of-powers.tex

102.9 Formulas for Sums of Powers

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:

Theorem 102.9.1.

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

□

I won't prove the above formula.

Exercise 102.9.1. Check by hand that the arithmetic sum formula is correct for $n = 1, 2, 3, 4$. □

Using the summation notation you can write the above as

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

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

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

I just subtract 1 to get

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

I then do some algebra to get this:

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

Here's the solution to this problem:

Example 102.9.1. Express $\sum_{i=2}^{n} i$ as a polynomial in descending powers of n.

Solution.

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

□

Now suppose I want

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

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

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

Example 102.9.2. Write

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

as a polynomial in descending powers of n.

Solution.

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

□

Exercise 102.9.2. Compute the sum $1+2+3+\cdots+1000$ by first rewriting it as a summation and then using the arithmetic sum formula. □

Exercise 102.9.3. Compute

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

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

Exercise 102.9.4. Compute

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

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

Exercise 102.9.5. Compute the sum

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

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

Exercise 102.9.6. Write the following as a polynomial in decreasing power of n:

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

□

Exercise 102.9.7. Write the following as a polynomial in decreasing power of n:

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

□

Exercise 102.9.8. 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:

Theorem 102.9.2.

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

I won't prove the above formula.

Exercise 102.9.9. Check by hand that the sum of squares formula is correct for $n = 1, 2, 3, 4$.

Exercise 102.9.10. Compute the sum $1^2 + 2^2 + 3^2 + \cdots + 1000^2$. □

Exercise 102.9.11. Write the following as a polynomial in decreasing power of n:

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

□

Exercise 102.9.12. 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.

File: bubblesort.tex

102.10 Bubblesort: Double for-loops

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,...,0:
    for j = 0 to 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]:
                  t = a[j]
                  a[j] = a[j + 1]
                  a[j + 1] = t
              j = j + 1
              goto LOOP2

ENDLOOP2: i = i - 1
          goto LOOP1

ENDLOOP1:
```

With time for each statement:

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

```
              j = 0                t4
LOOP2:        if j > i:            t5
                  goto ENDLOOP2    t6
              if a[i] < a[j + 1]: t7
                  t = a[i]         t8
                  a[i] = a[j]      t9
                  a[j] = t         t10
              j = j + 1            t11
              goto LOOP2           t12

ENDLOOP2: i = i - 1                t13
          goto LOOP1               t14

ENDLOOP1:
```

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

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

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

ENDLOOP2: i = i - 1                t13  n-1
          goto LOOP1               t14  n-1

ENDLOOP1:
```

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

```
              j = 0                t4   1
LOOP2:        if j > i:            t5   n
                  goto ENDLOOP2    t6   1
              if a[i] < a[j + 1]: t7   (n-1)
```

```
                t = a[i]          t8   (n-1)
                a[i] = a[j]       t9   (n-1)
                a[j] = t          t10  (n-1)
            j = j + 1             t11  (n-1)
            goto LOOP2            t12  (n-1)

ENDLOOPS:
```

All other cases of i values are similar.

So the worse case runtime is

$$\begin{aligned} f(n) = (t_1 + t_3) \\ &+ n \cdot t_2 \\ &+ (n-1) \cdot (t_4 + t_6 + t_{13} + t_{14}) \\ &+ \frac{(n-1)n}{2} \cdot (t_7 + t_8 + t_9 + t_{10} + t_{11} + t_{12}) \\ &+ \frac{(n-1)(n+2)}{2} \cdot t_5 \end{aligned}$$

Rewriting this as a polynomial in n, we get

$$\begin{aligned} f(n) = &\frac{t_5 + t_7 + t_8 + t_9 + t_{10} + t_{11} + t_{12}}{2} \cdot n^2 \\ &+ \left(t_2 + t_4 + t_6 + t_{13} + t_{14} - \frac{t_7 + t_8 + t_9 + t_{10} + t_{11} + t_{12} + t_5}{2} \right) \cdot n \\ &+ (t_1 + t_3 - t_4 - t_6 - t_{13} - t_{14} - 2 \cdot t_5) \end{aligned}$$

In other words the worse case runtime is of the form

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

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

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

For the best case:

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

              j = 0               t4   1+...+1      = n-1
```

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

ENDLOOP2: i = i - 1                   t13  n-1
          goto LOOP1                  t14  n-1

ENDLOOP1:
```

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

Exercise 102.10.1. The following is a variant of the bubblesort:

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

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

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

(b) Compute the big-O of the worse runtime of the above algorithm.

□

Exercise 102.10.2. The following algorithm computes the sum of values in a 2D array `x` of size n-by-n

```
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 worse runtimes are the same. Correct?] □

Exercise 102.10.3. The following algorithm computes the sum of upper triangular values in a 2D array x of size n-by-n:

```
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. □

Exercise 102.10.4. Here's another algorithm:

```
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. □

Exercise 102.10.5. Here's another algorithm:

```
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. □

Exercise 102.10.6. Here's another algorithm:

```
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. □

Exercise 102.10.7. Yet another algorithm:

```
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. □

Exercise 102.10.8. Yet another algorithm:

```
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. □

Exercise 102.10.9. Yet another algorithm:

```
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. □

File: chaining-up-big-O.tex

102.11 Some Shorthand Notation and Computational Tools

The purpose of this section is to give you the first bunch of theorems (computational tools) and a shorthand notation to begin computing big-O. (There will be more powerful theorems later.) I'll prove the theorems later. Right now, I just want to make sure that you know how to use the tools.

For a chain of inequalities, you have seen that if

$$a \leq b \text{ and } b \leq c$$

then

$$a \leq c$$

You have something similar in big-O:

Theorem 102.11.1. *If*

$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n))$$

then

$$f(n) = O(h(n))$$

I'll prove this later. Informally, you see that if $f(n)$ is bounded above by a multiple of $g(n)$ (for large n) and $g(n)$ is bounded above by $h(n)$ (for large n), then $f(n)$ is bounded by a multiple of $h(n)$ (for large n). It's not too shocking.

For instance, if you know that

$$5n = O(3n^2 + 10) \text{ and } 3n^2 + 10 = O(2n^5 + 1)$$

then you can conclude

$$5n = O(2n^5 + 1)$$

Although the above deduction is true, it's kind of lengthy to write. Instead of saying:

$$\text{“}5n = O(3n^2 + 10) \text{ and } 3n^2 + 10 = O(2n^5 + 1)\text{”}$$

you can say

$$\text{“}5n = O(3n^2 + 10) = O(2n^5 + 1)\text{”}$$

They mean the same thing. With this notational shorthand let me repeat what I said earlier ...

If you know that

$$5n = O(3n^2 + 10) = O(2n^5 + 1)$$

then you can conclude

$$5n = O(2n^5 + 1)$$

This is perfectly legal (mathematically speaking) and is understood by all mathematicians and computer scientists.

In general something like

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

or

$$\begin{aligned} f(n) &= O(g(n)) \\ &= O(h(n)) \\ &= O(i(n)) \end{aligned}$$

is the same as

$$\begin{aligned} f(n) &= O(g(n)) \\ g(n) &= O(h(n)) \\ h(n) &= O(i(n)) \end{aligned}$$

Note that the above is not a theorem. It's a notational shorthand. It's similar to the shorthand that

$$a \leq b \leq c$$

means

$$a \leq b \text{ and } b \leq c$$

Here's another fact that you can use:

Theorem 102.11.2.

(a) If $f(n) = O(g(n))$*, then*

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

(b) If $f(n) = O(g(n))$ *then*

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

This basically says that if a multiple of $g(n)$ grows faster than $f(n)$, then the growth of $f(n) + g(n)$ is basically determined by $g(n)$. The second part says this: When computing big-O, you can replace a term by the big-O of that term.

Here's another fact that should not shock you. Since we ignore constants, the following must be true:

Theorem 102.11.3. *If* $c \neq 0$ *is a constant then*

$$cf(n) = O(f(n))$$

This include the obvious fact that

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

Let me collect all the above facts in one place ...

Theorem 102.11.4.

(a) $f(n) = O(g(n)), g(n) = O(h(n)) \implies f(n) = O(h(n))$

(b) $f(n) = O(g(n)) \implies f(n) + g(n) = O(g(n))$

(c) $f(n) = O(g(n)) \implies f(n) + h(n) = O(g(n) + h(n))$

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

(e) $f(n) = O(f(n))$

(f) $f(n) = O(h(n)),\ g(n) = O(h(n)) \implies f(n) + g(n) = O(h(n))$

The above are theorems. The following is a notational convention:

(e) The statement

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

or

$$\begin{aligned} f(n) &= O(g(n)) \\ &= O(h(n)) \end{aligned}$$

means

$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n))$$

□

That's enough for now. I'll give you more powerful computational tools. Let me show you how to use the theorems and notational shorthand.

Example 102.11.1. Given

$$n = O(n^2) \tag{1}$$
$$1 = O(n) \tag{2}$$

prove that

$$10n^2 + 5n + 100 = O(n^2)$$

[Note that the given facts $n = O(n^2)$ and $1 = O(n)$ are in fact true and can be proven easily. I'll prove all the related facts about polynomial big-O in detail later.]

Solution. We make the following deductions that will be used later:

$$10n^2 = O(n^2) \quad \text{by Theorem 4d} \tag{3}$$
$$5n = O(n) \quad \text{by Theorem 4d} \tag{4}$$
$$100 = O(1) \quad \text{by Theorem 4d} \tag{5}$$

Now we prove the given statement:

$$\begin{aligned}
&10n^2 + 5n + 100 \\
&= O(10n^2 + 5n + 100) && \text{by Theorem 4e} \\
&= O(n^2 + 5n + 100) && \text{by (3), Theorem 4c} \\
&= O(n^2 + n + 100) && \text{by (4), Theorem 4c} \\
&= O(n^2 + n^2 + 100) && \text{by (1), Theorem 4c} \\
&= O(2n^2 + 100) \\
&= O(2n^2 + 1) && \text{by (5), Theorem 4c} \\
&= O(2n^2 + n) && \text{by (2), Theorem 4c} \\
&= O(2n^2 + n^2) && \text{by (1), Theorem 4c} \\
&= O(3n^2) \\
&= O(n^2) && \text{by Theorem 4d}
\end{aligned}$$

By Theorem 4a, we conclude that

$$10n^2 + 5n + 100 = O(n^2)$$

□

Note that there is no justification here:

$$
\begin{aligned}
&\ldots \\
&\quad = O(n^2 + n^2 + 100) \qquad\qquad \ldots \\
&\quad = O(2n^2 + 100) \\
&\ldots
\end{aligned}
$$

because that's just basic algebra.

Note also that there is more than one way to reach the the conclusion. (Just like programming or a game of chess.) Can you find another proof? Maybe a shorter (and correct!) one?

Here's another solution to the same problem ...

Example 102.11.2. Given

$$n = O(n^2) \tag{1}$$
$$1 = O(n) \tag{2}$$

prove that

$$10n^2 + 5n + 100 = O(n^2)$$

Solution. First we analyze the terms of the given sum. The first term gives us

$$10n^2 = O(n^2) \qquad \text{by Theorem 4d} \tag{3}$$

The second term gives us

$$\begin{aligned} 5n &= O(n) && \text{by Theorem 4d} \\ &= O(n^2) && \text{by (1)} \end{aligned}$$

which implies

$$5n = O(n^2) \qquad \text{by Theorem 4a} \tag{4}$$

And the third term gives us

$$\begin{aligned} 100 &= O(1) && \text{by Theorem 4d} \\ &= O(n) && \text{by (2)} \\ &= O(n^2) && \text{by (1)} \end{aligned}$$

which implies

$$100 = O(n^2) \qquad \text{by Theorem 4a} \tag{5}$$

Using the above results, we have

$$10n^2 + 5n = O(n^2) \qquad \text{by (3), (4), Theorem 4f} \tag{6}$$

and

$$10n^2 + 5n + 100 = O(n^2) \qquad \text{by (5), (6), Theorem 4f}$$

□

Make sure you study the above two proofs.

One more example ...

Example 102.11.3. Given

$$n = O(n^2) \tag{1}$$
$$n^{1.9} = O(n^2) \tag{2}$$
$$\sin n = O(1) \tag{3}$$
$$\ln n = O(n) \tag{4}$$
$$1 = O(n) \tag{5}$$

show that

$$n^{1.9} + 10\sin n + 5\ln n = O(n^2)$$

Solution. We have

$$\begin{aligned} 10\sin n &= O(\sin n) && \text{by Theorem 4d} \\ &= O(1) && \text{by 3} \end{aligned}$$

Therefore by Theorem 4a

$$10\sin n = O(1) \tag{5}$$

We also have

$$5\ln n = O(\ln n) \qquad \text{by Theorem 4d} \tag{6}$$

Therefore

$$\begin{aligned} & n^{1.9} + 10\sin n + 5\ln n \\ &= O(n^{1.9} + 10\sin n + 5\ln n) && \text{by Theorem 4e} \\ &= O(n^2 + 10\sin n + 5\ln n) && \text{by (2), Theorem 4c} \\ &= O(n^2 + 1 + 5\ln n) && \text{by (2), Theorem 4c} \\ &= O(n^2 + n + 5\ln n) && \text{by (5), Theorem 4c} \\ &= O(n^2 + n^2 + 5\ln n) && \text{by (1), Theorem 4c} \\ &= O(2n^2 + 5\ln n) \\ &= O(2n^2 + n) && \text{by (6), Theorem 4c} \\ &= O(2n^2 + n^2) && \text{by (1), Theorem 4c} \\ &= O(3n^2) \\ &= O(n^2) && \text{by Theorem 4d} \end{aligned}$$

Therefore by Theorem 4a

$$n^{1.9} + 10 \sin n + 5n \ln n = O(n^2)$$

[EXERCISES]

File: bubblesort-again-simplifying-the-for-loop.tex

102.12 Bubblesort again: Computing for-loop runtime quickly

Let me redo the runtime analysis for the bubblesort again. This time I'm going to show you some tricks for simplifying the computation of the runtime.

Remember that we're interested in the big-O of the runtime. So ultimately, we're going to replaced constants (the coefficients of the polynomial) by 1. Notice that I did this simplification at the end after computing the runtime. Now wouldn't it be a good idea to simplify my constants as early as possible?

Here's the algorithm with times for each statement: With time for each statement:

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

              j = 0                t4
LOOP2:        if j > i:            t5
                  goto ENDLOOP2    t6
              if a[i] < a[j + 1]:  t7
                  t = a[i]         t8
                  a[i] = a[j]      t9
                  a[j] = t         t10
              j = j + 1            t11
              goto LOOP2           t12

ENDLOOP2: i = i - 1                t13
          goto LOOP1               t14

ENDLOOP1:
```

I'm going to analyze the worse case again.

The runtime computation in the previous section was rather long. So I'm going to try to simplify it in several ways.

- The first question to ask is this: Can we stick to the for-loop without using goto statements? Converting to the format with goto statement

increases the number of statements to time. Can we avoid that?

- The second question is this: The big-O allows us to simplify the runtime function by throwing away constants (or rather replace them by 1). Why not perform this simplification step earlier? Another simplification done during big-O computation is that we retain the part (i.e. the term) that grows the fastest. Can we do that earlier too? In other words can we using big-O fudging as early as possible instead of at the last stage?

- The third question will only appear after I've addressed the above two. Once the simplification techniques above are addressed, you will see that runtime computations will become a lot simpler if there standard formulas. A minor point is that I will show you that the big-O of certain summations are not not changed if the lower and upper limits of the sum is changed by a constant amount.

Let's start with a simple for-loop ... let's look at the inner loop of the bubble-sort which depends on i:

```
...
              j = 0                t4   1
LOOP2:        if j > i:            t5   i + 1
                  goto ENDLOOP2    t6   1
              if a[i] < a[j + 1]: t7   i
                  t = a[i]         t8   i
                  a[i] = a[j]      t9   i
                  a[j] = t         t10  i
              j = j + 1            t11  i
              goto LOOP2           t12  i

ENDLOOP2:
...
```

The body of the inner loop is this:

```
              if a[i] < a[j + 1]: t7   i
                  t = a[i]         t8   i
                  a[i] = a[j]      t9   i
                  a[j] = t         t10  i
```

Let's call this runtime function $U(i)$. The runtime is

$$\begin{aligned} U(i) &= t_4 + (i+1)t_5 + t_6 + i(t_7 + t_8 + t_9 + t_{10} + t_{11}) \\ &= (t_5 + t_7 + t_8 + t_9 + t_{10} + t_{11})i + (t_4 + t_5 + t_6) \\ &= Ai + B \end{aligned}$$

for constants A and B. Phew! What a pain to compute!

Now I know that the body of the inner loop is constant time. When I'm computing time, I don't really care about the actual statements. So I will give the body a time of `t13` and view the inner loop like this:

```
...
                j = 0                t4   1
LOOP2:          if j > i:            t5   i + 1
                    goto ENDLOOP2    t6   1
                [BODY BLOCK]         t13  i
                j = j + 1            t11  i
                goto LOOP2           t12  i

ENDLOOP2:
...
```

Of course $U(i)$ becomes:

$$\begin{aligned} U(i) &= t_4 + (i+1)t_5 + t_6 + i(t_{13} + t_{11}) \\ &= (t_5 + t_{13} + t_{11})i + (t_4 + t_5 + t_6) \\ &= Ai + B \end{aligned}$$

Well ... the computation sure looks slightly simpler.

So the basic principle here is this: collapse chunks of statements into a block and assign a single time to that block will simplify the computation a little.

Analogous to this is that whenever you have constant expression you might want to rename it with a symbol. For instance in the above computation, you might write:

$$\begin{aligned} U(i) &= t_4 + (i+1)t_5 + t_6 + i(t_{13} + t_{11}) \\ &= A + (i+1)t_5 + iB && \text{for some constants } A,\ B \\ &= \ldots \end{aligned}$$

In this case of course $A = t_4 + t_6$ and $B = t_{13} + t_{11}$.

Let's redo the above in two piece. The computation of $U(i)$ is made up the time to execute the body of the loop and the time taken for the for-loop to control the execution of the body.

So now I want look at the "loop control", i.e., the code that is in charge of repeating the body for the correct number of times:

```
...
              j = 0                 t4   1
LOOP2:        if j > i:             t5   i + 1
                  goto ENDLOOP2     t6   1
              ...
              j = j + 1             t11  i
              goto LOOP2            t12  i

ENDLOOP2:
...
```

Next, I'm going to bunch up `t11` and `t12`:

```
...
              j = 0                     t4   1
LOOP2:        if j > i:                 t5   i + 1
                  goto ENDLOOP2         t6   1
              ...
              j = j + 1; goto LOOP2     t15  i

ENDLOOP2:
...
```

You see immediately that the time taken for the "loop control" is

$$\begin{aligned}\text{LOOP CONTROL TIME} &= t_4 + (i+1)t_5 + t_6 + it_{15} \\ &= A(i+1) + B\end{aligned}$$

where A *includes* the time taken to compute the boolean expression. Note that if the body runs i times, then the boolean expresssion is computed $i+1$ times. If A is a constant

$$\begin{aligned}\text{LOOP CONTROL TIME} &= Ai + (A+B) \\ &= (A+1)i + B \\ &= Ci + B\end{aligned}$$

for constants A, B, and C. Let me summarize this fact and call it TRICK1:

TRICK1. The total runtime of

```
for i = 1, 2, ..., n:
    [body(i)]                          Time to run body(i)
```

is of the form

$$\sum_{i=1}^{n}(\text{Time to execute body}(i)) + An + B$$

where A and B are constants. In general the total runtime of

```
for i = m, 2, ..., n:
    [body(i)]                          Time to run body(i)
```

is of the form

$$\sum_{i=m}^{n}(\text{Time to execute body}(i)) + A(n-m+1) + B$$

(Of course $n-m+1$ is the number of times the body of the loop is executed.)

Back to our bubblesort *without goto statements* and here's the second solution ...

Example 102.12.1. Compute the worse runtime of the bubblesort:

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

Solution. The inner most body

```
        if a[i] > a[i + 1]:
            t = a[i]
            a[i] = a[i + 1]
            a[i + 1] = t
```

has constant runtime, say A. Let $U(i)$ be the runtime of the inner for-loop. Using TRICK1, since the inner loop runs $i+1$ times:

$$U(i) = \sum_{j=0}^{i} A + B(i+1) + C = A(i+1) + B(i+1) + C = Di + E$$

for constants A, B, C, D, E. Using TRICK1 again, since the outer loop runs $n-1$ times the whole algorithm has a runtime of

$$T(n) = \sum_{i=0}^{n-2} U(i) + F(n-1) + G$$

Therefore

$$\begin{aligned}
T(n) &= \sum_{i=0}^{n-2} (Di + E) + F(n-1) + G \\
&= D\sum_{i=0}^{n-2} i + E\sum_{i=0}^{n-2} 1 + F(n-1) + G \\
&= D\frac{(n-2)(n-1)}{2} + E(n-1) + F(n-1) + G \\
&= \left(\frac{D}{2}\right) n^2 + \left(-\frac{3D}{2} + E + F\right) n + (D - E - F + G) \\
&= O(n^2)
\end{aligned}$$

□

Note that the above big-O computation does not require goto statements and massive number of statement counting. The solution is definitely simpler and less error prone.

Not only that, this method also allows you to see very quickly that the best case is also $O(n^2)$. I'll leave that as an exercise for you.

From the computation of the above example, you should see this one coming ...

ADVICE: For nested loops, try to compute the runtime by working from the innermost loop to the outmost.

[Despite what some intro books say, this is *not* always a good idea. But this is something you should try if you do see nested loops.]

Exercise 102.12.1. Compute the best runtime of the bubblesort. □

Exercise 102.12.2. Show that the big-O of the runtime of the following is $O(n)$.

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

□

Exercise 102.12.3. Show that the big-O of the runtime the following is $O(n)$.

```
for i = 0, ..., n - 1:
    x[i] = i
x[0] = 42
for i = n - 1, ..., 2:
    x[i] = x[i - 1] + x[i - 2]
```

□

Exercise 102.12.4. Show that the big-O of the runtime of the following is $O(n)$.

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

□

Exercise 102.12.5. Compute the big-O of the runtime of the following:

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

□

Exercise 102.12.6. Compute the big-O of the runtime of the folllowing:

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

□

Exercise 102.12.7. Compute the big-O of the runtime of the folllowing:

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

Note that in the above expression `(n - 1)/2`, I mean the floor of `(n - 1)/2`. In your computation, you can ignore floors and ceilings, i.e., your computations, the ceiling or floor of n/d can be replaced by n/d. □

Exercise 102.12.8. Compute the big-O of the runtime of the following:

```
for i = 0, ..., n - 1:
```

```
    for j = i, ..., (n - 1) / 2:
        x[i] = x[i] * x[i]
```

□

Exercise 102.12.9. What is the big-O of runtime of the following:

```
for i = 0, 1, 2, ..., n - 1:
    if n is even:
        x[i] = 0
    else:
        x[i] = 1
```

Exercise 102.12.10. What is the best and worse case big-O of the following:

```
for i = 0, 1, 2, ..., n - 1:
    if x[i] is even:
        for j = i + 1, ..., n - 1:
            x[i] = x[i] + x[j]
    else:
        x[i] = 0
```

Exercise 102.12.11. What is the big-O of the following:

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

Exercise 102.12.12. What is the big-O of the following:

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

Exercise 102.12.13. What is the big-O of the runtime of the following:

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

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

Exercise 102.12.14. What is the big-O of the best and worse case of the following:

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

Exercise 102.12.15. What is the big-O of the best and worse case of the following:

```
for i = 0, 1, 2, ..., n - 2:
    x[i] = x[i + 1]
    for j = 0, 1, 2, ..., i:
        if j is even:
            x[j] = x[i] + 1
        else:
            x[j] = x[i] - 1
        for k = n - 1, n - 2, ..., j:
            x[k] = x[j] + x[i]
```

Of course in general the runtime of the body of the for loop might not be a constant. For instance:

```
for i = 0, 1, 2, 3, ..., n - 1:
    f(i, n)
```

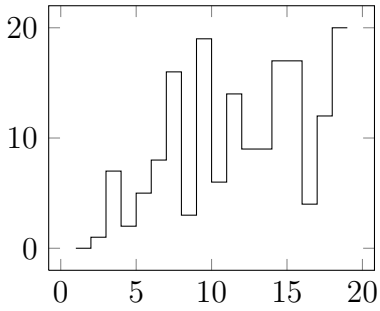
where f is a function.

Note that the case where the runtime of the innermost body of a loop is constant (or at least easy to compute) and where the number of times the body is executed is simple (example: i runs from 0 to $n-1$), the runtime of the whole algorithm is usually pretty easy to compute.

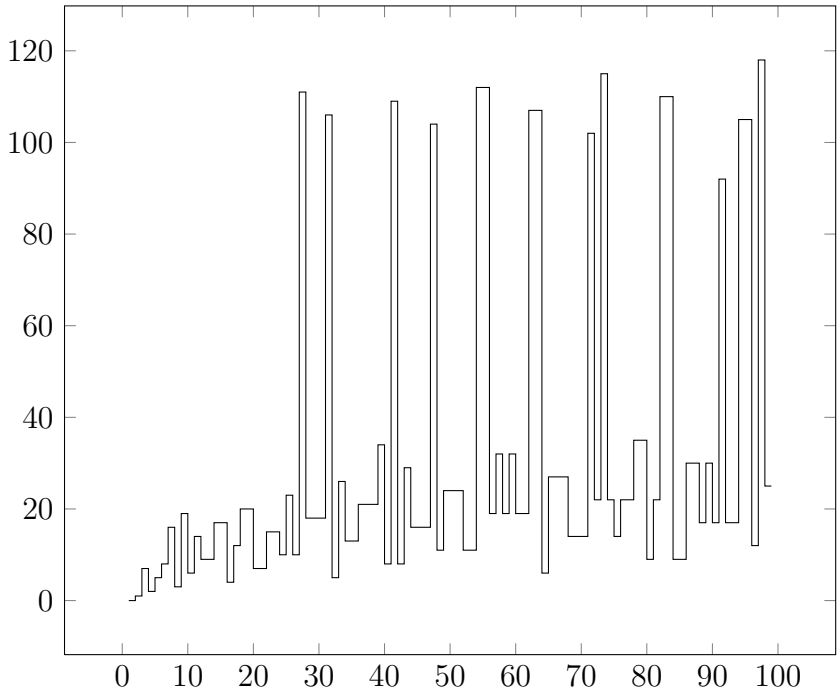
However if you do not know how many times the body of a loop is executed, then it's not so simple. Here's an example:

```
while n > 1:
    if n is even:
        n = n / 2
    else:
        n = 3 * n + 1
```

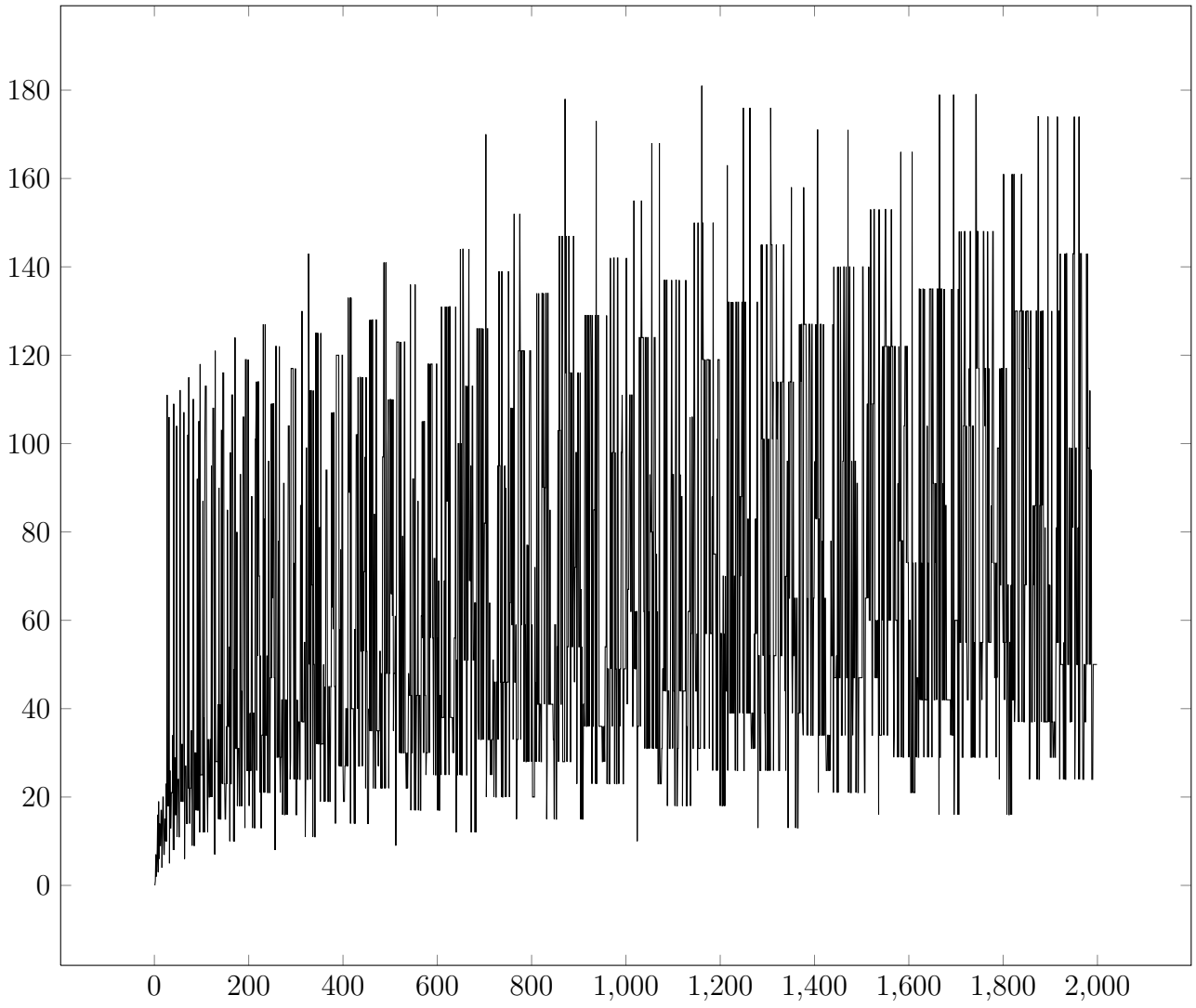
You can try to compute it several cases by hand. Here's the plot of the number of iterations of the body of the while loop for $n = 1, ..., 19$:



Not so clear, right? Here's the plot for $n = 1, ..., 99$:



It seems that the number of times the body of the while loop executes is always ≤ 120. Not quite. Here's the plot for $n = 1, ..., 1999$:



I hope that it's clear at least that the number of iterations for $n = 2^k$ is k. These are the optimistic cases. Do you see why?

I'll leave it to you to play around with the above while-loop.

File: bubblesort-again-summation.tex

102.13 Bubblesort again: Summation and big-O

In the section on speeding up computing the runtime on the bubblesort, I show that for the runtime for the following for-loop

```
for i = 1, ..., n:
    [body(i)]
```

is

$$T(n) = \sum_{i=1}^{n} [\text{Time to run body(i)}] + An + B$$

Recall that using this idea I computed the worse case runtime of the bubblesort ...

Example 102.13.1. Compute the worse runtime of the bubblesort:

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

Solution. The inner most body

```
        if a[i] > a[i + 1]:
            t = a[i]
            a[i] = a[i + 1]
            a[i + 1] = t
```

has constant runtime, say A. Let $U(i)$ be the runtime of the inner for-loop. Using TRICK1, since the inner loop runs $i+1$ times:

$$U(i) = \sum_{j=0}^{i} A + B(i+1) + C = A(i+1) + B(i+1) + C = Di + E$$

for constants A, B, C, D, E. Using TRICK1 again, since the outer loops runs $n-1$ times the whole algorithm has a runtime of

$$T(n) = \sum_{i=0}^{n-2} U(i) + F(n-1) + G$$

for constants F and G. Therefore

$$\begin{aligned}
T(n) &= \sum_{i=0}^{n-2} (Di + E) + F(n-1) + G \\
&= D\sum_{i=0}^{n-2} i + E\sum_{i=0}^{n-2} 1 + F(n-1) + G \\
&= D\frac{(n-2)(n-1)}{2} + E(n-1) + F(n-1) + G \\
&= \left(\frac{D}{2}\right) n^2 + \left(-\frac{3D}{2} + E + F\right) n + (D - E - F + G) \\
&= O(n^2)
\end{aligned}$$

□

Now I want to simplify in a different way ...

Notice that we compute the big-O at the last step. Can we simplify big-O-wise earlier? For instance the expression for $U(i)$ is

$$U(i) = Di + E$$

As a function of i, of course I can say

$$U(i) = Di + E = O(i)$$

(This is the big-O of i, not n.) It *does* allow us to throw away D and E.

But now the expression for $T(n)$ is this

$$T(n) = \sum_{i=0}^{n-2} U(i) + Fn + G$$

How does one substitute a *big-O expression* into an expression??? Can we do this:

$$T(n) = \sum_{i=0}^{n-2} i + Fn + G$$

i.e., replace $U(i)$ by the big-O of $U(i)$? Of course technically speaking the above is not an equality "=" anymore. Since it involves saying "$T(n)$ looks like this with big-O fudging", it's more accurate to say this:

$$T(n) = O\left(\sum_{i=0}^{n-2} i + Fn + G\right)$$

i.e., is the following true:

$$\sum_{i=0}^{n-2} (Di + E) + Fn + G = O\left(\sum_{i=0}^{n-2} i + Fn + G\right)$$

Or more generally we ask if the following is true:

In the computation of the big-O of a sum, I replace the terms of the sum by their big-O's? In other words, is the following true:

$$f(i) = O(g(i)) \implies \sum_{i=1}^{n} f(i) = O\left(\sum_{i=1}^{n} g(i)\right) \quad ???$$

Theorem 102.13.1. *If*

$$f(i) = O(g(i))$$

then

$$\sum_{i=1}^{n} f(i) = O\left(\sum_{i=1}^{n} |g(i)|\right)$$

Proof. Since $f(i) = O(g(i))$, there exists C and N such that for $i \geq N$,

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

Summing up, we have

$$\sum_{i=N}^{n} |f(i)| \leq C \sum_{i=N}^{n} |g(i)|$$

Let $D = \sum_{i=1}^{N-1} f(i)$. Then

$$\sum_{i=1}^{n} |f(i)| = D + \sum_{i=N}^{n} |f(i)| \leq D + C \sum_{i=N}^{n} |g(i)| \leq D + C \sum_{i=1}^{n} |g(i)|$$

If $\sum_{i=1}^{n} |g(i)|$ is a bounded function of n, this function must be $O(1)$. From the above inequality, $\sum_{i=1}^{n} |f(i)|$ must also be $O(1)$. Hence $\sum_{i=1}^{n} f(i)$ must be big-O of $O(1)$ as well and the result is proved.

If $\sum_{i=1}^{n} |g(i)|$ is an unbounded function of n, then there is some M such that for $n \geq M$,

$$\sum_{i=1}^{n} |g(n)| \geq D$$

Hence for $n \geq \max(N, D)$, we have

$$\sum_{i=1}^{n} |f(i)| \leq D + C \sum_{i=1}^{n} |g(i)| \leq (1 + C) \sum_{i=1}^{n} |g(i)|$$

Hence

$$\left| \sum_{i=1}^{n} f(i) \right| \leq \sum_{i=1}^{n} |f(i)| \leq (1 + C) \sum_{i=1}^{n} |g(i)|$$

Therefore

$$\sum_{i=1}^{n} f(i) = O\left(\sum_{i=1}^{n} |g(i)|\right)$$

□

Example 102.13.2. For instance if

$$f(i) = 5i + 10 = O(i)$$

then

$$\sum_{i=1}^{n} f(i) = O\left(\sum_{i=1}^{n} i\right)$$

Example 102.13.3. Let

$$f(i) = 10i + 42 \sin i = O(i)$$

then

$$\sum_{i=1}^{n} f(i) = O\left(\sum_{i=1}^{n} i\right)$$

I'm going to call the above TRICK2.

Now let's use the trick to replace big-O in the summation in the bubblesort ...

Example 102.13.4. Compute the worse runtime of the bubblesort:

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

Solution. The inner most body

```
        if a[i] > a[i + 1]:
            t = a[i]
            a[i] = a[i + 1]
            a[i + 1] = t
```

has constant runtime, say $A = O(1)$. Let $U(i)$ be the runtime of the inner loop. We have

$$\begin{aligned} U(i) &= \sum_{j=0}^{i} A + B(i+1) + C && \text{by TRICK1} \\ &= O\left(\sum_{j=0}^{i} 1 + B(i+1) + C\right) && \text{by TRICK2} \\ &= O\left((i+1) + B(i+1) + C\right) \\ &= O\left((1+B)i + (1+B+C)\right) \\ &= O\left(i\right) \end{aligned}$$

for constants B, C. The total runtime, $T(n)$, is:

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-2} U(i) + Fn + G && \text{by TRICK1} \\ &= O\left(\sum_{i=0}^{n-2} i + Fn + G\right) && \text{by TRICK2} \\ &= O\left(\frac{(n-2)(n-1)}{2} + Fn + G\right) \\ &= O\left(\frac{1}{2}n^2 - \frac{3}{2}n + 1 + Fn + G\right) \\ &= O(n^2) \end{aligned}$$

□

Note that in the computation of the big-O of $T(n)$ for the worse case of the bubblesort, you have to compute the big-O of a sum:

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

I have three more tricks to show you.

First of all I want to show you that you can frequently change the lower and upper limits of your sum by a constant amount without affecting the big-O, i.e.,

$$\sum_{i=0}^{n} i^2 = O\left(\sum_{i=1}^{n} i^2\right)$$

and

$$\sum_{i=0}^{n-1} i^2 = O\left(\sum_{i=1}^{n} i^2\right)$$

The second technique is that, since we're going to replace the term of our sum by their big-O, for instance

$$\sum_{i=1}^{n}(5i^2 + 7i + 10) = O\left(\sum_{i=1}^{n} i^2\right)$$

This means that if we want to compute big-O of for-loops quickly, we had better be able to compute the big-O of sums like

$$\sum_{i=1}^{n} i^3$$

or even

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

(for a constant k) or

$$\sum_{i=1}^{n} i^2 \ln^3 i$$

or in general the sums of standard functions used in describing big-O classes. Such a table of formulas of summations will allow us to compute runtimes quickly just like a table of differentiation rules of standard functions will allow you to compute the derivatives of general expressions quickly. I will deal with

the big-O of sums of the form $\sum_{i=1}^{n} i^k$ in the next section since this is the one that I will need for the bubblesort.

The last thing is that I will show you that the loop control time in the runtime of the for-loop is actually redundant.

File: bubblesort-sum-of-monomials.tex

102.14 Bubblesort again: big-O of Sums

The next theorem tells us two things: you can compute the big-O of sums of i^k and that change the limits of summations by a constant amount does not change the big-O. I'll do the first fact in this section.

Theorem 102.14.1. *If $k \geq 0$ is an integer, then*

$$\sum_{i=1}^{n} i^k = O(n^{k+1})$$

You already know that

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

therefore the big-O is $O(n^2)$. So it's not too surprising that for higher powers:

$$\sum_{i=1}^{n} i^k, \quad k > 1$$

there are actually exact formulas. However we can get away without these exact formulas (which are very complicated) since we only care about big-O. Here's how you can prove the theorem.

Proof. For each i in the sum,

$$1 \leq i \leq n$$

Therefore

$$i^k \leq n^k$$

We conclude that

$$\begin{aligned}\sum_{i=1}^{n} i^k &\leq \sum_{i=1}^{n} n^k \\ &= n^k \sum_{i=1}^{n} 1 \\ &= n^k n \\ &= n^{k+1}\end{aligned}$$

Hence

$$\sum_{i=1}^{n} i^k = O(n^{k+1})$$

□

Example 102.14.1. For instance the theorem tells us that

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

$$\sum_{i=1}^{n} i^2 = O(n^3)$$

$$\sum_{i=1}^{n} i^{100} = O(n^{101})$$

□

Exercise 102.14.1. Show that $\sum_{i=1}^{n} \ln i = O(n \ln n)$ □

[MORE EXAMPLES + EXERCISES]

File: changing-the-limits-of-a-summation.tex

102.15 Bubblesort again: Changing the Limits of a Summation

In the previous section, I showed you the following big-O formula:

$$\sum_{i=1}^{n} i^k = O(n^{k+1})$$

But if you look at the big-O computation of the worse case runtime for the bubblesort, you see that you the sum that appear is actually this:

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

RANT ... RANT ... RANT:

- The lower limit of the summation is 0 and not 1!!!
- The upper limit of the summation is $n-2$ and not n!!!

AARRGHHH!!! OK, don't panic ... let's see what I can do about them ...

Theorem 102.15.1. *From the above, we know that if $k \geq 0$ is an integer, then*

$$\sum_{i=1}^{n} i^k = O(n^{k+1})$$

The above is still true if the sum

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

is replaced by a sum of the form

$$\sum_{i=c}^{n-d}$$

where c and d are constants.

Example 102.15.1. For instance the theorem tells us that

$$\sum_{i=0}^{n} i = O(n^2)$$
$$\sum_{i=3}^{n-1} i^2 = O(n^3)$$
$$\sum_{i=2}^{n+2} i^{100} = O(n^{101})$$

□

Now I'm going to prove that changing the lower limit of the summation does not affect the big-O. In fact this is true for summation of *any* type of terms, not just i^k. The main idea is that the first few terms of a summation is of course ... *constant!!!*

For instance for the case where the terms are i^7 and the lower limit of the summation is changed to 4, I have

$$\begin{aligned}\sum_{i=4}^{n} i^7 &= \sum_{i=1}^{n} i^7 - 1^7 - 2^7 - 3^7 \\ &= \sum_{i=1}^{n} i^7 - C\end{aligned}$$

where $C = -1^7 - 2^7 - 3^7$ is a constant. Of course adding a constant to $\sum_{i=4}^{n} i^7$ does not change its big-O.

This gives me the hint that if I should look at an expression of the form

$$F(n) = G(n) + C$$

I want to be able to say:

$$F(n) = G(n) + C \implies F(n) = O\left(G(n)\right)$$

Can I? If not, what condition should I put on $F(n)$ and $G(n)$ to be able to say that?? Well ... if $G(n)$ is unbounded, it beats C after some point, then for $n \geq N$, then

$$G(n) \geq C$$

and then

$$F(n) = G(n) + C \leq G(n) + G(n) = 2G(n)$$

That works!!! But what if $G(n)$ is bounded? Well, then $G(n) = O(1)$ and

$$F(n) = O(G(n) + C) = O(1 + 1) = O(1)$$

and $F(n)$ and $G(n)$ has the same big-O again. So whether $G(n)$ is bounded or unbounded, we have the same result.

Theorem 102.15.2. *If c is a constants then*

$$\sum_{i=c}^{n} f(i) = O\left(\sum_{i=1}^{n} f(i)\right)$$

Proof. There is nothing to prove is $c = 1$. If $c > 1$, then

$$\sum_{i=c}^{n} f(i) = \sum_{i=1}^{n} f(i) - (f(1) + f(2) + \cdots + f(c-1))) = \sum_{i=1}^{n} f(i) + C$$

where C is a constant. And if $c < 1$, then

$$\sum_{i=c}^{n} f(i) = \sum_{i=1}^{n} f(i) + (f(c) + f(c+1) + \cdots + f(0))) = \sum_{i=1}^{n} f(i) + D$$

where D is a constant. Therefore in both cases, $\sum_{i=c}^{n} f(i)$ and $\sum_{i=1}^{n} f(i)$ differ by a constant. From our observation above, these two functions of n have the same big-O (or n). □

Now what about the upper limit? In the case where $f(i) = i^k$ for a constant k, if the upper limit is $n - 2$ instead of n:

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

Since $\sum_{i=1}^{n} i^k = O(n^{k+1})$, $-(n-1)^k = O(n^k)$, and $-n^k = O(n^k)$, we have

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

Now let me show you how to use the above to redo the runtime of our bubblesort ...

Example 102.15.2. Compute the big-O of the runtime of the bubblesort:

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

Solution. The runtime of the body of the inner loop is constant time say $A = O(1)$. The runtime of the inner loop is

$$U(i) = \sum_{j=0}^{i} A + B(i+1) + C$$

for constants B and C. Therefore

$$U(i) = O\left(\sum_{j=0}^{i} A + B(i+1) + C\right) = O\left(\sum_{j=0}^{i} 1 + i\right) = O(i+i) = O(i)$$

For the whole algorithm, the runtime is:

$$T(n) = \sum_{i=0}^{n-2} U(i) + Cn + D$$

Therefore

$$T(n) = O\left(\sum_{i=0}^{n-2} U(i) + Cn + D\right) \quad = O\left(\sum_{i=0}^{n-2} i + n\right) = O\left(n^2 + n\right) \quad = O(n^2)$$

□

[TODO: Talk about the case where the limits are floor/ceiling of proportions of n]

File: for-loop-throw-away-loop-control.tex

102.16 Bubblesort again: Throwing Away The Loop Control

One last trick that simplifies the computation of the runtime. Recall that the runtime for the for-loop

```
for i = 1, ..., n:
    [body(i)]
```

is this

$$T(n) = \sum_{i=1}^{n} [\text{Time to run body(i)}] + An + B$$

Recall that the time

$$An + B$$

is due to the time taken to perform the loop control on the execution of the body of the for-loop and the actual total time to execute the body is

$$\sum_{i=1}^{n} [\text{Time to run body(i)}]$$

Notice that when you compute the big-O of the summation

$$\sum_{i=1}^{n} [\text{Time to run body(i)}]$$

the runtime of the body(i) is at least constant time and

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

And the big-O of $An + B$ is n. Therefore in terms of big-O computation of $T(n)$,

$$\sum_{i=1}^{n} [\text{Time to run body(i)}]$$

would already include the big-O of $An + B$. Therefore, $An + B$ is redundant. (I fudged on some details here, but the idea is correct.)

The only time when the above is not quite right is when the runtime of body(i) is 0 ... which should *not* happen if you're *not* writing some ridiculous and useless code like this (in C++ notation):

```
for (int i = 0; i < n; i++)
{
}
```

I will assume that such bizarre cases will not happen from now on. If so, I conclude the following ...

The runtime for the for-loop

```
for i = 1, ..., n:
    [body(i)]
```

is this

$$T(n) = \sum_{i=1}^{n} [\text{Time to run body(i)}] + An + B$$

and the *big-O* of $T(n)$ is

$$T(n) = O\left(\sum_{i=1}^{n} [\text{Time to run body(i)}]\right)$$

So here's the last version of the runtime computation of the worse case of the bubblesort ...

Example 102.16.1. Compute the big-O of the runtime of the bubblesort:

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

Solution. The runtime of the body of the inner loop is constant time, say $A = O(1)$. Therefore the runtime of the inner loop, $U(i)$, is

$$U(i) = O\left(\sum_{j=0}^{i} A\right) = O\left(\sum_{j=0}^{i} 1\right) = O(i)$$

For the whole algorithm, the runtime $T(n)$ is:

$$T(n) = O\left(\sum_{i=0}^{n-2} U(i)\right) = O\left(\sum_{i=0}^{n-2} i\right) = O\left(n^2\right)$$

□

[EXERICSES]

[PUT THIS SOMEWHERE ELSE]

Exercise 102.16.1. Show that $\sum_{i=1}^{n} \ln i = O(n \ln n)$ □

File: formal-definition-of-big-O.tex

102.17 Formal Definition of big-O

Here's the big-O definition again:

Let $f(n)$ and $g(n)$ be two functions of n. We write

$$f = O(g)$$

and say "$f(n)$ is big-O of $g(n)$" if

$$\text{there exist } C \text{ and } N \text{ such that } |f(n)| \leq C|g(n)| \text{ for } n \geq N$$

So far I've been using graphs to give you an understanding of the big-O definition. However the clause

$$\text{"... for } n \geq N\text{"}$$

requires us to say something about arbitrarily large values for n and of course we can't draw infinitely large graphs.

Now I'm going to use math to show $f(n) = O(g(n))$. Occasionally I'll be using graphs as a tool to help. But hard facts must be proven by math.

Let's do some examples.

File: commercial-break-review-of-inequalities.tex

102.18 Commercial Break: Review of Inequalities

Here are some facts involves inequalities:

(b) If $x < y$ and $y < z$, then $x < z$.

(c) If $x < y$, then $x + c < y + c$.

(d) If $0 \leq c$, then $x - c \leq x \leq x + c$

(e) If $x < y$ and $c > 0$, then $cx < cy$.

(f) If $x < y$ and $c < 0$, then $cx > cy$.

Here are some facts involving the absolute value:

(a) By definition

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}$$

This of course means that if $x \geq 0$, then $|x| = x$.

(b) $|x| \geq 0$

(c) $|xy| = |x||y|$

(d) $|x + y| \leq |x| + |y|$

(e) $||x| - |y|| \leq |x - y|$

(e) If $0 \leq c < 1$, then $c|x| \leq |x|$.

(f) If $1 \leq c$, then $|x| \leq c|x|$.

(g) If $|x| \leq c$, then $-c \leq x \leq c$.

File: another-way-to-look-at-big-O.tex

102.19 Another Way to Look at big-O

Recall that if I say "$f = O(g)$" I mean you can find C and N such that for $n \geq N$,

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

In the above, $O(g)$ is purely symbolic. Like I said, when you see "$f = O(g)$", you just think of "there exist C and N such that for $n \geq N$, $|f(n)| \leq C \cdot |g(n)|$".

Now I'm going to define $O(g)$ as a *set* like this:

$$O(g) = \left\{ f \;\middle|\; \text{there is some } N \text{ and } C \text{ such that } |f(n)| \leq C \cdot |g(n)| \right\}$$

With this definition, I can say

$$f \in O(g)$$

This would be the say as saying

$$f = O(g)$$

in our previous definition. Whether you write "$f = O(g)$ or "$f \in O(g)$", they mean the same thing, i.e.,

$$\text{there are } C \text{ and } N \text{ such that for } n \geq N,\ |f(n)| \leq C \cdot |g(n)|$$

Besides seeing statements like

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

or

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

you might see something like this:

$$f(n) = 3n^2 + 4n + 5 + 6 \ln n = 3n^2 + O(n)$$

or

$$f(n) = 3n^2 + 4n + 5 + 6 \ln n \in 3n^2 + O(n)$$

What does this mean:

$$3n^2 + O(n)$$

Seems like the author is trying to add a function $3n^2$ with a set $O(n)$!!!

Basically the author is trying to say that he cares only about $3n^2$ and the *smaller terms* are bunched up as *some* function in $O(n)$, i.e.,

$$f(n) = 3n^2 + 4n + 5 + 6\ln n = 3n^2 + f_1(n) \text{ for some } f_1 \in O(n)$$

If he wants to be more accurate he can also say

$$f(n) = 3n^2 + 4n + 5 + 6\ln n = 3n^2 + 4n + O(\ln n)$$

he meant that $f(n)$ is $3n^2 + 4n$ together with a smaller function $f_1(n)$ which is in $O(\ln n)$:

$$f(n) = 3n^2 + 4n + 5 + 6\ln n = 3n^2 + 4n + f_1(n) \text{ for some } f_1(n) \in O(\ln n)$$

[PUT STUFF BELOW IN ANOTHER SECTION]

In general let's look at an expression of the form

$$f_1(n) + O(g_1(n)) = f_2(n) + O(g_2(n))$$

First of all let's define $f_1(n) + O(g_1(n))$. This is the sum of a function and a set. Basically you just add f_1 to all the functions in $O(g_1(n))$. More generally if you have a function f and a set of functions S, then we define

$$f + S = \{f + g \,|\, g \in S\}$$

With this definition,

$$f_1(n) + O(g_1(n)) = f_2(n) + O(g_2(n))$$

is the same as saying

$$\left\{ f_1 + g \;\middle|\; g \in O(g_1(n)) \right\} \subseteq \left\{ f_2 + g \;\middle|\; g \in O(g_2(n)) \right\}$$

Remember that when it comes to O and Θ equations, $=$ always means $\subseteq$!!

If you stop to think about it, not only can you define $f+S$ where f is a function

and S is a set of functions, you can also define $S + f$ as $\{g + f \mid g \in S\}$ too. Of course since $f + g = g + f$, we have $f + S = S + f$. Right? Furthermore, you can also define fS and Sf.

Exercise 102.19.1. Let S, T, U be sets of functions and f be a function. Equality here is really equality and not $\in$ or $\subseteq$.

1. Define $S + T$ and prove that $S + T = T + S$ (duh) and $(S + T) + U = S + (T + U)$ (duh). Therefore we can write $S + T + U$ without fear of ambiguity or lawsuits or flamemail. Of course with your definition we can just define $f + S$ as $f + S$.

2. Define fS and Sf. Prove that $fS = fS$ (duh).

3. Define ST. Prove that $ST = TS$ and $(ST)U = S(TU)$ (two duhs). So again we can write STU.

Exercise 102.19.2. Try this out yourself: What does this means

$$f + O(g) + O(h) = a + O(b) + O(c)$$

where f, g, h, a, b, c, are functions. Don't forget that in this case $=$ means $\subseteq$.

File: basic-facts-about-big-O.tex

102.20 Basic Facts about Big-O

Here are some basic facts you should know.

Theorem 102.20.1. *Let f, g, h, f_i, and g_i be functions and $c \neq 0$ be a constant.*

(a) *If $|f(n)| \leq |g(n)|$ for all $n \geq 1$, then $f = O(g)$.*

(b) $f = O(f)$

(c) *If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.*

(d) $f = O(cg)$ *iff* $f = O(g)$

(e) $cf = O(g)$ *iff* $f = O(g)$

(f) *If $f_i = O(g_i)$ $(i = 1, 2)$ then*

$$f_1 + f_2 = O\big(\max(|g_1|, |g_2|)\big), \quad f_1 f_2 \in O(g_1 g_2)$$

(g) $f_i = O(g)$ $(i = 1, 2) \implies f_1 + f_2 = O(g)$

(h) $f \in O(g) \implies O(f) \subseteq O(g)$.

Here are some examples on how to use the above theorem.

Example 102.20.1. Prove that $n^3 + 1 = O(n^3)$.

Solution. We have

$$n^3 = O(n^3) \qquad \text{by Theorem 1(b)} \tag{1}$$

Also,

$$\begin{aligned} & 1 \leq |n^3| \text{ for } n \geq 1 \\ \therefore \quad & 1 = O(n^3) \qquad \text{by Theorem 1(a)} \end{aligned} \tag{2}$$

Using (1) and (2),

$$n^3 + 1 = O(n^3) \qquad \text{by Theorem 1(g)}$$

□

Example 102.20.2. Prove that $7n^5 - 3n = O(n^5)$.

Solution. We have

$$\begin{aligned} 7n^5 &= O(7n^3) && \text{by Theorem 1(a)} \\ \therefore\ 7n^5 &= O(n^3) && \text{by Theorey 1(d)} \end{aligned} \tag{1}$$

Also,

$$\begin{aligned} &|-3n| \leq 3n^3 \text{ for } n \geq 1 \\ \therefore\ &-3n = O(3n^3) && \text{by Theorem 1(a)} \\ \therefore\ &-3n = O(n^3) && \text{by Theorem 1(d)} \end{aligned} \tag{2}$$

From (1) and (2), we have

$$\begin{aligned} 7n^5 - 3n &= 7n^5 + (-3n) \\ &= O(n^3) && \text{by Theorem 1(g)} \end{aligned}$$

□

Exercise 102.20.1. Show that $n^2 = O(n^3)$. □

Exercise 102.20.2. Show that $5n^7 + 42n^2 - 5 = O(n^7)$. □

Exercise 102.20.3. Show that $5n^7 + \sin((2n+1)^2\pi/4) = O(n^7)$. □

Exercise 102.20.4. Show that $5n + 0.5^n = O(n)$. □

Exercise 102.20.5. Show that $(-1)^n n^2 + 5 = O(n^2)$. □

Exercise 102.20.6. Show that $e^{-n} \sin n = O(1)$. □

Exercise 102.20.7. Investigate the following question: Suppose $f = O(g)$. Is it true that $h^f \in O(h^g)$? Of course the function h^f is just given by $(h^f)(n) = h(n)^{f(n)}$. □

File: the-big-O-classes-polynomial.tex

102.21 The big-O Classes: Polynomials

I have already talked about constant functions

$$f(n) = c = O(1)$$

and polynomials of degree 1:

$$f(n) = an + b = O(n)$$

(see previous sections).

It's time to look at polynomial functions such as

$$f(n) = 5n^{42} + n^5 + 1234n - 1$$

and figure out a simple function $g(n)$ such that $f(n) = O(g(n))$. In fact in the previous section, I talked about cases such as

$$f(n) = n^3 + 1 = O(n^3)$$

In this section, I'm going to settle the big-O question of *all* polynomials.

Theorem 102.21.1.

(a) If $1 \leq a \leq b$, then $n^a = O(n^b)$.

(b) If p is a polynomial of degree d, then $p = O(n^d)$

(c) If $c > 0$ is a constant, then $n^d = O(c^n)$.

Proof

(a) For $n \geq 1$, since $b - a > 0$,

$$n^{b-a} \geq 1$$

Therefore

$$n^a \leq n^a \cdot n^{b-a} = n^b$$

i.e.,

$$|n^a| \leq C|n^b|$$

where $C = 1$ and $n \geq N$ where $N = 1$. Hence $n^a = O(n^b)$.

(b) If p is a polynomial of degree d, then p is of the form

$$p = c_0 + c_1 n + c_2 n^2 + \cdots + c_d n^d$$

where c_i are constants. From (a),

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

for $i = 0, 1, 2, ..., d$. From the Theorem 1e:

$$c_i n^i = O(n^d)$$

From Theorem 1g:

$$c_0 + \cdots + c_d n^d = O(n^d)$$

□

The following are some big-O asymptotic classes:

$$O(1) \subset O(n) \subset O(n^2) \subset O(n^3) \subset \cdots$$

Every polynomial is of one of the above complexity classes. For instance

$$f(n) = 5n^3 - 100n^2 + \frac{1}{2}n + 23 = O(n^3)$$

Of course this $f(n)$ is also $O(n^r)$ for $r > 3$.

You might have a function that looks like this:

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

i.e., the powers need not be integers. In this case

$$f(n) = 7n^{2.7} + 5n^2 + 1 = O(n^{2.7})$$

since $n^{2.7}$ grows faster that n^2.

In general if $1 \leq r < s$, then

$$O(n^r) \subset O(n^s)$$

Note that the inclusion is strict, i.e.,

$$O(n^r) \neq O(n^s)$$

in other words there is a function $f(n)$ in $O(n^s)$ that is not in $O(n^r)$.

File: average-time.tex

102.22 Average Time

Recall that we have this is our linear search algorithm:

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

Here's the algorithm with times:

```
                                time
         index = -1             t1
         i = 0                  t2
LOOP:    if i == n:             t3
             goto ENDLOOP       t4
         if x[i] is target:     t5
             index = i          t6
             goto ENDLOOP       t7
         i = i + 1              t8
         goto LOOP              t9
ENDLOOP:
```

Recall that the best and worse times are

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

I have also computed the average runtime for the case where the `target` is in the array at index position $k = 0, 1, 2, ..., n-1$ with equal likelihood.

$$T_a(n) = O(n)$$

Now I'm going to change the average runtime scenario a little and compute a *new* average runtime.

I'm now going to assume the besides the `target` appear at index position $0, 1, 2, ..., n-1$, I also want to include the case where `target` is *not* in the array `x` at all. So altogether, in the new average runtime, we have $n+1$ cases.

I want to assume that all cases are *equally likely*. Let's call out new average runtime $T'_a(n)$.

First let me recall that if I write T_k for the runtime for the case where `target` is at index position k and T_n is the runtime where the `target` is not in the array `x`, then they are of the form

$$\begin{aligned} T_k &= A + Bk, \quad (k = 0, 1, 2, ..., n-1) \\ T_n &= C + Dn \end{aligned}$$

Of course you know now that the constants are not that important when we are computing big-O. But anyway, referring to an earlier section we have:

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

Recall that our *old* average runtime is

$$T_a(n) = \frac{1}{n}\left(T_0 + T_1 + \cdots + T_{n-1}\right) = O(n)$$

This averages over n cases.

For our new average runtime function we have

$$T'_a(n) = \frac{1}{n+1}\left(T_0 + T_1 + \cdots + T_{n-1} + T_n\right)$$

Note that we're now averaging over $n+1$ cases. Now I will compute the big-O

of our new average runtime:

$$\begin{aligned}
T'_a(n) &= \frac{1}{n+1}\left((A+B\cdot 0)+(A+B\cdot 1)+\cdots(A+B\cdot(n-1))+C+Dn\right)\\
&= \frac{1}{n+1}\left(nA+B\cdot(0+1+\cdots+(n-1))+C+Dn\right)\\
&= \frac{1}{n+1}\left(nA+B\cdot\frac{n(n-1)}{2}+C+Dn\right)\\
&= \frac{1}{n+1}\left(\left(\frac{B}{2}\right)n^2+\left(A-\frac{1}{2}B+D\right)n+C\right)\\
&= \left(\frac{B}{2}\right)\frac{n^2}{n+1}+\left(A-\frac{1}{2}B+D\right)\frac{n}{n+1}+C\frac{1}{n+1}
\end{aligned}$$

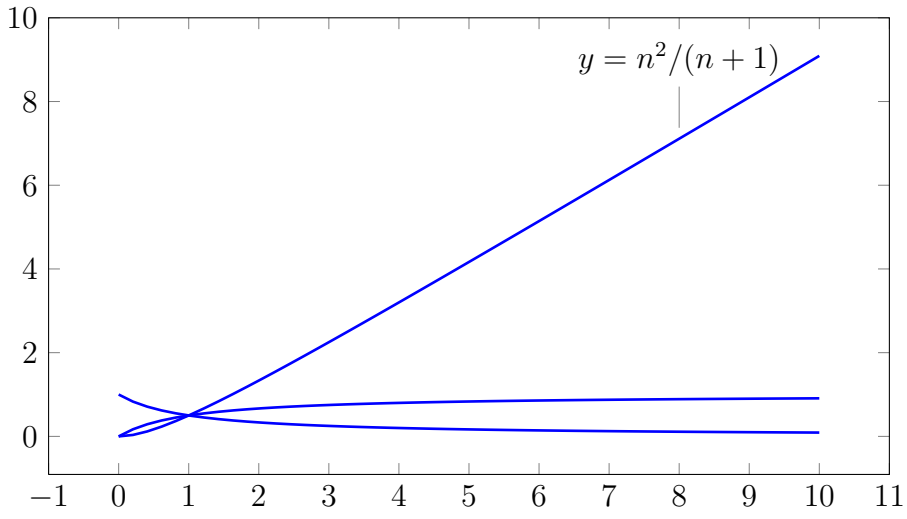
Clearly, of the 3 functions of n,

$$\frac{n^2}{n+1},\quad \frac{n}{n+1},\quad \frac{1}{n+1}$$

the one that grows the fastest is

$$\frac{n^2}{n+1},$$

Here's a plot of the three functions:



Although at this point I can say

$$T'_a(n) = O\left(\frac{n^2}{n+1}\right)$$

I prefer to bound $n^2/(n+1)$ by a simpler (and cleaner) function. In fact it's really simple:

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

There are several ways to prove this. Here's the simplest proof:

$$\frac{n^2}{n+1} \leq \frac{n^2}{n+0} = n$$

Hence

$$T'_a(n) = O(n)$$

Notice that both average runtimes gives the same big-O:

$$T_a(n) = O(n)$$
$$T'_a(n) = O(n)$$

Exercise 102.22.1. Consider $n+2$ cases of the linear search.

- `target` is at index position 0 or 1 or ... or $n-1$
- `target` is larger than all values in array `x`
- `target` is less than all values in array `x`

and they are equally likely.

(a) Compute the average runtime under the above assumptions.

(b) Compute the big-O of the average runtime of (a) in the form of n^k.

□

Exercise 102.22.2. Here are the same cases as the previous problem.

- `target` is at index position 0 or 1 or ... or $n-1$

- `target` is larger than all values in array `x`
- `target` is less than all values in array `x`

After running the linear search for a long time on a particular system I realize that there is a particular value that is searched for very frequently. So I move this value to index 0. Specifically, the case where the search for `target` at index 0 is twice as frequent other cases.

(a) Compute the average runtime under the above assumptions.

(b) Compute the big-O of the average runtime of (a) in the form of n^k.

□

Exercise 102.22.3. For another system, I realized that the following cases are most helpful:

- `target` is at index position 0 or 1 or ... or $n-1$. The case where `target` is at index 0 is twice as likely as each of the other cases where `target` is at index i for $i = 1, 2, 3, ..., n-1$.
- `target` is not in the array `x`. This case is 10 times more likely that the case where `target` is at index 1.

(a) Compute the average runtime under the above assumptions.

(b) Compute the big-O of the average runtime of (a) in the form of n^k.

□

File: average-time-using-probability-theory.tex

102.23 Average Time Using Probability Theory

Now I'm going to rephrase our computations of the various average times in terms of probability theory. Of course you'll get the same results. So why do we bother? Because by doing so we can bring in facts/formulas/techniques/big guns from probability theory without having to re-invent the wheel.

Recall that for out linear search, we looked at this average time scenario:

- `target` is at index 0, 1, 2, ..., n - 1 in array `x`
- `target` is not in array `x`

When phrased in terms of probability theory the above are called events. For instance "`target` is at index 2" is an event. However I want to simplify this and just call it "2". I'm going to call the event "`target` is not in the array" the event "n". So here's the set of all possible events

$$S = \{0, 1, 2, ..., n\}$$

The set of all events of a particular *experiment* is called the sample space of the experiment. Of course in this case the *experiment* is the act of finding `target` in `x`.

Now let me add our "frequency" assumption:

- `target` is at index 0, 1, 2, ..., n - 1 in array `x`
- `target` is not in array `x`
- All the above cases are equally likely.

In terms of probability theory, this is the same as assignment a measure of likelyhood to all events such that they add up to 1 (i.e. to 100%). I'm going to using a function for this assignment of measure of likelihood:

$$p : S \to \mathbb{R}$$

and of course since the events in S are equally likely and there are $n+1$ events,

the function p is given by

$$p(x) = \frac{1}{n+1}$$

Next I will attach a quantity to each event that I want to count or measure. In our case, I attach to each event the runtime. For instance for the event "`target` is at index 2" I attach the quantity T_2. This assignment is called a random variable. I'm going to use X for this assignment. So here's the assignment:

$$X : S \to \{T_0, T_1, ..., T_n\}$$

where

$$X(i) = T_i$$

Now everything is set up.

So what's our average time? Or in probability theory we would say what is the "expected" time or the "expectation of X"? It's written $E[X]$ and the formula is what I have already shown you:

$$E[X] = \frac{1}{n+1}(T_0 + T_1 + \cdots T_n)$$

However I want to rewrite it in a different form:

$$\begin{aligned} E[X] &= \frac{1}{n+1}(T_0 + T_1 + \cdots + T_n) \\ &= \frac{1}{n+1}T_0 + \frac{1}{n+1}T_1 + \cdots + \frac{1}{n+1}T_n \\ &= p(0)X(0) + p(1)X(1) + \cdots + p(n)X(n) \\ &= \sum_{x \in S} p(x)X(x) \end{aligned}$$

Here's a summary of how to setup quantities in order to use probabily theory to compute average values:

- State your experiment. In our case: "Using the linear search algorithm, find the index where `target` occurs in an array `x` of size `n`. If it's not found set index to -1."
- State the sample space, i.e., all the events. In our case: "The index is 0, 1, 2, ..., `n` - 1 if `target` is in `x` or the index is -1 when `target` is not in `x`. To simplify notation we call these events 0, 1, 2, $n-1$, n. Let

$S = \{0, 1, 2, ..., n\}$.

- State the probability function. In our case (because we assume the events are equally likely):

$$p : S \to \mathbb{R}$$

where

$$p(x) = \frac{1}{n+1}$$

- State the random variable. In our case: “

$$X : S \to \mathbb{R}$$

where

$$X(i) = A + Bn, \quad i = 0, 1, 2, ..., n-1$$

and

$$X(n) = C + Dn$$

- Compute the average or the expectation of X using

$$E[X] = \sum_{x \in S} p(x)X(x)$$

File: review-of-logarithm.tex

102.24 Review of Logarithm

Here's the definition of logarithm:

$$\log_a b = c$$

if

$$b = a^c$$

For instance, since

$$100 = 10^2$$

we have

$$\log_{10}(100) = 2$$

Exercise 102.24.1. What is $\log_3 81$? □

Exercise 102.24.2. What is $\log_4 2$? □

The point of the logarithm is to simplify large numbers by focusing on powers after the numbers are expressed as powers of a fixed base. It works particularly well for products. For instance if you look at this:

$$100^{0.2} \cdot 10^{5+x} = 100000000$$

Notice that all numbers can be converted to base 10:

$$(10^2)^{0.2} \cdot 10^{5+x} = 10^8$$

or

$$10^{0.4} \cdot 10^{5+x} = 10^8$$

and therefore you get

$$10^{0.4+5+x} = 10^8$$

from which you get the equation

$$0.4 + 5 + x = 8$$

Notice that this equation involves much smaller numbers than

$$100^{0.2} \cdot 10^{5+x} = 100000000$$

With the concept of logarithm to base 10 this is exactly what you would get if you use log-to-base-10 calculations with logarithm formulas like

- $\log_{10}(10) = 1$
- $\log_{10}(ab) = \log_{10}(a) + \log_{10}(b)$
- $\log_{10}(a^b) = b \log_{10}(a)$

$$\begin{aligned}
\log_{10}(100^{0.2} \cdot 10^{5+x}) &= \log_{10}(100000000) \\
\log_{10}(100^{0.2}) + \log_{10}(10^{5+x}) &= \log_{10} 10^8 \\
\log_{10}(10^{0.4}) + \log_{10}(10^{5+x}) &= 8 \log_{10} 10 \\
0.4 \log_{10} 10 + (5+x) \log_{10} 10 &= 8 \\
0.4 + (5+x) &= 8
\end{aligned}$$

Theorem 102.24.1. *Here are some formulas for logarithms.*

(a) $\log_b b = 1$

(b) $\log_b xy = \log_b x + \log_b y$

(c) $\log_b x^y = y \log_b x$

(d) $\log_b a = \frac{\log_c a}{\log_c b}$

Exercise 102.24.3. What is x is $3^{x+2} = 3^3$? □

Exercise 102.24.4. What is x is $3^{x+2} = 9$? □

Exercise 102.24.5. What is x is $3^{x+2} = 1$? □

Exercise 102.24.6. What is x is $27 \cdot 3^{x+2} = \frac{1}{9}$? □

Exercise 102.24.7. What is x is $27 \cdot 3^{x+2} = 3^{2x-1}$? □

File: powers-of-logarithm.tex

102.25 The big-O Classes: Logarithms and Their Powers

What about powers of $\ln n$ and also composition of the log function such as

$$(\ln n)^2$$

and

$$\ln(\ln n)$$

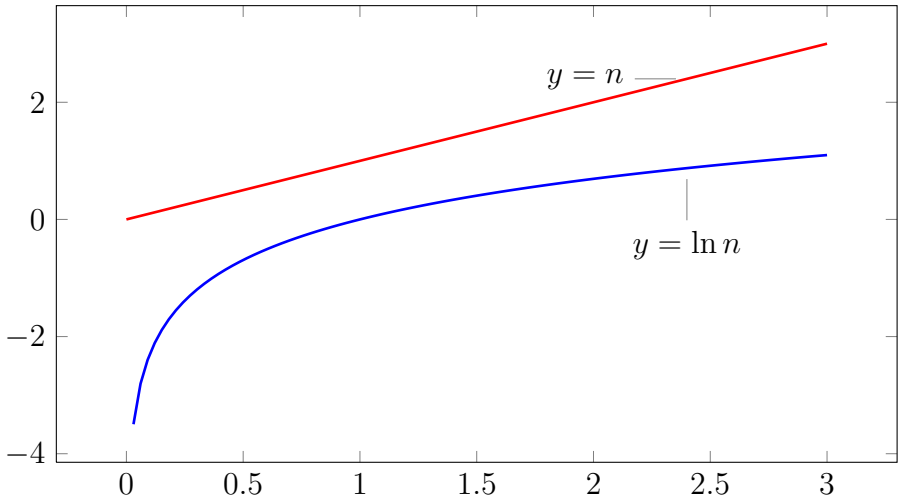
We define the following

- The k–th power of ln is
$$\ln^k n = (\ln n)^k$$
i.e., ln function to the power of k. For instance
$$\ln^3 n = (\ln n)^3$$
Note that the first power of ln is just ln:
$$\ln^1 n = \ln n$$

- The k–th composition of ln is
$$\ln^{(k)} n = \ln(...(\ln k)...)$$
i.e., composition of ln function k times. For instance
$$\ln^{(3)} n = \ln(\ln(\ln n))$$

How does $\ln n$ (or $\ln^k n$) compare with powers of n^d?

Here's $\ln n$ and n:



The function $\ln n$ seems to bend downward. It actually grows unboundedly, i.e.,

$$\lim_{n\to\infty} \ln n = \infty$$

although it does grow rather slowly when compared to n.

Here's the picture when for $1 \leq x \leq 50$:

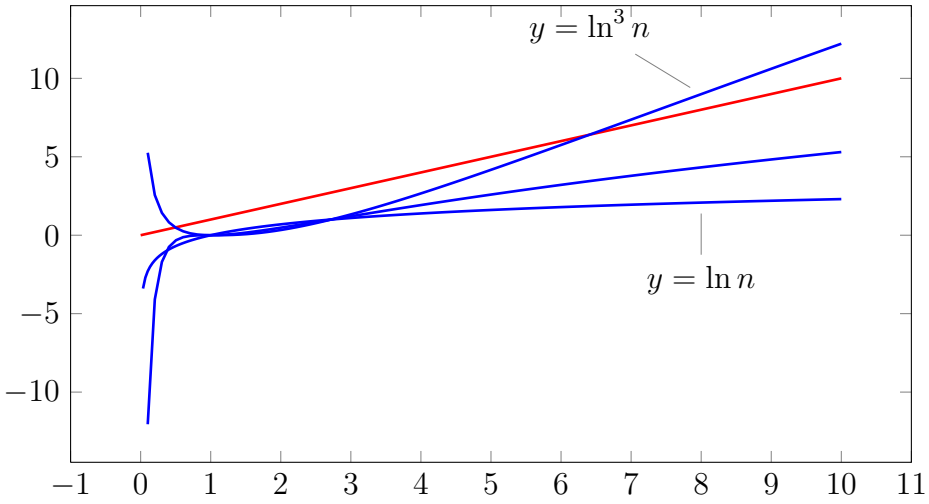
From the graph you would guess

$$\ln n = O(n)$$

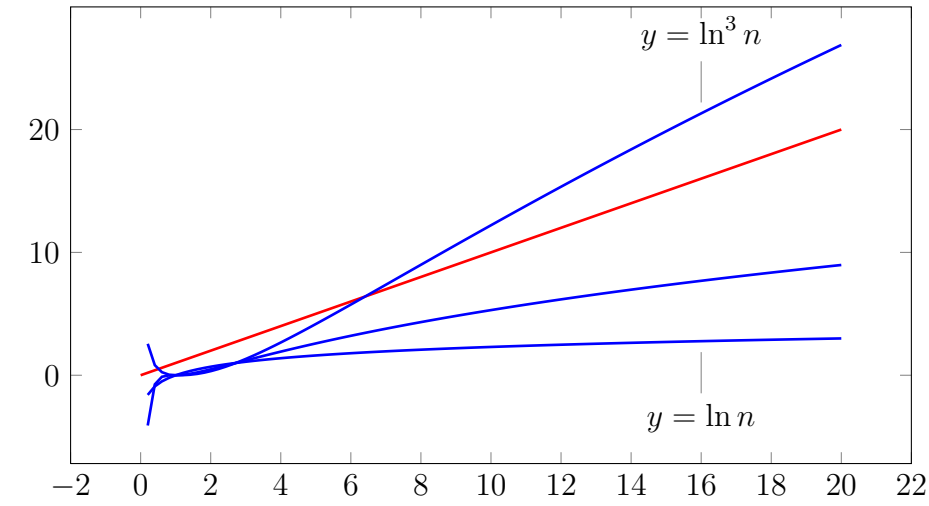
and that is in fact true: you can prove the above fact using l'Hôpital's rule:

$$\lim_{n\to\infty} \frac{\ln n}{n} = \lim_{n\to\infty} \frac{1/n}{1} = \lim_{n\to\infty} \frac{1}{n} = 0$$

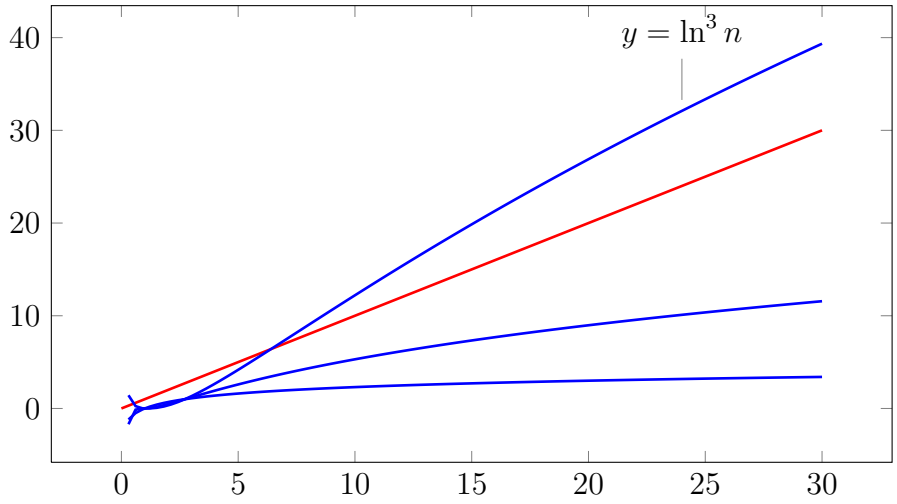
What if I beef up $\ln n$? Let's look at powers of $\ln^k n$ for $k = 1, 2, 3$.



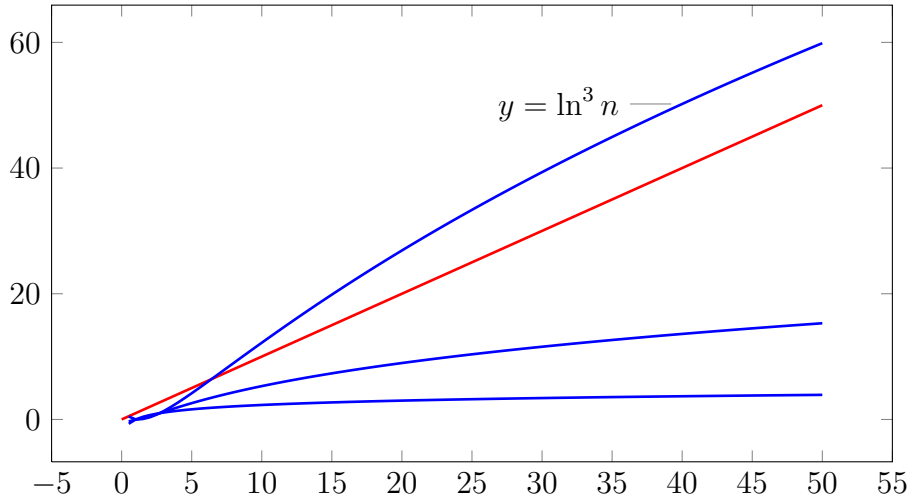
Looks like $\ln^3 n$ beats n. But you know that graphs cannot be trusted (I already told that, right?). So we check the graphs for n up to 20:



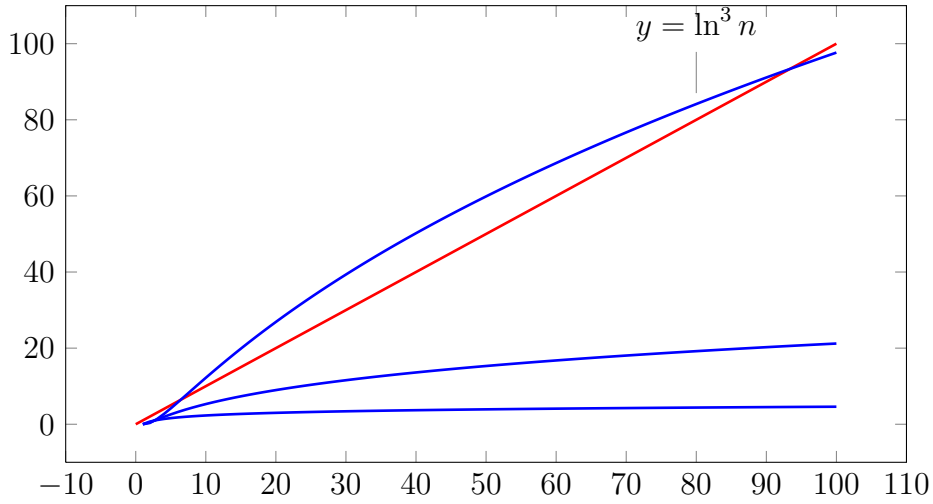
Phew! ... seems to be ok ... wait let's try up to $n = 30$:



But now you begin to see something disturbing ... do you notice that the $\ln^3 n$ is bending ever so slightly download? Here's the graph up to $n = 50$:



Now the concavity of $\ln^3 n$ becomes more pronounced! Oh no! Let's try up to $n = 100$:



Aarrghhhh! Even $\ln^3 n$ can't beat n

When does n beat $\ln^k n$? Well

$$\begin{aligned} & n \geq \ln^k n \\ \iff \quad & \ln n \geq \ln(\ln^k n) \\ \iff \quad & \ln n \geq k \ln(\ln n) \\ \iff \quad & \frac{\ln n}{\ln(\ln n)} \geq k \end{aligned}$$

Now why is the last fact a good fact (if it's true at all)? Because the left-hand-side is independent of k, that's why. It's a comparison of $\ln n$ and $\ln \ln n$. If you look at the above graph, you'd see that it's likely that $\ln n / \ln \ln n$ grows unboundedly, i.e. we guess:

$$\lim_{n\to\infty} \frac{\ln n}{\ln \ln n} = \infty$$

This looks like a job for l'Hôpital's rule again.

$$\lim_{n\to\infty} \frac{\ln n}{\ln \ln n} = \lim_{n\to\infty} \frac{1/n}{1/(\ln n) \cdot (1/n)} = \lim_{n\to\infty} \ln n = \infty$$

This means that

$$\ln^k n = O(n)$$

for all $k \geq 0$ (for $k = 0$, $\ln^0 n = 1$ and for $k = 1$, $\ln^1 n = \ln n$.) Of course

$$\ln^k n = O(\ln^{k+1} n)$$

So to our complexity classes

$$O(1) \subset O(n) \subset O(n^2) \subset O(n^3) \subset \cdots$$

we can now add more classes to get:

$$\begin{aligned}
&O(1) \subset O(\ln n) \subset O(\ln^2 n) \subset \cdots O(n)\\
&O(n) \subset O(n \ln n) \subset O(n \ln^2 n) \subset \cdots O(n^2)\\
&O(n^2) \subset O(n^2 \ln n) \subset O(n^2 \ln^2 n) \subset \cdots O(n^3)\\
&\cdots
\end{aligned}$$

In other words in $O(n^r) \subset O(n^{r+1})$ we expand to get

$$O(n^r) \subset O(n^r \ln n) \subset O(n^r \ln^2 n) \subset O(n^r \ln^3 n) \subset \cdots \subset O(n^{r+1})$$

Theorem 102.25.1.

(a) $\log_a n = O(n)$.

Proof

(d) This following from l'Hôpitals rule:

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

if the right hand side makes sense. Therefore

$$\lim_{n\to\infty} \frac{\ln n}{n} = \lim_{n\to\infty} \frac{1/n}{1} = \lim_{n\to\infty} \frac{1}{n} = 0$$

This implies that for any $\epsilon > 0$, there is some N such that for $n \geq N$, $|(\ln n)/n| \leq \epsilon$, i.e.,

$$|\ln n| \leq \epsilon |n|$$

In particular, for $C = 1$, there is some N such that for $n \geq N$

$$|\ln n| \leq C|n|$$

Hence $\ln n = O(n)$. Since

$$\log_a n = \frac{\ln n}{\ln a}$$

we also have $\log_a n = O(n)$. □

By the way, the proof technique using l'Hôpital's rule proves more generally that if $f(n)$ and $g(n)$ are positively valued for all large n, and such that

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

for a constant C, then $f = O(g)$. Why? Because if that's the case, then for every $\epsilon > 0$, there is some N such that for $n \geq N$, we have

$$\left|\frac{f(n)}{g(n)} - C\right| < \epsilon$$

Therefore if we choose $\epsilon = 1$, then there is some N such that for $n \geq N$,

$$\left|\frac{f(n)}{g(n)} - C\right| < 1$$

i.e.,

$$-1 < \frac{f(n)}{g(n)} - C < 1$$

i.e.,

$$\frac{f(n)}{g(n)} < C + 1$$

and therefore

$$f(n) < (C+1)g(n)$$

Exercise 102.25.1. Prove that $\ln(n^3) = O(\ln n)$. □

Exercise 102.25.2. Prove that $\ln(1000 + n^3) = O(\ln n)$. □

Exercise 102.25.3. Prove that $\ln(1000n + n^{1000}) = O(\ln n)$. □

Exercise 102.25.4. Prove that $\ln(n + 10^{10}) = O(\ln n)$. □

Exercise 102.25.5. Find the best big-O in the form of $O(n^a \ln^b n)$ for the

following function: $\ln(10^{42}n^{12})$ □

Exercise 102.25.6. Find the best big-O in the form of $O(n^a \ln^b n)$ for the following function: $(n+10)\ln(20n^{12})$ □

Exercise 102.25.7. Find the best big-O in the form of $O(n^a \ln^b n)$ for the following function: $\ln(10^{42}n^{12}+n^5)$. □

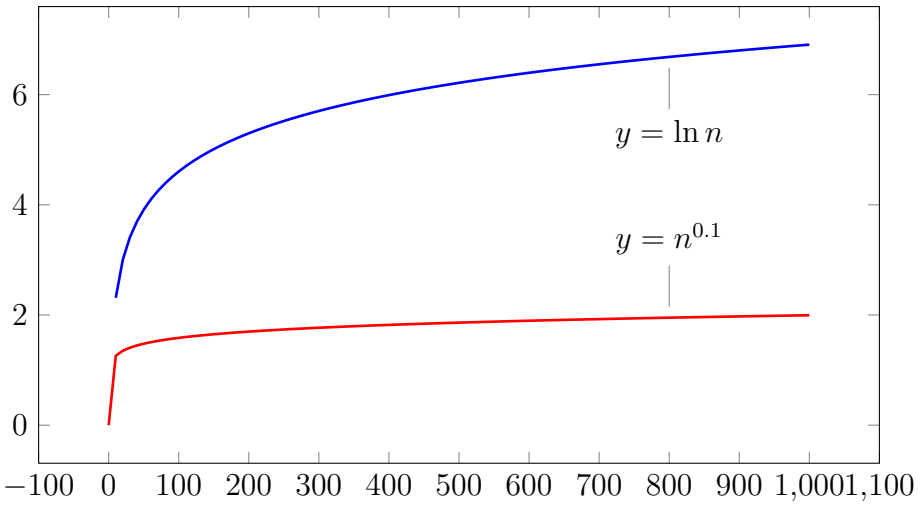
Exercise 102.25.8. Find the best big-O in the form of $O(n^a \ln^b n)$ for the following function: $\ln(10^n+4)$. □

Exercise 102.25.9. Find the best big-O in the form of $O(n^a \ln^b n)$ for the following function: $\ln(n^{n+\ln n}+\ln n)+n^2$. □

Exercise 102.25.10. Find the best big-O in the form of $O(n^a \ln^b n)$ for the following function: $n^2\ln(n^{n+5}+4)$. □

Exercise 102.25.11. Find the best big-O in the form of $O(n^a \ln^b n)$ for the following function: $n^2\ln(n^{n+5}+n^6)+n^3$. □

Exercise 102.25.12. The following are plots of $y=n^{0.1}$ and $y=\ln n$ for $0\le n\le 1000$:

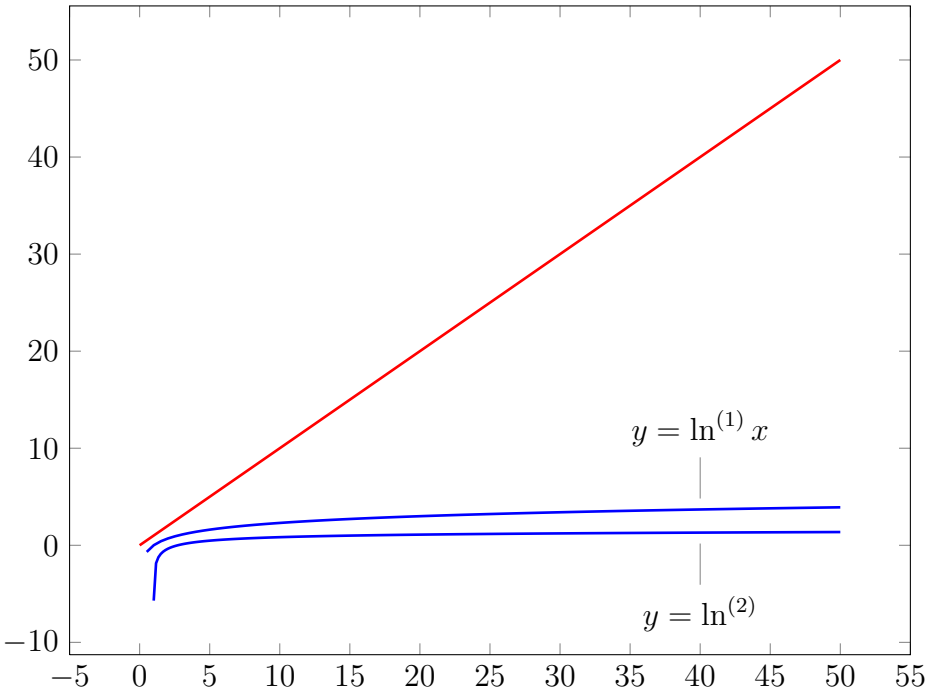


Prove or disprove $n^{0.1}=O(\ln n)$? □

File: the-big-O-classes-composition-of-logarithm.tex

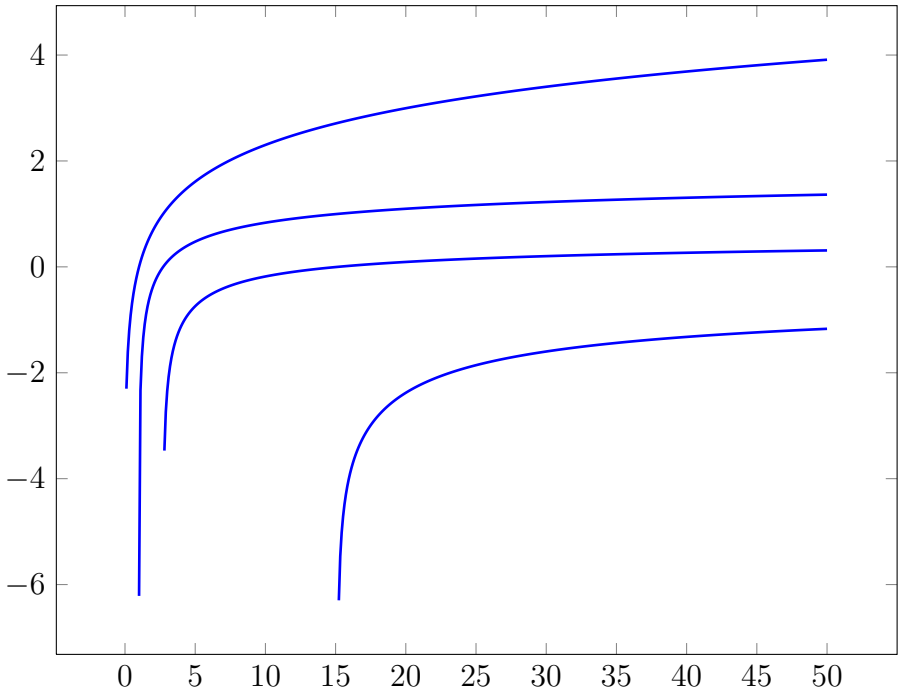
102.26 The big-O Classes: Composition of Logarithm

What about $\ln^{(k)} n$? Here's a plot of $n, \ln^{(1)} n, \ln^{(2)} n$ (you do see quickly just from definition that $\ln^{(1)} n = \ln n$, right?)



Exercise 102.26.1. Why is it not surprising that $\ln^{(1)} n$ beats $\ln^{(2)} n$? □

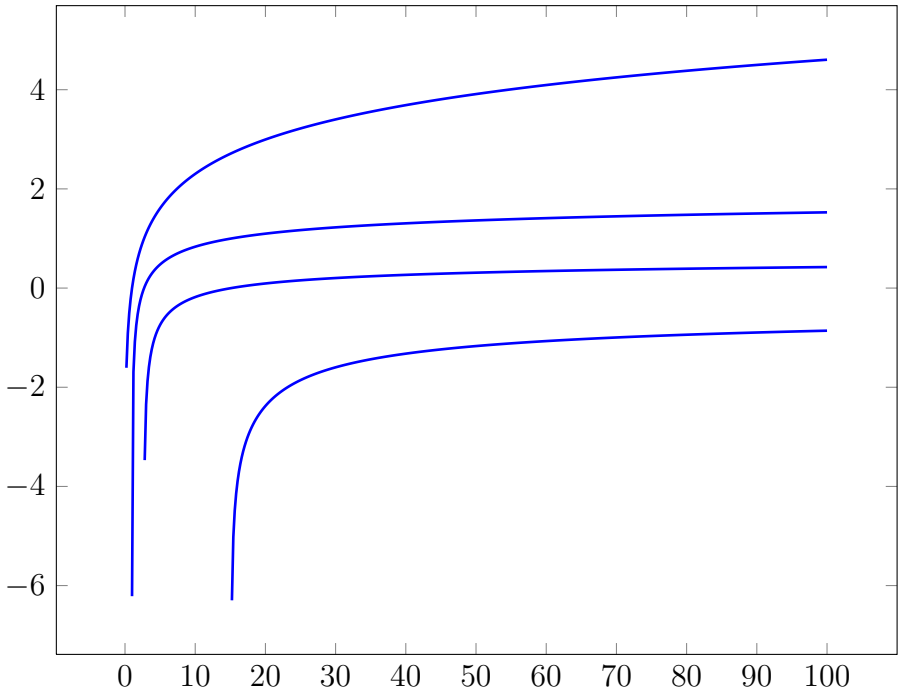
Let's forget about the plot for n and just look at the plots for $\ln^{(k)} n$ for $k = 1, 2, 3, 4$:



So we suspect that for large values of n:

$$\ln^{(4)} n < \ln^{(3)} n < \ln^{(2)} n < \ln^{(1)} n = \ln n$$

Let's plot the functions again but for a domain of up to 100 just to visually verify our guess:



In other words in $O(n^r) \subset O(n^{r+1})$ we expand to get

$$\cdots \subset O(n^r \ln^{(3)}) \subset O(n^r \ln^{(2)}) \subset O(n^r \ln^{(1)})$$
$$\| $$
$$O(n^r \ln n) \subset O(n^r \ln^2 n) \subset O(n^r \ln^3 n) \subset \cdots$$

File: review-of-exponentials.tex

102.27 Review of Exponentials

You should know all the following formulas:

Theorem 102.27.1.

(a) *If* $b \neq 0$, *then* $b^0 = 1$.

(b) $b^x \cdot b^y = b^{x+y}$

(c) $(b^x)^y = b^{xy}$

(d) $b^{-x} = \frac{1}{b^x}$

Note that 0^0 is undefined.

File: the-big-O-classes-exponential.tex

102.28 The big-O Classes: Exponentials

At this points I've shown you several standard complexity classes:

$$O(n^r), \quad O(\ln^r n), \quad O(\ln^{(r)} n)$$

and their various products such as

$$O(n^r \ln^s n)$$

and their relationship such as

$$O(n^r) \subset O(n^s) \text{ if } r < s$$

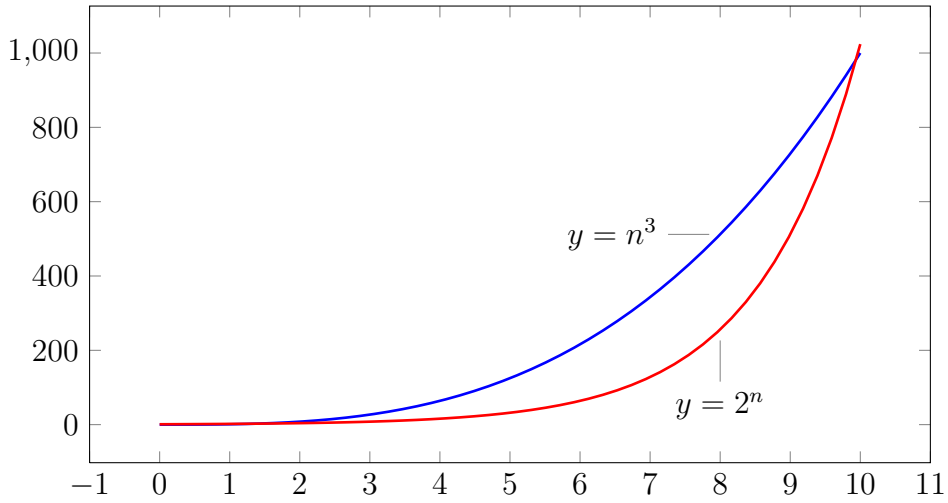
There's another very important class which is used to bound algorithms that run with exponential time. Here's an example:

$$O(2^n)$$

In general, given a constant c, one can talk about

$$O(c^n)$$

Algorithms with exponential runtimes are really bad. Here's a plot to give you an idea:



Let's look at c^n for different values of c. If $c \leq 1$, then

$$c^n \leq 1$$

for $n \geq 0$. Therefore

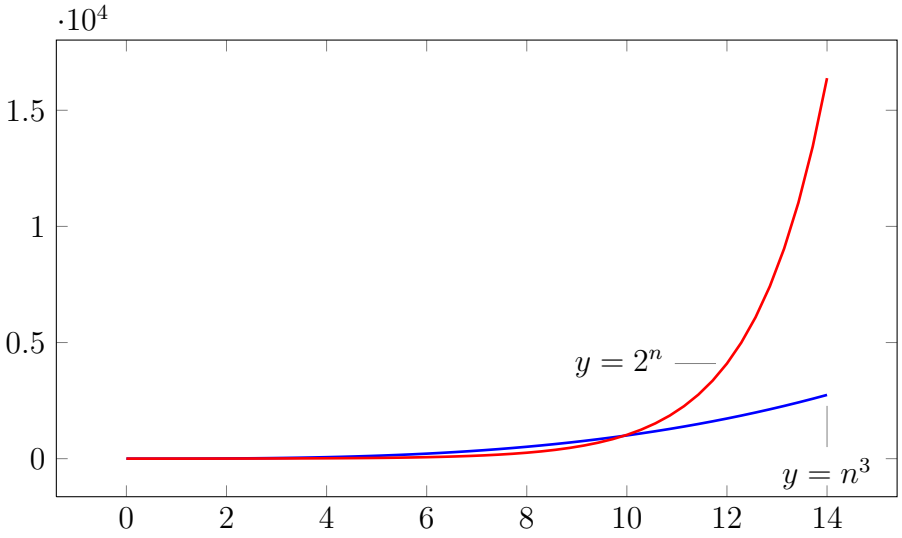
$$O(c^n) \subset O(1)$$

Theorem 102.28.1.

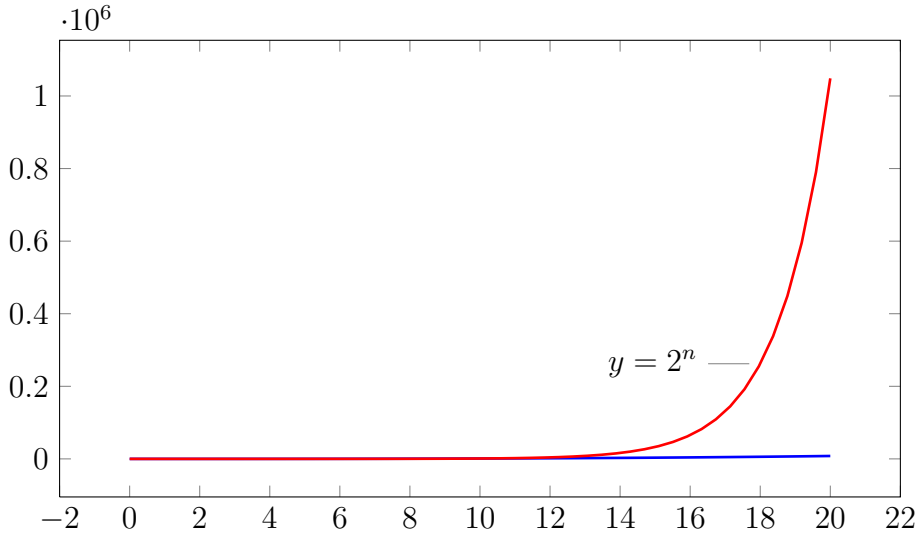
(a) If $c \leq 1$, then $c^n = O(1)$.

(b) If $1 \leq c < d$, then $O(c^n) \subset O(d^n)$

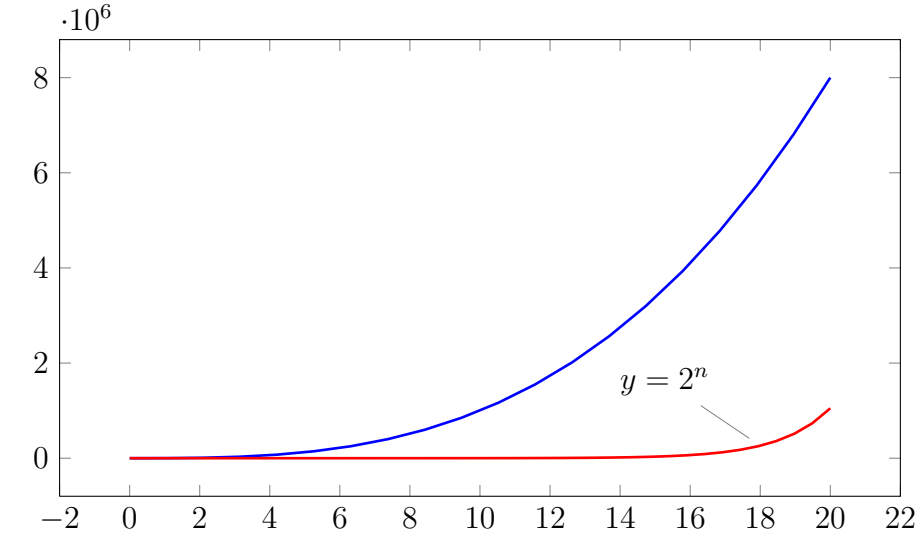
It seems that n^3 is worse than 2^n for the most part. But look at this:



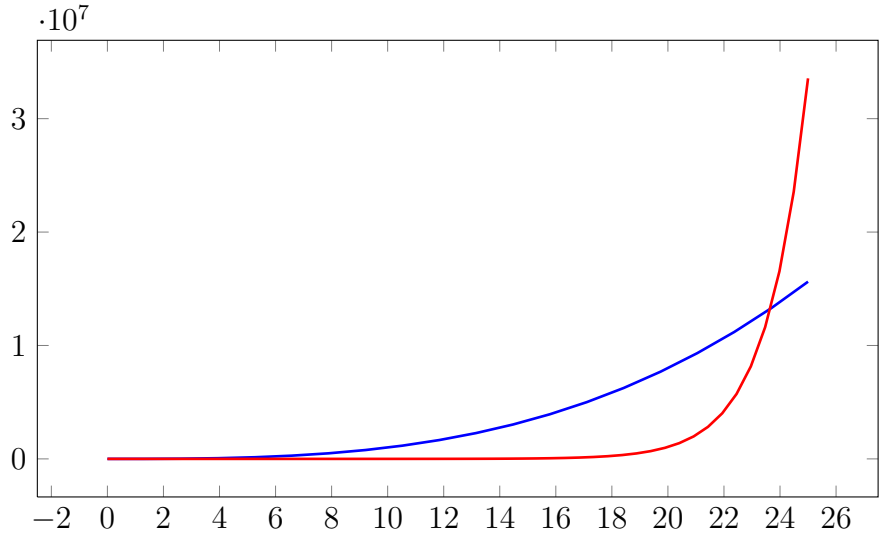
and then this:



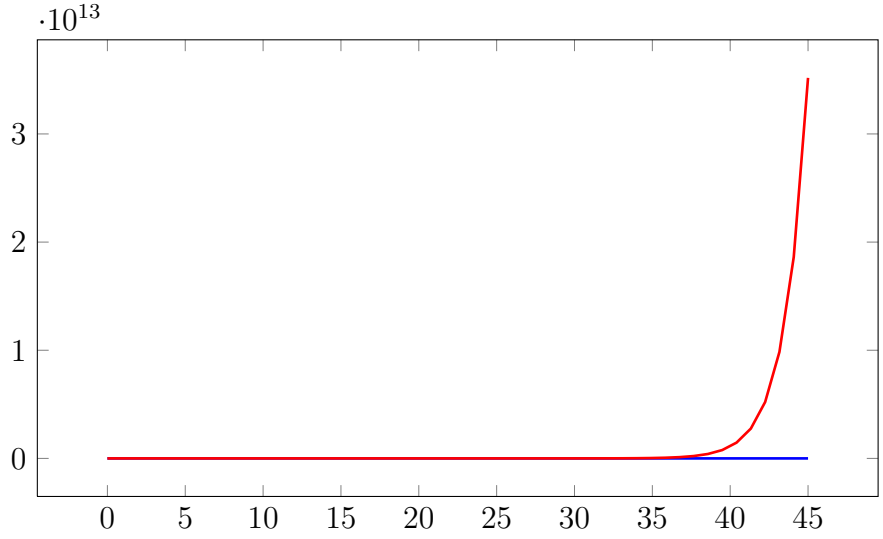
To make matters worse, even if I give n^3 a huge multiple, say I use $1000n^3$, 2^n still ultimate still beats $1000n^3$. Here's the plot of $1000n^3$ and 2^n up to $n = 20$:



and then up to $n = 30$ where you see 2^n beating $1000n^3$ for $n \geq 24$:



In fact, no amount of "multiple help" you give to n^3 will save it from being beaten by 2^n for large values of n. Here's the plot of 2^n and $1000000n^3$ for $0 \leq n \leq 45$ with 2^n practically flattening $1000000n^3$:



In fact, for *any* constant $c > 1$, c^n will alway beat n^d for any constant d even if you multiply n^d by a huge constant, i.e., for any constant $c > 1$ and any constants d_0 and d_1, there is some constant N such that

$$|d_0 \cdot d_1^n| \leq c^n$$

for $n \geq N$. Of course this means

$$|d_0 \cdot n^{d_1}| \leq 1 \cdot |c^n|$$

for $n \geq N$ and therefore

$$d_0 \cdot n^{d_1} = O(c^n)$$

Therefore

$$O(n^d) \subset O(c^n)$$

for any constants $c > 1$ and d.

Theorem 102.28.2. *For constants $c > 1$ and d,*

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

i.e.,

$$O(n^d) \subset O(c^n)$$

Solution.

$$\begin{aligned} & n^d \leq c^n \\ \iff \quad & \ln n^d \leq \ln c^n \\ \iff \quad & d \ln n \leq n \ln c \\ \iff \quad & \left(\frac{d}{\ln c}\right) \ln n \leq n \end{aligned}$$

Since we already know that $\ln n = O(n)$, we have

$$\alpha \ln n = O(n)$$

for any constant α. In partifular for $\alpha = d/\ln c$,

$$\left(\frac{d}{\ln c}\right) \ln n = O(n)$$

Therefore there are constants C and N such that

$$\left(\frac{d}{\ln c}\right) \ln n \leq Cn$$

for $n \geq N$. The above inequality is the same as

$$d \ln n \leq Cn \ln c$$

which is the same as

$$\ln n^d \leq C \ln c^n$$

which is the same as

$$n^d \leq Cc^n$$

Hence $n^d = O(c^n)$. □

Therefore we now know that exponential time algorithms in $O(c^n)$ (of course for $c > 1$) is always worse than any polynomial time algorithm.

It's not difficult to compare between exponential classes:

Theorem 102.28.3. *If* $1 \leq c < d$ *then*

$$O(c^n) \subset O(d^n)$$

The proof is easy.

Exercise 102.28.1. Prove that $2^{n+4}(n+42)^3 = O(2^n)$. □

Exercise 102.28.2. Prove that $3^{n/2} = O(2^n)$. □

Exercise 102.28.3. Prove that $\ln(n \ln n) = O(\ln n)$. [MORE TO LOG SECTION] □

[MORE EXERCISES]

File: other-big-O-classes.tex

102.29 Other big-O Classes

So far we have big-O classes for polynomials, logarithms, and exponentials (as well as their products).

There are other big-O classes for functions like

$$n^n, \quad n!$$

Clearly

$$n! \leq n^n$$

just by looking

$$1 \cdot 2 \cdot 3 \cdots n \leq n \cdot n \cdot n \cdots n$$

So clearly

$$O(n!) \subseteq O(n^n)$$

Could the two classes be the same? Is it possible for a large enough constant multiple of $n!$ to beat n^n?

[TODO: subexponential, superexponential, etc.]

File: other-types-of-asymptotic-bounds.tex

102.30 Other Types of Asymptotics Bounds

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

Here's another type of asymptotic bound:

Definition 102.30.1. We write $f \in \Omega(g)$ if there exist C and N such that for $n \geq N$,

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

In other words a multiple of g bounds f from below. We also say that f is **asymptotically bounded below** by g or f is the **big-Ω** of g.

Definition 102.30.2. You can put the two definitions together and get this definition: $f = \Theta(g)$ if $f \in O(g)$ and $f \in \Omega(g)$. We will say in this case that f is **asymptotically equivalent to** g or f is **big-Θ** of g.

Here are some basic facts you should know. For the following f, g, f_i, and g_i are functions.

1. $O(cf) = O(f)$, $\Omega(cf) = \Omega(f)$, $\Theta(cf) = \Theta(f)$
2. $\Omega(f) \subseteq O(f) \cap \Theta(f)$
3. If $f_i = O(g_i)$ $(i = 1, 2)$ then

$$f_1 + f_2 = O\big(\max(|g_1|, |g_2|)\big), \quad f_1 f_2 \in O(g_1 g_2)$$

4. $f_i = O(g)$ $(i = 1, 2) \implies f_1 + f_2 = O(g)$
5. $f \in O(g) \implies O(f) \subseteq O(g)$.

Besides *inequality* types asymptotic bounds, there are two other *limit* types asymptotic properties:

File: function-call.tex

102.31 Function Call

The runtime analysis for function call is similar when it comes to computing the runtime of the body of the function. The only other thing you need to be careful about the cost of parameter passing and the cost of function return.

Suppose you have a function that takes an array and returns the first value:

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

You would say right away that

```
    return x[0];
```

uses the following amount of time: Suppose you call `f()` from `main()`:

1. Your computer needs time to "go to" `f()`
2. Your computer needs time to set up parameter `x`
3. Your computer needs time to access `x[0]`
4. Your computer needs time to "go back" to `main()`

The time taken for each step is as follows:

1. Your computer needs time to "go to" `f()`: constant
2. Your computer needs time to set up parameter `x`: constant
3. Your computer needs time to access `x[0]` for returning: constant
4. Your computer needs time to "go back" to `main()`: constant

The algorithmic part, i.e., the code in the body of `f()`:

3. Your computer needs time to access `x[0]`: constant

You have already seen that before – so no surprises. The rest are new. You can always assume that function calling and function returning is always constant time:

1. Your computer needs time to "go to" `f()`: constant
4. Your computer needs time to "go back" to `main()`: constant

The crucial thing is this:

2. Your computer needs time to set up parameter `x`: constant constant

Whether this is constant or not depends on how much memory is used by your parameter or parameters. In the case of

```
int f(int x[])
```

```
{
    return x[0];
}
```

the parameter is a pointer, i.e., the above is really the same as

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

or, since I'm not changing the values of the array that `x` points to,

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

(Don't remember? You had better re-study your pointer notes.) If you are using a 64-bit computer, you need to request for 64 bits for `x`. But that's a constant. No matter large the array `x` points to, `x` consumes 64 bits. That's why the time taken to set up `x` is a constant.

Therefore the total runtime is $O(1)$.

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 & 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 $O(1)$.

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

```
int f(const 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` is a copy of `y`, 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`. And if `y` is a vector of 1000000000 values, then it takes even more time to set up `x`, requesting for memory and then copying the 1000000000 values from `y` to `x`. Therefore in this the set up time for `x` is actually $O(n)$ where n is the size of `y`. Therefore this algorithm has a total runtime of $O(n)$.

In the same way if you return a "large" value, 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 $O(1)$. 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.

This then finally explains why in CISS245, for "large" variables such as `struct` variables or objects, we always pass by pointer or pass by reference.

So if your paramters are always basic type variables (such as `int`, `float`, `double`, `bool`, `char`) or pointers or references, then the runtime of functions depends only on the runtime of the body of the function –

Exercise 102.31.1. What is the runtime of executing the following function (including function calling and returning). Assume the size of the vector `x` is n:

```
std::vector & g(std::vector & x)
{
    return x;
}
```

Exercise 102.31.2. What is the runtime of executing the following function `f` (including function calling and returning). Assume the size of all vectors involved is n:

```
int h(std::vector x)
{
    return x[0] + 1;
}

int g(std::vector x)
{
    return x[0] - 1;
}

void f(std::vector x)
{
    if (x[0] < 0)
    {
        g(x);
    }
    else
    {
        h(x);
    }
}
```

Speed up the code if possible. What's the new runtime after your improvement?

Exercise 102.31.3. 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 swap(std::vector x, int i, int j)
{
    int = x[i];
    x[i] = x[j];
    x[j] = x[i];
    return x;
}

std::vector bubblesort(std::vector x)
{
    int n = x.size();
    for (int i = 0; i < n - 1; ++i)
    {
        if (x[i] > x[i + 1])
        {
            x = swap(x, i, i + 1);
        }
    }
    return x;
}
```

What is the maximum amount of memory used? Rewrite it to speed it up and reduce memory consumption. Note: Both functions must return something. What is the new runtime and memory used?

Exercise 102.31.4. 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 ...)

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

std::vector bubblesort(std::vector x, int last_index)
{
    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
        bubblesort(x, last_index - 1);
        return x;
    }
}
```

What is the maximum amount of memory used?

File: linear-recursion.tex

102.32 Linear Recursion

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

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

Or in C/C++:

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

Of course this is your standard Fibonacci. (Review your recursion notes from CISS240.)

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 ... well ... because there are two pieces in the definition, right?)

The cases

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

are usually called base cases. The case that has recursion

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

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

How do you analyze the runtime of the above recursive function? Here's what you do. First you let

$$T(n)$$

denote the total time to execute `f(n)`. That includes the function call, the execution of the body, 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

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

If `n` is `1` you get

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

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 no prizes for guessing that 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. 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:

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

(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 we have

$$t_5 = T(n-1) + T(n-2) + B$$

where B is a constant.

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. Of course in analyzing runtimes, you prefer to put this $T(n)$ into one of the standard runtime big-O classes such as

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

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.

Not to panic. All you need to do is to understand how to do numeric recursion

(i.e. substitution). For instance

$$\begin{aligned}
T(5) &= T(4) + T(3) + d \\
&= (T(3) + T(2) + d) + (T(2) + T(1) + d) + d && \text{substitute} \\
&= T(3) + 2T(2) + T(1) + 3d && \text{cleanup ... ur mom told u right?} \\
&= (T(2) + T(1) + d) + 2(T(1) + T(0) + d) + T(1) + 3d && \text{substitute} \\
&= T(2) + 4T(1) + 2T(0) + 6d && \text{cleanup} \\
&= T(1) + T(0) + d + 4T(1) + 2T(0) + 6d && \text{substitute} \\
&= 5T(1) + 3T(0) + 7d && \text{cleanup}
\end{aligned}$$

This is all well and good ... but we need $T(n)$, not $T(5)$!!! Ultimately do you see that $T(n)$ must be of the form:

$$T(n) = \text{BLAH1} \cdot T(1) + \text{BLAH2} \cdot T(0) + \text{BLAH3} \cdot d$$

where BLAH1, BLAH2 and BLAH3 must be formulas involving n. Ultimately we're going to ditch the constants $T(1)$, $T(0)$ and d. What matters is really BLAH1, BLAH2, BLAH3. It turns out that BLAH1 and BLAH2 are of the form

$$(\text{some constant})^n$$

One of the constants is approximately 1.618 while the other is approximately -0.618.

That already tells you that the runtime of the above algorithm to compute the n–th Fibonacci number is pretty bad: it's ...

exponential!!!

That's why if you use the above to compute $f(i)$ for $i = 0, 1, 2, ...$, you will see that your program will slow down to almost a grinding halt after $i = 50$.

It turns out that for such problems you can rewrite the program to run in linear time, i.e., $O(n)$. In fact that there are *several* different ways to do that. I'll talk about this is another set of notes (on techniques for re-designing algorithms to improving runtime) since for this set of notes I'm only going to focus on measuring runtimes.

Of course I still have not explained how I got this mysterious 1.618. But I'm going to take a pause here and give you some exercises first ...

First of all, it's clear that a *recursive algorithm* like the above will give you a corresponding *recursing runtime*. Here are some exercises to verify that.

Exercise 102.32.1. Using the same technique above, compute a recursive runtime for the following algorithm. (Don't worry about writing the formula in closed form.)

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

Exercise 102.32.2. Using the same technique above, compute a recursive runtime for the following algorithm. (Don't worry about writing the formula in closed form.)

```
def f(n):
    if n == 0:
        return 5
    else:
        return 7 * f(n - 1) + 2
```

Exercise 102.32.3. Using the same technique above, compute a recursive runtime for the following algorithm. (Don't worry about writing the formula in closed form.)

```
def f(n):
    if n == 0:
        return 7
    elif n == 1:
        return 10
    else:
        return n * f(n - 1) +  f(n - 2)
```

Exercise 102.32.4. Using the same technique above, compute a recursive runtime for the following algorithm. (Don't worry about writing the formula in closed form.)

```
def f(n):
    if n == 0:
        return 7
    elif n == 1:
        return 10
    else:
        return f(n - 1) * f(n - 2)
```

File: linear-recursive-runtime.tex

102.33 Linear Recursive Runtime

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-resursive* closed form.

Before I show you how to do that, let's just get a feel the such runtime functions by calculating with 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:

$$T(n) = \begin{cases} A & \text{if } n = 0, 1 \\ 2T(n-1) + 3T(n-2) + B & \text{if } n > 1 \end{cases}$$

for some constants A and B. Later I'll show you now 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:

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

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.

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 is said to be a "closed form" formula.

In fact 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

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

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

First of all, you ignore the constant c and look at this recursion instead:

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

You need to know that $T(n)$ must be an exponential function "roughly" of the form

$$T(n) = r^n$$

where r is a constant. The immediate goal now is to compute r. You substitute this into the above equation ... *but without the c!!!* ...

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

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

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

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

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

(remember ... ignore the tail-end constant for now!) 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 the if the recurrence relation on $T(n)$ is

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

(we'll worry about the missing d later ...) then

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

Now, $|r_2| = 0.618...$ whereas $r_1 = 1.618....$ Therefore

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

which means that the algorithm has exponential runtime.

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

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

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 (b) above, the $T(n)$ is just

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

(Unfortunately for case (b) you would have using complex numbers.) For the third case 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]

File: fast-linear-recursion.tex

102.34 Speeding up Linear Recursion

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

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

For instance the Fibonacci function gives the above runtime:

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

The above runtime is exponential.

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 `f(i)` (`i = n - 2, ...`) when you compute `f(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 `f(0)`, `f(1)`, ..., `f(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 `f(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 `f(n)`, save it in the table, then return it. Now the function looks like this:

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

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

```
        return table[n]
```

Exercise 102.34.1. Assuming your `table` is infinite, compute the runtime of the above implementation. □

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

Exercise 102.34.2. Modify the above problem to use the `table` only when `n` is less than 1000.

Exercise 102.34.3. Note that we use `table[5]` to store $f(5)$, i.e., `table[0, ..., 999]` corresponds to $f(0), ..., f(999)$. What if you want to store $f(10)$, ..., $f(1009)$ in `table[0,...,999]` instead?

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 the 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-to-top technique. You can use a for-loop or recursion to implement the bottom-to-top technique.

Exercise 102.34.4. Implement this new Fibonacci function (a) using a for-loop and (b) using recursion. Test them. Compute the runtime. □

Exercise 102.34.5. 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 102.34.6. 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.) □

File: floor-ceiling.tex

102.35 Floor and Ceiling

No ... I'm not creating a diversion to building construction although what I'm going to talk about is analogous to what you're standing on and what's above your head.

Given a real number x. The **floor** of x, written $\lfloor x \rfloor$ is x if x is an integer or the integer before x. For instance

$$\lfloor 3 \rfloor = 3, \quad \lfloor 1.7 \rfloor = 1, \quad \lfloor -2.3 \rfloor = -3$$

The **ceiling** of x, written $\lceil x \rceil$ is x if x is an integer or the integer after x. For instance

$$\lceil 3 \rceil = 3, \quad \lceil 1.7 \rceil = 2, \quad \lceil -2.3 \rceil = -2$$

Note that in your C/C++ programs when you do something like

```
int y = 7;
int x = y / 2;
```

you know that the "/" is integer division because `y` and `2` are integers. Therefore `x` has a value of `3`. In mathematical notation, when you write a/b, the "/ " is *real* division regardless of whether a and b are real or integers. So if I convert the above to mathematical notation I would have to write

$$y = 7$$
$$x = \lfloor y/2 \rfloor$$

When (not if) you want to make use of mathematical tools to study algorithms, you have to translate your algorithms into mathematical notation in order to move from the program world to the math world. Since the meaning of "/" in a program might be different from the meaning of "/" in math, you have to be very careful.

The point is that when you convert a `double` value to `int` value by type casting, the *fractional part* of the `double` value is lost. Therefore for the following C/C++ code:

```
double y = 3.14159;
int x = y;
```

the value of `x` is 3. (This is old CISS240 stuff, right?) Therefore the integer cast is like the floor function in math. But that's only true for *positive* value because for the following:

```
double y = -3.14159;
int x = y;
```

the value `x` is `-3` whereas, in math, the floor of -3.14159 is -4. So remember the double-to-int cast (same for float-to-int of course) is like the floor only for values ≥ 0.

Exercise 102.35.1.
(a) Compute $\lfloor 3.5 \rfloor$.
(b) Compute $\lceil 3.5 \rceil$.
(c) Compute $\lfloor 3 \rfloor$.
(d) Compute $\lceil 3 \rceil$.

□

Exercise 102.35.2. Let $x = 5$. Compute

$$\left\lfloor \frac{x+0}{4} \right\rfloor + \left\lfloor \frac{x+1}{4} \right\rfloor + \left\lfloor \frac{x+2}{4} \right\rfloor + \left\lfloor \frac{x+3}{4} \right\rfloor$$

and

$$\left\lceil \frac{x+0}{4} \right\rceil + \left\lceil \frac{x+1}{4} \right\rceil + \left\lceil \frac{x+2}{4} \right\rceil + \left\lceil \frac{x+3}{4} \right\rceil$$

□

Exercise 102.35.3. Consider the following algorithm where `x`, `y`, and `z` are arrays. `x` has size `n` $= 25$.

```
n0 = n / 3
for i = 0, 1, 2, ..., n0 - 1:
    y[i] = x[i]
for i = n0, n0 + 1, ..., n - 1:
```

```
    z[i - n0] = x[i]
```

(a) How many values were copied over to `y`?
(b) How many values were copied over to `z`?
(c) If `n` has value 26, how many values were copied over to `y`?
(d) If `n` has value 26, how many values were copied over to `z`?
(e) If `n` has value 27, how many values were copied over to `y`?
(f) If `n` has value 27, how many values were copied over to `z`?

Exercise 102.35.4. Plot $y = \lfloor x \rfloor$. □

Exercise 102.35.5. Plot $y = \lceil x \rceil$. □

Exercise 102.35.6. Find all real numbers x such that

$$\lfloor x \rfloor = 2$$

□

Exercise 102.35.7.

1. Write a function in C/C++ that accepts a double `x` and returns the ceiling of value of `x`. Call your function `ceiling`. (Test your code ... I don't have to tell you that, right?)
2. Do the same for floor.

□

Exercise 102.35.8. Plot $y = \frac{\lceil x \rceil + \lfloor x \rfloor}{2}$. Solve

$$\frac{\lceil x \rceil + \lfloor x \rfloor}{2} = x$$

□

Exercise 102.35.9. Plot $y = 2 \left\lfloor \frac{x}{2} \right\rfloor$. Solve

$$2 \left\lfloor \frac{x}{2} \right\rfloor = x$$

□

Exercise 102.35.10. Plot $y = \lceil x/2 \rceil + \lfloor x/2 \rfloor$. Solve

$$\lceil x/2 \rceil + \lfloor x/2 \rfloor = x$$

□

Exercise 102.35.11. Let $a \geq 0, b > 0, c > 0$ be integers. Write a program to check

$$\lfloor \lfloor a/b \rfloor / c \rfloor = \lfloor a/(bc) \rfloor$$

for $0 \leq a \leq 1000000$, $1 \leq b \leq 1000000$, and $1 \leq b \leq 1000000$. If the above is false, list a reasonably small counterexample. Otherwise prove it. □

Exercise 102.35.12. Let $a \geq 0, b > 0, c > 0$ be integers. Write a program to check

$$\lceil \lceil a/b \rceil / c \rceil = \lceil a/(bc) \rceil$$

for $0 \leq a \leq 1000000$, $1 \leq b \leq 1000000$, and $1 \leq b \leq 1000000$. If the above is false, list a reasonably small counterexample. Otherwise prove it. □

File: divide-and-conquer.tex

102.36 Divide-and-Conquer Algorithms

Another type of recursive runtime function that 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).

Exercise 102.36.1. You are given the following recursive function:

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

Compute $f(7)$ performing your computation like a computer, evaluating one operation of performing only one substitution at a time.

Exercise 102.36.2. You are given the following recursive function:

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

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 go the integer before x and ceiling $\lceil x \rceil$ means the integer after x. For instance $\lfloor 3.4 \rfloor = 3$ and $\lceil 3.4 \rceil = 4$.)

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

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

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

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

$$x = [1, 3, 4, 6, 7, 8, 10, 12, 13, 17]$$

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

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

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

$$x = [1, 3, 4, 6, 7]$$

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

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

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

$$x = [6, 7]$$

This will result in either since `target` or when x becomes empty.

Note the the size of the array looks like this:

$$10 \to 5 \to 2$$

Each time, after looking at $x[\texttt{mid}]$, 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 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` tell the function where's the starting index and ending index of the current search. Initially of course $\texttt{left} = 0$ and $\texttt{right} = n - 1$ where n is the size of the array x.

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

Now let's look at a high level overview of mergesort. It's easier to go an example by hand. The main idea when you call mergesort with an array x, 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 array 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 102.36.3. Write a merge function. □

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 102.36.4. 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 roughly An) then call mergesort with these two halves (this requires $2T(n/2)$) and then merge the two return results (this requires roughly Bn). This results in a runtime of $T(n) = 2T(n/2) + An$.

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 merge sort.

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 $T(n) = T(n/2) + ...$
- In the case of mergesort, the mergesort then recursively work on each half (both halves are used), resulting in a runtime that looks like $T(n) = 2T(n/2) + ...$

In general divide-and-conquer algorithms involves splitting up the work into roughly equal smaller pieces, recursively performing the algorithm on the smaller pieces with possibly work done on the return value from the function call to work on the smaller pieces.

Exercise 102.36.5. 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? □

Exercise 102.36.6. Here's our simple sum to n algorithm:

```
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? □

Exercise 102.36.7. 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? □

Exercise 102.36.8. 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. □

Exercise 102.36.9. 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. □

File: master-theorem.tex

102.37 Master theorem

So far I have been talking about recurrences like

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + f(n)$$

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

$$a_n = c_1(n) a_{n-1} + c_2(n) a_{n-2} + f(n)$$

where $c_i(n)$ are also expressions in n. There are also more complex ones like

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

Then there are the divide-and-conquer type recurrence relations. Here's one:

$$a_n = a_{n/2} + 1$$

and here's another

$$a_n = n a_{n/3} + \log n$$

For such recurrences I might write $a(n)$ for a_n. So here's the first example:

$$a(n) = a(n/3) + 1$$

Note in the case of, for instance, runtime analysis the domain of $a(n)$ is the set of positive integers. Therefore we really meant to say

$$a(n) = a(\lfloor n/3 \rfloor) + 1$$

There are times when I *will* be very specific and write

$$a(n) = a(\lfloor n/3 \rfloor) + 1$$

or

$$a(n) = 2a(\lceil n/3 \rceil) + n$$

or

$$a(n) = a(\lceil n/3 \rceil) + a(n - \lfloor n/3 \rfloor) + 1$$

Exercise 102.37.1.

1. Let
$$a(n) = a(\lfloor n/2 \rfloor) + 1$$
and $a(0) = 1$ Compute (by hand) and write down the table for $a(n)$ for $n = 0, 1, 2, ..., 9$.
2. Let
$$b(n) = b(\lceil n/2 \rceil) + 1$$
and $b(0)$ Write down the table for $b(n)$ for $n = 0, 1, 2, ..., 9$.

Compute more values for $a(n)$ and $b(n)$ ... as in *lots* of n's and plot them. Compare $a(n)$ and $b(n)$ when n is large. Can you find good approximation for $a(n)$ and $b(n)$? (Here good mean functions such as n^α, $\ln^\alpha n$, $n^\alpha \ln^\beta n$, c^n, etc. □

Exercise 102.37.2.

1. Let
$$a(n) = 2a(\lfloor n/2 \rfloor) + 1$$
and $a(0) = a(1) = 1$ Compute (by hand) and write down the table for $a(n)$ for $n = 0, 1, 2, ..., 9$.
2. Let
$$b(n) = b(\lfloor n/2 \rfloor) + b(\lceil n/2 \rceil) + 1$$
and $b(0) = b(1) = 1$ Write down the table for $b(n)$ for $n = 0, 1, 2, ..., 9$.

Plot more values for $a(n)$ and $b(n)$ ... as in *lots* of n's. Compare $a(n)$ and $b(n)$ when n is large. Can you find good approximation for $a(n)$ and $b(n)$? (Here good mean functions such as n^α, $\ln^\alpha n$, $n^\alpha \ln^\beta n$, c^n, etc. □

```
def a(n):
    if n in [0, 1]: return 1
    else:
        return 2 * a(n/2) + 1
def b(n):
    if n in [0, 1]: return 1
    else:
        return 2 * b(round(n/2.0)) + 1

for n in range(0, 9):
    print(n, a(n), b(n))

for n in range(100000, 100011):
    print(n, float(a(n))/float(b(n)))
```

```
[student@localhost asymptotics] python tmp1243525234.py
0 1 1
1 1 1
2 3 3
Traceback (most recent call last):
  File "tmp1243525234.py", line 11, in <module>
    print(n, a(n), b(n))
  File "tmp1243525234.py", line 4, in a
    return 2 * a(n/2) + 1
  File "tmp1243525234.py", line 4, in a
    return 2 * a(n/2) + 1
  File "tmp1243525234.py", line 4, in a
    return 2 * a(n/2) + 1
  [Previous line repeated 995 more times]
  File "tmp1243525234.py", line 2, in a
    if n in [0, 1]: return 1
RecursionError: maximum recursion depth exceeded in comparison
```

Exercise 102.37.3.

1. Let
$$a(n) = a(\lfloor n/2 \rfloor) + a(n - \lfloor n/2 \rfloor) + n$$
and $a(0) = a(1) = 1$ Compute (by hand) and write down the table for $a(n)$ for $n = 0, 1, 2, ..., 9$.
2. Plot more values for $a(n)$ and $b(n)$ ... as in *lots* of n's. Can you find good approximation for $a(n)$? (Here good mean functions such as n^α, $\ln^\alpha n$, $n^\alpha \ln^\beta n$, c^n, etc.

□

More generally, we are interested in recurrences like this:

$$a(n) = c_1 \cdot a(n/c_2) + b(n)$$

where c_1 and c_2 are constants.

Example 102.37.1. 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

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

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

Example 102.37.2. 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)
```

Therefore

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

The cost of computing mid and merging is $An + B$.

Let's compute a closed form for $a(n)$. We can perform

$$\begin{aligned} a(n) &= a(n/2) + c \\ &= a(n/2^2) + 2c \\ &= a(n/2^3) + 3c \\ &= \cdots \\ &= a(n/2^m) + mc \end{aligned}$$

At some point $n/2^m = 1$ and we reach

$$a(n) = a(1) + mc$$

From $n/2^m = 1$ we get

$$n = 2^m$$

Hence $m = \log_2 n$. Note we have

$$\begin{aligned} a(n) &= a(1) + c \log_2 n \\ &= A \log_2 n + B \end{aligned}$$

for some constants A and B. Therefore $a(n) = O(\log n)$.

Hmmm ... I wonder if this can be generalized ... Well suppose

$$a(n) = a(n/c_1) + c_2$$

Doing the same computation I get

$$\begin{aligned} a(n) &= a(n/c_1) + c_2 \\ &= a(n/c_1^2) + 2c_2 \\ &= a(n/c_1^3) + 3c_2 \\ &= \dots \\ &= a(n/c_1^m) + mc_2 \end{aligned}$$

At some point $n/2^m = 1$ and hence

$$m = \lg n$$

Therefore

$$\begin{aligned} a(n) &= a(1) + c_2 \lg n \\ &= A \lg n + B \end{aligned}$$

for constants A and B.

Suppose a divide and conquer algorithm looks like this:

```
ALGORITHM A
INPUT: Some work w of size n

Let w1,w2,...,wa be subwork of w of sizes n1,n2,...,na

Pre-recursion work
Perform A on w1
Perform A on w2
...
Perform A on wa
Post-recursion work
```

Let $T(n)$ be the time needed to perform the algorithm on a work of size n. Suppose the time take for non-recursion work is $f(n)$. Suppose that the non-

recursive parts take $f(n)$ time. The total time is then

$$T(n) = T(n_1) + \cdots + T(n_a) + f(n)$$

Now I will assume that

$$n_1 = n_2 = \ldots = n_a = n/b$$

where $a \geq 1$ and $b > 1$. (See ??? for divide-and-conquer where the subwork has different sizes.) The above becomes

$$T(n) = aT(n/b) + f(n)$$

Now I want to know what's the big-O of $T(n)$. But before that ... here are some examples just to put things into context. I'm going to list some runtimes with the corresponding a and b.

Example 102.37.3. Consider the binary search algorithm on a sorted array. We have

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

in the worse case when the target is not found. Of course when the target is at the middle of the array, the algorithm terminates right away – this is of course not the worse case. So in this case,

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

□

Example 102.37.4. Consider the mergesort algorithm on an array. We have

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

So in this case,

$$a = 2 = b, \quad f(n) = An + B = \Theta(n)$$

□

Let's play around with the above runtime recurrence

$$T(n) = aT(n/b) + f(n)$$

It's going to be pretty ugly for a general n. Since we continually have to compute n/b, let's try it with $n = b^k$. Then

$$\begin{aligned}
T(b^k) &= aT\left(b^{k-1}\right) + f(b^k) \\
&= a\left(aT\left(b^{k-2}\right) + f(b^{k-1})\right) + f(b^k) \\
&= a^2T\left(b^{k-2}\right) + af(b^{k-1}) + f(b^k) \\
&= a^2\left(aT\left(b^{k-3}\right) + f(b^{k-3})\right) + af(b^{k-1}) + f(b^k) \\
&= a^3T\left(b^{k-3}\right) + a^2f(b^{k-2}) + af(b^{k-1}) + f(b^k) \\
&= \cdots \\
&= a^kT\left(b^0\right) + \left(a^{k-1}f(b^1) + \cdots + af(b^{k-1}) + f(b^k)\right)
\end{aligned}$$

Now don't forget that I introduced k just to make n look like b^k. So when I need to, I have to wipe out k. This is easy:

$$n = b^k \iff \log_b n = k$$

In other words, later on, I have to replace k by $\log_b n$.

The resulting expression

$$T(b^k) = a^kT\left(b^0\right) + \left(a^{k-1}f(b^1) + \cdots + af(b^{k-1}) + f(b^k)\right)$$

is a sum of two terms so it's a matter of which of the two, i.e.,

$$a^kT\left(b^0\right)$$

or

$$a^{k-1}f(b^1) + \cdots + af(b^{k-1}) + f(b^k)$$

is "bigger", asymptotically speaking.

We can simplify the first term a little:

$$a^kT\left(b^0\right) = \Theta(a^k) = \Theta(a^{\log_b n})$$

$a^{\log_b n}$ is kind of weird since it's not one of the standard representative asymptotic functions like n^α or $\log^\alpha$ or $n^\alpha \log^\beta n$ or c^n or etc. But note that

$$a^{\log_b n} = n^{\log_b a}$$

Why? Look at this:

$$\begin{aligned}(\log_b n)(\log_b a) &= (\log_b a)(\log_b n)\\ \therefore\ \log_b a^{\log_b n} &= \log_b n^{\log_b a}\\ \therefore\ a^{\log_b n} &= n^{\log_b a}\end{aligned}$$

So the first term is

$$a^k T\left(b^0\right) = \Theta\left(a^{\log_b n}\right) = \Theta\left(n^{\log_b a}\right)$$

The second term

$$a^{k-1} f(b^1) + \cdots + a f(b^{k-1}) + f(b^k)$$

looks like a pain. But ... hmmm it sure looks like a geometric sum. Of course I can't say anything more. Why? Well ... there's just nothing to say because I don't have more information about the function $f(n)$. The sum clearly adds up the amount of work (k pieces of work) which takes $f(n)$ time for a bunch of values of n. The question is of course: how much work is involved and therefore how much time, i.e., how big is $f(n)$? Now from some examples such as binary search, mergesort, etc. we can have

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

(think binary search) or

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

(think mergesort). So let's say we impose some condition to make $f(n)$ "nice":

$$f(n) = \Theta(n^c)$$

say

$$f(n) = An^c$$

to be concrete (ignoring unimportant terms in $f(n)$.) Then I can at least say that

$$\begin{aligned}&a^{k-1} f(b^1) + \cdots + a f(b^{k-1}) + f(b^k)\\ &= a^{k-1} \cdot A(b^1)^c + \cdots + a \cdot A(b^{k-1})^c + a \cdot A(b^k)^c\\ &= A\left(a^{k-1}(b^c)^1 \cdots + a(b^c)^{k-1} + (b^c)^k\right)\end{aligned}$$

Well ... too bad ... it's still pretty yucky. It still looks somewhat like a geometric sum. We impose the condition $f(n) = \Theta(n^c)$ to make $f(n)$ nice–

looking. Let's make it even nicer. When I look at the term

$$a^{k-1}(b^c)^1$$

in the above sum (and also the rest), I see that if b^c is an a–power, I would get a sum of a–powers. So how about I checkout the case when

$$b^c = a$$

This means

$$c = \log_b a$$

The above becomes

$$\begin{aligned}
&a^{k-1}f(b^1) + \cdots + af(b^{k-1}) + f(b^k) \\
&= A\left(a^{k-1}(b^c)^1 \cdots + a(b^c)^{k-1} + (b^c)^k\right) \\
&= A\left(a^{k-1}(a)^1 \cdots + a(a)^{k-1} + (a)^k\right) \\
&= Aka^k
\end{aligned}$$

OK, this is not a geometric sum ... but ... at least it's simplified! Since $k = \log_b n$,

$$\begin{aligned}
&a^{k-1}f(b^1) + \cdots + af(b^{k-1}) + f(b^k) \\
&= Aka^k \\
&= A\log_b n \cdot a^{\log_b n} \\
&= \Theta\left(\log_b n \cdot a^{\log_b n}\right)
\end{aligned}$$

Don't forget that the awkward looking $a^{\log_b n}$ is the same as $n^{\log_b a}$. So

$$\begin{aligned}
&a^{k-1}f(b^1) + \cdots + af(b^{k-1}) + f(b^k) \\
&= \Theta\left(\log_b n \cdot a^{\log_b n}\right) \\
&= \Theta\left(\log n \cdot n^{\log_b a}\right)
\end{aligned}$$

(The $\log_b n$ is replaced by $\log n$, since log bases are not important in big-O or big-Θ.)

Altogether we have the following. If $n = b^k$, then

$$T(n) = a^k T\left(b^0\right) + a^{k-1}f(b^1) + \cdots + af(b^{k-1}) + f(b^k)$$

where

$$a^k T\left(b^0\right) = \Theta\left(n^{\log_b a}\right)$$

and if $f(n) = \Theta(n^c)$ where $c = \log_b a$, then

$$a^{k-1} f(b^1) + \cdots + a f(b^{k-1}) + f(b^k) = \Theta\left(\log n \cdot n^{\log_b a}\right)$$

I conclude that

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

Before we celebrate, remember that I made the assumption

$$f(n) = \Theta(n^c), \quad c = \log_b a$$

What if

$$f(n) = \Theta(n^c), \quad c \neq \log_b a$$

We're missing the case $c < \log_b a$ and $c > \log_b a$. Also, what if $f(n) = O(n^c)$? Or what if $f(n) = O(\log n)$? Etc.

Here's the famous **master theorem** ...

Theorem 102.37.1. (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 $b > 1$. Then

(a) Let $f(n) = O(n^c)$ where $c < \log_b a$. Then

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

(b) Let $f(n) = \Theta(n^c \log^k n)$ where $c = \log_b a$. Then

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

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

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

for sufficciently large n, then

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

Example 102.37.5. For binary search, if we let $T(n)$ be the runtime, then

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

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

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 102.37.6. For mergesort search

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

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

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

□

□

Exercise 102.37.4. 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