Search

Searching

Searching is a basic problem in computing

We will look at the binary search algorithm

Most computer scientists think it is trivial, because we've absorbed it into our bloodstreams, and it is the basis of a lot of other ideas

But while we are looking at how to do it, quite a few general issues from computer science will arise

Example problem

Suppose we have an array of letters like this:

'a'	'c'	'd'	'f'	'g'	'h'	'j'	'm'	'n'

And suppose we want to find where the letter `'c'` is

Issue: numbers

One issue that the example problem raises is that computing is ***not*** all about numbers, it is about handling all sorts of data

The example is the same as the one in the wikipedia entry on the binary search algorithm, except that the article uses numbers, giving the wrong impression

Numerical programming is a minority interest - how many numerical apps do you know?

Issue: data as numbers

On the other hand, numerical programming was the starting point for computer science

And numerical operations are what processors provide, and almost all key programming areas need them, e.g. graphics

In the example problem, the letters are sorted, which involves comparing them as numerical codes (`'a'` is stored as character code `97`, `'b'` as `98`, and so on)

Issue: avoiding numbers

Programs shouldn't use numbers unnecessarily

For example, knowing that `'a'` is `97` and `'A'` is `65`, you could convert a letter to upper case by:

```
ch = ch - 32;
```

But you can do it more readably, without the "magic" number:

```
ch = ch + ('A' - 'a');
```

Trust the compiler to optimise

Linear search

Probably the simplest solution is:

```
int search(char ch, int n, char a[n]) {
    int result = -1;
    for (int i=0; i<n; i++) {
        if (a[i] == ch) result = i;
    }
    return result;
}
```

Returning `-1` for an unsuccessful search is a slight cheat, but extremely common (alternatives are having two return values, or using an exception - but C doesn't make either of them easy)

Issue: simplicity

The linear search solution illustrates a simplicity issue: sometimes simplest is best

The linear search algorithm is best if (a) correctness is the most important thing or (b) speed of programming is the important thing or (c) the problem is so small that the linear approach is actually the fastest

The moral is: start simple, and only make things more complicated later if you need to

Early return

The linear search can be speeded up by stopping early:

```
int search(char ch, int n, char a[n]) {
    for (int i=0; i<n; i++) {
        if (a[i] == ch) return i;
    }
    return -1;
}
```

This has hidden complexity: (a) the `for` statement ***says*** its going to run through all the indexes, but that's a ***lie*** and (b) the function returns from two different places

Logical simplicity

We've seen that some programmers think it is better to write this, which has no ***hidden*** complexity:

```
int search(char ch, int n, char a[n]) {
    int result = -1;
    bool found = false;
    for (int i=0; i<n && ! found; i++) {
        if (a[i] == ch) {
            result = i;
            found = true;
        }
    }
    return result;
}
```

Issue: optimisation

Stopping a search early is a small optimisation (though a natural one)

When should you optimise?

Optimisation rules

Things you should know about optimisation are:

Bottlenecks: slow speed is almost always due to a bottleneck (so it is not a good idea to try to make every part of your program fast)

Measurement: guesses about where bottlenecks are usually wrong, so find out by measuring

Optimise last: that means it is best to put correctness and readability first, and do optimisation last

Can we do better?

Is there a better way to solve the searching problem?

If you wanted to look up a word in a printed dictionary, would you go through each page one by one?

No – you would open the dictionary in the middle, work out which half your word was in, and then repeat the halving process

The binary search algorithm

This algorithm is poorly named, because it doesn't have much to do with binary, except for a vague "twoness"

A better name is "interval halving search"

The idea of the algorithm is: when searching in a sorted array, repeatedly halve the range that you are looking in

How difficult?

The idea of the algorithm is easy, but the devil is in the details

It is easy to program, but also easy to get wrong

It has been said that most professional programmers and most textbooks get it wrong (because it is natural to program partly by logic and partly by trial and error)

Let's look at what it takes to get it right

Loop invariant

As we go round the loop, what do we keep track of?

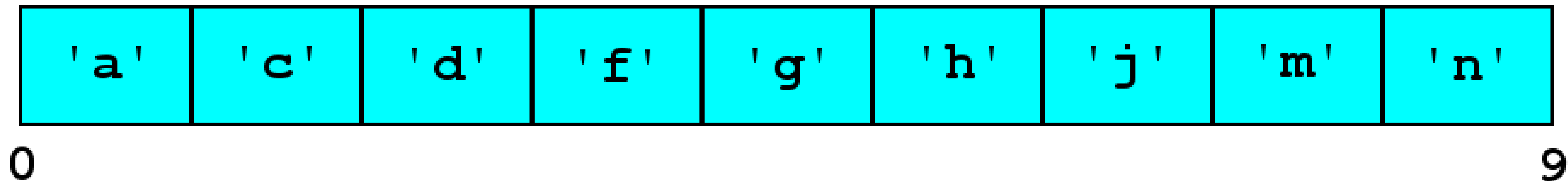
The answer is indexes `start` and `end` which represent the range we have narrowed the search down to

But does this mean "the item might be at an index from `start` to `end` inclusive" or "the item is between positions `start` and `end`", i.e. are we counting or measuring the array?

We ***must*** choose one, stick to it, and not get confused

Getting started

I'm going to choose measuring (unlike Wikipedia)




```
int search(char ch, int n, char a[n]) {
    int start = 0, end = n;
    ...
}
```

Loop end

When does the loop end?

Either when the item we look at is the one we want, or if the range has nothing left in it

```
int search(char ch, int n, char a[n]) {
    int start = 0, end = n;
    bool found = false;
    while (! found && end > start) {
        ...
    }
}
```

Issue: pointers

In the early days of computing, it would have been regarded as better (more efficient) to use pointers than index numbers for the range

But now we think (a) indexes are better for humans, making correctness more likely and (b) indexes are better for the compiler, because there are more possibilities for optimising

Middle choice

Within the loop, we are going to look at the middle element

The middle is the average of `start` and `end`

```
int mid = (start + end) / 2;
```

The integer division handles the case when there isn't an exact middle element, which is fine

But is this the right way to find the average?

Overflow bug

This line has an overflow bug

```
int mid = (start + end) / 2;
```

If `start` and `end` are around a billion, the addition may go over the 2-billion limit on `int`

Is it possible to avoid the overflow bug? Yes!

```
int mid = start + (end - start) / 2;
```

Now the program works over the whole range of `int`

Issue: cockroaches

Some people would say this isn't an important bug, because it is unlikely that a user will have an array of size over a billion

But a better point of view is that this is the worst possible kind of bug (a cockroach?)

Testing doesn't detect it, and it is too rare to get reported, so it survives forever, waiting to bite

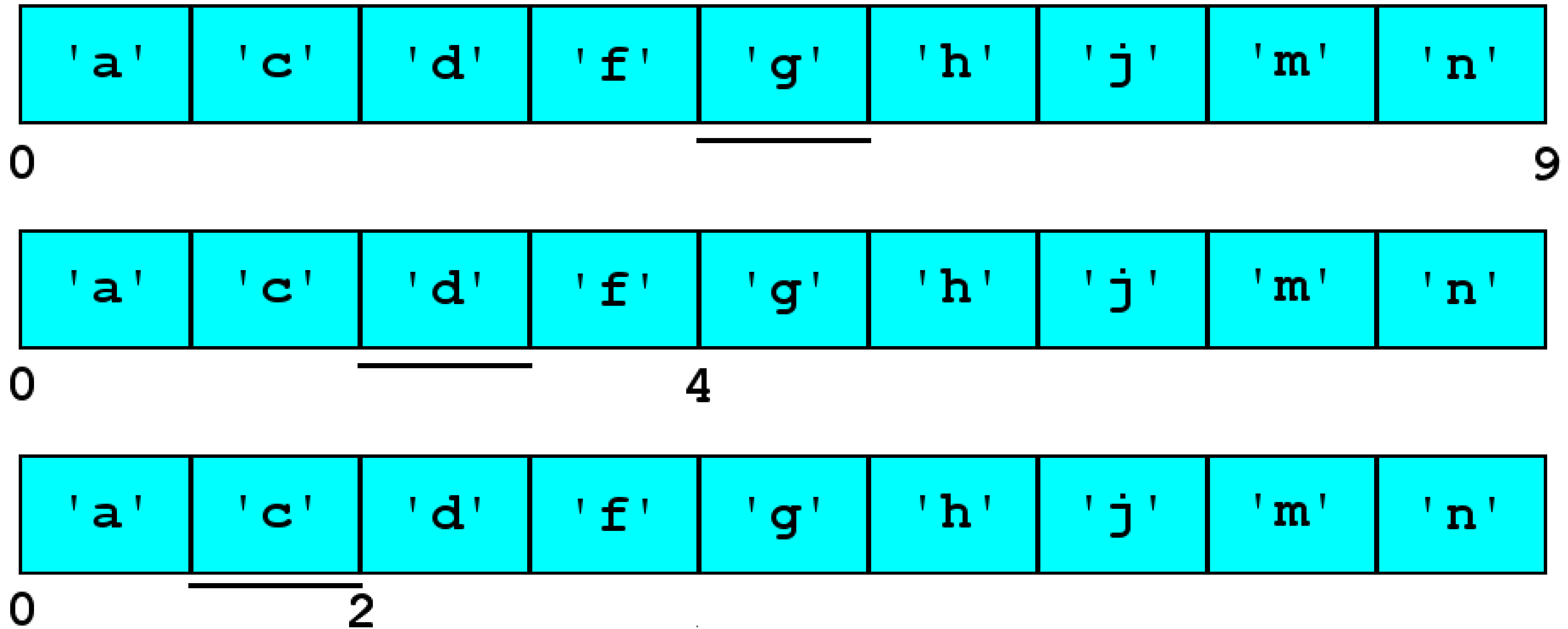
Loop test

We can finish off the function:

```
int search(char ch, int n, char a[n]) {  search.c
    int start = 0, end = n, mid;
    bool found = false;
    while (! found  && end > start) {
        mid = start + (end - start) / 2;
        if (ch == a[mid]) found = true;
        else if (ch < a[mid]) end = mid;
        else start = mid + 1;
    }
    return found ? mid : -1;
}
```

Visualise

You should visualise, and test, some searches, including failures - searching for `'c'` looks like this:



Issue: optimisation

Many programmers would instinctively try to optimise by making only one test (`<=` or `>`, say) instead of two, and check equality at the end after finding the item

- this is less symmetrical, so more liable to error
- a common error is an occasional infinite loop because the range doesn't decrease
- there is only an improvement if there are a quintillion items, way beyond `int`, i.e. never

Issue: precision

This chapter illustrates that woolly programming is useless - you need to think or draw pictures to get your head straight before programming

It also illustrates that trial-and-error programming in general is useless - it doesn't get to the right answer and, like evolution, it is slooooow

As a programmer, you need to be aiming for 100% precision

But it's not quite as hard as it sounds, because the computer itself helps you to get there

Issue: automated testing

There are various techniques you can use to program better, but in the end, you can only achieve precision by proof or testing

Most of the time, proof is too expensive (and difficult), so the heart of good development is testing

Testing is potentially boring, but whenever anything is boring, the computer should be doing it, so what you want is ***automated testing***

That is the first sign of a developer, rather than just a programmer

Linear time

Linear search takes some constant time to get started, plus some constant times `n` steps, i.e. `a + b * n`

The constants depend on the computer, the language, the compiler etc. - impossible to estimate

So we ignore the constants and say that it takes ***O(n)*** time ("big O of n time" or "order n time" or "linear time")

This is called "big O notation" or "computational complexity"

Log time

For binary search, the time is the number of times you have to halve ***n***

If ***n=2,4,8,16,32,64,128,256,512,1024***, you can halve it ***1,2,3,4,5,6,7,8,9,10*** times

This is the "logarithm to base ***2***" of ***n***, i.e. what power of ***2*** makes ***n***

Logs with different bases only have constant factor differences, so we just write ***O(log(n))*** for the time binary search takes

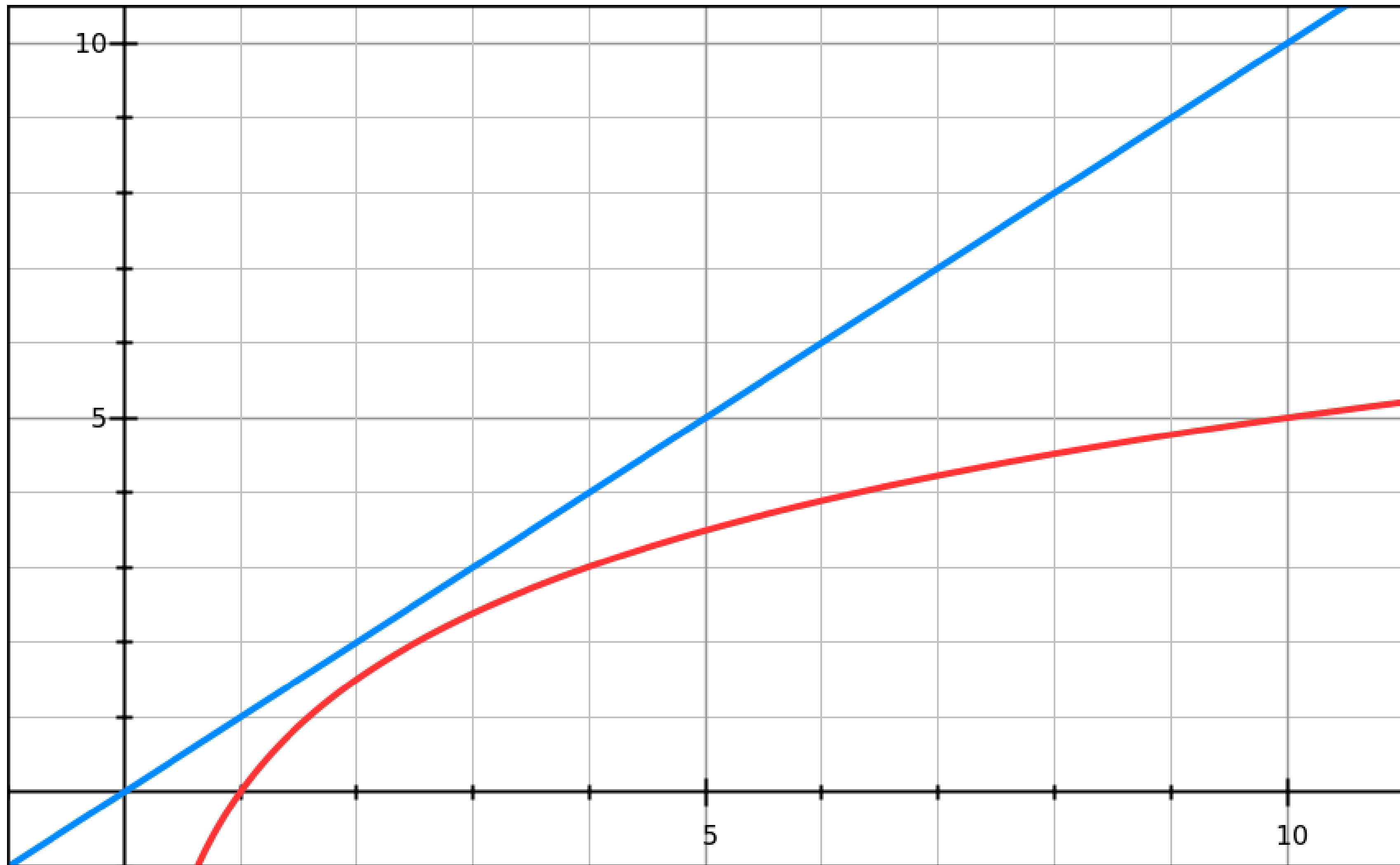
Graphs

If you like graphs ('charts') go to graphsketch.com and put in ***f(x)=x*** and ***g(x)=5*log(x)***

The ***5**** is a wild guess representing the extra logical complications of the binary search algorithm

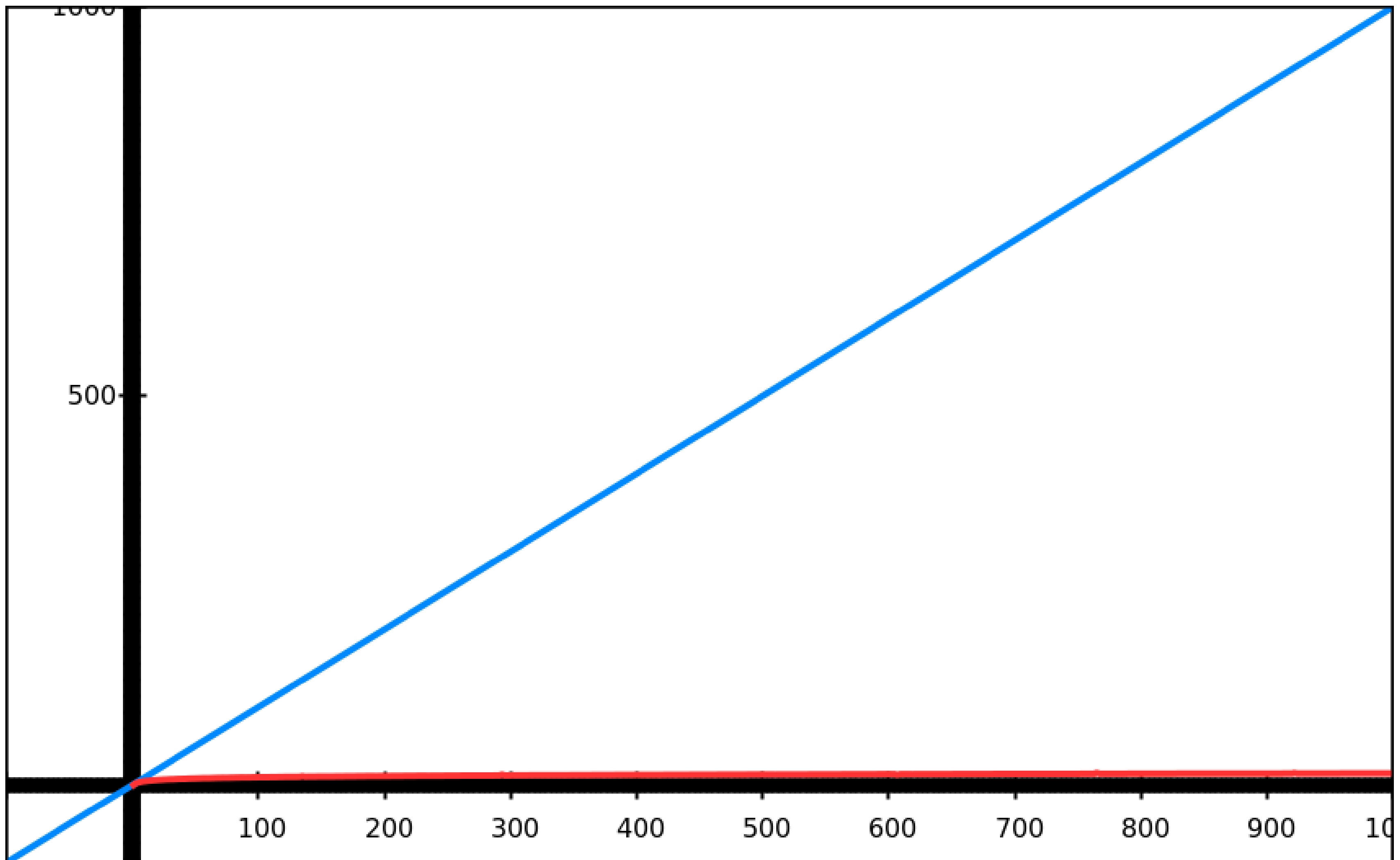
Graphs

For small numbers, the difference is negligible:



Graphs

For big numbers, the difference is 'infinite':



Issue: design

Although little optimisations should be left to the end, if done at all, optimisations which change the ***order*** of efficiency may be important

What they depend on most is the overall design of the program, before you start writing it

So the last word on optimisation is "design efficiency in, but don't sweat the small stuff", i.e. pay attention to ***design*** efficiency, but not ***code*** efficiency