A closer look at big-*O* notation.

We all know that in a formula $y = ax + b$ the values of both a (slope) and b (intercept) are important.

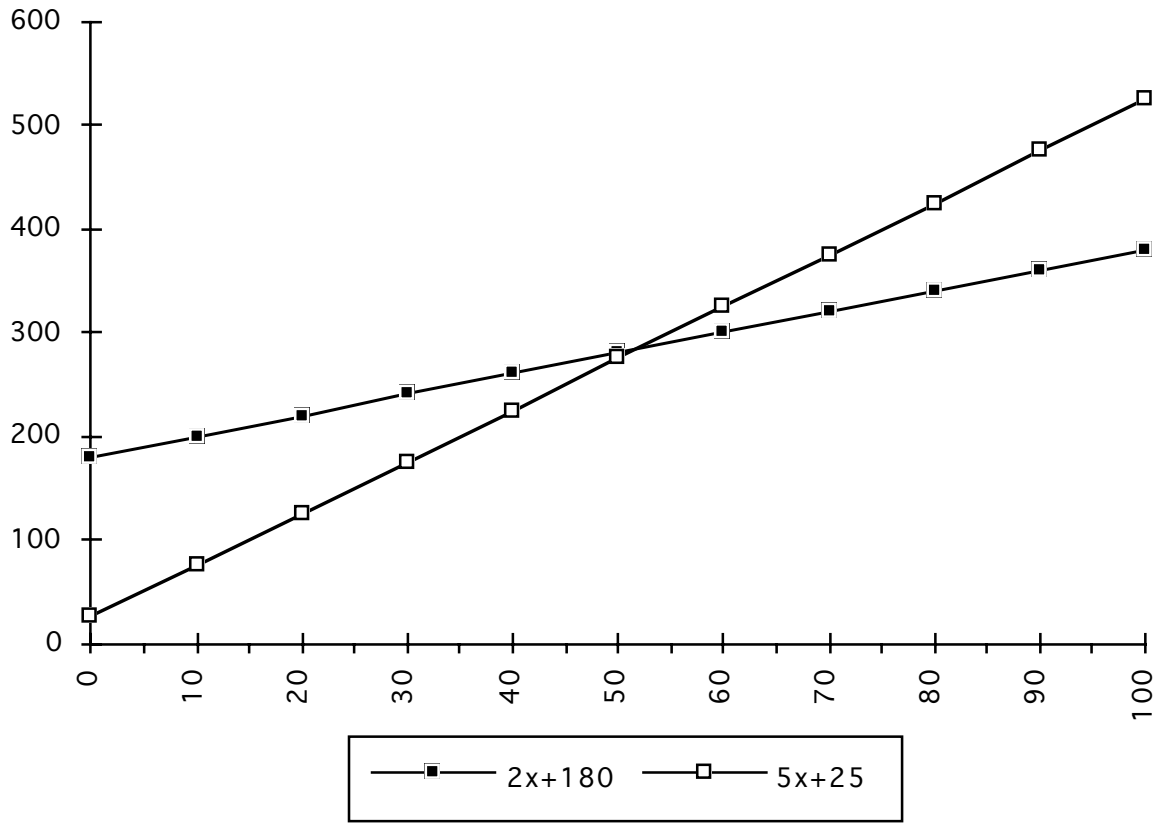
If y is the cost of executing a program on a problem of size x, then

- b determines the value at $x = 0$ – the ***fixed cost*** of execution;
- a determines how fast the cost grows as the problem size increases.

a is the ***constant of proportionality*** of a linear-cost program.

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

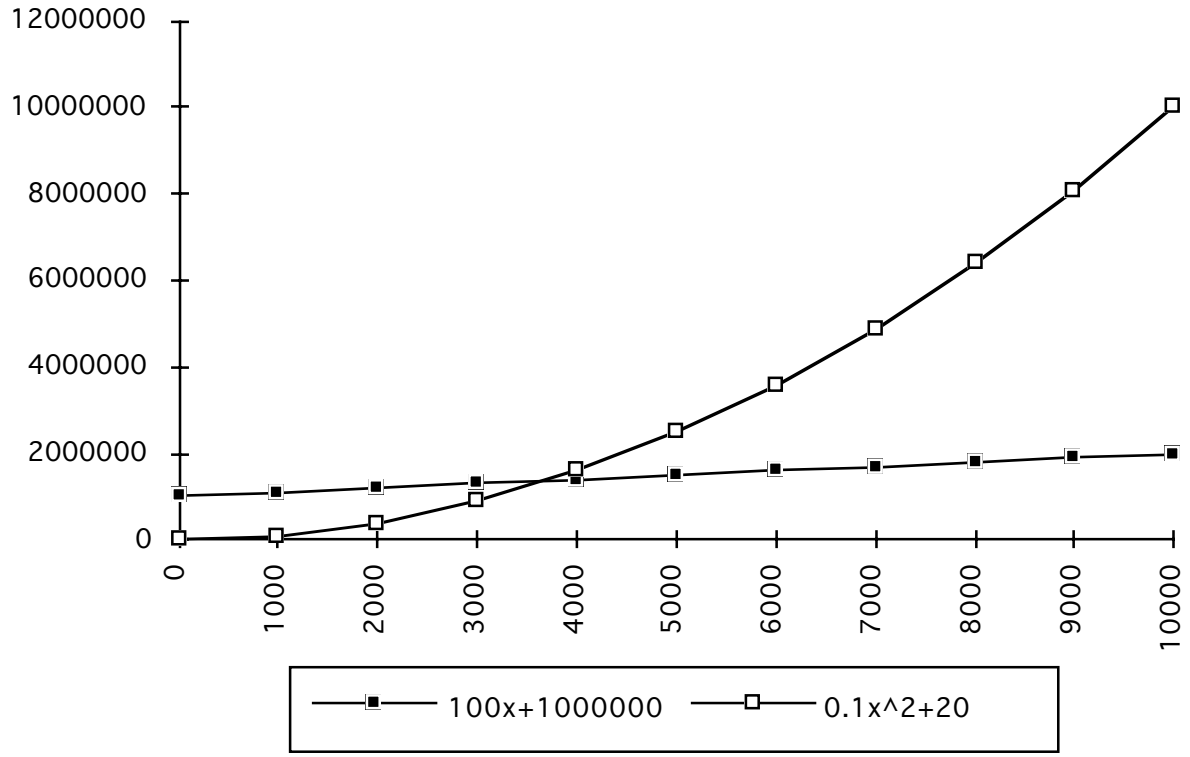
It's obviously untrue that all linear-cost programs have the same execution time:



The $2x$ cost grows more slowly, so the corresponding program is to be preferred on 'sufficiently large' problems.

But all the problems we ever consider may be smaller than 50, so the $5x$ program might be better for us.

You should already be persuaded that whatever the constants of proportionality, x^2 formulæ will overtake x formulæ at sufficiently large values of x:



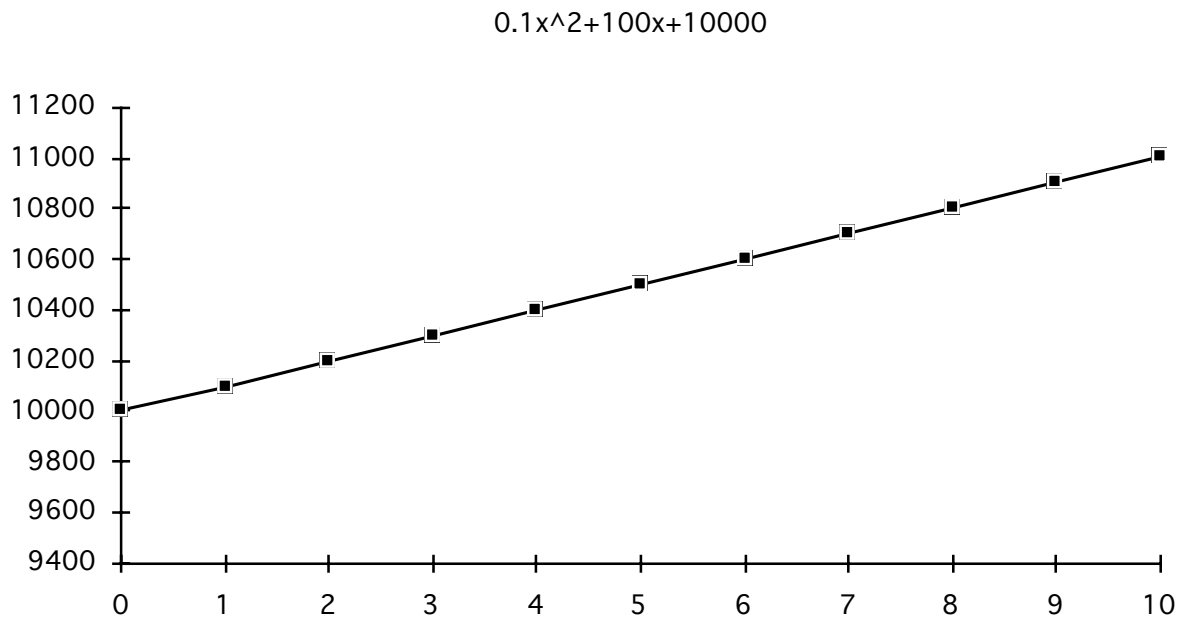
No matter what the disparity in fixed costs (20 vs 1 million), no matter what the cost of the inner loop (0.1 vs 100), the quadratic program will cost more than the linear program on sufficiently large problems.

> *And the same is true for all the other powers: $O(N^k)$ is worse than $O(N^j)$ on sufficiently large problems whenever $k > j$ no matter what the fixed costs or the constants of proportionality.*

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

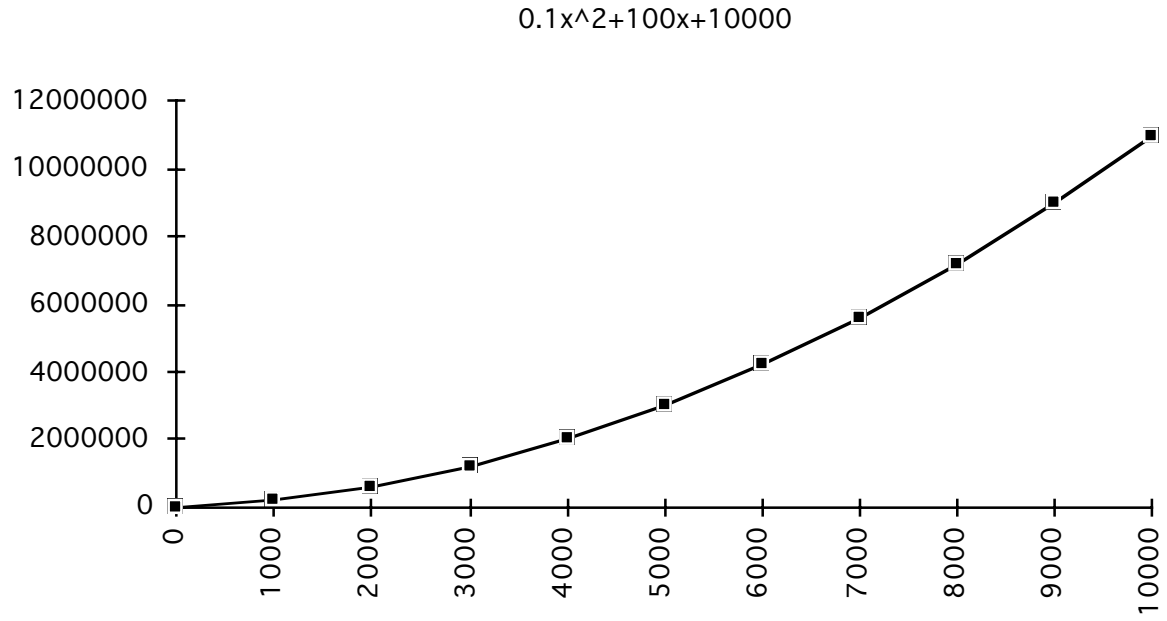
It doesn't matter if a quadratic program has a large linear component. Eventually it will grow just like x^2.

At small scales it might look linear:



Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

Over a larger scale it looks simply quadratic:



The higher power eventually ***dominates*** the lower, no matter what the constants of proportionality.

So if the cost of executing a program on a problem of size N is given by a polynomial formula $a_0 + a_1 N + a_2 N^2 + ... + a_n N^k$ where $a_n \neq 0$, we say it is $O(N^k)$, ***neglecting smaller powers of N*** (because on large problems N^k will dominate).

And then we say that $O(N^k)$ is to be preferred to $O(N^j)$ whenever $k > j$ ***neglecting the constants*** $a_0, a_1, ..., a_n$ (because on large problems N^k will dominate N^j).

This notation is ***a convenient approximation***.

- It shouldn't tempt us to neglect the constants of proportionality when comparing two $O(N^k)$ algorithms.
- We should be aware that $O(N^k)$ may be worse than $O(N^j)$ on small problems, even though $k > j$.
- Experiment rules.

No interesting algorithm is $O(N^k)$ where $k < 0$. I hope you can justify this assertion.

One last wrinkle.

We sometimes write algorithms which are mixed, because of different constants of proportionality.

For example, an algorithm which is $O(N^2)$ for small values of N, and $O(N \lg N)$ for larger values – because the N^2 algorithm is quick and easy to set up on small problems, perhaps.

Such an algorithm, in the limit, is $O(N \lg N)$.

Hence the definitions on p121 of Weiss. Big-O notation gives upper bounds on execution costs.

> *He also gives definitions of* $\Omega(...)$ *(big-Omega, a notation for lower bounds),* $\Theta(...)$ *(big-Theta, upper and lower bounds) and* $o(...)$ *(little-o, upper bound only).*

In this course we are mostly concerned with ***worst-case*** calculations, and with finding an ***upper bound*** on the worst case of a program's execution.

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

On logarithms: *log*, *ln* and *lg*.

In various examples, as we shall see, we prefer $O(\lg N)$ to $O(N)$, because $\lg N$ grows more slowly than N.

When $N > 0$ and $b^x = N$, we say that $\log_b N = x$.

Here b is the ***base***, and x is the ***logarithm***.

$\log_b N$ is the power to which you must raise b to get N.

A logarithm is rarely a whole number ...

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

Fact 0. $b^{\log_b N} = N$.

That's the definition of a logarithm!

Fact 1. If $N = J \times K$, then $\log_b N = \log_b J + \log_b K$.

$b^{x+y} = b^x \times b^y$, *so* $b^{\log_b J + \log_b K} = b^{\log_b J} \times b^{\log_b K} = J \times K = N$.

This is why logarithms were popular in my schooldays: they convert multiplication problems into addition problems.

Fact 2. If $\log_b N = x$, then $\log_b\left(N^2\right) = 2x$.

$b^{2x} = b^x \times b^x = N \times N = N^2$

Fact 2a. In general, $\log_b\left(N^y\right) = y \log_b N$.

Another reason for the popularity of logarithms: they convert exponentiation problems into multiplication problems.

Fact 3. $\log_b N = k \log_c N$, where k is a constant.

$N = c^{\log_c N}$, *by definition.*
$\log_b N = \log_b \left(c^{\log_c N} \right)$, *taking* $\log_b$ *of both sides.*
$\log_b \left(c^{\log cN} \right) = \log_c N \times \log_b c$, *by fact 2a.*
$\log_b c$ *is a constant, because c is a constant.*
So $\log_b N = k \log_c N$, *where k is a constant.*

So the base doesn't matter in big-O calculations.

Therefore $O(\log_b N)$ programs run just like $O(\log_c N)$ programs, neglecting constants of proportionality.

Computer scientists are especially interested in base 2.

For all sorts of reasons:

- $\lg N$ is the number of bits in the binary numeral representation of N;

  therefore $\lg N$ *is the number of bits needed to represent all the numbers 0..N in binary numeral notation;*

- $\lg N$ is the number of times you must double (starting from 1) before you reach or exceed N;
- $\lg N$ is the number of times you must halve (starting from N) before you reach 0.

The last point is the crucial one in this course: we shall consider algorithms which work by ***repeated halving***, stopping when they reach a problem of size 0 (in $\lg N$ steps) or 1 (in $\lg N - 1$ steps).

For these reasons we use a special notation for base-2 logarithms.

Calculating execution costs.

Mostly addition and multiplication.

All costs assessed on the kind of machine we are using as a model: sequential, no significant parallel executions.

0. The cost of arithmetic, comparison and storage operations is constant in time and zero in space.

Some arithmetic or comparison operations might take longer than others, because of the size of the data. This does not contradict point 0.

T1. The execution time of (time taken to evaluate) the formula `f1 op f2` is $T_{\texttt{f1}} + T_{\texttt{f2}} + T_{\texttt{op}}$, where $T_{\texttt{op}}$ is some small constant depending on the operator `op` and the types of the formulæ `f1` and `f2`.

What goes for binary operators goes similarly for all the other kinds of operators - but see below for choice instructions and choice formulæ.

T2. If the execution time of I_1 is T_1 , and the execution time of I_2 is T_2, then the execution time of $I_1; I_2$ is $T_1 + T_2$.

T3. The execution time of the instruction `for (INIT; COND; INC) BOD` is $T_{\texttt{INIT}} + T_{\texttt{COND}} + \left(T_{\texttt{BOD(v0)}} + T_{\texttt{INC}} + T_{\texttt{COND}}\right) + \ldots + \left(T_{\texttt{BOD(vN)}} + T_{\texttt{INC}} + T_{\texttt{COND}}\right)$, where $v_0, v_1, \ldots, v_N$ are the successive values set up by `INIT` and `INC` to control the execution of `BOD`.

It follows that if $T_{\texttt{BOD}}$ is independent of the values v_i, and if $T_{\texttt{COND}}$, $T_{\texttt{INC}}$ and $T_{\texttt{BOD}}$ are all $O(f(N))$ execution time and $T_{\texttt{INIT}}$ is $O(f(N))$ or better, then the for *is $O(N \times f(N))$ execution time.*

while *instructions can be treated as a special kind of* for, *without* `INIT` *or* `INC`.

I think I can neglect the cost of jumps.

T4. The execution time of the instruction `if (COND) THEN else ELSE` is either $T_{\texttt{COND}} + T_{\texttt{THEN}}$ (if `COND` is non-zero) or $T_{\texttt{COND}} + T_{\texttt{ELSE}}$ (otherwise).

The same goes for choice formulæ `COND ? THEN : ELSE`.

Single-armed choice instructions `if (COND) THEN` *can be treated as* `if (COND) THEN else {}`

I neglect the cost of jumps.

T5. The execution time of the block `{ decls instrs }` is $T_{\text{decls}} + T_{\text{instrs}}$.

T6. The execution time of the variable declaration *type* `x` is a small constant $T_{\texttt{varalloc}}$; the execution time of the initialised declaration *type* `x = val` is $T_{\texttt{varalloc}} + T_{\texttt{val}}$.

Variable allocation is pretty cheap, but initialisations can be as costly as you like.

Variable declared in for *instructions are allocated by the smallest enclosing block, but the initialisation takes place when the* for *is executed.*

T7. The execution time of the method declaration *type* `f(`*params*`)` is zero.

Declaration is cheap, but execution may be expensive.

T8. The execution time of the method (function / procedure) call `f(`*args*`)` is a small constant $T_{\texttt{call}}$ plus the time to evaluate the arguments *args* and the time to execute the method body.

T9. The execution time of the formula `new` *class*`(`*args*`)` is difficult to determine. It includes at least the time to evaluate the arguments *args* and to execute the class body, considered as a block.

The difficulty arises because this formula requires use of a garbage collector.

S1. If the space used by I_1 is S_1 , and the space used by I_2 is S_2, then the space used by $I_1;I_2$ is $S_1 \cup S_2$.

Space can be reused; time can't be.

Space can be ***reclaimed*** *and reused.*

Space is allocated in two ways: in variable declarations and in new *formulæ.*

S2. The space allocated by the variable declaration *type* `x` is a small constant $S_{\texttt{var}}$. The space is reclaimed when the block which contains the declaration terminates.

The same goes for variable declarations in for *instructions.*

S3. The space allocated by the method declaration *type* `f(`*params*`)` is a small constant $S_{\texttt{method}}$. The space is reclaimed when the block which contains the declaration terminates.

S4. The block which is a method body terminates when the method returns.

So the variable and method space allocated by that block execution is reclaimed when the method returns.

S5. The space allocated by a `new` formula is that allocated by the declarations in the corresponding `class` body. It is reclaimed only when the garbage collector is good and ready.

We don't know when the garbage collector will be ready: it depends on all sorts of difficult considerations.

*In effect there are two kinds of space: variable/method (**stack**) space and object (**heap**) space.*

Some examples.

We try to work inside-out, calculating the properties of the smallest components first.

1. `for (int i=n; i>m; i--) A[i]=A[i-1];`

   $T_{\texttt{INIT}}$, $T_{\texttt{COND}}$, $T_{\texttt{INC}}$ *and* $T_{\texttt{BOD}}$ *all* $O(1)$ *(constant) execution time: time taken by the* for *is therefore* $O(m-n)$ *(when* $m < n$*) or* $O(1)$ *(when* $n = m$*).*

   The for *executes just one declaration: space used is therefore* $O(1)$.

2.
```
int common = false;
for (int i=0; i<N; i++)
  for (int j=0; j<M; j++)
    if (A[i]==B[j]) common = true;
```

The if *(line 4) consists of a constant-time test and a constant-time assignment. It's worst-case constant time,* $O(1)$.

The inner for *(lines 3-4) has constant-time components, and executes its body M times. It's worst-case linear time,* $O(M)$.

The outer for *(lines 2-4) has constant-time* INIT, COND *and* INC, *and its* BOD *has an execution time independent of i and* $O(M)$. *So the outer* for *is* $O(N \times M)$.

The whole is a constant-time declaration followed by an $O(N \times M)$*-time* for*; the whole is* $O(N \times M)$ *in execution time.*

The if *allocates no space: it's* $O(0)$ *in space.*

The inner for *is equivalent to a block which allocates one variable: it's* $O(1)$ *in space.*

The outer for *allocates one variable and repeatedly executes the inner* for*, a block which begins by allocating one variable and ends by reclaiming it. So the outer* for *uses two variables: it's* $O(1)$ *in space.*

The whole allocates one variable and then executes a for *which is* $O(1)$ *in space: the whole is* $O(1)$ *in space*

3.
```
for (int i=0; i<N; i++) {
  Value min = A[i];
  int minp = i;
  for (int j=i+1; j<N; j++)
    if (A[j]<A[i]) {
      min = A[j]; minp = j;
    }
  A[minp] = A[i]; A[i] = min;
}
```

The if *instruction (lines 5-7) has a constant-time test and a constant-time sequence of assignments. It's worst-case* $O(1)$ *in execution time.*

The inner for *(lines 4-7) has constant-time components, and executes its body* $N-i-1$ *times. It's worst-case linear in execution time,* $O(N-i)$.

Lines 2, 3 and 8 are $O(1)$*, and lines 4-7 are* $O(N-i)$*, so lines 2-8 are* $O(N-i)$ *in execution time.*

The outer for *has constant-time* `INIT`*,* `COND` *and* `INC`*, and a* `BOD` *whose execution time depends on* $N-i$*. So its execution time is* $O(1)+O(N)+O(N-1)+\ldots+O(0)$*, which is a triangular pattern whose area is proportional to* $N\times(N+1)/2$*, and that makes it* $O(N^2)$ *in execution time.*

Space analysis of example 3 is similar to example 2: lines 2-8 allocate three variables and so are $O(1)$ in space.

The whole is equivalent to a block which allocates one variable and then repeatedly executes a block (lines 2-8). That block repeatedly allocates and reclaims three variables. So the whole uses four variables: it's $O(1)$ in space.