

Algorithms, Order, Sorting

PHYS2G03

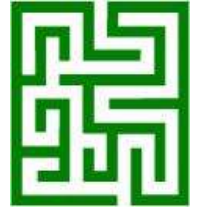
© James Wadsley,
McMaster University

Problems and Algorithms

If a problem has a known solution method, then it should also be possible to clearly state the way to solve it

When a solution has several steps it is called an **algorithm** to solve the problem

An Algorithm



- Problem: Finish a maze with a single entrance and exit on the outer boundary

- Solution (Right-hand Algorithm):
 1. Proceed forward to an intersection
 2. If you are at the exit you are **done**
 3. Turn to face the first route on the right
 4. **Go to** 1 and repeat

Algorithms

An algorithm doesn't have to be the fastest or smartest way to solve the problem

For programming people tend to choose algorithms that are:

1. Easy to program
2. Guaranteed to finish

Algorithms built of algorithms


- A complex problem can usually be broken down into manageable parts:

This is the key to programming

- Each part may have one or more algorithms that could be used to solve it
- In sketching out an algorithm its acceptable to leave out details of self-contained parts for which you have a solution

Algorithm for Theseus' Quest

1. Go to Crete
2. Go to Maze
3. Solve Maze
4. Kill Minotaur
5. Sign Autographs



**The algorithm
for this part
could be the
right hand one
or involve a
ball of string**

Algorithms and Pseudo-Code

When you are designing a program, you want to come up with logical steps to complete the task

- An algorithm is a general description
- A description that is similar to a programming language is called pseudo-code

Pseudo-Code

- Pseudo Code is an algorithm written out in the way you'd program it
- Pseudo-code doesn't have to be specific to any one language
- Pseudo-code reflects the common features of programming languages

Mathematical Algorithms

- Some problems can be solved in closed form via application of a formula

e.g. Solve: $ax^2+bx+c = 0$ for x

Answer: $x = (-b \pm \sqrt{b^2-4ac})/2a$

This is a very simple kind of algorithm (but efficient)

Mathematical Algorithms

Others may require multiple steps with decisions or repetition

- Long Division of integers
- Euclid's Algorithm to find the Greatest Common Divisor (GCD)
- Estimating π or $\sin(x)$
- Calculate constant gravity: ballistics with drag
- Calculating forces between objects (e.g. gravity or electrostatics)
- Sorting numbers

(This kind of thing is what programmers usually think off when they think about Algorithms)

Complexity of Algorithms:

Order of Algorithms


Order of Algorithms

- The order of an algorithm (sometimes called complexity of the algorithm) is determined by the leading (largest) term in the expression for the number of instructions
- When stating order a multiplicative constant is neglected
- Order N is written $O(N)$
(often a script O is used: $\mathcal{O}(N)$)

Order Examples:

- A single loop

```
for (i=0;i<n;i++) {  
    statement1  
    statement2  
    ...  
}
```



This is $O(n)$ – the number of times the
“work” is done is n

Example: Ballistics with drag

$$a = \frac{dv}{dt} = g - D v^2$$

- The behavior of any particle only depends on its own properties (velocity v , acceleration a)
- So for n particles, I'd only need to loop over them once to calculate acceleration: $O(N)$
- A single loop

```
for (i=0; i<n; i++) {  
    // Calculate drag on particle i  
    a[i] = g - D * v[i] * v[i];  
}
```

Order Examples:

- A single loop
for (i=0;i<n/2;i++) {
 statement1
 statement2
}

This is also $O(n)$ – the number of steps is linear in n

Order Examples:

- A double nested loop

```
for (i=0;i<n;i++) {  
    for (j=0;j<n;j++) {  
        statements ...  
    }  
}
```

} This work done
n² times

This is $O(n^2)$ – the number of steps goes as n squared

Example: Gravity in 1D

$$\frac{d v_i}{dt} = \frac{G M_j (x_j - x_i)}{|x_j - x_i|^3}$$

- The behavior of any object depends on how close it is to nearby objects (with positions x , velocities v , acceleration a and mass m)
- So for each one of my n particles, I'd only need to loop all the neighbours to get the total acceleration: $O(N^2)$

```
for (i=0; i<n; i++) {  
    a[i]=0;  
    for (j=0; j<n; j++) {  
        if (i==j) continue; // skip yourself!  
        //Calculate acceleration for particle i  
        a[i] = a[i] + G*M[j]*(x[j]-x[i])/  
            abs((x[j]-x[i])*(x[j]-x[i])*x[j]-x[i]);  
    }  
}
```

Note “brute force” gravity like this is $O(N^2)$ in 3D as well
– depends on number of objects

Order Examples:

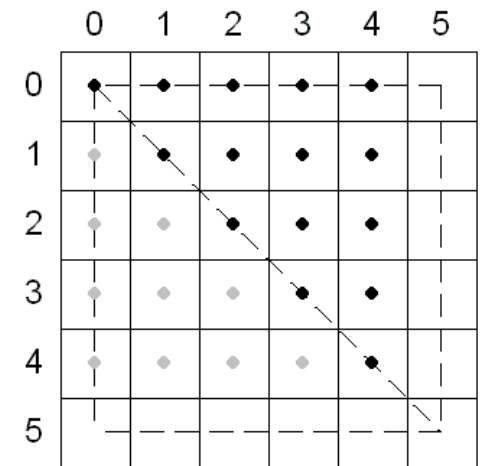
- A double nested do loop with a twist
for (i=0;i<n;i++) {
 for (j=0; **j<=i;** j++) { // loop 0 to i
 statementA;
 }
}

What is the order of this example?

[Pause the lecture here a bit and think...]

Order Examples:

- A double nested do loop with a twist
for (i=0;i<n;i++) {
 for (j=0;j<=i;j++) { // loop 0 to i
 statementA;
 }
}



Total times statementA executed
= $\frac{1}{2} n(n+1)$ Leading term is n^2 so $O(n^2)$

Example: Gravity in 1D (alternate)

$$\frac{d v_i}{dt} = \frac{G M_j (x_j - x_i)}{|x_j - x_i|^3}$$

- The behavior of any particle depends on how close it is to nearby particles (with positions x , velocities v , acceleration a and mass m)
- So for each one of my n particles, I'd only need to loop all the neighbours to get the total acceleration: Still $O(N^2)$ – no work saved directly but does make equal and opposite force thing a bit clearer

```
// zero accelerations first
for (i=0;i<n;i++) a[i]=0;
// Now add in contributions
for (i=0;i<n;i++) {
    for (j=0;j<i;j++) {
        //Calculate acceleration for particle i and j
        f = G*(x[j]-x[i])/
            abs((x[j]-x[i])*(x[j]-x[i])*x[j]-x[i]);
        a[i] = a[i] + f*M[j];
        a[j] = a[j] - f*M[i];
    }
}
```

Order Examples: Recursion

- Good recursive algorithms tend to be logarithmic: $O(\log_2 N)$
- Anything else is likely to result in using up a lot of stack memory
- e.g. Recursive factorial is not great (see: `/home/2G03/func/factorial.cpp`)
 - Factorial costs $O(N)$ to but it calls itself N times and has big memory overhead

Recursive: Fibonacci Algorithm

$$F_n = F_{n-1} + F_{n-2}$$

- What is the Order of a solution with a recursive method?
- What is the theoretical best you can do using the formula ?

Fibonacci main program

```
int main() {  
    // A Program for calculating fibonacci numbers using a recursive function.  
    int n;  
    std::cout << "Welcome to the Fibonnaci program. Enter number n\n";  
    std::cin >> n;  
    std::cout << "The " << n << "th Fibonacci number is " << fib(n) << "\n";  
}
```

Fibonacci function

Recursive: calls itself

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

Results of recursive function

- Function `fib(n)` recursive call generates 2 calls every time it is called
- What is the order of this approach to calculating the Fibonacci numbers?

[Pause the lecture here a bit and think...]

Results of recursive function

- Function `fib(n)` recursive call generates 2 function calls every time it is called
- What is the order of this approach to calculating the Fibonacci numbers?
- Fibonacci `fib(N)` results in $O(2^N)$ calls total
(This is absolutely brutal)

Results of recursive function

- Function fib(n) recursive call generates 2 function calls every time it is called

- Fibonacci fib(N) $O(2^N)$ calls total

- Note:

 - $O(2^N)$ CPU Work

 - $O(2^N)$ Memory Usage!

Is there a better way?

Fibonacci – a better way

- If you store the last two Fibonacci numbers you can count up to any Fibonacci number you want
- This is a simple loop and is thus $O(N)$

Fibonacci – a better way

Fibonacci function

```
int fib(int nend) {  
    int n,F_n,F_n_minus_1, F_n_minus_2;  
    if (nend<=1) return 1;  
  
    F_n_minus_2 = 1;  
    F_n_minus_1 = 1;  
    n = 2;  
    for (;;) {  
        F_n = F_n_minus_1 + F_n_minus_2  
        if (n == nend) return F_n;  
        F_n_minus_2 = F_n_minus_1;  
        F_n_minus_1 = F_n;  
        n++;  
    }  
}
```

Overall Order

- Complex algorithms have several parts
- One part may be $O(N^2)$ another $O(N\log_2 N)$ and so forth ...
- Unless a specific range for N is given, we normally talk in terms of the leading terms for very large N

Overall Order

- Repeatedly applying an algorithm increases the overall order
- Applying an $O(N)$ algorithm $N/2$ times results in an $O(N^2)$ algorithm
- Applying an $O(\log_2 N)$ algorithm $3N$ times results in an $O(N \log_2 N)$ algorithm

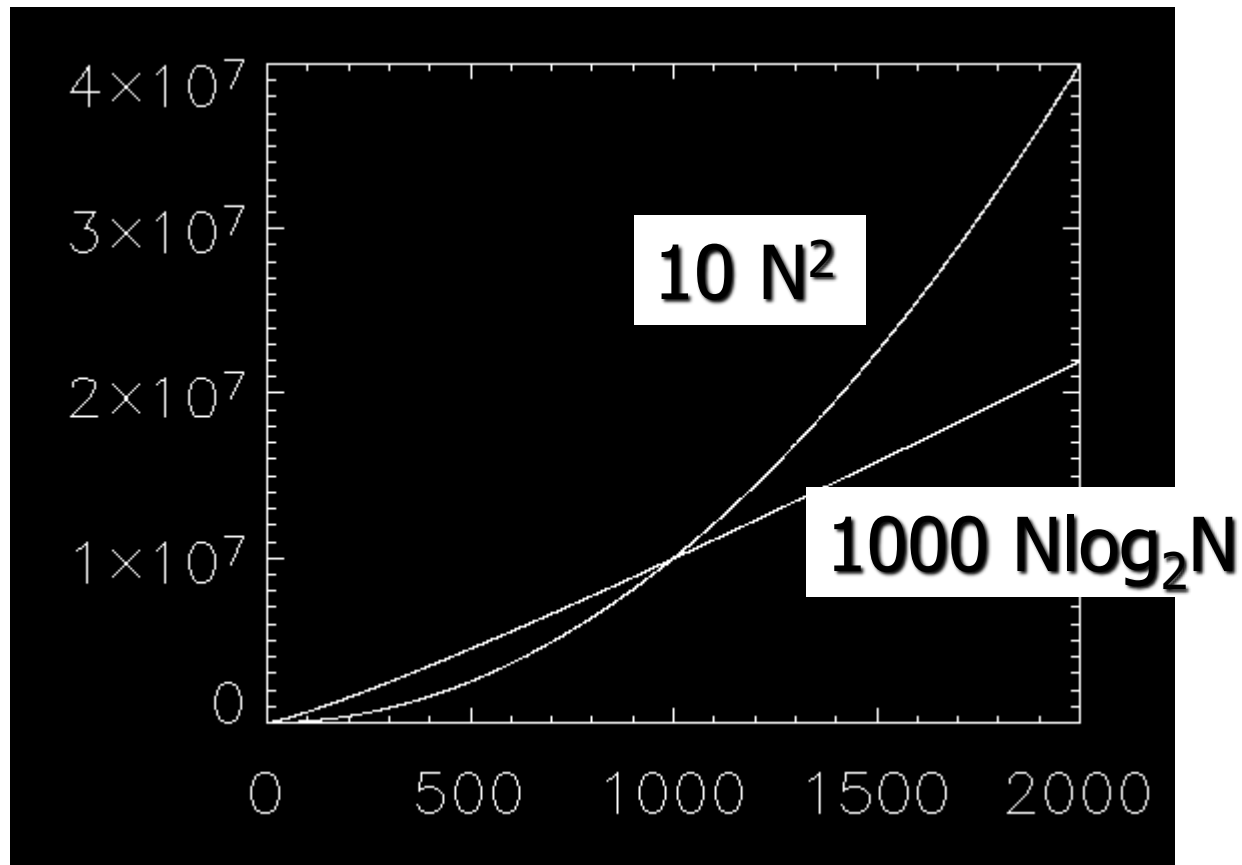
Worst Case Scenarios

- An algorithm can be very good on average but beware worst case scenarios
- Even if your typically $O(N)$ algorithm only hits an $O(N^3)$ configuration one in a hundred times, that is bad news for large N

Fastest Algorithm

- It is important to choose an algorithm appropriate to the size of problem you want to run
- Order alone doesn't tell the whole story – sometimes the constant is important
- E.g. $10 N^2$ is better than $1000 N \log_2 N$ if N is less than 1000

$O(N^2)$ vs. $O(N\log_2 N)$



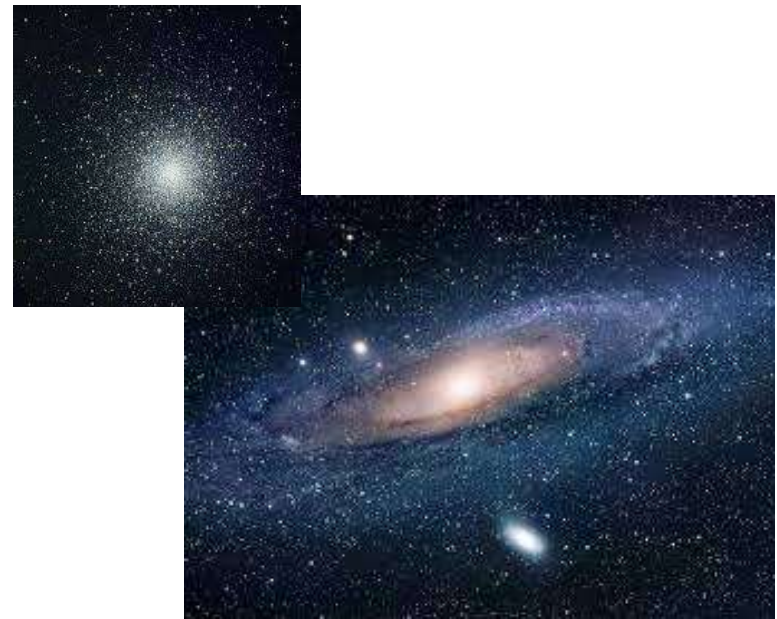
Cross-over
point is
 $N=1000$

Order – Why do we care?

- Scientific computing applications are either limited by memory or computer time
- A more efficient (in memory) or faster algorithm can revolutionize a field because it enables much larger, more realistic simulations

Example: Stars under gravity

- $O(N^2)$ Limited to 100,000 or fewer: a star cluster
- $O(N \log N)$ Run billions of stars: a realistic galaxy



Example: Gravity Solving

- Solving for Gravity in the solar system (<9 planets). This is best done with direct N^2 (as per the prior examples). Total about < 100 acceleration terms only
- Solving gravity in a galaxy (billions of stars 10^9) needs a better approach.
 - Using a tree-code based on sorting is $O(N \log_2 N)$
 - Can reduce it to around a trillion calculations 10^{12} instead of 10^{18}

Sorting

- Sorting is an everyday activity but very useful for managing data
- It is much easier to find something in a sorted list
- Sorting is the basis for building **trees**, very useful data structure for managing data (std::objects), searching and doing calculations

Order of Sorting Algorithms

- Sorts rely on comparison operations (and often exchanges as well)
- Often sorting methods are evaluated by counting the total number of comparisons required to sort N items

Sorting Algorithms: Overview

■ $O(N^2)$

Insertion, **Bubble**, **Shell**

■ $O(N \log_2 N)$

Heap, Merge, Quicksort

■ $O(N)$

Bucket

Insertion Sort

This is the sorting you do by hand

For example: An office assistant, given a new file, goes through the previously sorted files in order and inserts the new file at the right place

6 5 3 1 8 7 2 4

source:
Wikipedia

Insertion Sort: Order

- With N items to sort, your list grows by 1 every time an item is inserted
- On average, you have to check half the current list to find the right place
- Worst Case: $N(N+1)/2$ comparisons
- Typical Work: $N(N+1)/4$ comparisons

$O(N^2)$

Bubble Sort

- The Bubble sort uses exchanges to sort
- The entire data-set is passed through, exchanging neighbouring pairs that are in the wrong order (it ends if it did not have to swap anything)

6 5 3 1 8 7 2 4

source:
Wikipedia

Bubble Sort: Order

- It is typically the same order as an insertion sort but tends to have a larger constant
- It is most inefficient for random data $O(N^2)$
- BUT: For ordered data a single pass is sufficient: $O(N)$

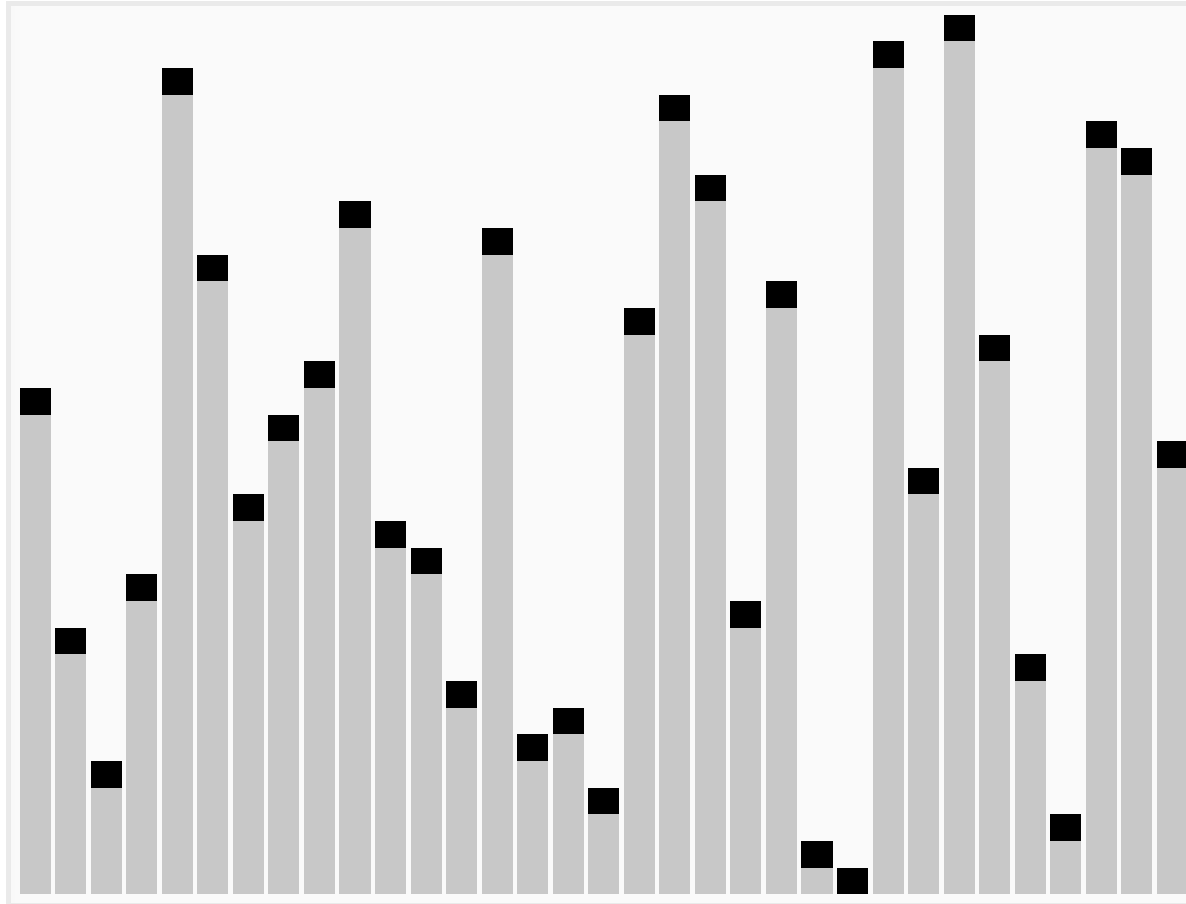
Quicksort (C.A.R. Hoare, 1962)

- Quicksort uses a divide and conquer strategy
- The unsorted data can be partitioned into two lists where all elements in list one are less than and all elements in list two are greater than a value
- This requires one loop: N compares

Quicksort: Recursion

- Now you have two partitioned lists
- You can repeat the operation on the two sublists (two subsets with roughly $\sim N/2$ elements each)
- Doing the two sublists also requires $N/2 * 2 = N$ compares total
- Assuming fairly even partitioning you only have to do the partition $\log_2 N$ times to end up with a completely sorted list

Quicksort in Action



source:
Wikipedia

Quicksort: Order

- The fastest sort for totally random data:
 $O(N \log_2 N)$ with the smallest constant
- In the worst case the partition puts only one element in one list and $N-1$ in the other: Worst case $O(N^2)$
- The Worst case is for ordered data but it is easy to tweak the algorithm not to hit this case

Heapsort and Mergesort

- Heapsort and Mergesort are $O(N \log_2 N)$ algorithms in the worst case as well
- Slightly slower than Quicksort on average
- Mergesort is suited to Parallel
- Heapsort is suited to dynamic data structures like queues

Limits to sorting

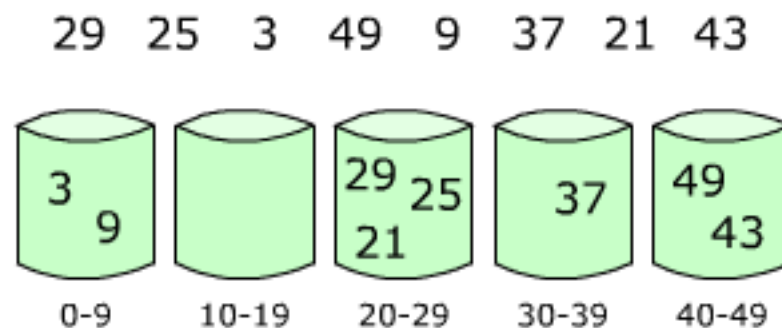
Theoretically, the best you can do via comparisons is $O(N \log_2 N)$

Can you do better?

YES – but not with comparisons

Bucket Sort

- What if the data is fairly evenly distributed?
- For example: Incoming mail to a office building can be quickly sorted by first letter of the surname. This is $O(N)$
- The trick is getting a complete sorted list after the initial pidgeon-holing



source:
Wikipedia

Bucket: Algorithm

- Consider items labeled 1 to 100 in random order
- If I have 100 bins with numbered 1 to 100, I can insert everything into the right bin and I am done
- *Pseudo Code*
 - for each item(i), $i=1,n$:
 - put item(i) into bin number item(i).label

Bucket: A More General Case

- 100 Random numbers in the range 1-100
- 100 bins
- Typically there will be 1 per bin on average, quite often 0 and sometimes more (Poisson statistics)
- Looking at empty bins takes work
- Bins with more than one item need further sorting (More Bucket sorts?)
- Still $O(N)$ – just a bigger constant

Bucket sort: Limitations

- It works best for evenly distributed data
- Highly correlated data can be worse than $O(N \log_2 N)$ sorting – there are no guarantees for arbitrary data
- Comparison based sorting can guarantee a $O(N \log_2 N)$ worst case

Why is sorting relevant to Scientific Computing?

Many physical interactions are short range or strong when close
e.g. protein molecules, collisions, gravity

In general it is only necessary to interact with a subset of the other objects for accurate forces

- Naively testing all possible interactions is $O(N^2)$
- Pre-sorting the objects in space [$O(N \log N)$] means you only do relevant interactions (can be $O(N)$ if finite range like molecular dynamics or $O(N \log N)$ if long range like gravity or electrodynamics)
- Forces are needed for the RHS of your ODE (for dynamics)
- Note: Integrating the $6N$ ODEs explicitly is $O(N)$ (not an important cost)

Objects: Hidden Costs

- C++, Python and so on offer complicated objects that have hidden costs over simple arrays
- e.g. `std::vector` and python list objects
- Even simple actions, like inserting a value into the list have costs: Could be $O(N)$ or $O(\log N)$ – different objects have different properties
- Building a list in a dumb way could be $O(N^2)$!
- For big data sets (millions) it can matter a lot. For small problems its not worth worrying about.

Order Cost: built-in functions

- C/C++, Python have standard functions for sorting and other actions
- In general they always use clever (near optimal) algorithms when available
- C++ `std::sort` and C `qsort` both use an $O(N \log_2 N)$ approach
- <https://en.cppreference.com/w/cpp/algorithm/sort#Complexity>