Order Notation in Practice

Roger Orr

OR/2 Limited

What does complexity measurement **mean**?

– ACCU 2014 –

What *is* Order Notation?

- This notation is a way of describing how the **number of operations** performed by an algorithm varies by the **size of the problem** as the size increases
- You've probably heard of order notation before – if you have studied computer science then the next section is likely to be revision

Why do we care?

- Almost no-one* is actually interested in the **complexity** of an algorithm
- What we normally care about is the **performance** of a function
  - The complexity measure of an algorithm will affect the performance of a function implementing it, but it is by no means the *only* factor

(*Present audience possibly excepted)

Ways to measure performance

- There are a number of different ways to measure the performance of a function
- Typical measures include:
  - Wall clock time
  - CPU clock cycles
  - Memory use
  - I/O (disk, network, etc)
  - Power consumption
  - Number of <> brackets used

Complexity measurement

- Complexity measurement is (normally) used to approximate the number of **operations** performed
- This is then used as a **proxy** for CPU clock cycles
- It ignores 'details' such as memory access costs that have become increasingly important over time
- It often is a measure of **one** operation

Introduction to Order Notation

- A classification of algorithms by how they respond to changes in size.
- Uses a big O (also called Landau's symbol, after the number theoretician Edmund Landau who invented the notation)
- We write $f(x) = O(g(x))$ to mean

  There exists a constant C and a value N

  such that $|f(x)| < C|g(x)| \ \forall \ x > N$

Example of Order Notation

- If $f(x) = 2x^2 + 3x + 4$
- Then $f(x) = O(x^2)$
- If $h(x) = x^2 + 345678x + 456789$
- Then $h(x) = O(x^2)$
- Note that, in these two cases, the values of C and N are likely to be different:
  - For f we can use (3, 4)

    For g we can use (2, 345679) or (4000, 87)

Example of Order Notation

- Note that f and h are **both** $O(x^2)$ although they're different functions.
- For the purposes of **order** classification, it doesn't matter what the multiplier C is nor how big the value N is.
- Note too that formally O is a “<=” relationship. So $j(x) = 16$ is also $O(x^2)$
- If $f(x) = O(g(x))$ and $g(x) = O(f(x))$ then we can write $f(x) = \boldsymbol{\theta}(g(x))$

Some common orders

- Here a some common orders, with the slower growing functions first:
  - O(1) – constant
  - O(log(x)) – logarithmic
  - O(x) – linear
  - $O(x^2)$ – quadratic
  - $O(x^n)$ – polynomial
  - $O(e^x)$ - exponential

Order arithmetic

- When two functions are combined the order of the resulting function can (usually) be inferred
- When adding functions, you simply take the biggest order
  - eg. $O(1) + O(n) = O(n)$
- When multiplying functions, you multiply the orders
  - eg. $O(n) * O(n) = O(n^2)$

Order arithmetic for programs

- For a function making a sequence of function calls the order of the function is the same as the highest order of the called functions

```
void f(int n) {
  g(n); // O(n.log(n))
  h(n); // O(n)
}
```

- In this example f() = O(n.log(n))

Order arithmetic for programs

- For a function using a loop the order is the product of the order of the loop count and the loop body

```
void f(int n) {
  int count = g(n); // count is O(log(n))
  for (int i = 0; i != count; ++i) {
    h(n); // O(n)
}
```

- In this example too f() = O(n.log(n))

Order for standard algorithms

- Many standard algorithms have a well-understood order. One of the best known non-trivial examples is probably **quicksort** which “everyone knows” is O(n.log(n)).

Order for standard algorithms

- Many standard algorithms have a well-understood order. One of the best known non-trivial examples is probably **quicksort** which “everyone knows” is O(n.log(n)).
- Except when it **isn't**, of course!
  - On **average** it is O(n.log(n))
  - The **worst** case is $O(n^2)$
- Also, this is the **computational** cost, not the **memory** cost

Order for standard algorithms

- The C++ standard mandates the complexity of many algorithms.
- For example, `std::sort`:

  “Complexity: O(N log(N)) comparisons.”
- and `std::stable_sort`:

  “Complexity: It does at most $N \log^2(N)$ comparisons; if enough extra memory is available, it is N log(N).”
- and `std::list::sort`:

  “Complexity: Approximately N log(N) comparisons”

Order for standard operations

- The C++ standard also mandates the complexity of many operations.
- For example, `container::size`:

  "Complexity: constant."
- and `std::list::push_back`:

  "Complexity: Insertion of a single element into a list takes constant time and exactly one call to a constructor of T."

Order for standard algorithms

- .Net lists complexity for some algorithms.
- For example, `List<T>.Sort`:

  “On average, this method is an O(n log n) operation, where n is Count; in the worst case it is an O(n ^ 2) operation.”
- Java does the same
- For example, `Arrays.sort`:

  “This implementation is a stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered...”

Order for standard operations

- However, neither Java not .Net seem to provide much detail for the cost of *other* operations with containers
- This makes it harder to reason about the performance impact of the choice of container and the methods used.

Let's try some experiments

- So that's the theory; what happens when we try some of these out in an actual program on real hardware?
  - YMMV (different clock speeds, amount of memory, speed of memory access and cache sizes)

strlen()

- Should be simple enough: O(n) where n is the number of bytes in the string.

```
int strlen(char *s) /* source: K&R */
{
  int n;

  for(n = 0; *s != '\0'; s++)
  {
    n++;
  }
  return n;
}
```

- Anyone looked inside strlen recently?

strlen() – more than you wanted to know

```
strlen:
    mov     rax,rcx                     ; rax -> string
    neg     rcx
    test    rax,7                       ; test if string is aligned on 64 bits
    je      main_loop
    xchg    ax,ax
str_misaligned:
    mov     dl,byte ptr [rax]           ; read 1 byte
    inc     rax
    test    dl,dl
    je      byte_7
    test    al,7
    jne     str_misaligned              ; loop until aligned
main_loop:
    mov     r8,7EFEFEFEFEFEFEFFh
    mov     r11,8101010101010100h
    mov     rdx,qword ptr [rax]         ; read 8 bytes
    mov     r9,r8
    add     rax,8
    add     r9,rdx
    not     rdx
    xor     rdx,r9
    and     rdx,r11
    je      main_loop
    mov     rdx,qword ptr [rax-8]       ; found zero byte in the loop
    test    dl,dl
    je      byte_0                      ; is it byte 0?
    test    dh,dh
    je      byte_1                      ; is it byte 1?
    shr     rdx,10h
    ...

byte_1:
    lea     rax,[rcx+rax-7]
    ret
byte_0:
    lea     rax,[rcx+rax-8]
    ret
```

strlen()

- Naively we compare time for:

```
timer.start();
strlen(data1);
timer.stop();
```

- The call appears to take no time at all ....
- Gotcha: strlen() use can be optimised away if the return value is not used.
- **It's important to check you're measuring what you think you're measuring!**

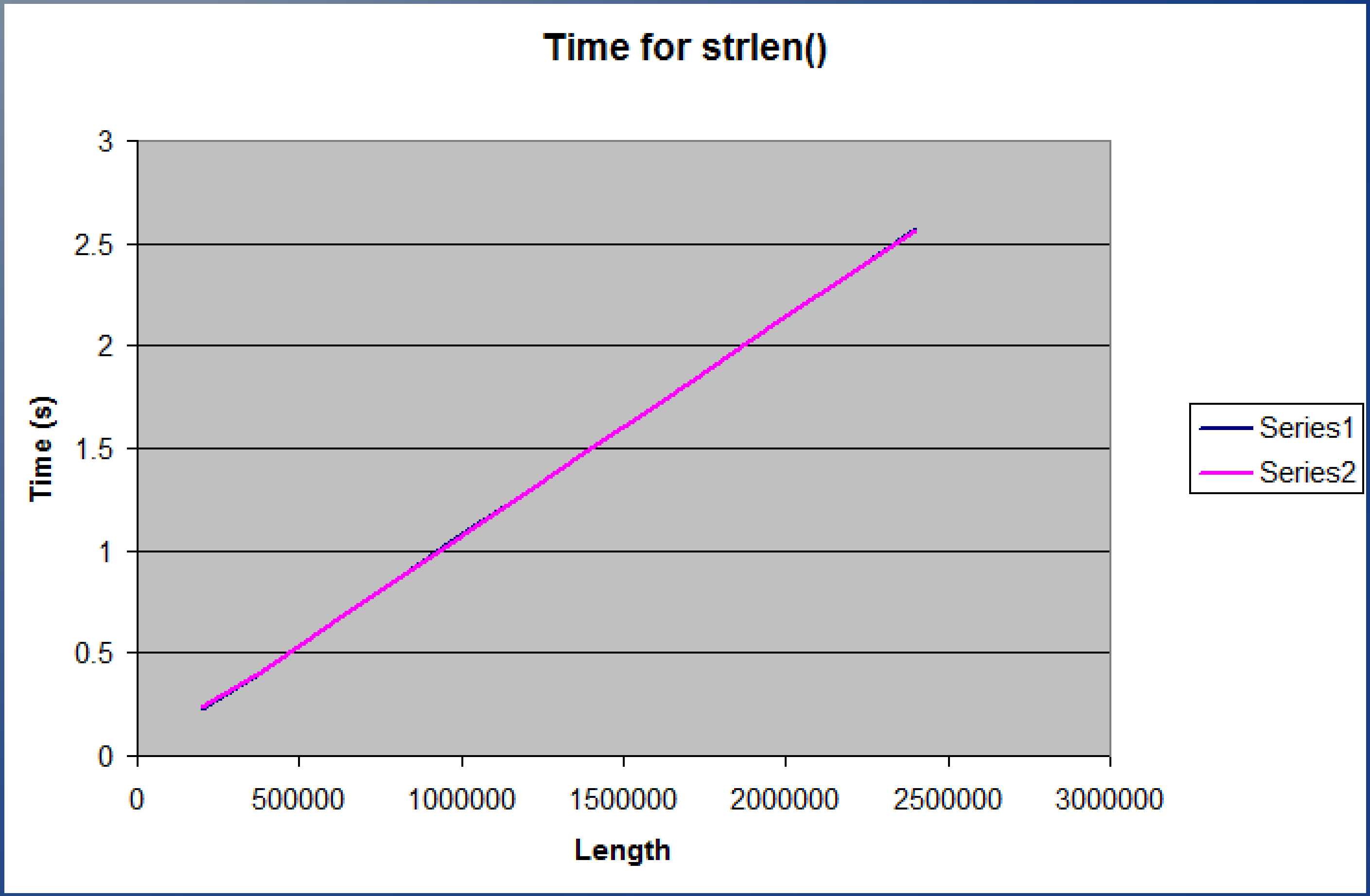
strlen()

- Set up a couple of strings:

```
char const data1[] = "1";
char const data2[] = "12345...67890...";
```

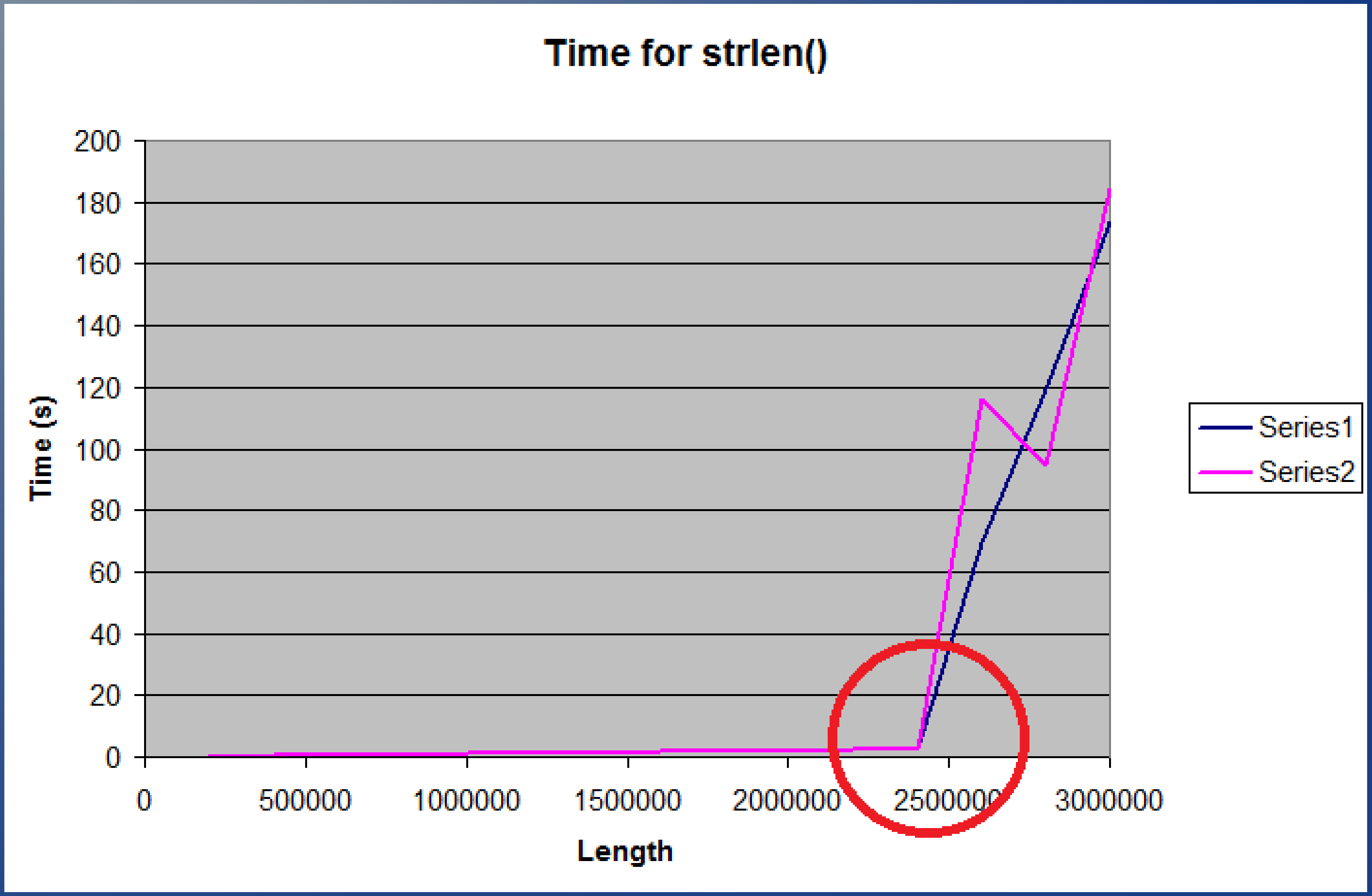
- Compare time for `v1 = strlen(data1)` against `v2 = strlen(data2)`
- Gotcha: strlen() of a **constant** string can be evaluated at compile time: O(1)
- **It's important to check you're measuring what you think you're measuring!**

strlen() - O(n)



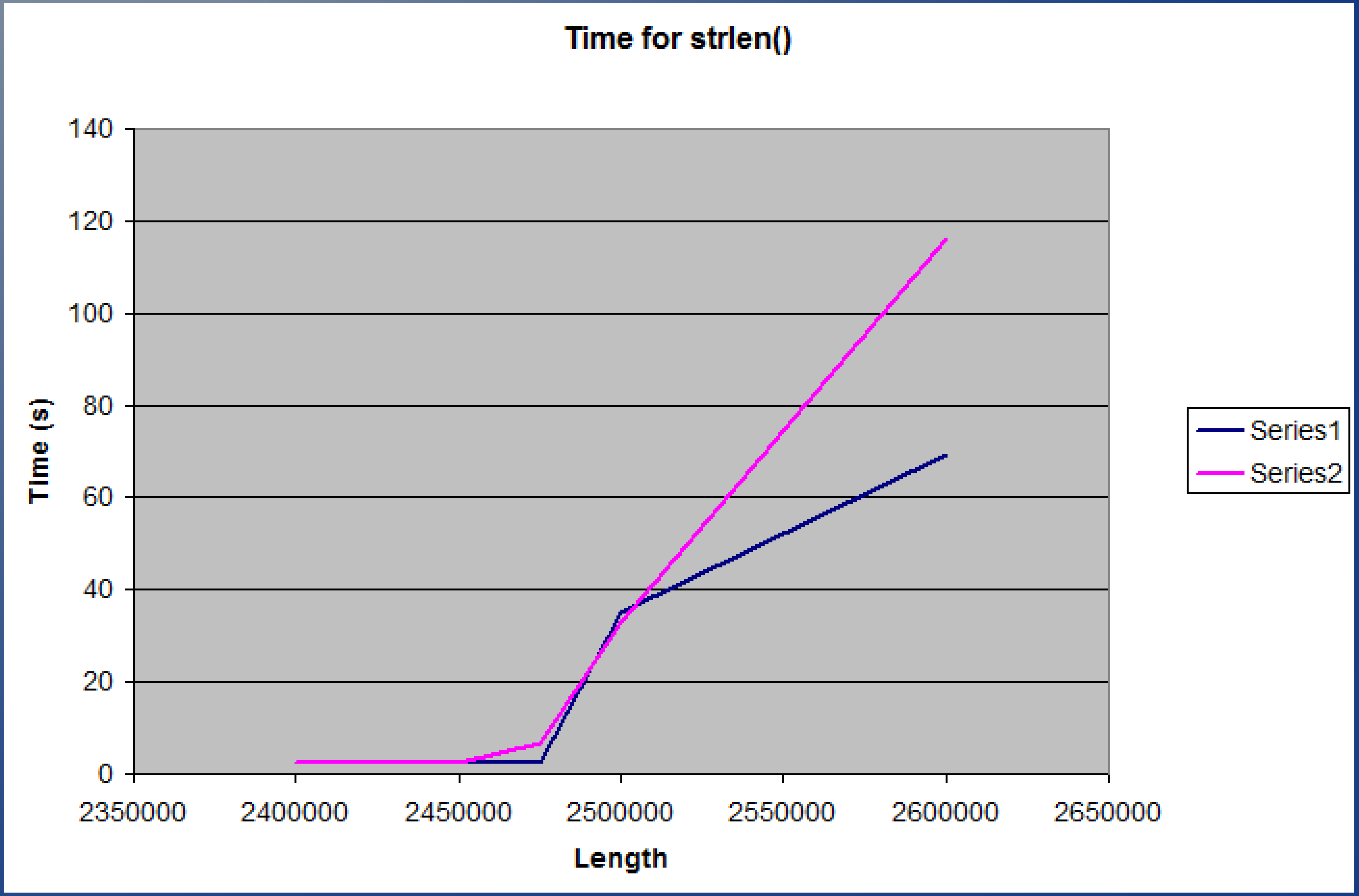
Linear and consistent

strlen() - O(dear)

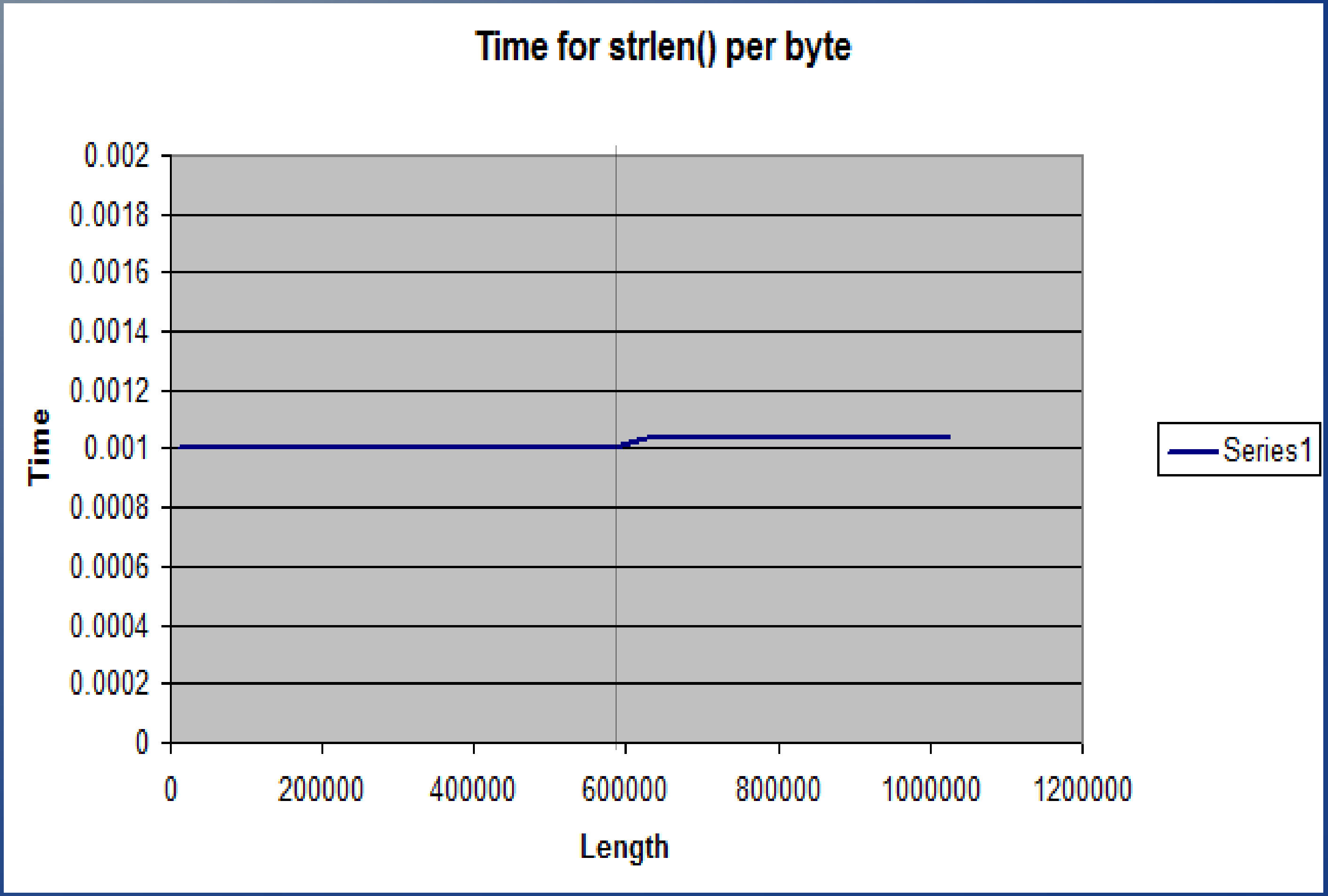


Discontinuous (and no longer as consistent)

strlen() - zoom in



strlen() - small n



This machine has 64K L1 + 512K L2 cache per core

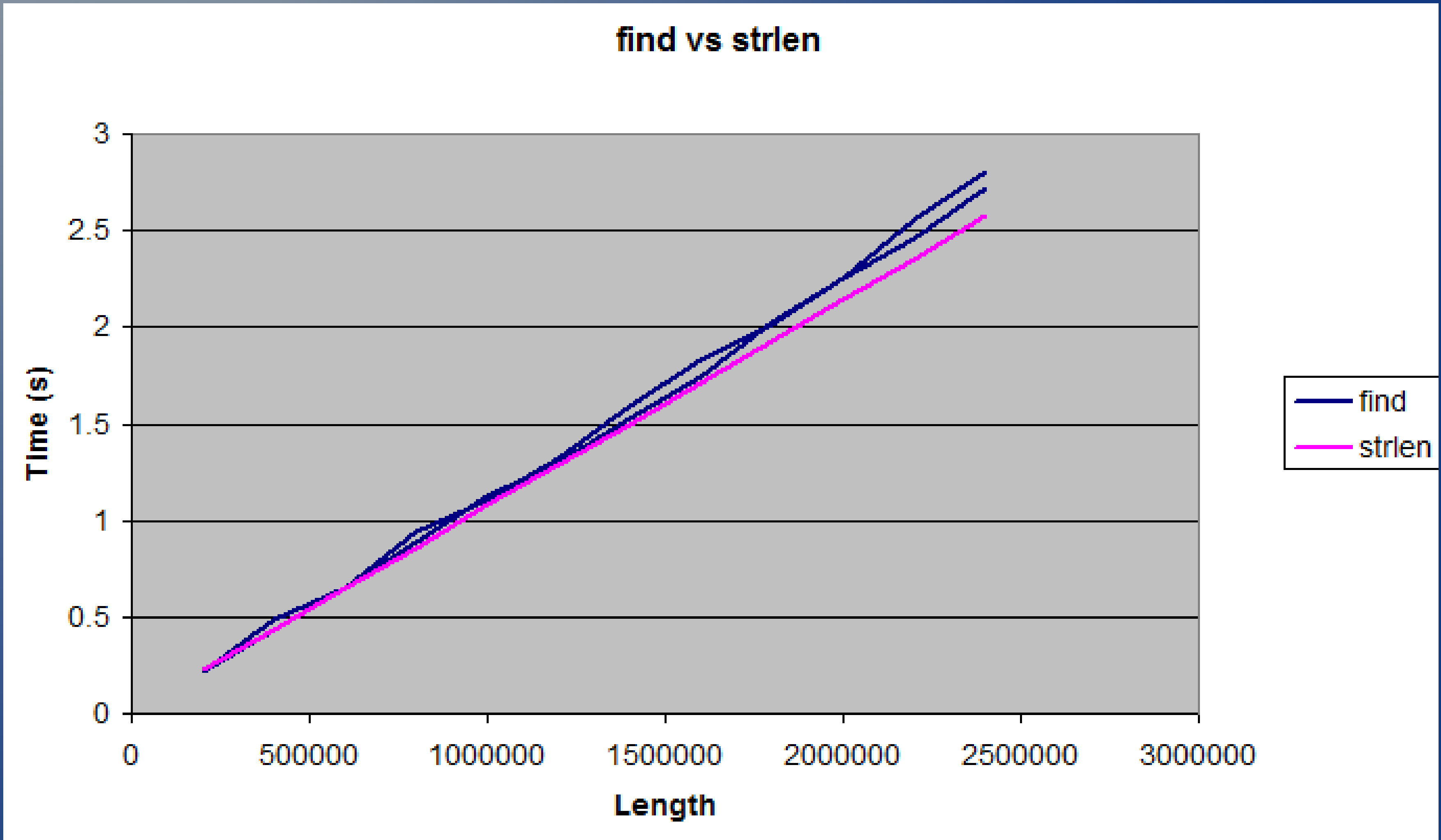
strlen()

- O(n) to a very good approximation for n between cache size and available memory
- Small discontinuity around cache size
- O(n) when swapping, but the factor 'C' is much bigger (250 – 300 times bigger here)

string::find()

- Let's swap over from using strlen() to using string::find('\0')
- Exactly the same sort of operation but with a very slightly more generic algorithm
- We expect this will behave just like strlen()

string::find()



find vs strlen
3
2.5
2
1.5
1
0.5
0
Time (s)
0
500000
1000000
1500000
2000000
2500000
3000000
Length
find
strlen

Sorting

- Let's start with a (deterministic) **bogo** sort

```
template <typename T>
void bogo_sort(T begin, T end)
{
  do
  {
    std::next_permutation(begin, end);
  } while (!std::is_sorted(begin, end));
}
```

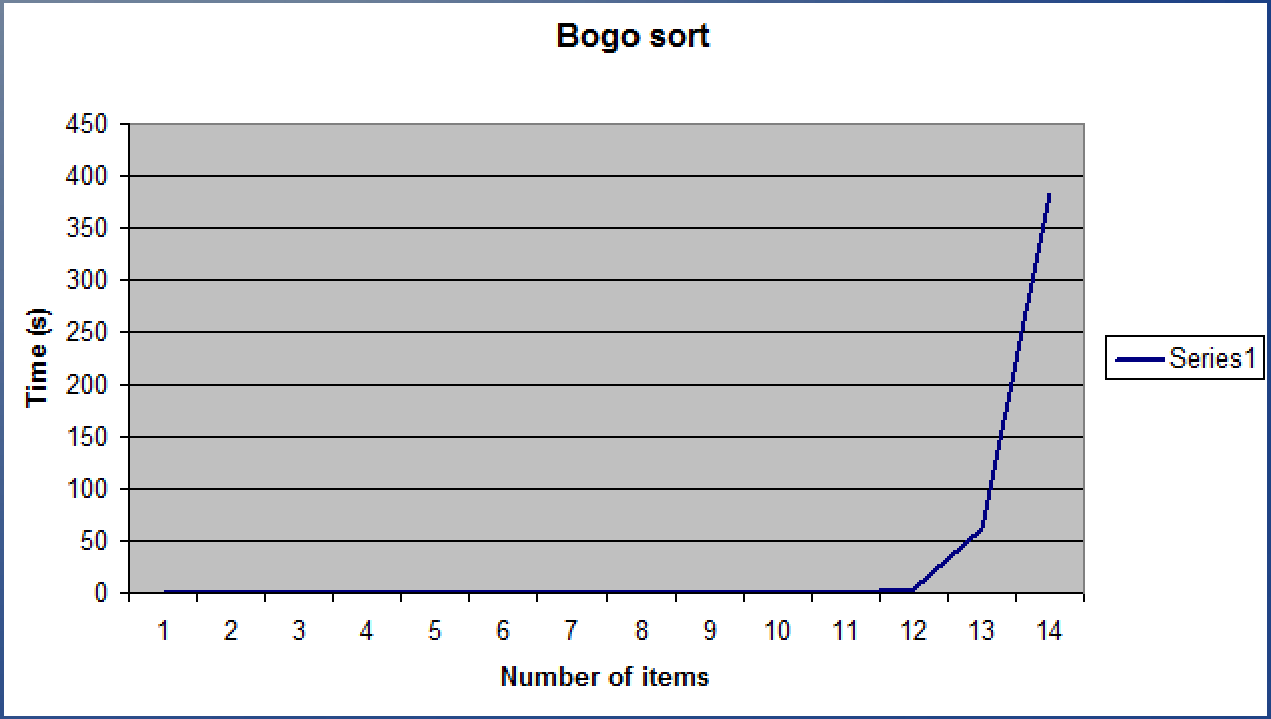
- NSFW
- $O(n \times n!)$ comparisons

Sorting

- Timings
    - 10,000 items – 1.13ms
    - 20,000 items – 2.32ms
    - 30,000 items - 3.55ms
    - 40,000 items - 4.72ms
- O(n) – but … how?
- I cheated and set the initial state carefully
- Be very careful about best and worst cases!

Sorting

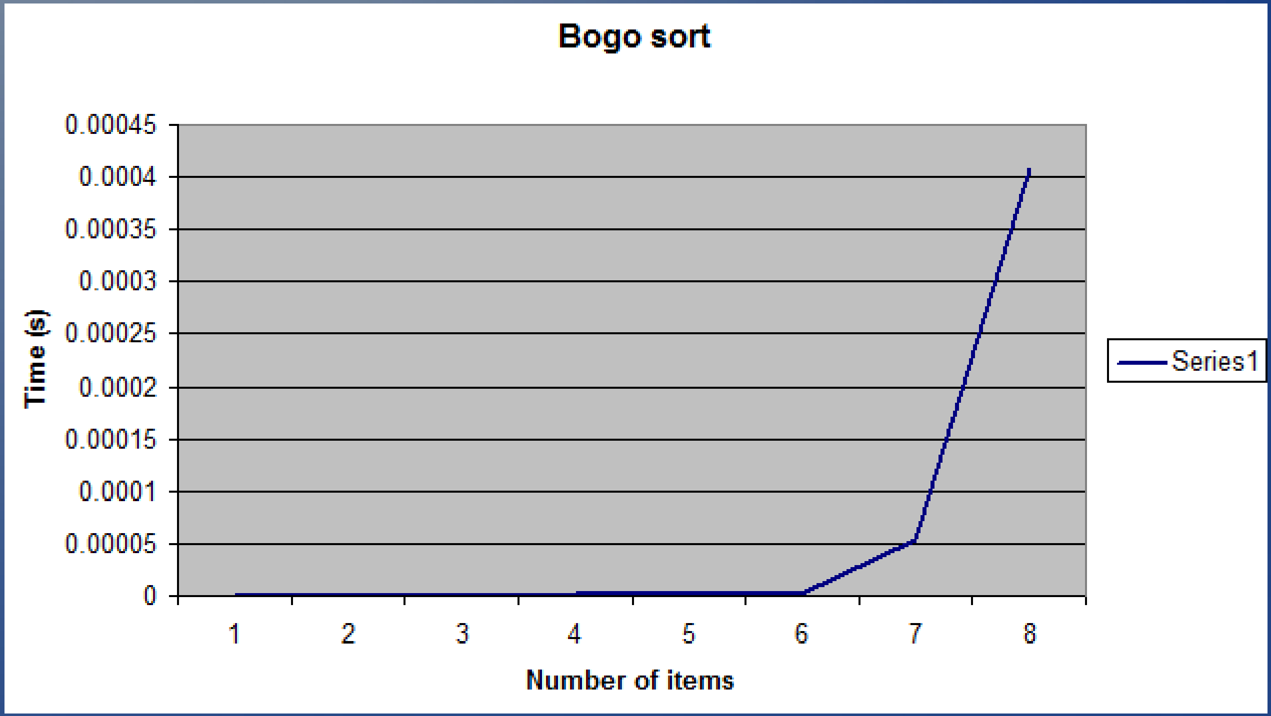
- Timings (randomised collection)



- I got bored after **14** items
- It looks like we hit a 'wall' at 13/14

Sorting

- Timings (randomised collection)



- Same graph after **8** items
- Note: the 'wall' effect depends on **scale**

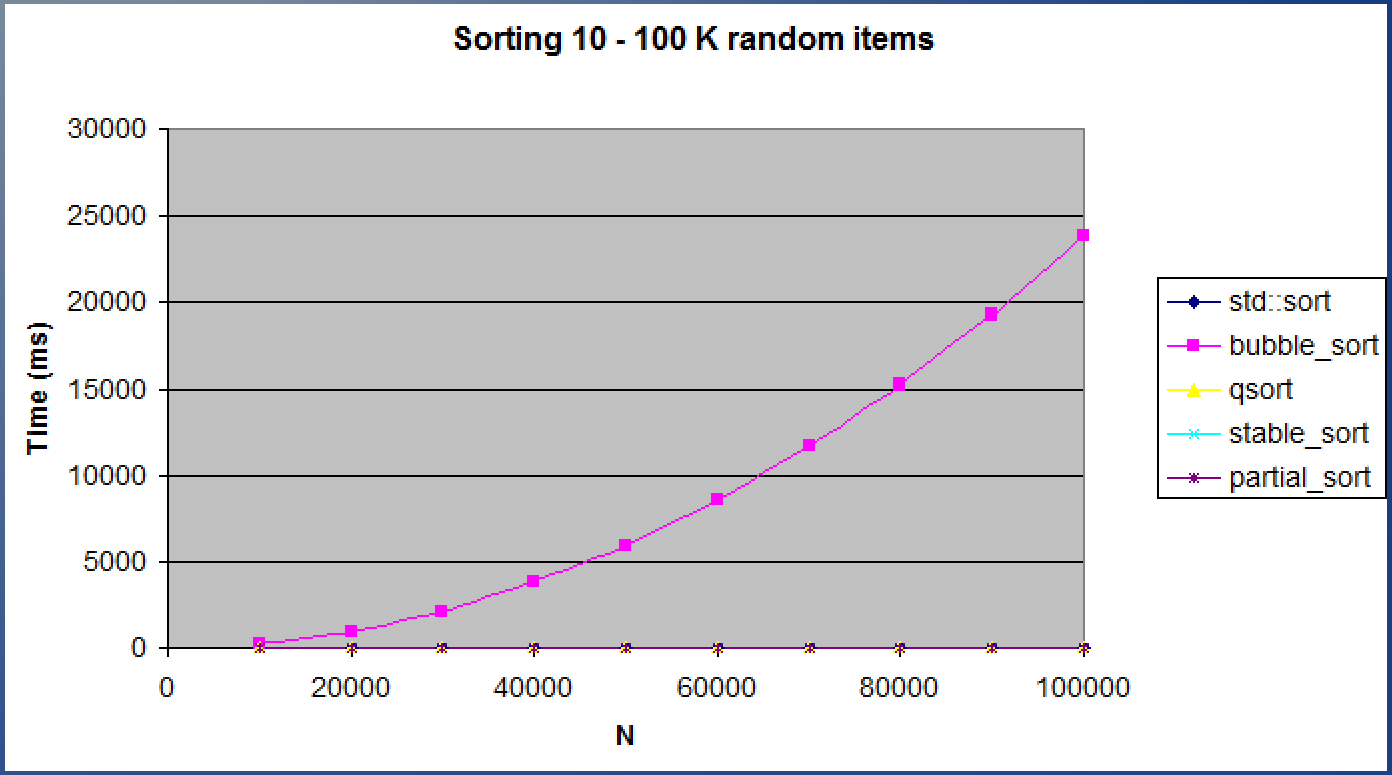
Sorting

- std::sort
  - the best known in C++
- qsort
  - the equivalent for C
- bubble_sort
  - easy to explain and demonstrate
- stable_sort
  - retain order of equivalent items
- partial_sort
  - sort 'm' items from 'n'

Sorting

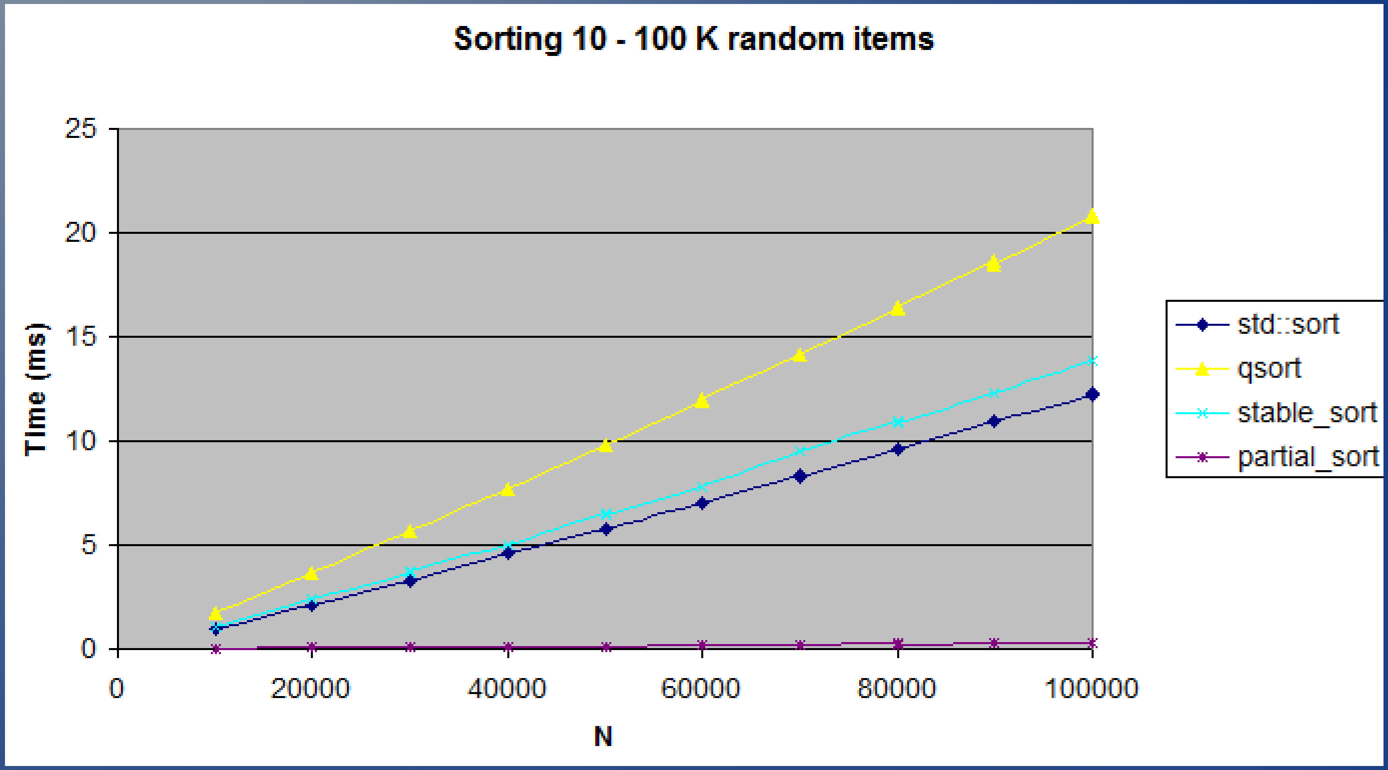
- I must mention AlgoRythmics – illustrating sort algorithms with Hungarian folk dance
- https://www.youtube.com/watch?v=ywWBy6J5gz8
- Helps to give some idea of how the algorithm **works**
- Also shows the importance of the multiplier **C** in the formula

Sorting



- “I'd like to go back in time and kill the inventor of bubblesort” - Andrei Alexandrescu

Sorting



- Granted

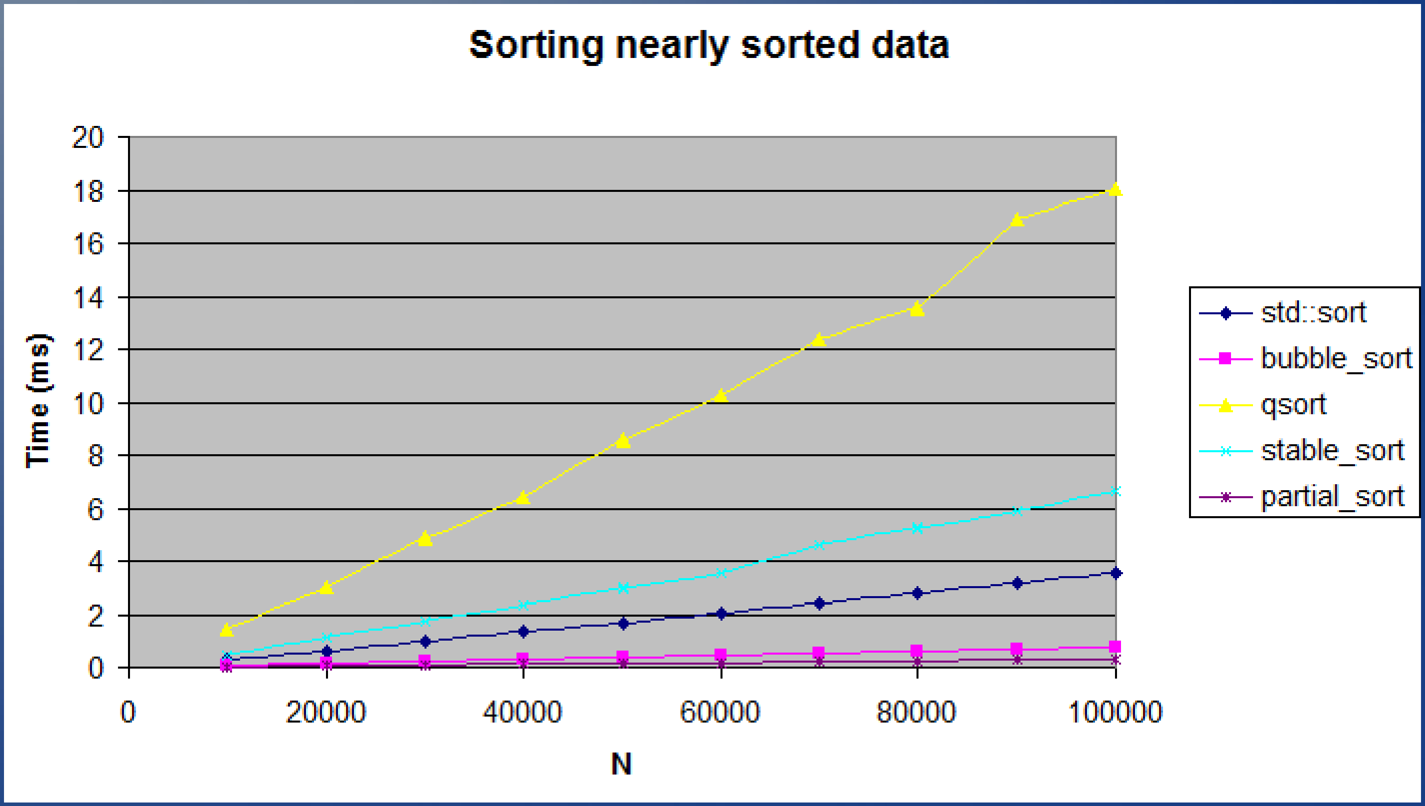
Sorting

- std::sort is faster than qsort
  - don't tell the C programmers
- You do pay (a little) for stability
- partial_sort is a “dark horse” - do you really need the **full** set sorted?

- That was with *randomised* input
- A **lot** of real data is **not** randomly sorted

Sorting

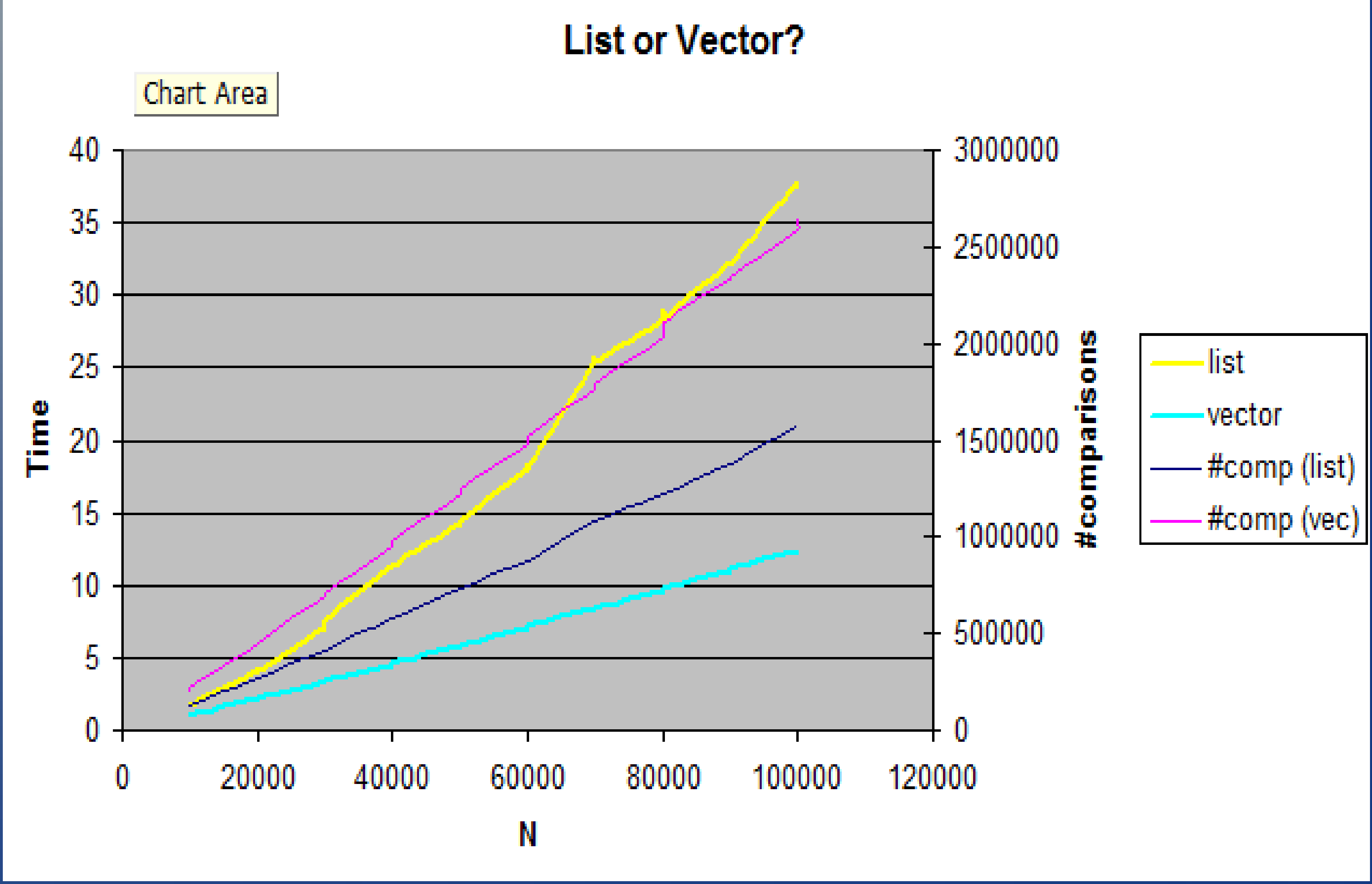
- bubble_sort's revenge



List or vector?

- The complexity of std::sort is the same as std::list::sort – so what's the difference?
- Must copy the whole object in a vector
- Can just swap the pointers in a list

List or vector?



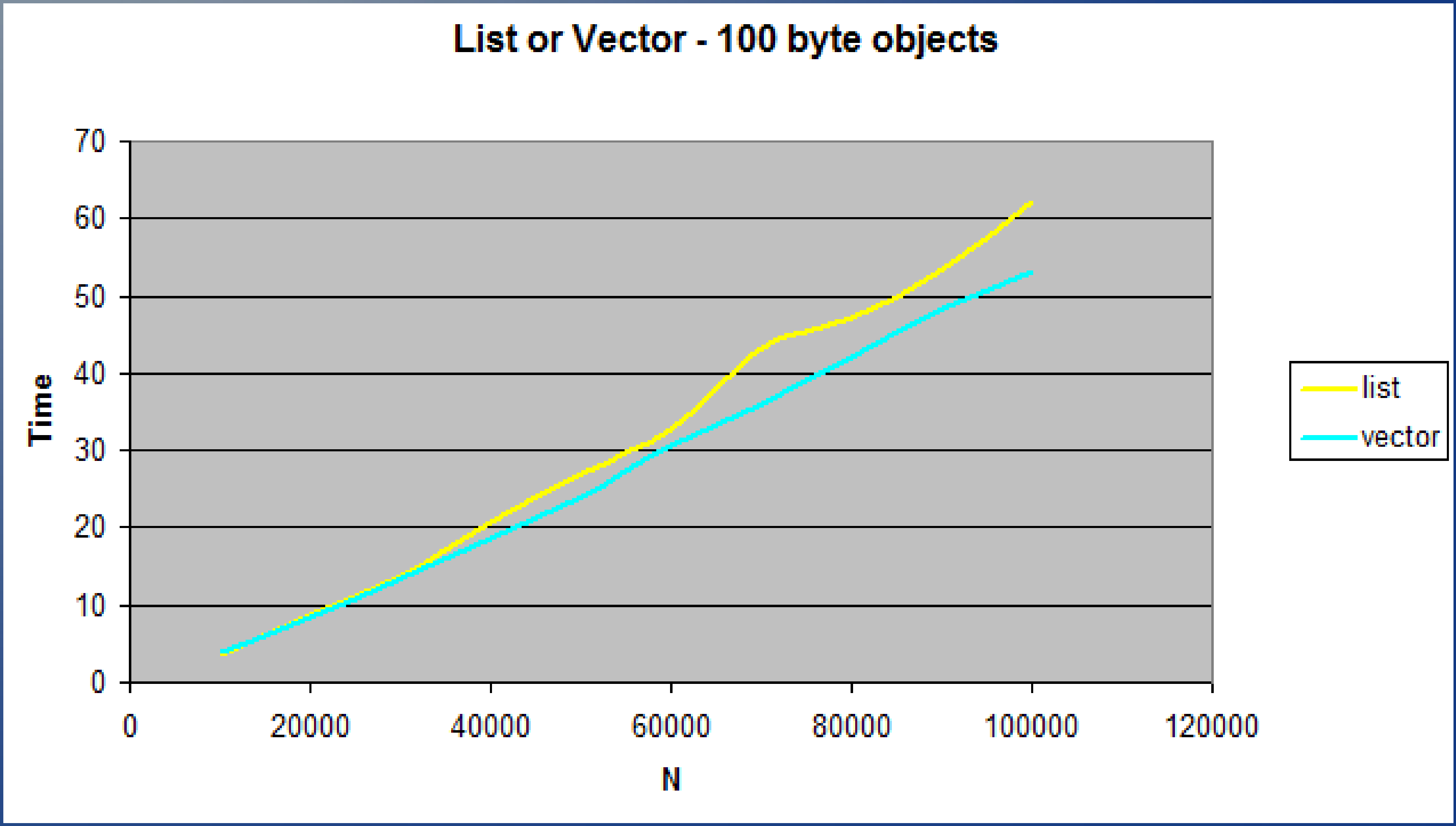
List or vector?

- So at this data size list is over twice as slow as vector to sort but uses just over half as many comparisons
- Perhaps measure sort complexity in other terms than just the number of comparisons
- However note that the items sorted in this example are quite small (wraps an int)

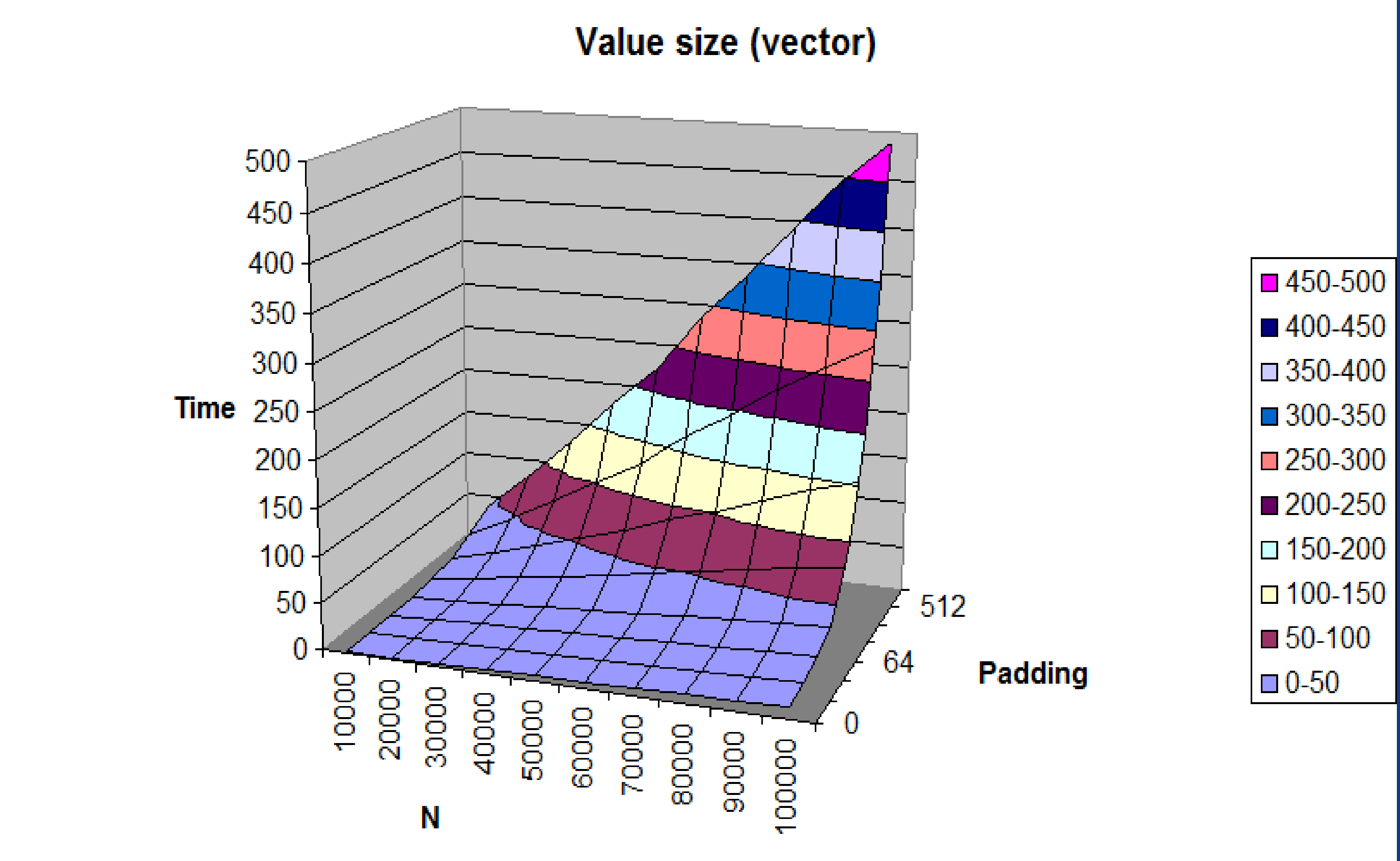
List or vector?

- The performance will depend on the size of the object being copied
- With a bigger object footprint
  - Same number of comparisons
  - Same number of pointer swaps (list)
  - More bytes copied (vector)
- Repeat the test with a bigger data structure (we won't display the # of comparisons)

List or vector?



List or vector?

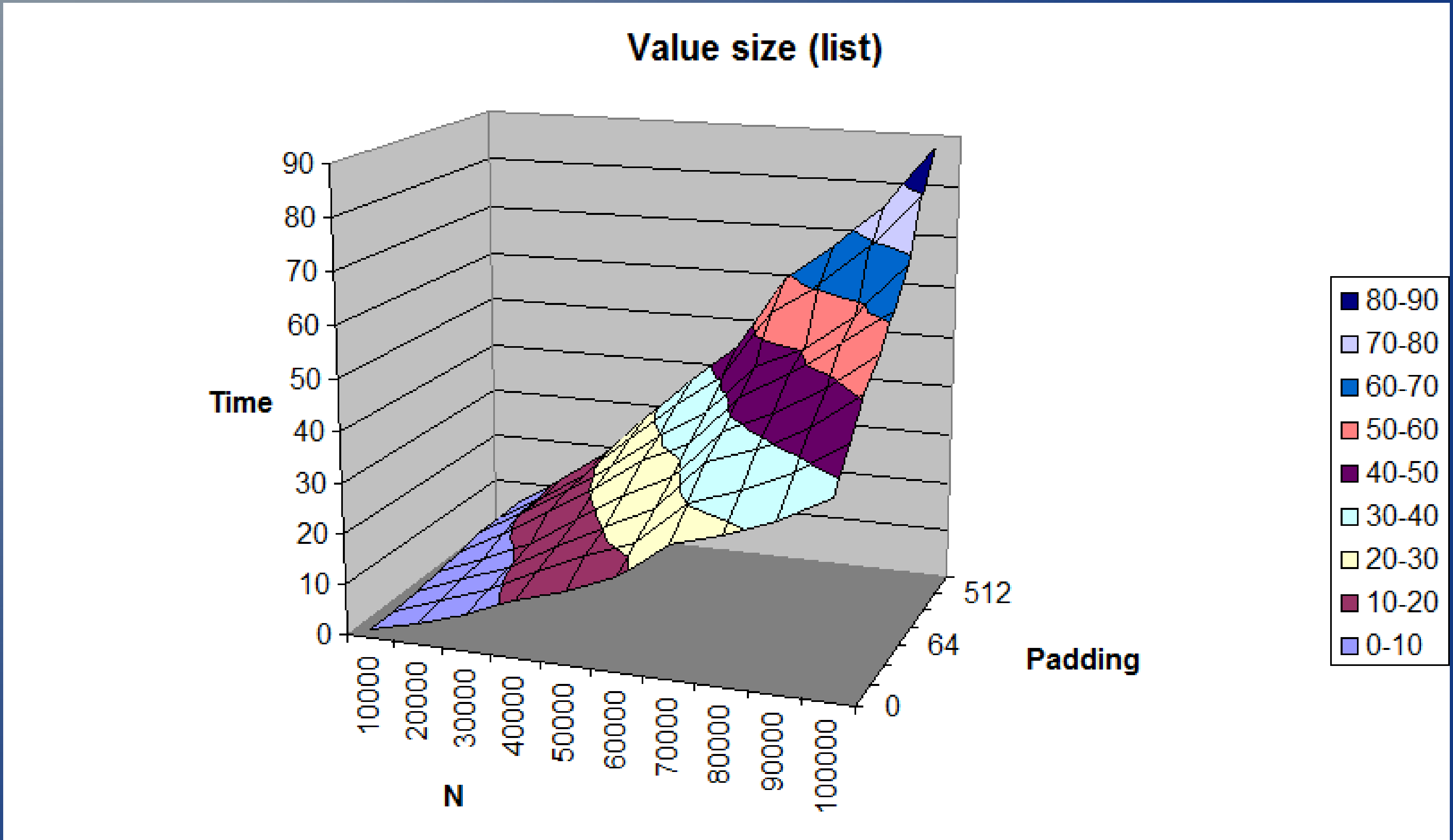


List or vector?

- This is what we expect: the performance depends very heavily on the size of the object being copied

  So, in this test on this hardware, the break-even point comes at somewhere around 100 bytes for the object footprint
- This is bigger than I was expecting
- For comparison here is the effect on sorting the **list** when we change the object footprint

List or vector?



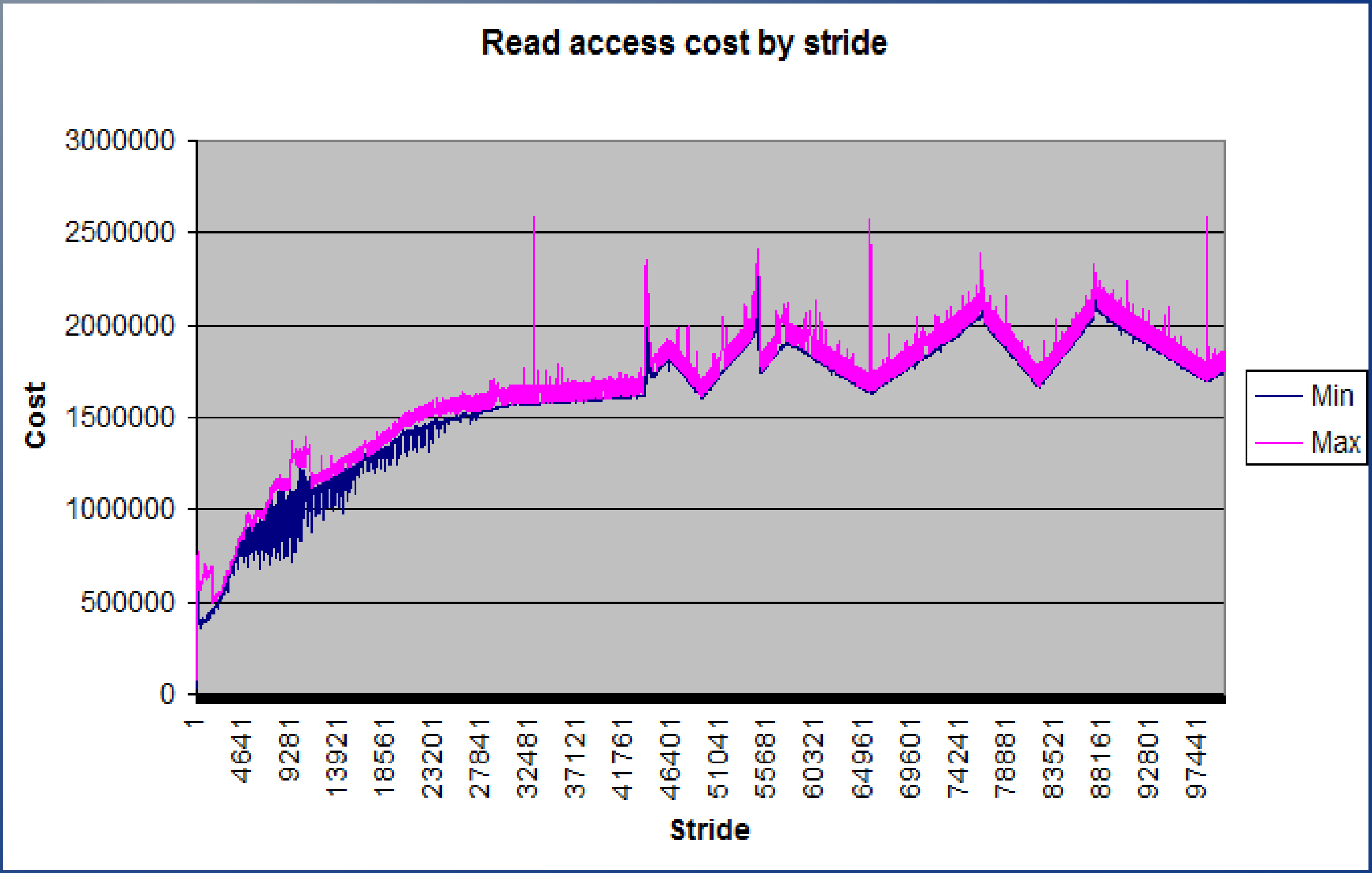
List or vector?

- This is less expected: it is about 2 – 3 times slower to sort a list of 1Kb objects than a list of **int** objects.
- The only difference is the memory access pattern: objects are further apart and so cache use is less efficient.
- But once you're further apart than a cache line (64bytes) why does *more* size still make a difference?

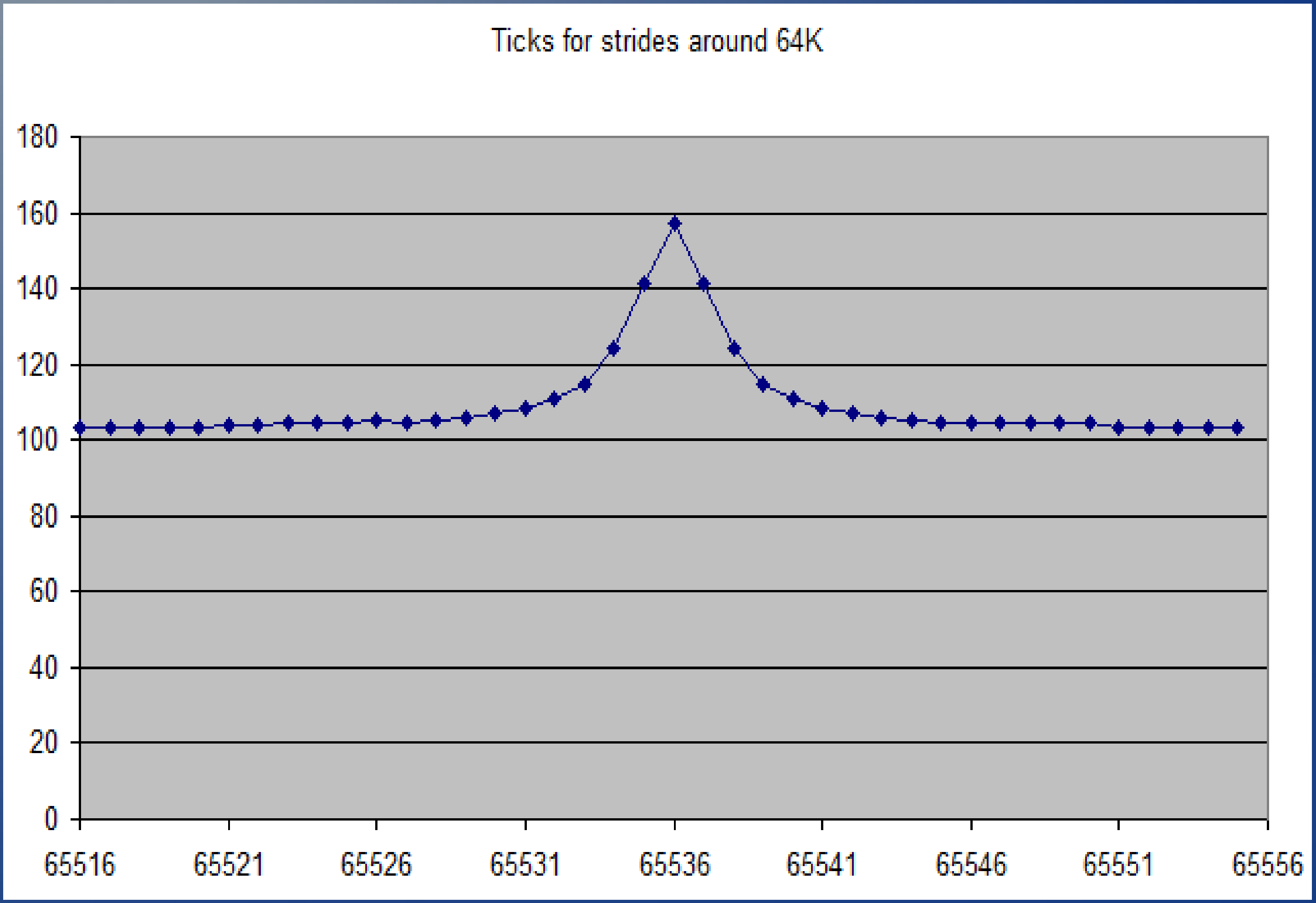
Back to basics

- Allocate a range of memory and access it sequentially with 'n' steps of size 'm'.
- There is an overall trend, of sorts, with some anomalies
- The specifics will vary depending on the hardware you're running on and will depend on both the **size** and **associativity** of the various caches

Back to basics



Back to basics



Back to basics

- While the specifics vary, the principle of **locality** is important
- If it is **multiplicative** with the algorithmic complexity it can change the complexity measure of the overall function

Cost of inserting

- Suppose we need to insert data into a collection and the performance is an issue
- What might be the effect of using:
  - std::list
  - std::vector
  - std::deque
  - std::set
  - std::multiset

Cost of inserting

- std::list “constant time insert and erase operations anywhere within the sequence”
- std::vector “linear in distance to end of vector”
- std::deque “linear in distance to nearer end”
- std::set & std::multiset “logarithmic”

- We also need the time to find the insert point

Cost of inserting

- Randomly inserting 10,000 items:
- std:list ~600ms
  - very slow – cost of **finding** the insertion point in the list
- std::vector ~37ms
  - Much faster than list even though we're copying each time we insert
- std::deque ~310ms
  - Surprisingly poor – spilling between buckets
- std::set ~2.6ms   our winner!

Cost of inserting

- May be worth using a helper collection if the target collection is costly to create
  - Use std::set as the helper and construct std:list on completion ~4ms
  - Use a std::map of iterators into the list so list built in right order ~4.8ms
- The helper collection will increase the overall memory use of the program

Cost of **sorted** inserting

- Inserting 10,000 **sorted** items:
- std:list ~0.88ms
  - Fast insertion (at **known** insert point)
- std::vector ~0.85ms (end) / 60ms (start)
  - **Much** faster when appending
- std::deque ~3ms
  - Roughly equal cost at either end; a bit slower than a vector
- std::set ~2ms (between vector and deque)

Cost of inserting

- What about **order** notation effects?
- If we use 10x as many items:
  - std:list ~600s (1000x)
  - std::vector ~3.7s (100x)
  - std::deque ~33s (100x)
  - std::set ~66ms (33x)
- The **find** cost for list dwarfs the **insert** cost, which is often a hidden complexity

Cost of inserting

- Can we beat std::set ?
- Try naïve std::**unordered**_set() - very slightly slower at 10K (~2.8ms vs ~2.6ms) but better at 100K (~46ms vs ~66ms)
- However, in this **particular** case we have additional knowledge about our value set and so can use a *trivial* hash function
- Now std::unordered_set() takes ~2.3ms (10K) and ~38ms (100K)

Conclusion

- The algorithm we choose is obviously important for the overall performance of the operation (measured as elapsed time)
- As data sizes increase we eventually hit the limits of the machine; the best algorithms are those that involve least swapping
- For smaller data sizes the characteristics of the cache will have some effect on the performance

Conclusion

- While complexity measure is a good tool we must bear in mind:
- What are N (the relevant size) and C (the multiplier)?
- Have we identified the function with the dominant complexity?
- Can we re-define the problem to reduce the cost?

Making it faster

- We've seen a few examples already of making things faster.
- Compile-time evaluation of strlen() turns O(n) into O(1)
  - Can you pre-process (or cache) key values?
  - Swapping setup cost or memory use for runtime cost

Making it faster

- Don't calculate what you don't need
- We saw that, if you only need the top 'n', partial_sort is typically much faster than a full sort
- If you know something about the characteristics of the data then a more specific algorithm might perform better
  - strlen() vs find()
  - Sorting *nearly* sorted data
  - 'Trivial' hash function

Making it faster

- Pick the best algorithm to work with memory hardware
  - Prefer sequential access to memory
  - Smaller is better
  - Splitting compute-intensive data items from the rest can help – at a slight cost in the complexity of the program logic and in memory use

Some other references

- Scott Meyers at ACCU “CPU caches”:
  http://www.aristeia.com/TalkNotes/ACCU2011_CPUCaches.pdf
- Ulrich Drepper “What Every Programmer Should Know About Memory”:
  http://people.redhat.com/drepper/cpumemory.pdf
- Herb Sutter's experiments with containers:
  http://www.gotw.ca/gotw/054.htm
- and looking at memory use:
  http://www.gotw.ca/publications/mill14.htm
- Bjarne Stroustrup's vector vs list test:
  http://bulldozer00.com/2012/02/09/vectors-and-lists/ (esp slides 43-47)
- Baptiste Wicht's list vs vector benchmarks:
  http://www.baptiste-wicht.com/2012/12/cpp-benchmark-vector-list-deque/