Speed Is Found In The Minds Of People

Prepared for CppCon 2019

Andrei Alexandrescu, Ph.D.
`andrei@erdani.com`

September 17, 2019

What Algorithm Is This?

```
  test    esi, esi
  jle     .L5
  lea     eax, [rsi-1]
  cmp     eax, 3
  jbe     .L5
  mov     edx, esi
  pxor    xmm0, xmm0
  shr     edx, 2
  sal     rdx, 4
  add     rdx, rdi
.L5:
  movdqu  xmm2, XMMWORD PTR [rax]
  add     rax, 16
  cmp     rax, rdx
  jne     .L5
  movdqa  xmm1, xmm0
  jmp     .L5
```

Official Announcement

Herb Sutter Book Signing After This Talk

*Meaning of **shedload** in English*

shedload

noun [C] UK informal

UK /ˈʃed.ləʊd/ **US** /ˈʃed.loʊd/

a large amount:

- *The film has recently won **a** shedload **of** awards.*
- *Shedloads **of** cash are needed to improve the failing health service.*

+ Thesaurus: synonyms and related words

What's the Deal with Sorting?

Sorting

- Arguably the most researched CS problem
- Many algorithms include sorting as a step
- Staple of programming classes

- Every programmer must implement sort

Why is Quicksort Popular?

- Fundamentally easy to code and analyze
  - Can ‘spend’ on corner cases, optimization
- Fast on average
  - Just like computers!
- Little work on (almost) sorted data
  - “Idempotence should be cheap”
- Cache friendly on large inputs
- Well balanced across compares and swaps

Naïve Implementation

```
template <typename It>
void qsort(It first, It last) {
  while (last - first > 1) {
    auto cut = partition_pivot(first, last);
    qsort(cut, last);
    last = cut;
  }
}
```

Less Naïve Implementation

```
template <typename It>
void qsort(It first, It last) {
  while (last - first > THRESHOLD) {
    auto cut = partition_pivot(first, last);
    qsort(cut, last);
    last = cut;
  }
  small_sort(first, last);
}
```

- `THRESHOLD` is 32 on VS
- 16 on gnu
- clang: 30 for trivially constructible/assignable types, 6 otherwise
- Glossing over pathology (`introsort`—more later)

Challenge

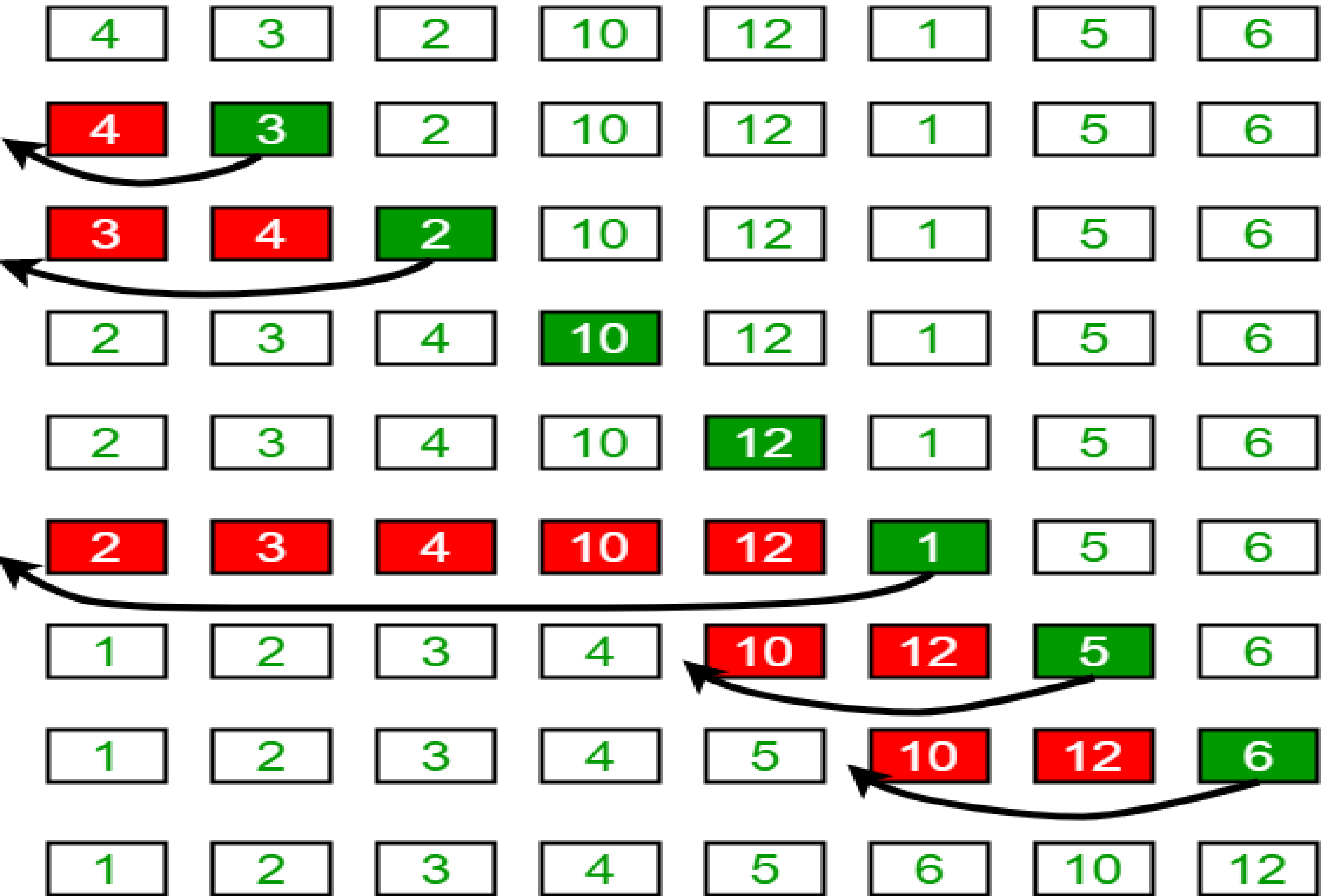
- Usual `small_sort`: optimistic linear insertion sort
- Large inputs: “solved” (with caveats)
- Small inputs: “solved”
  - Optimal solutions known for ≤ 15 elements
  - However, code size remains an issue
- Medium inputs: *difficult*

- Challenge: essentially increase `THRESHOLD`

Investigating `small_sort`

- `std::sort`: optimistic insertion sort
- Starting from the left:
  - Linear search (going backwards) position of current in sorted subarray on the left
  - Insert it there
- Worst: $C(n) = S(n) = \frac{n(n-1)}{2}$
- Average: $C_a(n) = S_a(n) = \frac{n(n-1)}{4}$
- $C_a(32) = S_a(32) = 248$

Insertion Sort Execution Example



Why Not *Binary* Insertion Sort?

- Same number of moves
- $C(n) = \sum_{i=1}^{n-1} \lceil \log_2 i \rceil$
- $C(32) = 155$ (compare to 248)
- Less work for same result $\Rightarrow$ win!

Looking Good

- Test on 1M random `double`s, threshold 32
- `std::sort`: 25.33M comparisons
- With binary insertion: 22.14M comparisons
- Cool, 15% reduction of comparisons!
- Same number of moves (13.79M)

Oopsies

- `std::sort`: 60.75 ms
- With binary insertion: 68.58 ms
- “Cool,” 13% pessimization!
- Increasing `THRESHOLD` only makes it worse

Some Like It Boring

- Linear searches are highly predictable
  - Literally one fail per search
  - Average success: $R(n) = \frac{n-4}{n}$, $R(32) = 87.5\%$
  - Branch prediction works swimmingly
- Binary searches have maximum entropy
  - Each extracts 1 bit of information
  - Average success: $R(n) = 50\%$
  - Branch prediction is powerless

Unpleasant Realization

- All research: minimize $C(n)$
- All textbooks: minimize $C(n)$
- Extracting max info per comparison is a central goal

- Reality: Informational entropy of comparisons radically affects performance

If You Thought It Ain't Weird Enough

- See "Binary Search Eliminates Branch Mispredictions"
- Concludes repeated binary search faster
- Specialized case (search only, powers of 2)

- Branchless binary search exists
- Coded it, tried it
- Slower... even than branchy binary search!

What to Do?

"I Want Someone Smart but also Boring"

- Reducing swaps may be more productive
- Idea: start from the *middle* and expand
- Swap left with right if left $>$ right
- Insert from left
- Insert from right
- ... until done

- Advantage: fewer swaps
- (Credit: "Silicon Valley" show)

Middle-Out Insertion Sort

```
template <class It>
void middle_out_sort(It first, const It last) {
  const size_t size = last - first;
  if (size <= 1) return;
  first += size / 2 - 1;
  auto right = first + 1 + (size & 1);
  for (; right < last; ++right, --first) {
    if (*first > *right) swap(*first, *right);
    unguarded_linear_insert(right);
    unguarded_linear_insert_right(first);
  }
}
```

(Aside)

- This is not terribly original
- Cottage industry of insertion sort variations
  - Two/k at a time insertion
  - Shell sort
  - Binary merge sort
  - Library sort

Hold On To Your Hat

- Test on 1M random `double`s, threshold 32
- Middle-out insertion:
  - 23.75M comps (7% better)
  - 12.15M moves (13% better)
- However, time is identical within 1%
- Changing threshold does not help

More Ideas?

Going the Other Way

- Computing systems are unfathomably complex
- Optimization is complicated and surprising
- Doing something sensible had opposite effect
- We often try clever things that don't work

- How about trying something silly then?

Showerthought

Average distance of moving elements around is too damn high, so let’s reduce that

Enter `stupid_small_sort`

```
template <typename It>
void stupid_small_sort(It begin, It end) {
  make_heap(begin, end, greater<>());
  insertion_sort(begin, end);
}
```

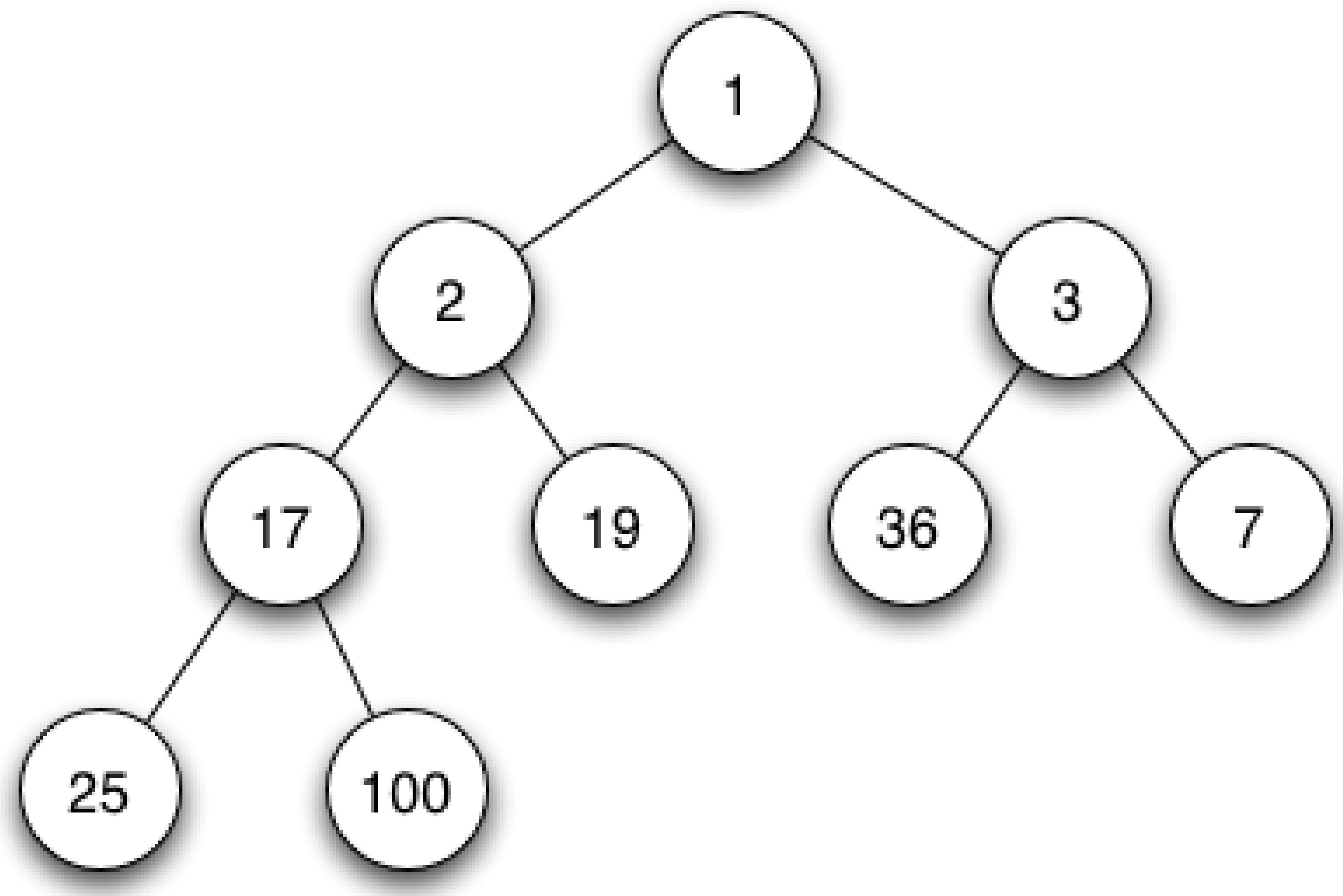
- Like Smoothsort, just stupid
- Many comparisons still predictable
- Fewer swaps

stupid_small_sort, improved

```
template <typename It>
void stupid_small_sort(It begin, It end) {
  make_heap(begin, end, greater<>());
  // Bug in slides:
  // unguarded_insertion_sort(begin + 3, end);
  unguarded_insertion_sort(begin + 2, end);
}
```

"Oh. Well, yuk. Shellsort was that same sort of idea but both simpler and probably better."

— Mathematician upon hearing this idea


1
2
3
17
19
36
7
25
100

Counts—Not Bad!

- Test on 1M random `double`s, threshold 32
- `std::sort`: 25.33M comparisons, 13.79M swaps
- Heapify+insertion sort: 23.51M comparisons, 11.56M swaps

- Improvement: 9% comps, 20% swaps

Net Optimization

- `std::sort`: 60.54 ms
- Heapify+insertion sort: 65.92 ms
- Disaster! (9% pessimization)
- Recall we improved all metrics significantly

- Researcher's view:
  - Direction is right
  - We do too many ancillary operations
  - We can increase `THRESHOLD`
  - Time for micro-optimizations

Enter Micro-optimization

Classic heapify (Rosetta Code) 1/2

- Floyd's algorithm:
  - From mid-array to zero (outer loop):
  - Insert into the subheap below (inner loop)
  - Done after inserting element at index zero

```
void to_heap(vector<int>& arr) {
  int i = (arr.size() / 2) - 1;
  for (; i >= 0; --i)
    shift_down(arr, i, arr.size());
}
```

Classic heapify (Rosetta Code) 2/2

```
void shift_down(vector<int>& heap, int i, int max) {
  while (i < max) {
    auto i_big = i;
    auto c1 = (2 * i) + 1;
    auto c2 = c1 + 1;
    if (c1 < max && heap[c1] > heap[i_big])
      i_big = c1;
    if (c2 < max && heap[c2] > heap[i_big])
      i_big = c2;
    if (i_big == i) return;
    swap(heap[i], heap[i_big]);
    i = i_big;
  }
}
```

Classic heapify

- Inner loop: 5 compare/jump decisions
- 3 add/shift
- 6 assignments (max)
- Must reduce this

GNU make_heap

```
template <typename _RandomAccessIterator, typename _Distance, typename _Tp, typename _Compare>
void __adjust_heap(_RandomAccessIterator __first, _Distance __holeIndex,
    _Distance __len, _Tp __value, _Compare __comp) {
  const _Distance __topIndex = __holeIndex;
  _Distance __secondChild = __holeIndex;
  while (__secondChild < (__len - 1) / 2) {
    __secondChild = 2 * (__secondChild + 1);
    if (__comp(__first + __secondChild, __first + (__secondChild - 1)))
      __secondChild--;
    *(__first + __holeIndex) = _GLIBCXX_MOVE(*(__first + __secondChild));
    __holeIndex = __secondChild;
  }
  if ((__len & 1) == 0 && __secondChild == (__len - 2) / 2) {
    __secondChild = 2 * (__secondChild + 1);
    *(__first + __holeIndex) = _GLIBCXX_MOVE(*(__first
              + (__secondChild - 1)));
    __holeIndex = __secondChild - 1;
  }
  __decltype(__gnu_cxx::__ops::__iter_comp_val(_GLIBCXX_MOVE(__comp)))
    __cmp(_GLIBCXX_MOVE(__comp));
  std::__push_heap(__first, __holeIndex, __topIndex,
    _GLIBCXX_MOVE(__value), __cmp);
}
```

GNU, clang, VS `make_heap`

- Use moves instead of swaps
- Special cases the sibling-less last leaf
- Inner loop: 2 compare/jump, 4 arith, 2 assign
- Outer loop: large fixup code to handle last node

Optimization is Imagination

- Move fixup code outside outer loop
- Integrate conditionals as arithmetic

- Debate every penny like the lawyer of an insurance company
  - Fight for every cycle in the inner loop!

Let's Do This

```
template <typename It>
void insertion_sort_heap(It first, It last) {
  assert(first < last); // 0 size handled outside
  const size_t size = last - first;
  if (size < 3) {
      sort2(first[0], first[size == 2]);
      return;
  }
  make_heap(first, size);
  unguarded_insertion_sort(first + 2, last);
}
```

Heapifying

```
template <typename It>
void make_heap(const It a, const size_t size) {
  assert(size > 2); // other sizes handled outside
  size_t firstParent = (size - 3) / 2;
  size_t firstRightKid = (firstParent + 1) * 2;
  for (;; --firstParent, firstRightKid -= 2) {
    ... outer loop ...
  }
  if (size & 1) return;
  // Fixup for only child
  push_heap(a, a + size);
}
```

Outer Loop

```
for (;; firstRightKid -= 2, --firstParent) {
    const auto lucifer = a[firstParent];
    auto parent = firstParent;
    auto rightKid = firstRightKid;
    for (;;) {
      ... inner loop ...
    }
    if (parent != firstParent)
        write: a[parent] = lucifer;
    if (firstParent == 0) break;
}
```

The Pit of Hell: Inner Loop

```
for (;;) {
  const auto jr = rightKid -
    (a[rightKid - 1] <= a[rightKid]);
  const auto crt = a[jr];
  if (lucifer <= crt)
      break;
  a[parent] = crt;
  parent = jr;
  rightKid = (jr + 1) * 2;
  if (rightKid >= size)
      goto write; // SO SUE ME
}
```

Ended Up Writing push_heap, Too

```
for (auto i = size - 1;; ) {
  const auto parent = (i - 1) / 2;
  const auto ai = a[i], ap = a[parent];
  if (ap <= ai) break;
  a[parent] = ai;
  a[i] = ap;
  if (parent == 0) break;
  i = parent;
}
```

- Inefficiencies in GNU, clang, VS directly caused by the use of structured loops

Always* Use Infinite Loops

* Except in most cases

Analysis

- 3 comparisons but only 2 compare/jump
    - GNU, clang, VS end up doing more work
- 2 arith, 2 assigns

- Let’s put this to test!

Meh

- Test on 1M random `double`s, threshold 32
- `std::sort`: 60.54 ms
- `insertion_sort_heap`: 61.85 ms
- Getting close (2%) but not breaking even

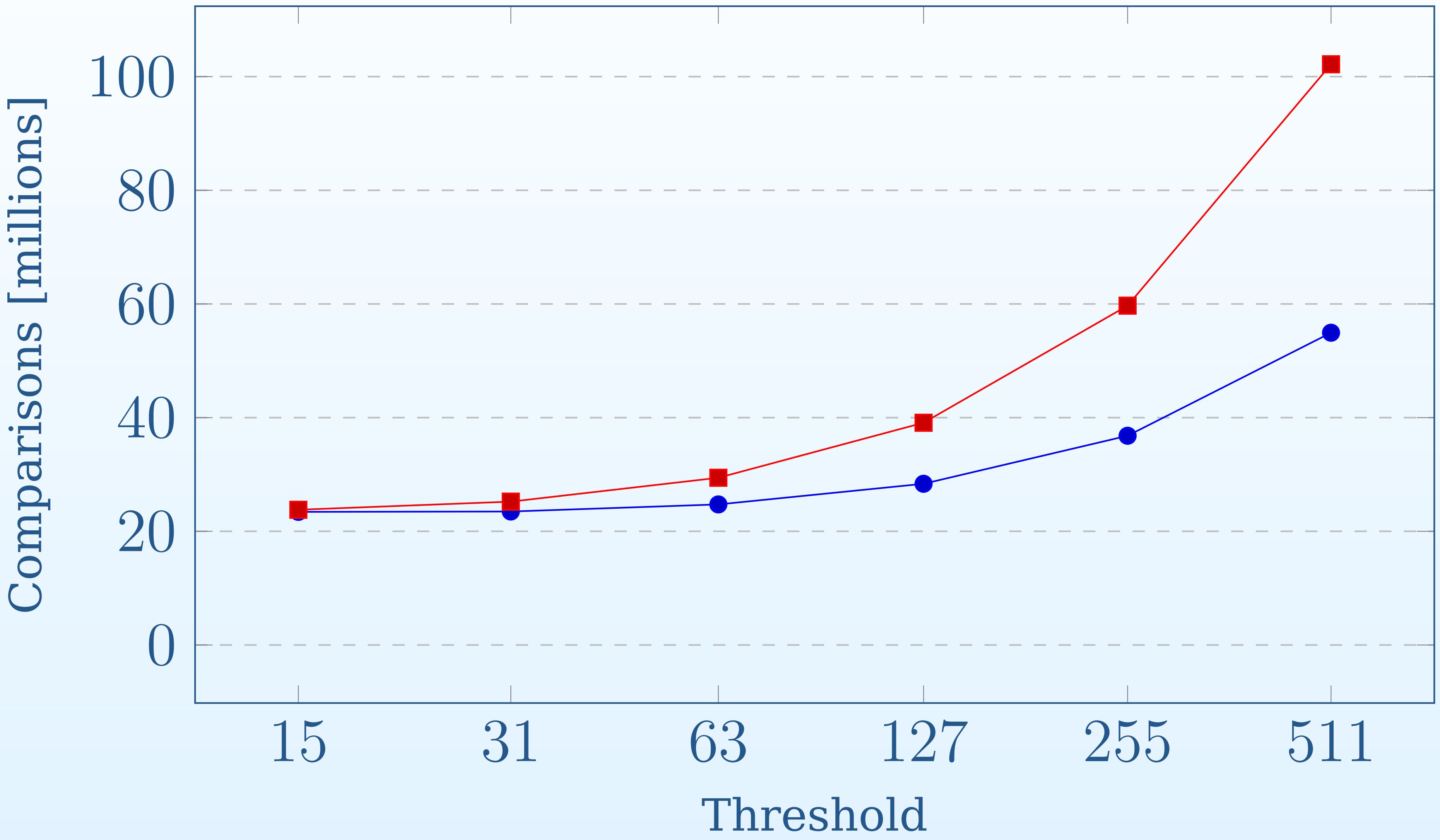
- But wait. . .
- We can increase `THRESHOLD` without compromising counts

Finally! A Significant Win

- `THRESHOLD=63`
- Reduces comparisons by 2% (for all types)
- Reduces swaps by 1.5% (for all types)
- Reduces runtime by 3% (for `double`)
    - Elaborate types only get better

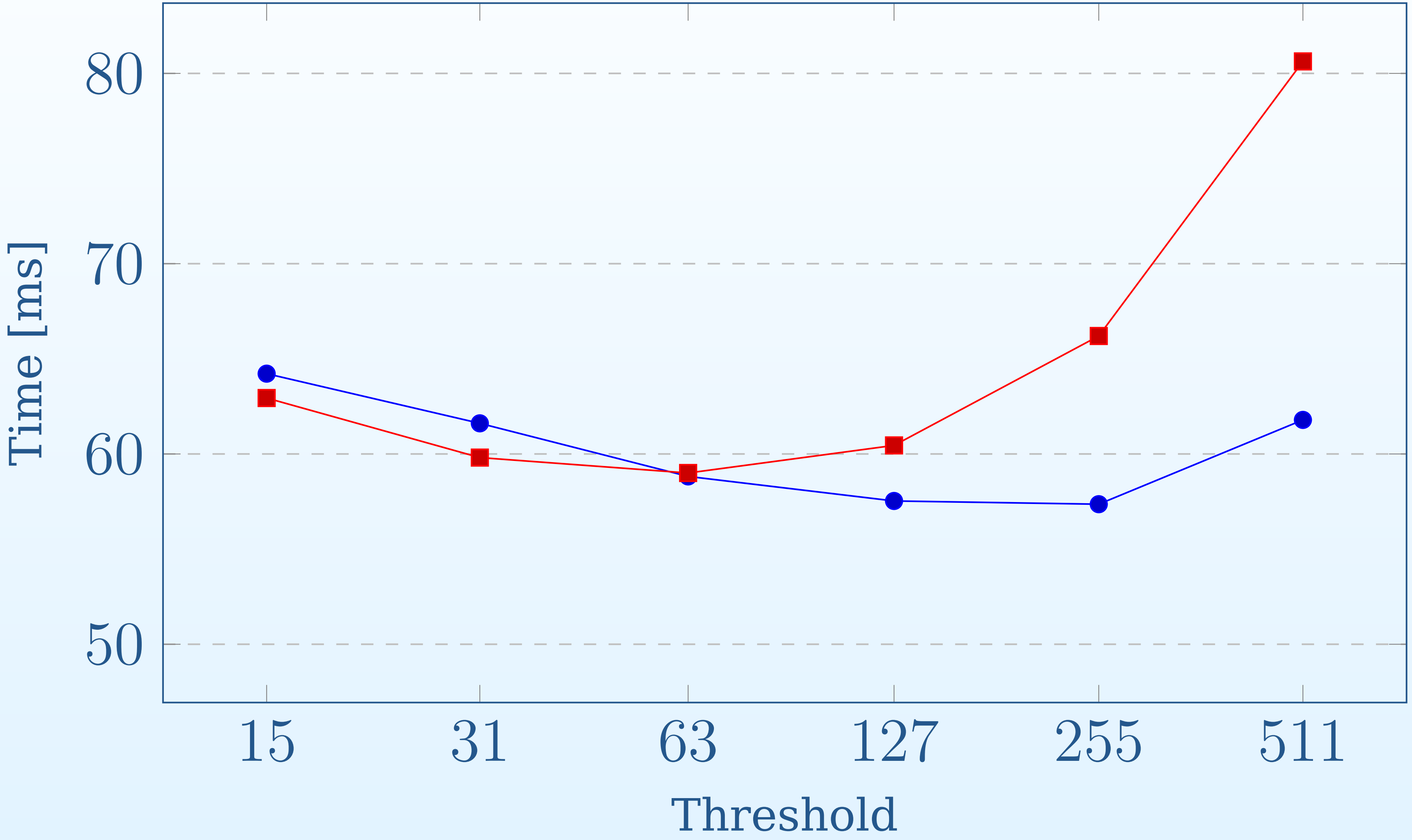
"Gasp! The speaker has gone completely mad!"

Comparisons (baseline: red)



Moves

Time



Trying Silly Things

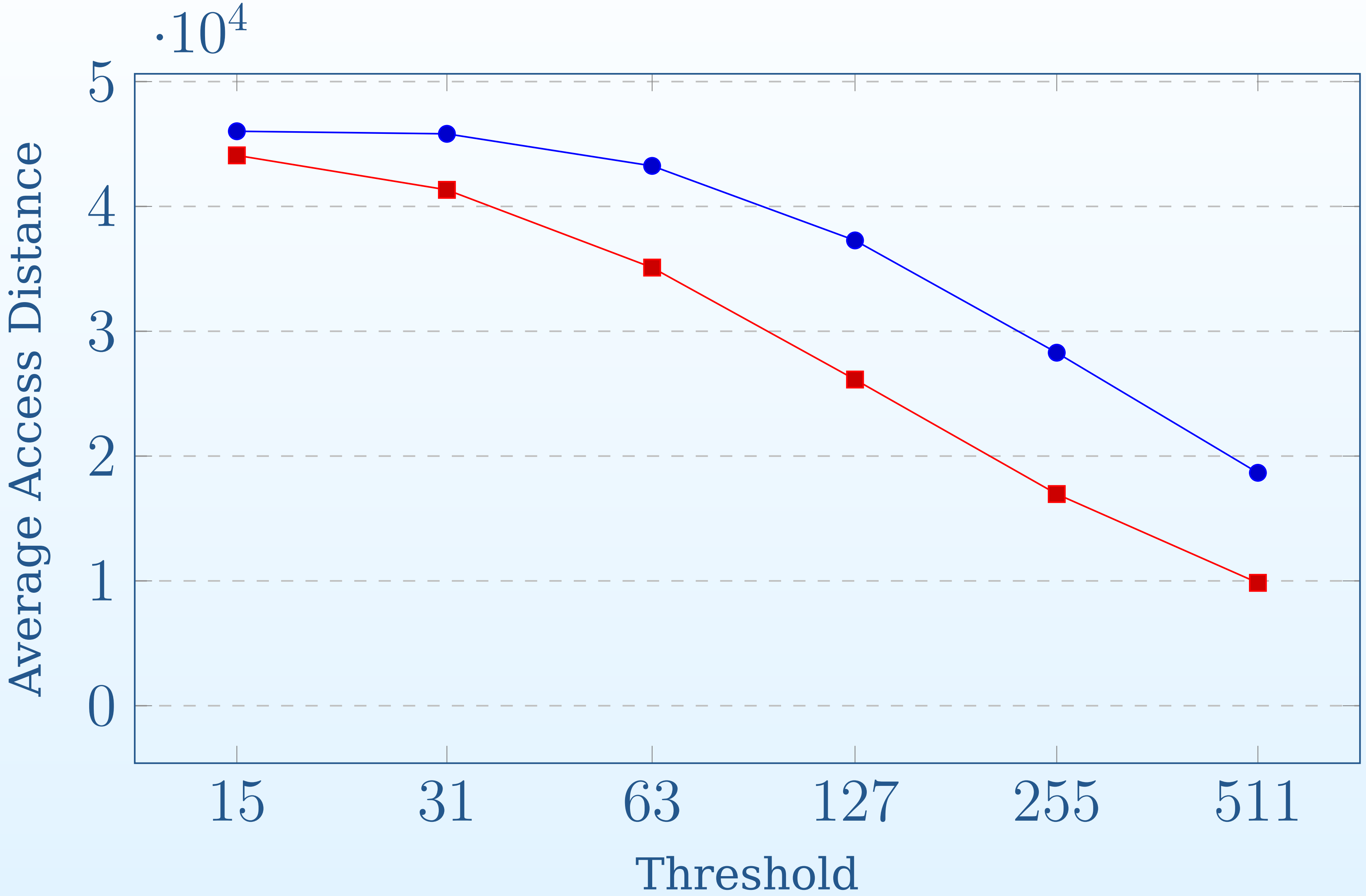
- Increasing `THRESHOLD` further:
  - Comparisons increase
  - Swaps increase
  - Time *continues to drop*
- Sweet spot (for heap insert, `double`): 255
  - 36.81M comparisons (46% worse than baseline)
  - 25.7M moves (88% worse)
  - Time: **57 ms** (*4% better*)

A Helpful Metric

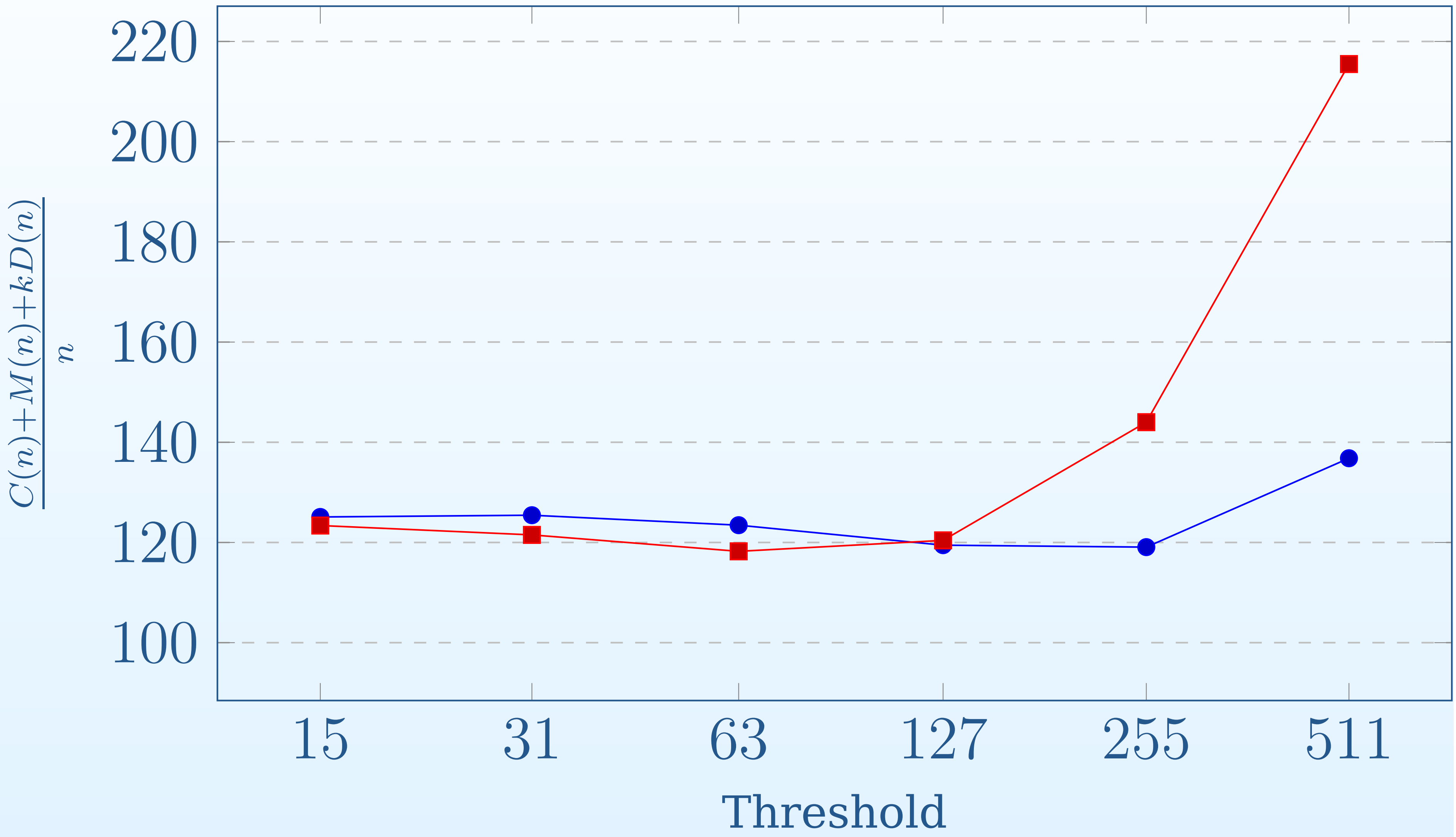
- Collect $D(n)$, average distance between two subsequent array accesses
- A proxy for (non-)locality of array access
- Quicksort: $D(n)$ is large
- Insertion sort, heap+insertion sort: $D(n)$ is smaller
- $D(n)$ decreases as `THRESHOLD` increases

- $C(n)$ and $S(n)$ don't tell the whole story
- $D(n)$ helps independently of cache particulars

Average Access Distance



Blended Cost $(C(n)+M(n)+kD(n))/n$



"But not all data is random!"

- More concerns? More measurements!

- Sorted: $0, 1, 2, \ldots, n-1$
- Reversed: $n-1, n-2, \ldots, 0$
- Organpipe: $0, 1, 2, \ldots, n/2, n/2-1, \ldots, 1, 0$
- Rotated: $1, 2, \ldots, n-1, 0$
- Random01: $0, 1, 0, 0, 1, 0, 1, 1, \ldots$

- Key: cannot specialize `THRESHOLD` on shape!

Sorted Data

- 1M sorted `doubles`
- Plain insertion sort: 10.92 ms
- Heap insertion sort: 9.99 ms

- Result: heap wins (9%)

Reversed Data

- $n-1, n-2, \ldots, 0$
- Plain insertion sort: 8.43 ms
- Heap insertion sort: 8.13 ms

- Result: heap wins (3.7%)

Organ Pipe Data

- $0, 1, 2, \ldots, n/2, n/2 - 1, \ldots, 1, 0$
- Plain insertion sort: 47.28 ms
- Heap insertion sort: 49.47 ms

- Result: plain wins (4.6%)
- (TODO: investigate anomalies)

Rotated Data

- $1, 2, \ldots, n-1, 0$
- Plain insertion sort: falls back to heapsort
- Heap insertion sort: falls back to heapsort
- GNU pivot choice leads to quadratic time

- Result: the GNU libstdc++ team has homework

Random 0/1 Data

- $n/2$ zeros and $n/2$ ones randomly shuffled
- Plain insertion sort: 14.27 ms
- Heap insertion sort: 11.45 ms

- Result: heap wins (24.6%)

Result

Heapifying Before Insertion Sort Significantly Faster On Most Tested Distributions

Coda

First-Order Conclusions

- Throw the structures & algos book away
- Research papers & industry is where it's at
- Devise proper perf metrics and proxies
- Measure everything like crazy

- Try silly things!

First-Order Conclusion

Code that wants to be fast is left-leaning.

Second-Order Conclusions

- What is the perfect `std::sort` for C++?
- Use hardcoded versions for very small sizes
- Use Radix Sort for small integers, default ordering
- Choose `THRESHOLD` depending on:
  - Cost of moving
  - Cost of comparison
  - `sizeof` element
  - Data contiguity
- Customize depending on trivial move, copy
- *Tension with Generic Programming*

How to Encode Cost of Operations?

- Difficult problem
- But heuristics go a long way

- Implementation: User Defined Attributes!
- Assign fixed costs to primitive operations
- Users can affix & propagate UDAs with functions
- Introspection queries the UDA

Tension

- Generic Programming
  - Define broad categories
  - Write algorithms tailored along them
- Design by Introspection
  - Can’t categorize everything for everyone
  - Just crack types open, introspect, customize
  - Unprecedently compact renderings of algos

Second-Order Conclusion

Generic Programming
Is Why We Can't Have
Nice Things

Timeline

- 1990s: OOP

Timeline

- 1990s: OOP
  - Inheritance and virtuals, wheeee!...

Timeline

- 1990s: OOP
  - Inheritance and virtuals, wheee!...
- 2000s: Generic Programming

Timeline

- 1990s: OOP
  - Inheritance and virtuals, wheee!...
- 2000s: Generic Programming
  - Iterators and algos, wheee!...

Timeline

- 1990s: OOP
  - Inheritance and virtuals, wheee!. . .
- 2000s: Generic Programming
  - Iterators and algos, wheee!. . .
- 2020s: Design By Introspection

Timeline

- 1990s: OOP
  - Inheritance and virtuals, wheee!...
- 2000s: Generic Programming
  - Iterators and algos, wheee!...
- 2020s: Design By Introspection
  - Inspect and Customize Everything Everywhere

Timeline

- 1990s: OOP
  - Inheritance and virtuals, wheee!. . .
- 2000s: Generic Programming
  - Iterators and algos, wheee!. . .
- 2020s: Design By Introspection
  - Inspect and Customize Everything Everywhere, wheee?. . .

The End