We're preparing for your visit...

Quicker Sorting

Dietmar Kühl

Quicker Sorting

Dietmar Kühl
API Technology London
Bloomberg L.P.

Copyright Notice

Overview

- quickly sorting out Quick-Sort
- implement and compare variations of Quick-Sort
- turning the concrete implementation generic

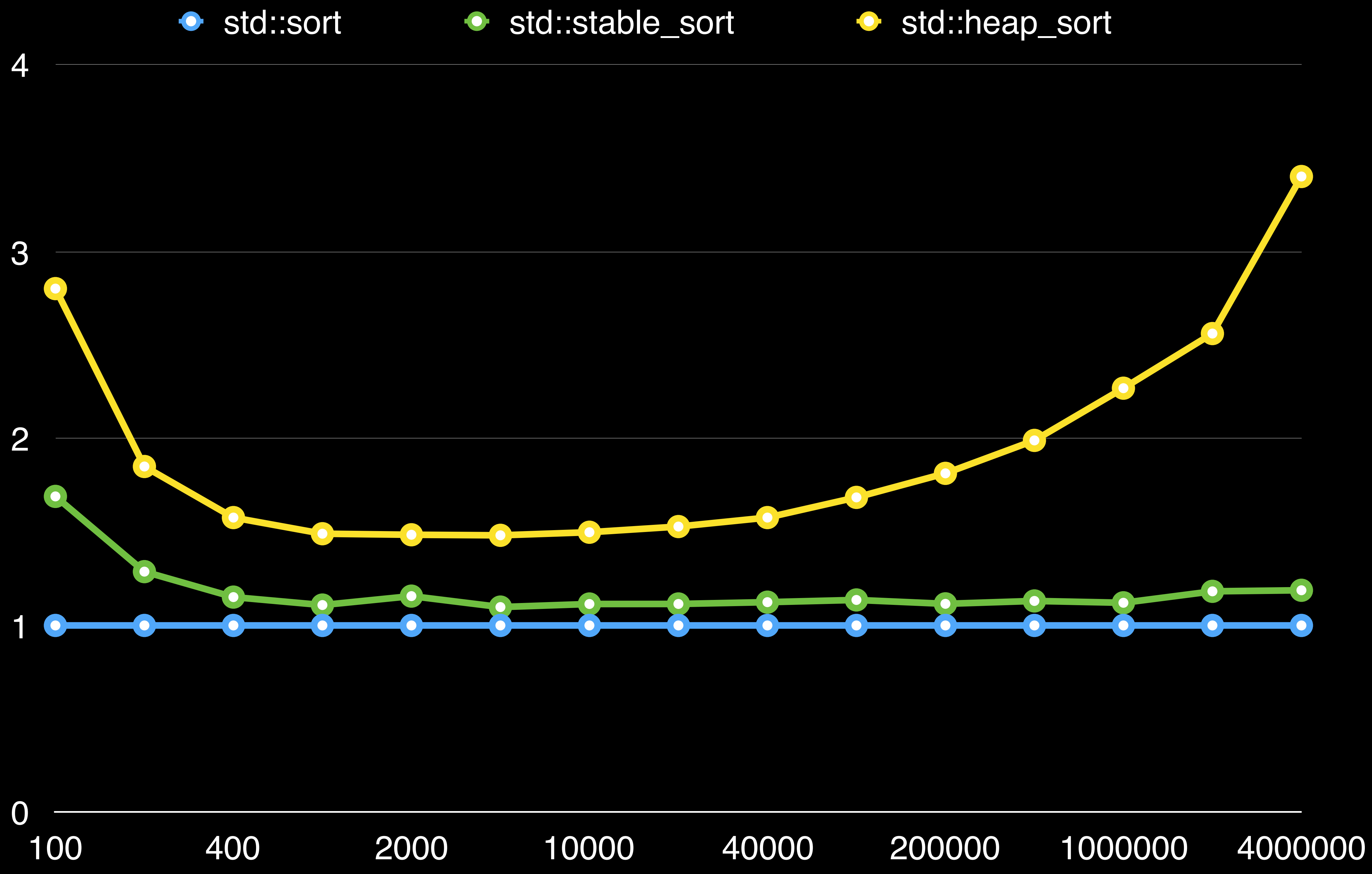
C++ Standard Library

- complexity of C++ Standard Library sort algorithms is O(n log n)
- std::sort() - in-place, fast
- std::stable_sort() - stable, not in-place
- std::make_heap()/std::sort_heap()

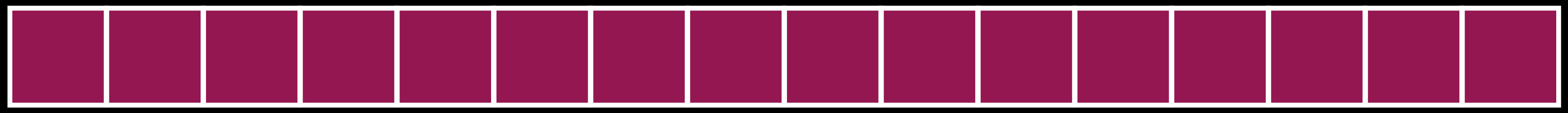
Data Set

- multiple random sequence
    - with varying degrees of equal values
- one each ascending/descending sequence
- one sequence with all the same value

Results

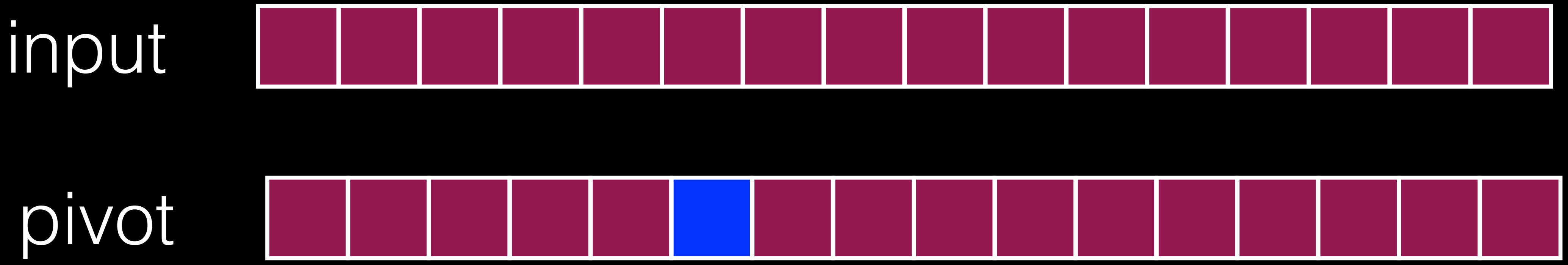

std::sort
std::stable_sort
std::heap_sort
4
3
2
1
0
100
400
2000
10000
40000
200000
1000000
4000000

Quick-Sort



input

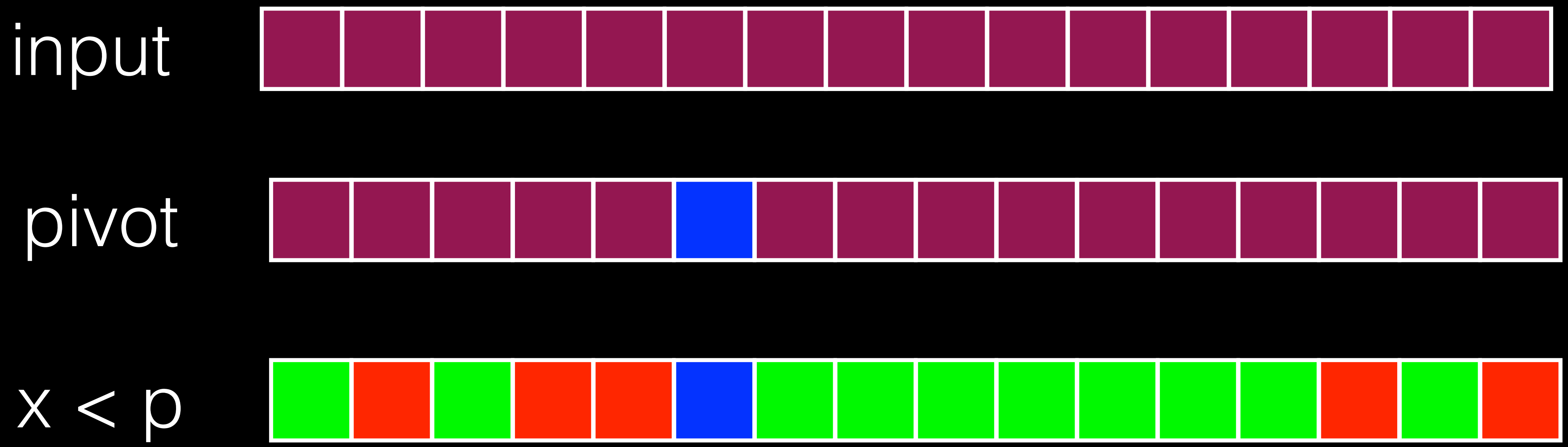
Quick-Sort



input

pivot

Quick-Sort

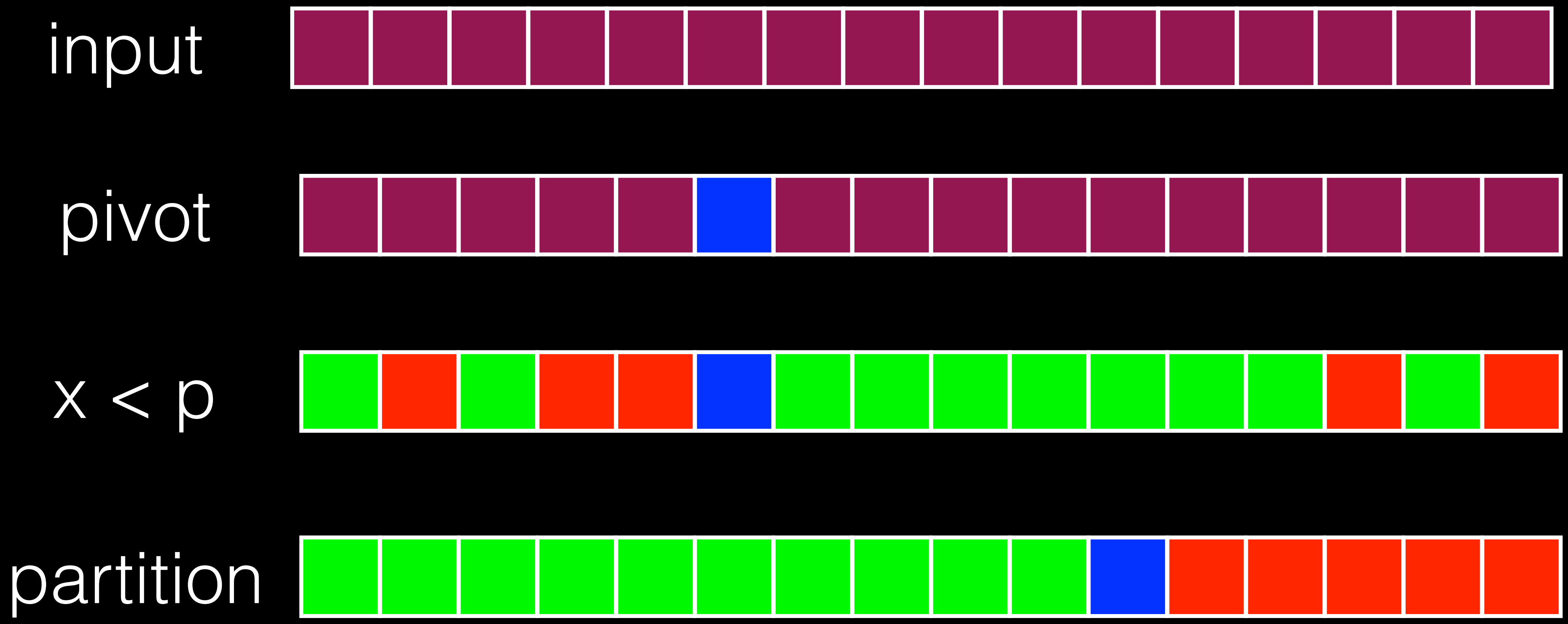


input

pivot

x < p

Quick-Sort



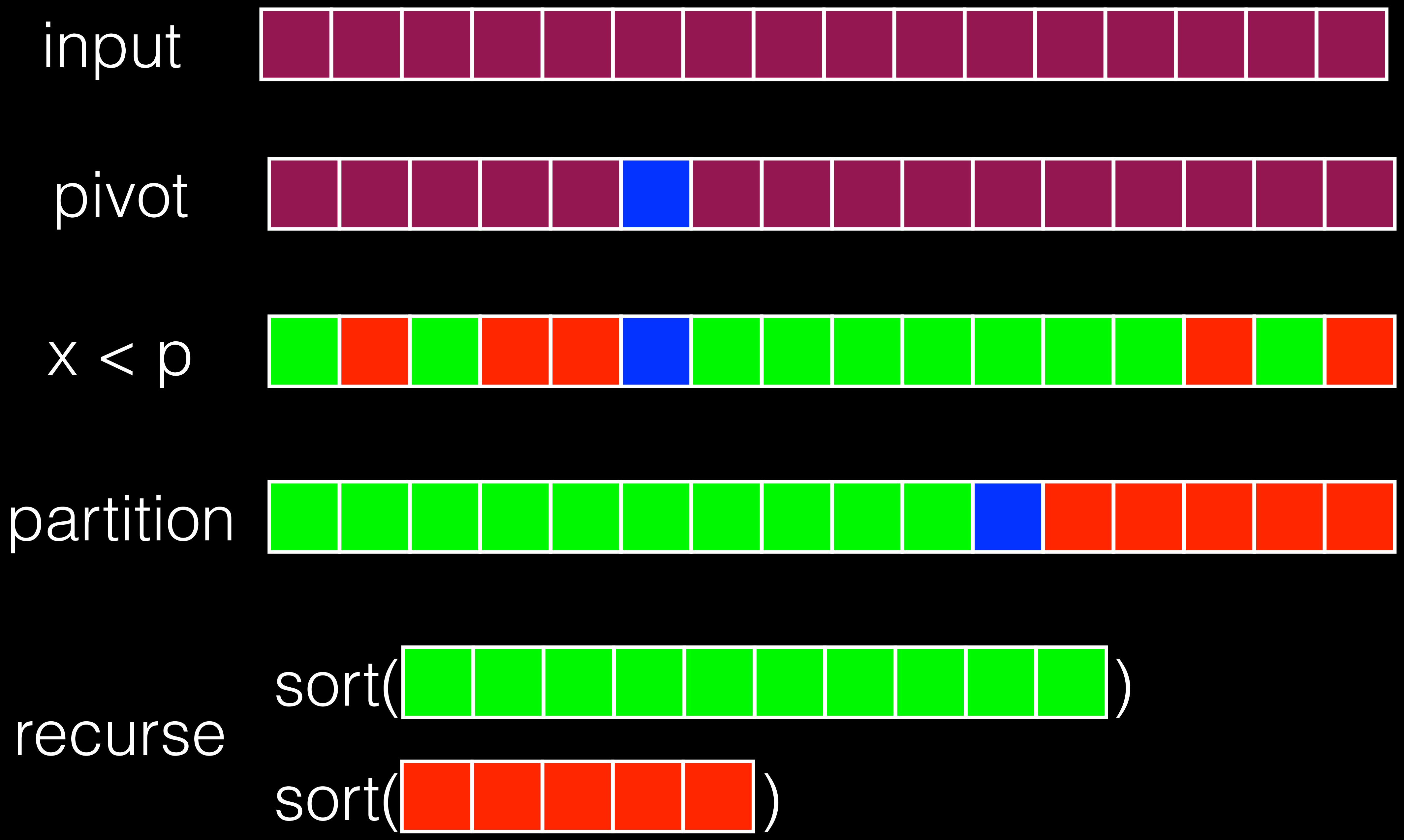
input

pivot

x < p

partition

Quick-Sort



input

pivot

x < p

partition

recurse

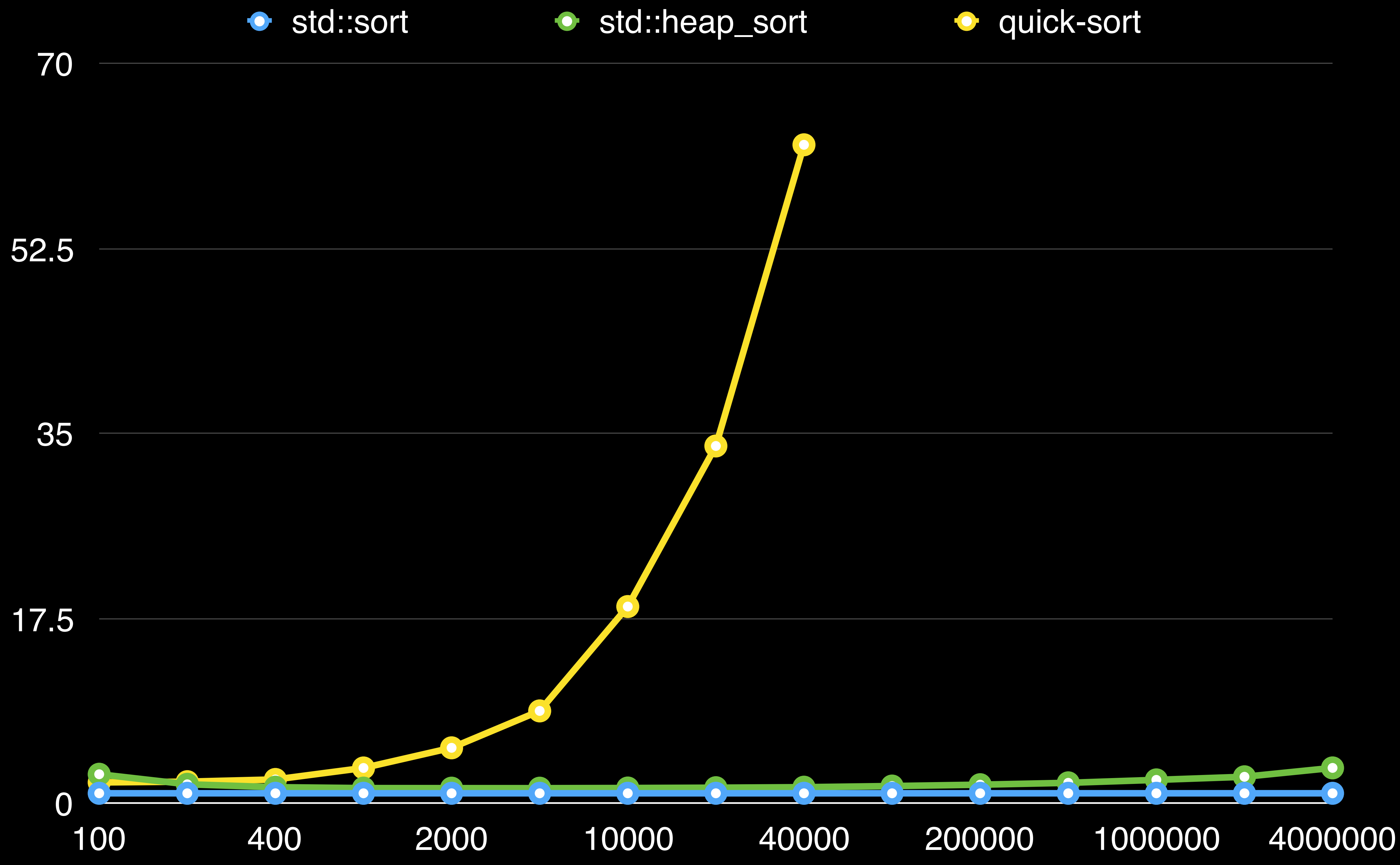
sort( )

sort( )

Simple Implementation

```
void sort(int* begin, int* end) {
    std::size_t len = std::distance(begin, end);
    if (len <= 1) return;
    int* pivot = end - 1;
    int* mid = std::partition(begin, pivot,
                    [=](int value){ return value < *pivot; });
    swap(*mid, *pivot);
    sort(begin, mid);
    sort(mid + 1, end);
}
```

Results



std::sort
std::heap_sort
quick-sort
70
52.5
35
17.5
0
100
400
2000
10000
40000
200000
1000000
4000000

Use a Pivot

- popular choice: median of 3 (or more)
- works well with sorted or reverse sorted inputs
- adds extra costs to determine the median
- … using last is bad, though

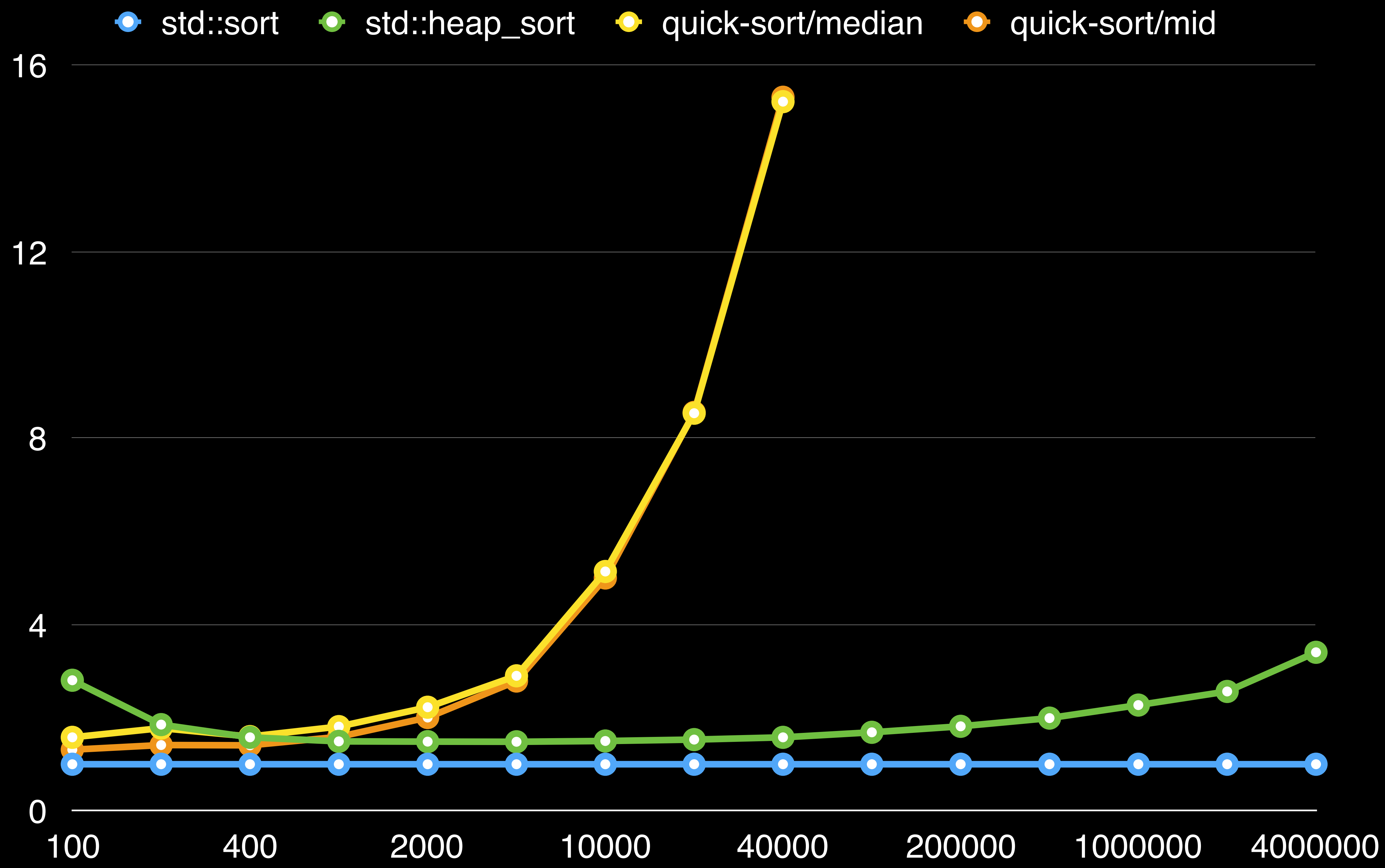
Pivot Implementation

```
void sort(int* begin, int* end) {
    std::size_t len = std::distance(begin, end);
    if (len <= 1) return;
    int* pivot = end - 1, * mid = begin + len / 2;
    small::sort(begin, pivot, mid);
    mid = std::partition(begin, pivot,
                              [=](int arg){ return arg < *pivot; });
    swap(*mid, *pivot);
    sort(begin, mid);
    sort(mid + 1, end);
}
```

Sorting 3 Elements

```
template <typename I>
static inline void small::sort(I a, I b, I c) {
    if (*b < *a) {
        if (*c < *b) std::iter_swap(a, c);
        else if (*c < *a) { auto t(*a); *a = *b; *b = *c; *c = t; }
        else std::iter_swap(a, b);
    } else {
        if (*c < *a) { auto t(*c); *c = *b; *b = *a; *a = t; }
        else if (*c < *b) std::iter_swap(b, c);
    }
}
```

Results



std::sort
std::heap_sort
quick-sort/median
quick-sort/mid
16
12
8
4
0
100
400
2000
10000
40000
200000
1000000
4000000

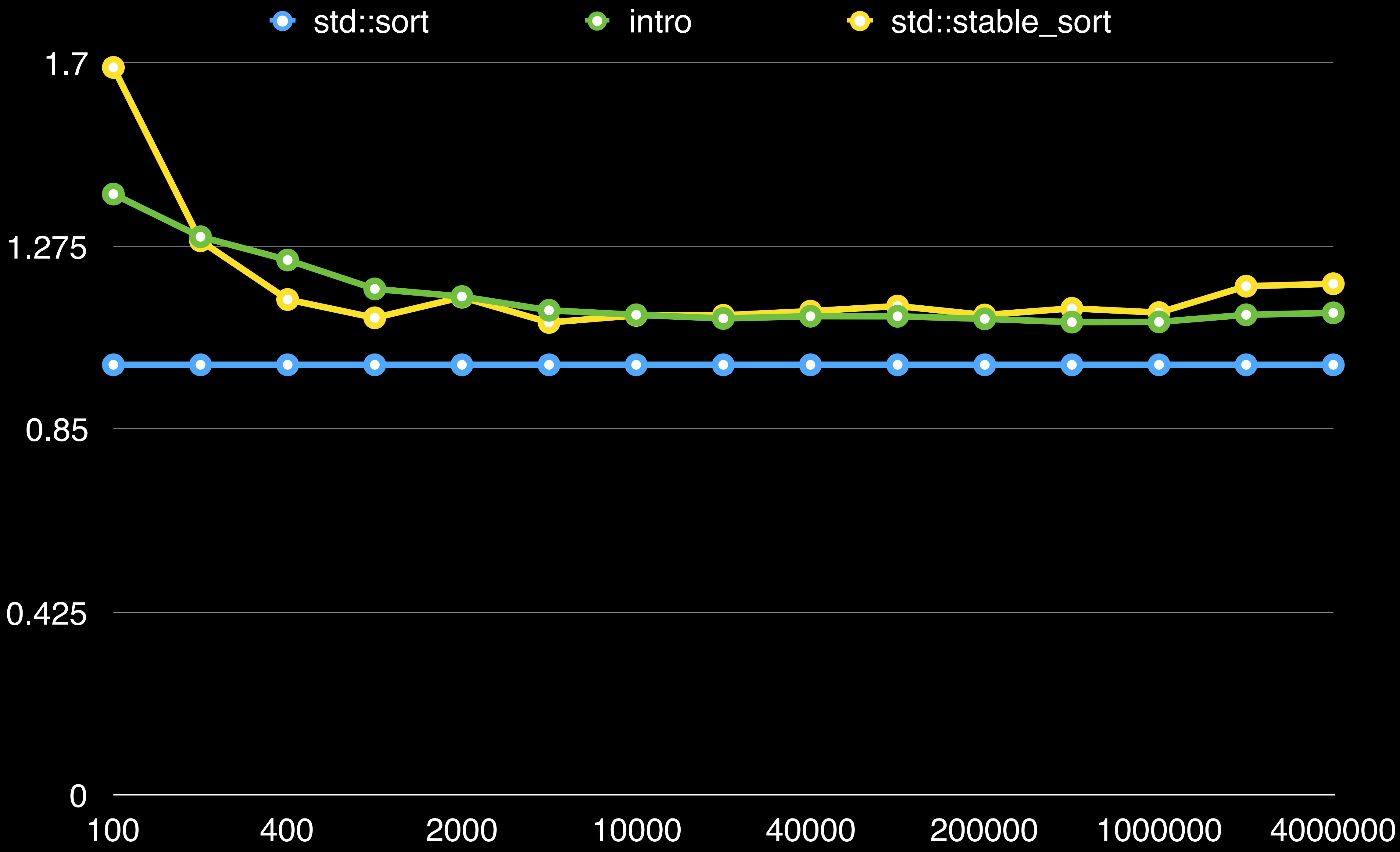
Intro Sort

- monitor depth of recursion:
  - no more than 2 log n recursive calls
- too deep ⇒ fallback to a different algorithm
  - in-place required: heap sort
  - enough spare memory: merge sort

Intro Sort

```
void sort(int* begin, int* end, int depth, int max) {
    if (++depth < max) {
        ...
        sort(begin, mid, depth, max);
        sort(mid + 1, end, depth, max);
    }
    else { std::stable_sort(begin, end); }
}
void sort(int* begin, int* end) {
   int s(end-begin), m(0); while (s >>=1) { ++m; }
    sort(begin, end, 0, 2 * m);
}
```

Results


std::sort
intro
std::stable_sort
1.7
1.275
0.85
0.425
0
100
400
2000
10000
40000
200000
1000000
4000000

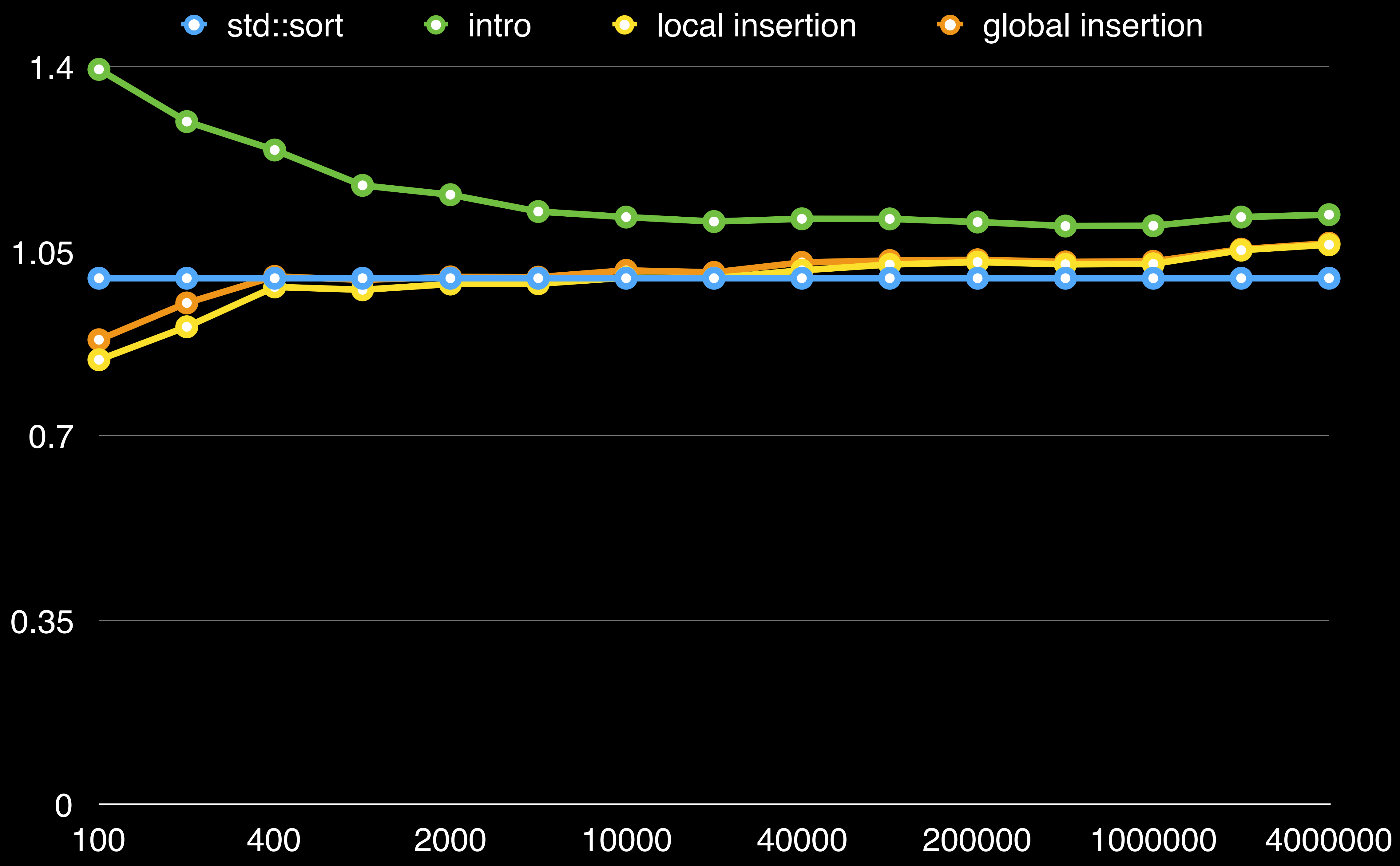
Insertion Sort

- quick sort isn’t really effective on small ranges
- insertion sort is more effective in that case
- two potential approaches:
    - stop when ranges too small and post-process
    - sort small ranges with insertion sort

Local Insertion Sort

```
if (len <= Size) {
    if (begin == end) return;
    for (int* it = begin; ++it != end; )
        if (*it < *(it - 1)) {
            int* c = it, t = *c;
            *c = *(c - 1);
            while (begin != --c && t < *(c - 1))
                *c = *(c - 1);
            *c = t;
        }
    return;
}
```

Results


std::sort
intro
local insertion
global insertion
1.4
1.05
0.7
0.35
0
100
400
2000
10000
40000
200000
1000000
4000000

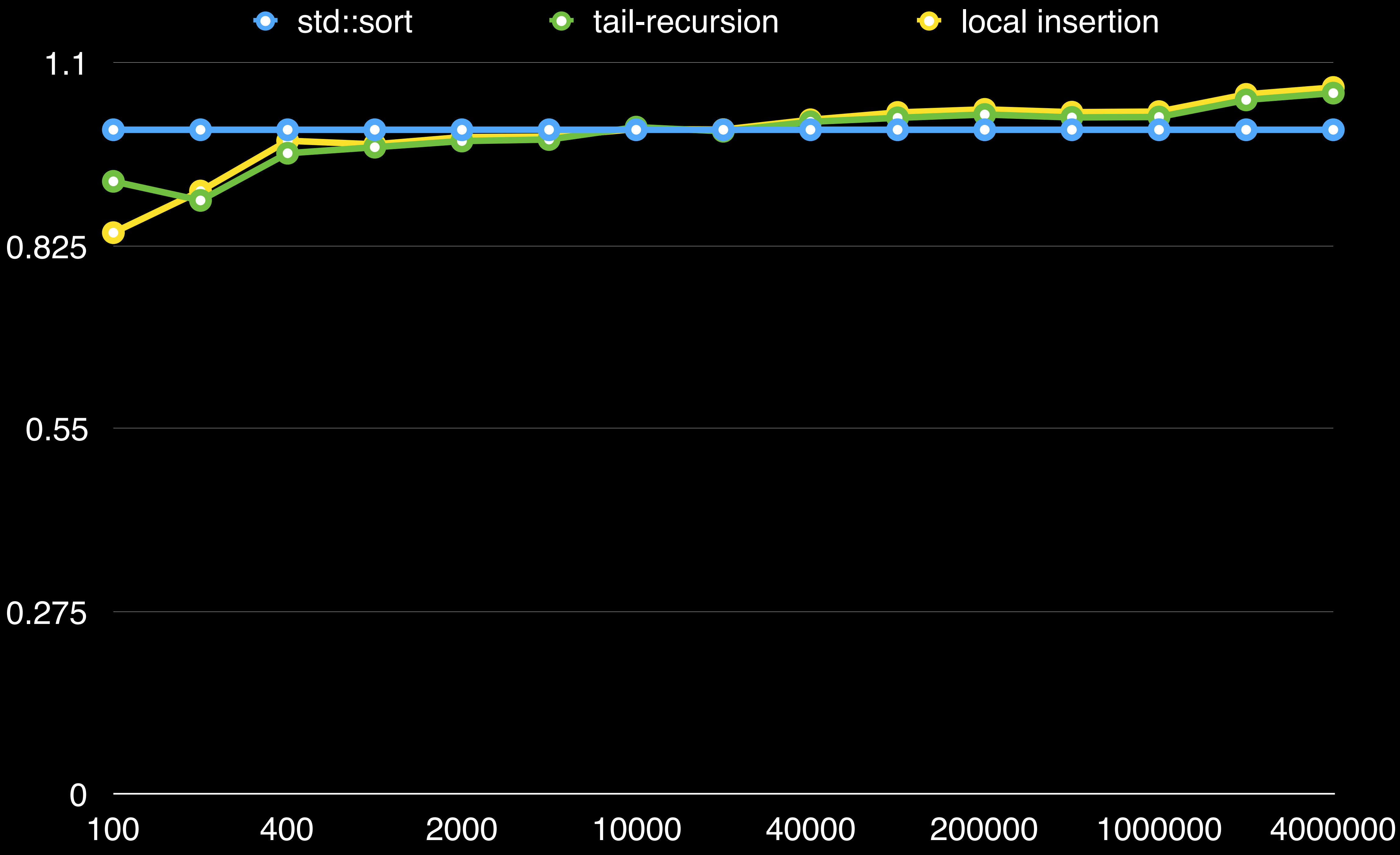
“Tail Recursion”

- using more stack is slower
- instead of call use a loop for the bigger part
- equivalent to tail call optimisation
    - however, C++ compilers are not too strong

Tail Recursion

```
while (20 < (len = std::distance(begin, end))) {
    if (++depth < max) {
        ...
        if (mid - begin < end - mid) {
            sort(begin, mid, depth, max);
            begin = mid + 1; }
        else {
            sort(mid + 1, end, depth, max);
            end = mid; }
    }
}
```

Results


std::sort
tail-recursion
local insertion
1.1
0.825
0.55
0.275
0
100
400
2000
10000
40000
200000
1000000
4000000

Partition

- so far using std::partition():

  ```
  mid = std::partition(begin, end, predicate);
  ```

- really the core part of the algorithm

Partition

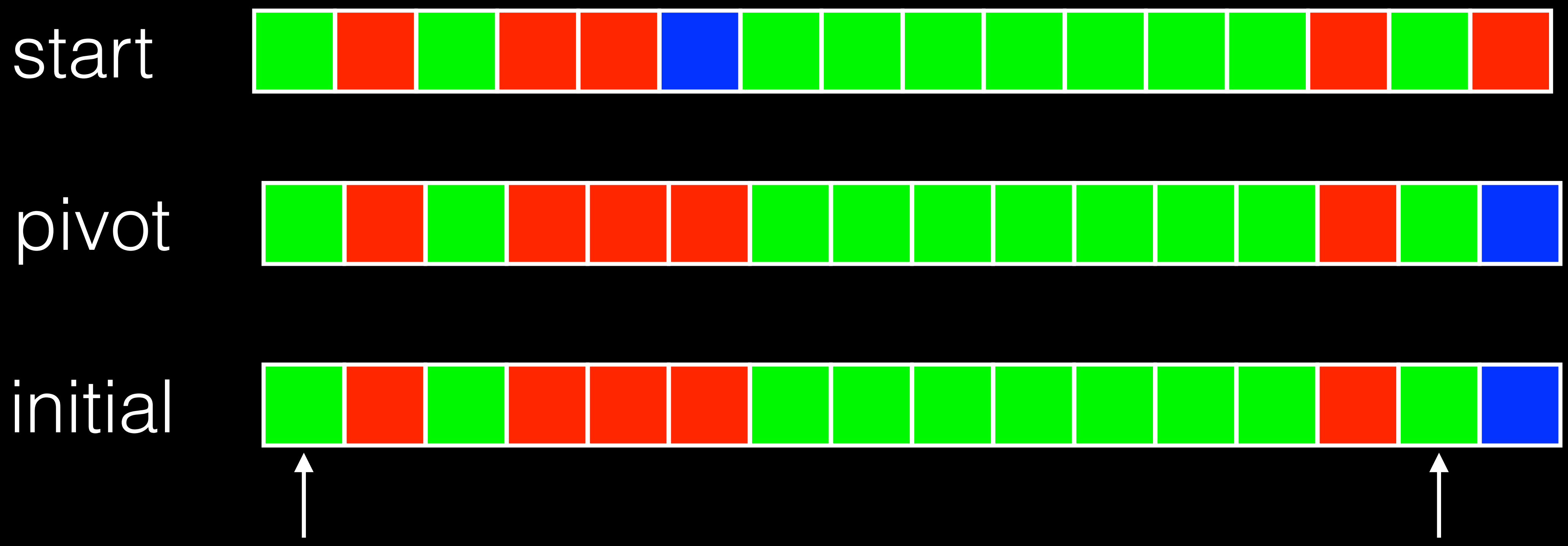
start	🟩	🟥	🟩	🟥	🟥	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟥

Partition

start 🟩🟥🟩🟥🟥🟦🟩🟩🟩🟩🟩🟩🟩🟥🟩🟥

pivot 🟩🟥🟩🟥🟥🟥🟩🟩🟩🟩🟩🟩🟩🟥🟩🟦

Partition

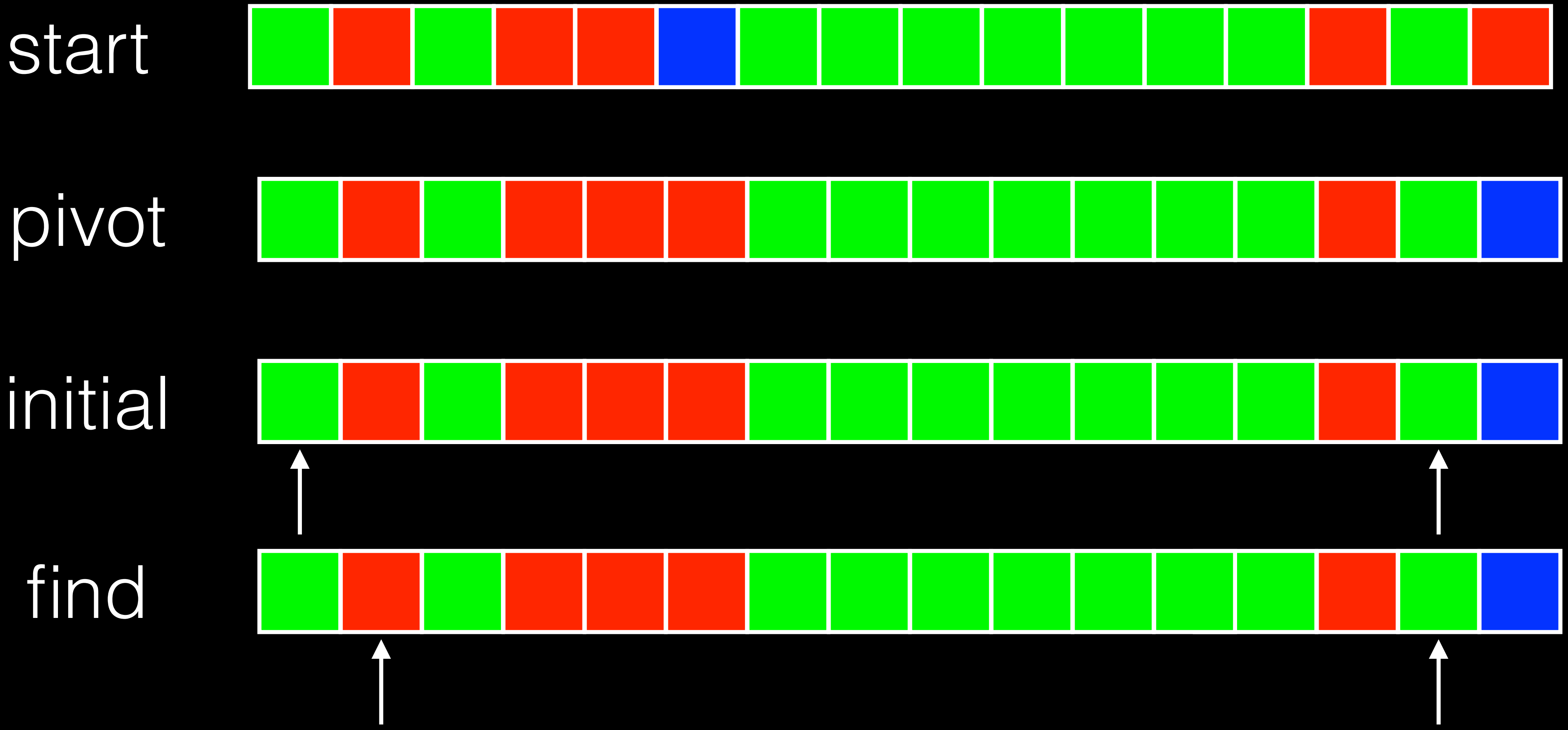


start

pivot

initial

Partition



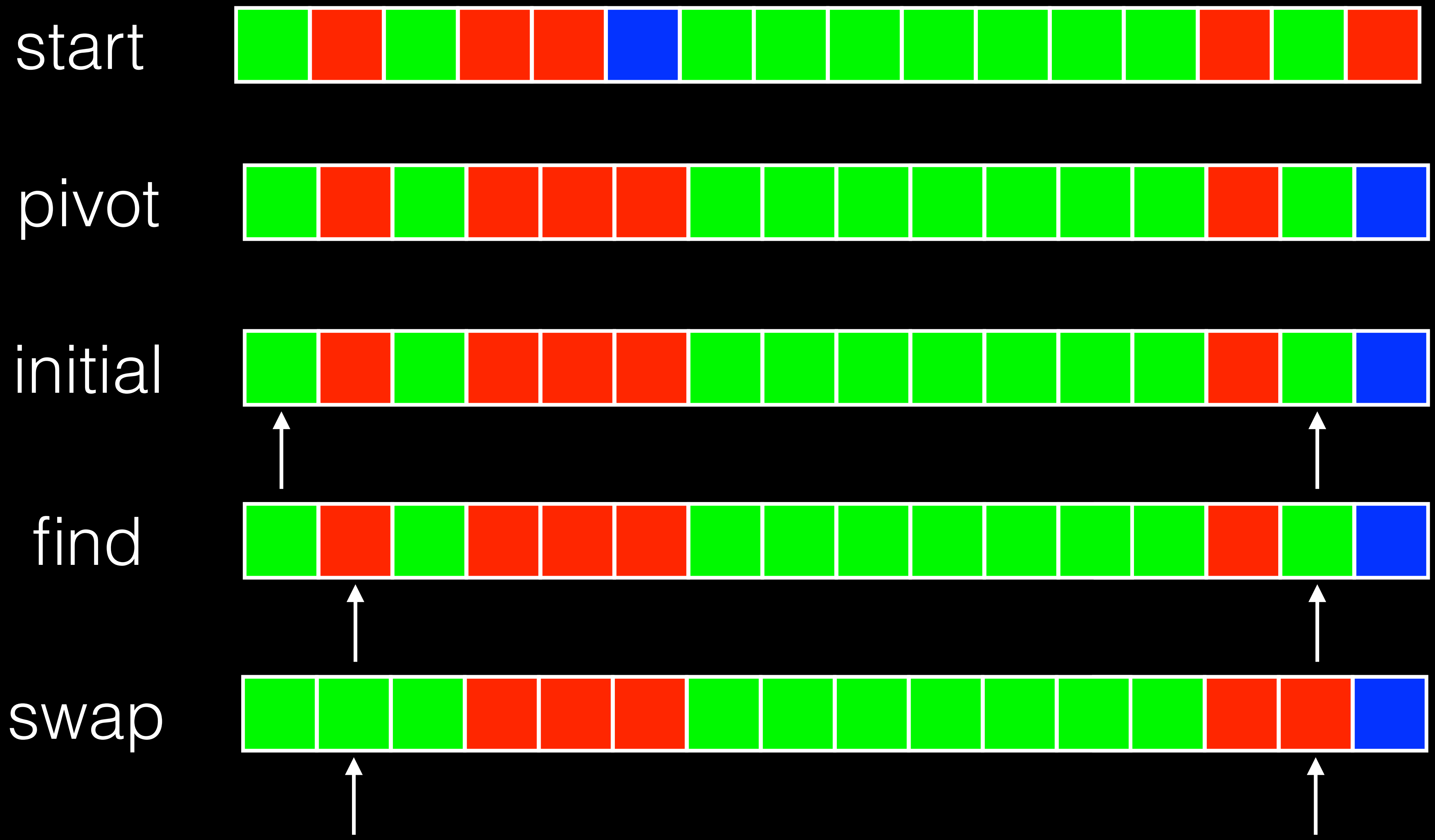
start

pivot

initial

find

Partition



start

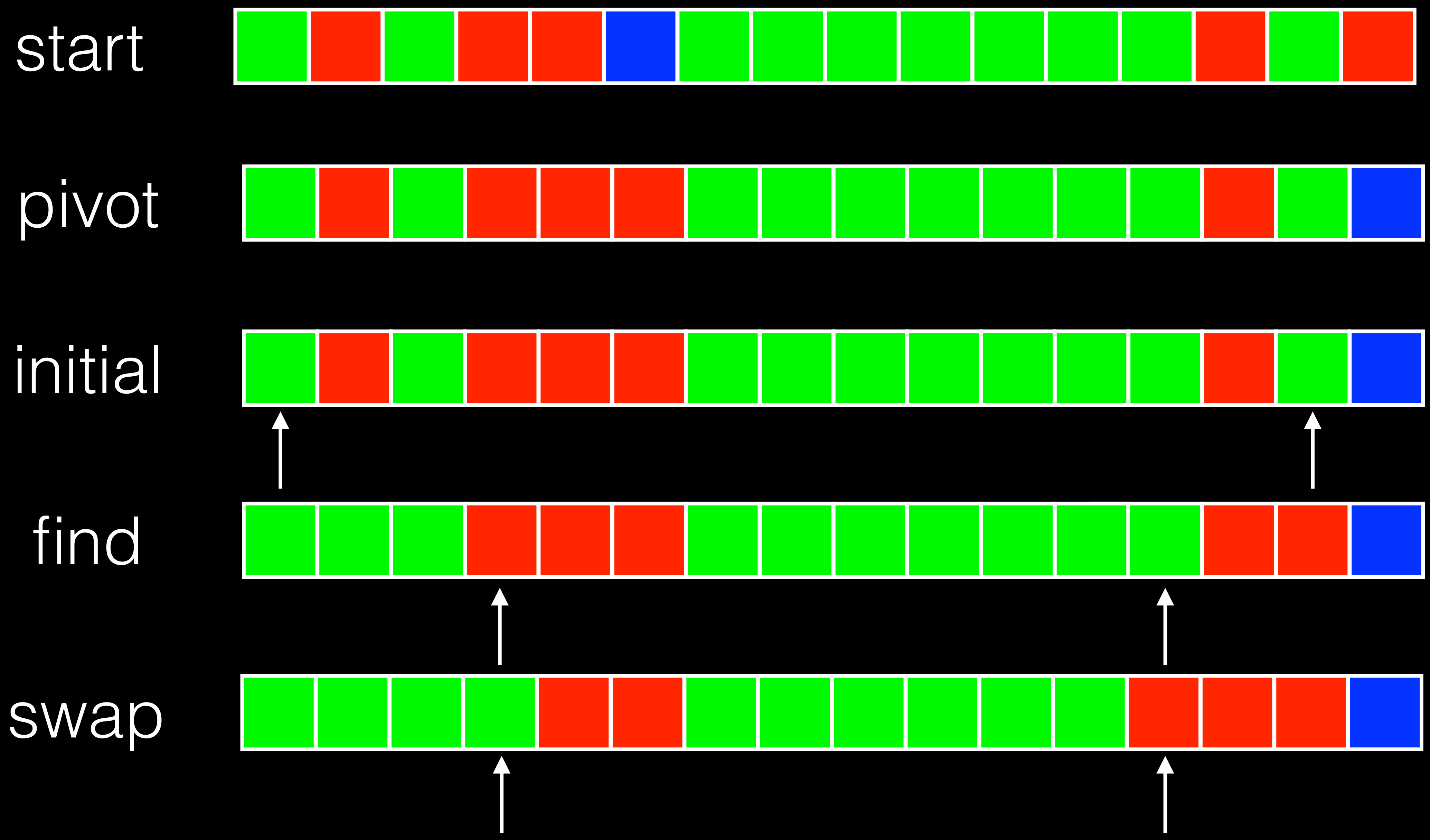
pivot

initial

find

swap

Partition



start

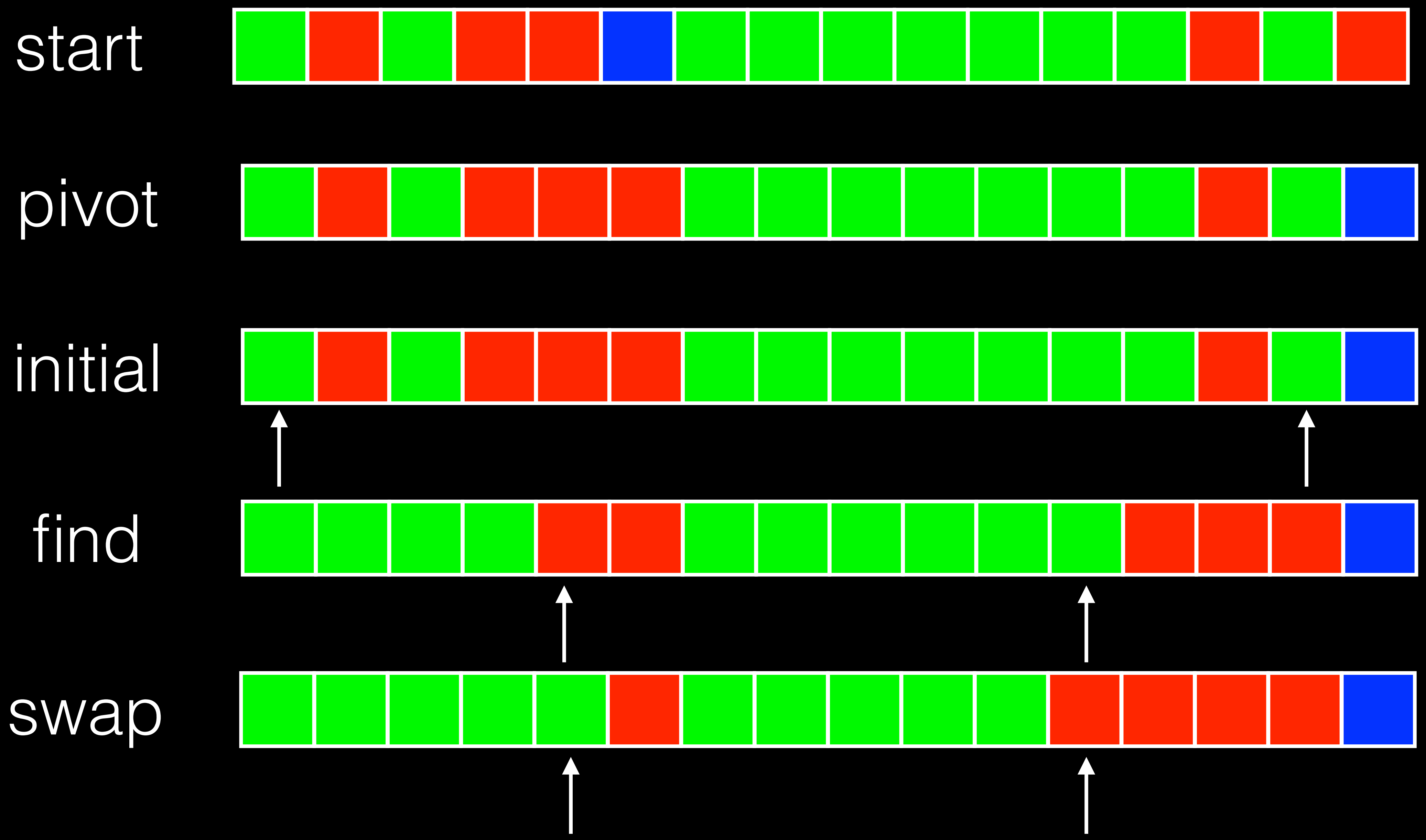
pivot

initial

find

swap

Partition



start

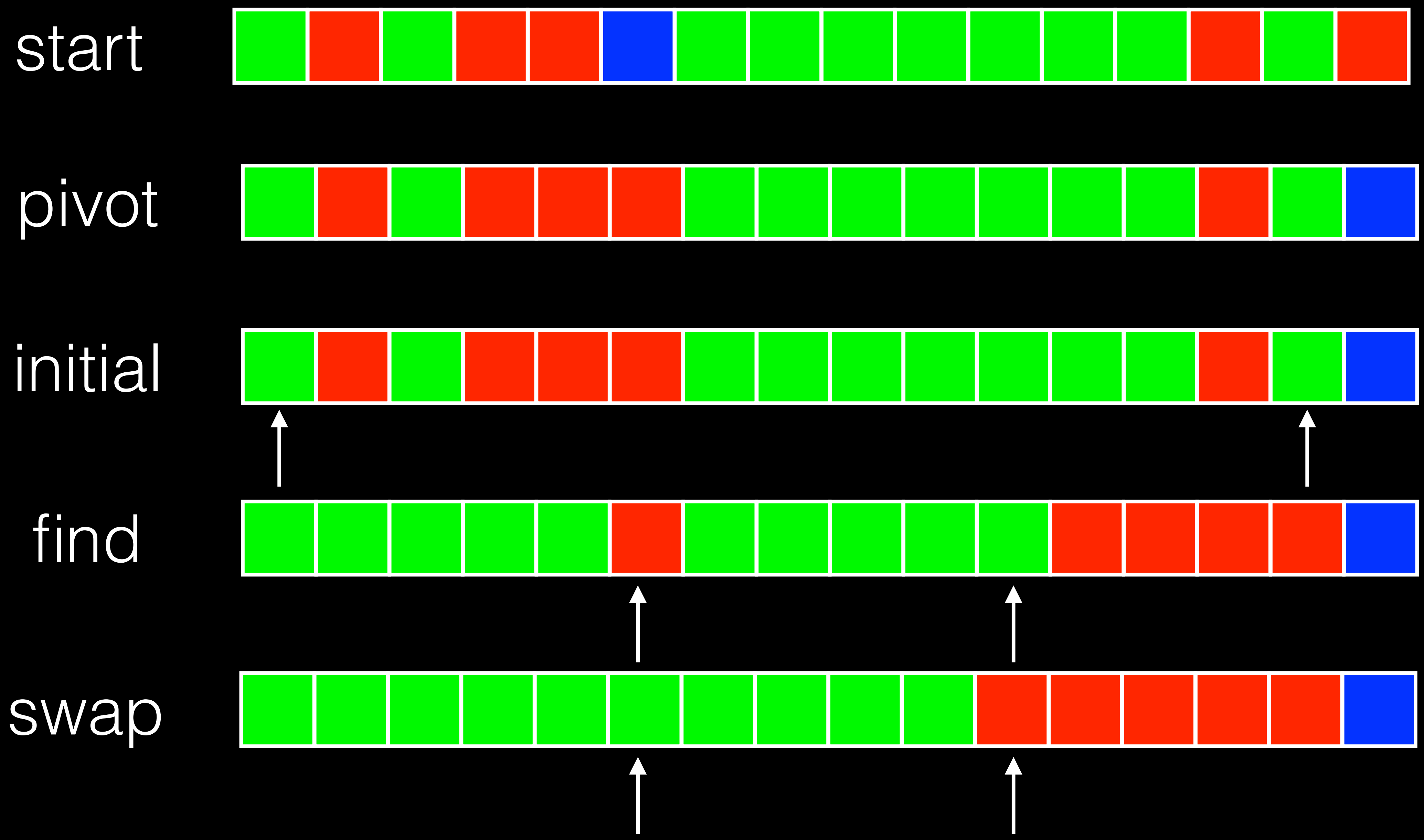
pivot

initial

find

swap

Partition



start

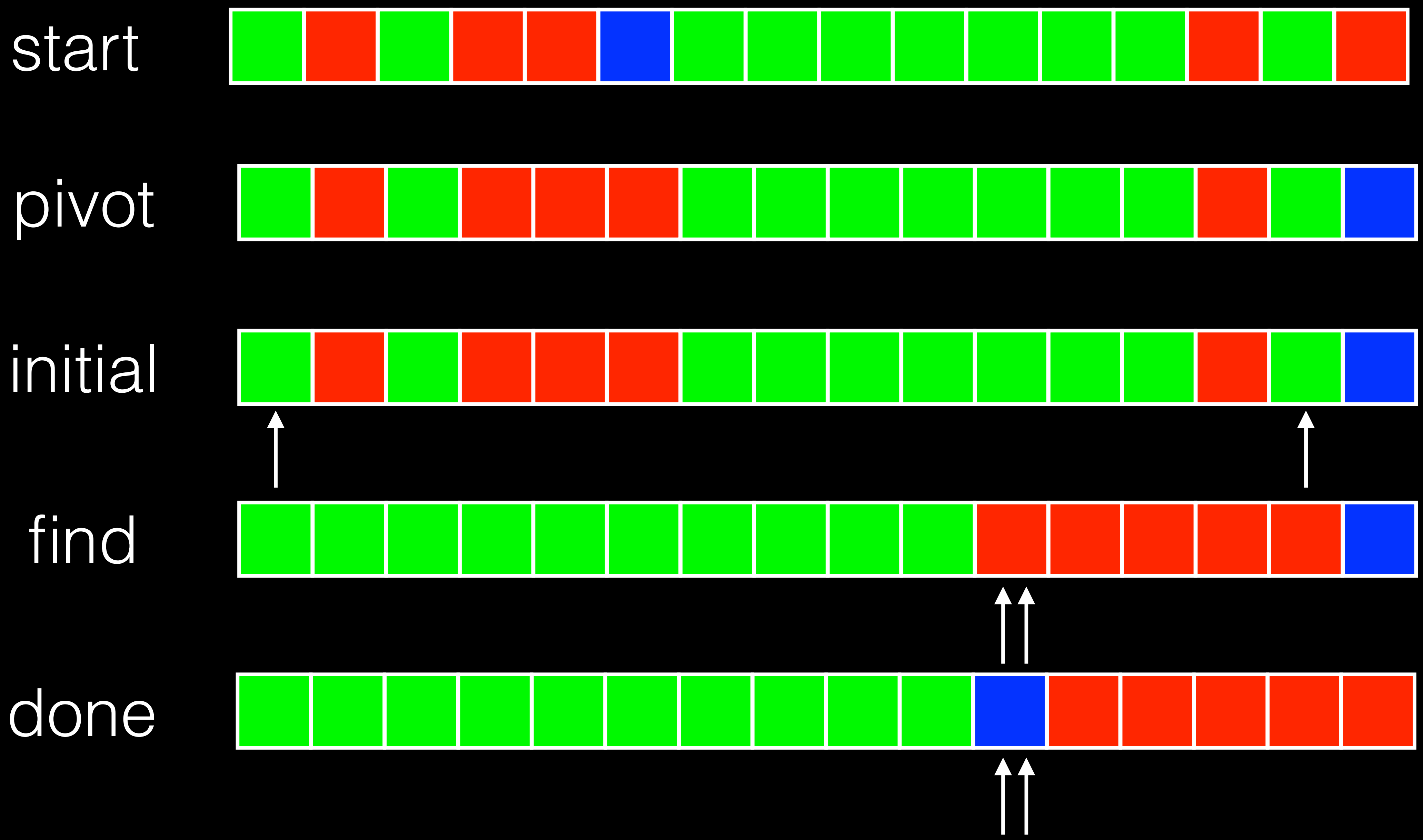
pivot

initial

find

swap

Partition



start

pivot

initial

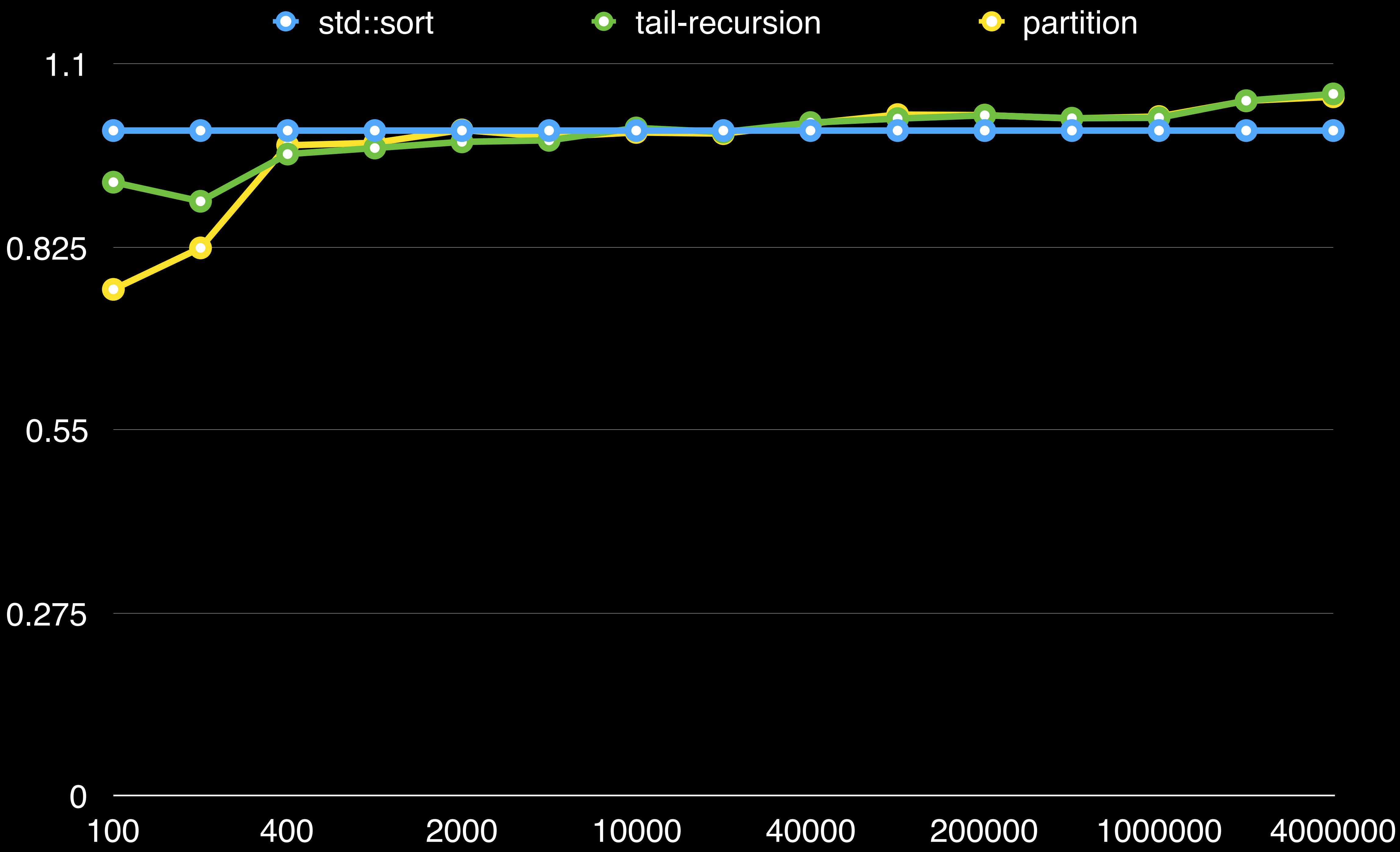
find

done

Partition

```
…
int* pbegin = begin, * pend = end;
while (true) {
    while (pbegin != pend && *pbegin < *pivot)
        ++pbegin;
    while (pbegin != pend && !(*--pend < *pivot));
    if (pbegin == pend) break;
    std::iter_swap(pbegin, pend);
    ++pbegin;
 }
mid = pbegin;
…
```

Results


std::sort
tail-recursion
partition
1.1
0.825
0.55
0.275
0
100
400
2000
10000
40000
200000
1000000
4000000

Sentinel Partition

- inner loops makes two checks per iteration:
  - determine if the end is reached
  - determine if the predicate applies
- can the check for the end be avoid?
  - use an object matching the predicate to stop

Sentinel on the Right

```
…
int* pbegin = begin, * pend = end;
while (true) {
    while (pbegin != pend && *pbegin < *pivot)
        ++pbegin;
    while (pbegin != pend && !(*--pend < *pivot));
    if (pbegin == pend) break;
    std::iter_swap(pbegin, pend);
    ++pbegin;
 }
mid = pbegin;
…
```

Sentinel on the Right

```
…
int* pbegin = begin, * pend = end;
while (true) {
    while (*pbegin < *pivot)
        ++pbegin;
    while (pbegin != pend && !(*--pend < *pivot));
    if (pbegin == pend) break;
    std::iter_swap(pbegin, pend);
    ++pbegin;
 }
mid = pbegin;
…
```

Initial Sentinel Partition

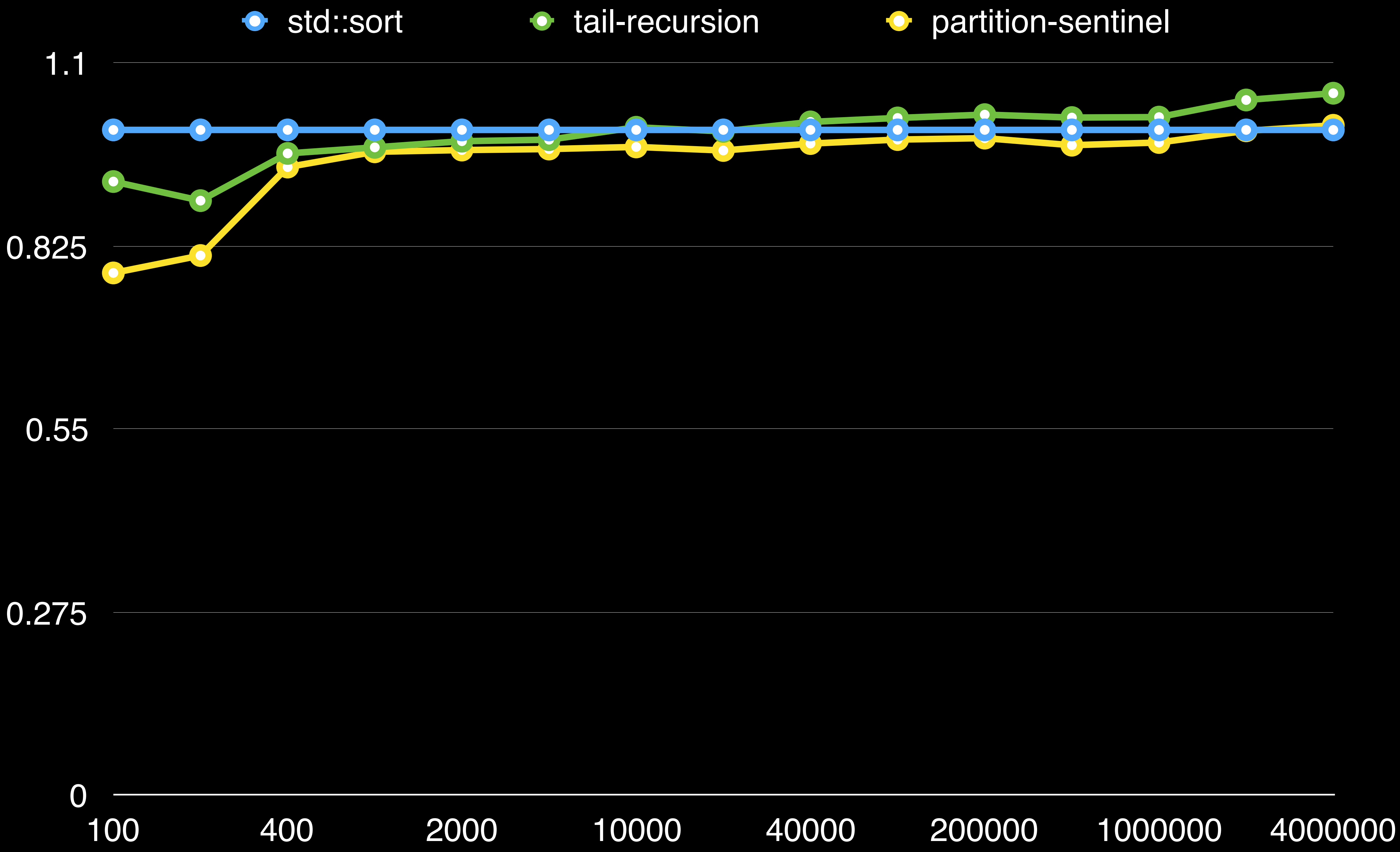
make sure there is a sentinel on the left side, too:

```
int* pbegin = begin, * pend = end;
if (pbegin != pend && !(*pbegin < p)) {
    while (pbegin != pend && !(*--pend < p)) {
    }
    if (pbegin != pend) {
        std::iter_swap(pbegin, pend);
        ++pbegin;
    }
}
```

Sentinel Partition Core

```
if (pbegin != pend) {
    while (true) {
        while (*pbegin < p) ++pbegin;
        while (!(*--pend < p));
        if (pend <= pbegin) break;
        std::iter_swap(pbegin, pend);
        ++pbegin;
    }
 }
mid = pbegin;
```

Results


std::sort
tail-recursion
partition-sentinel
1.1
0.825
0.55
0.275
0
100
400
2000
10000
40000
200000
1000000
4000000

Small Ranges

- many of the final ranges are small
- not worth to kick off an insertion sort
- manually sort ranges with up to 4 elements
- bigger ranges still use normal insertion sort

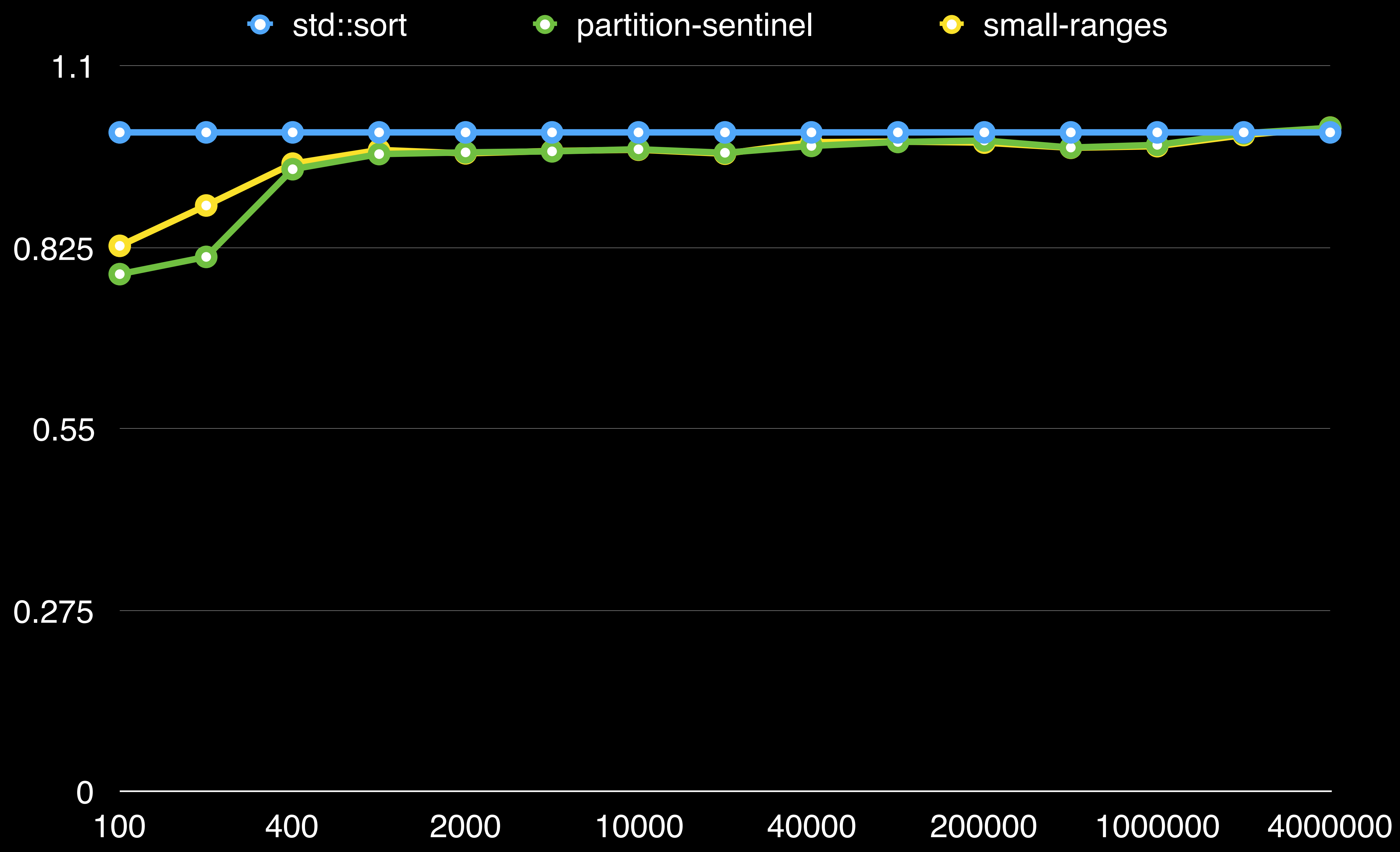
Small Range Dispatch

```
switch (std::distance(b, e)) {
 case 0:
 case 1: return;
 case 2: if (*--e < *b) swap(*b, *e); return;
 case 3: small::sort(b, b + 1, b + 2); return;
 case 4: small::sort(b, b + 1, b + 2, b + 3); return;
 }
```

Sorting Four Elements

```
void small::sort(I p0, I p1, I p2, I p3) {
    sort(p0, p1, p2);
    if (*p3 < *p2) {
        if (*p3 < *p1) {
            auto tmp = *p3; *p3 = *p2; *p2 = *p1;
            if (tmp < *p0) { *p1 = *p0; *p0 = tmp; }
            else *p1 = tmp;
        }
        else swap(p2, p3);
    }
}
```

Results


std::sort
partition-sentinel
small-ranges
1.1
0.825
0.55
0.275
0
100
400
2000
10000
40000
200000
1000000
4000000

Partition Did Nothing

- the partition may not do anything
  - sequence is sorted already
  - notably: all elements are the same
- gamble time on testing if it is sorted
  - … and fix a few local inconsistency

Monitor Swaps

splitting the check may trigger small range optimisation

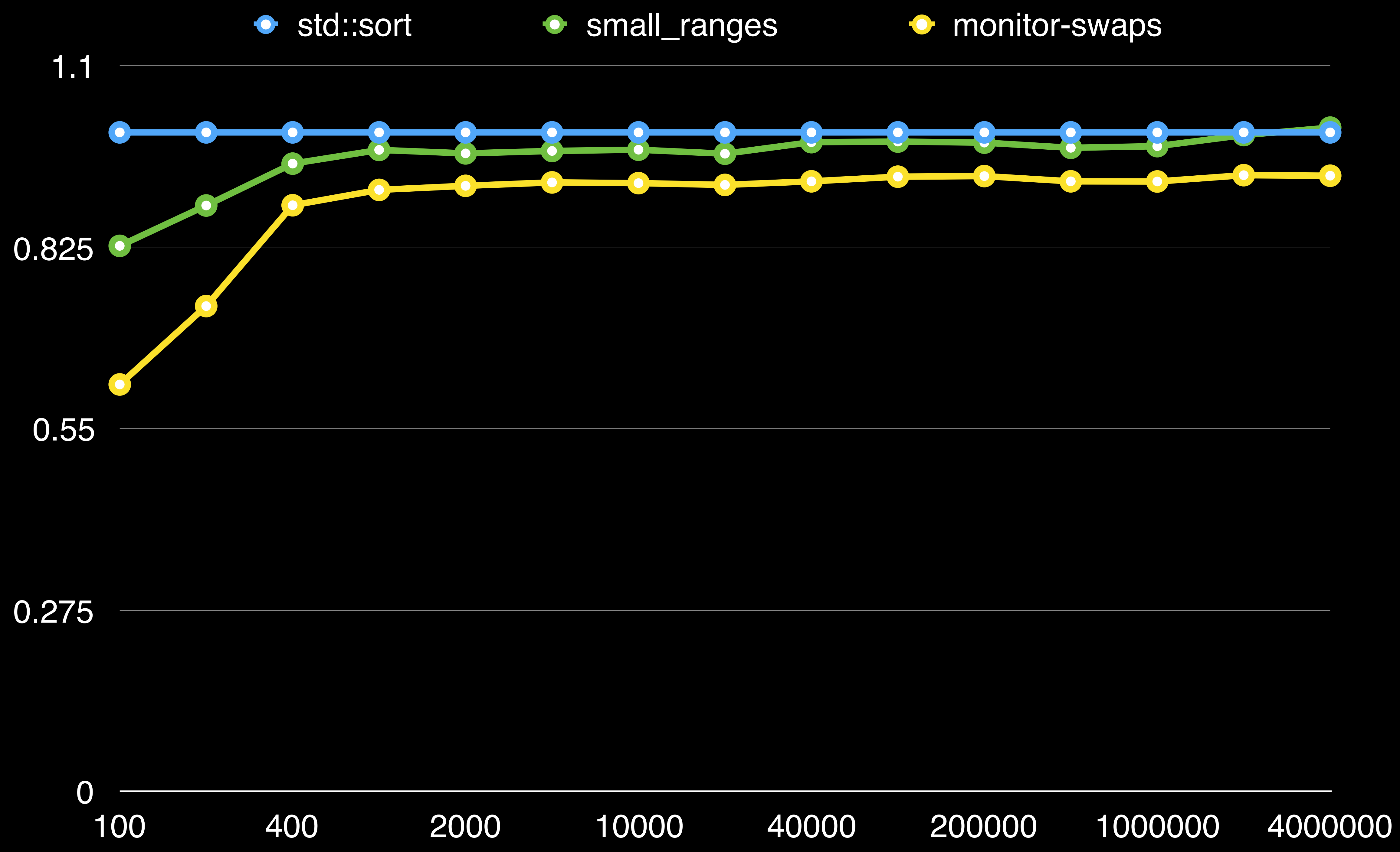
```
unsigned swaps = 0;
… // partition incrementing swaps

if (swaps == 0
    && incomplete_insertion(begin, mid)
    && incomplete_insertion(mid + 1, end)) {
    return;
}
```

Partial Insertion Sort

```
switch (std::distance(b, e)) { ... }
for (int* it = b, cout = 0; ++it != e; )
    if (*it < *(it - 1)) {
        if (++count == limit) return false;
        int* c = it, t = *c; *c = *(c - 1);
        while (b != --c && t < *(c - 1)) {
            *c = *(c - 1);
            if (++count == limit) { *(c - 1) = t; return false; }
        }
        *c = t;
}
return true;
```

Results


std::sort
small_ranges
monitor-swaps
1.1
0.825
0.55
0.275
0
100
400
2000
10000
40000
200000
1000000
4000000

Putting It All Together

```
bool incomplete_insertion(int* begin, int* end) {
    switch (std::distance(begin, end)) {
    case 0:
    case 1:
        return true;
    case 2:
        if (*--end < *begin) std::swap(*begin, *end);
        return true;
    case 3:
        small::sort(begin, begin + 1, begin + 2);
        return true;
    case 4:
        small::sort(begin, begin + 1, begin + 2, begin + 3);
        return true;
    case 5:
        small::sort(begin, begin + 1, begin + 2, begin + 3, begin
+ 4);
        return true;
    }
    constexpr int limit = 8;
    int  count(0);
    for (int* it = begin; ++it != end; )
        if (*it < *(it - 1)) {
            if (++count == limit) return false;
            int* c = it, t = *c;
            *c = *(c - 1);
            while (begin != --c && t < *(c - 1)) {
                *c = *(c - 1);
                if (++count == limit) {
                    *(c - 1) = t;
                    return false;
                }
            }
            *c = t;
        }
    return true;
}
void sort(int* begin, int* end, int depth, int max) {
    std::size_t len;
    while (20 < (len = std::distance(begin, end))) {
        if (++depth < max) {
            int* pivot = end - 1, * mid = begin + len / 2;
```

```
std::swap(*mid, *pivot);

unsigned swaps = 0;
int p = *pivot;
auto pred = [=](int arg){ return arg < p; };
int* pbegin = begin, * pend = end;
if (pbegin != pend && !pred(*pbegin)) {
    while (pbegin != pend && !pred(*--pend)) {
    }
    if (pbegin != pend) {
        std::iter_swap(pbegin, pend);
        ++swaps;
        ++pbegin;
    }
}
if (pbegin != pend) {
    while (true) {
        while (pred(*pbegin)) {
            ++pbegin;
        }
        while (!pred(*--pend)) {
        }
        if (pend <= pbegin) {
            break;
        }
        std::iter_swap(pbegin, pend);
        ++pbegin;
        ++swaps;
    }
}
mid = pbegin;
std::swap(*mid, *pivot);

if (swaps == 0
    && incomplete_insertion(begin, mid)
    && incomplete_insertion(mid + 1, end)) {
    return;
}
if (mid - begin < end - mid) {
    sort(begin, mid, depth, max);
    begin = mid + 1;
```

```
            }
            else {
                sort(mid + 1, end, depth, max);
                end = mid;
            }
        }
        else {
            std::stable_sort(begin, end);
            return;
        }
    }
    switch (std::distance(begin, end)) {
    case 0:
    case 1:
        return;
    case 2:
        if (*--end < *begin) std::swap(*begin, *end);
        return;
    case 3:
        small::sort(begin, begin + 1, begin + 2);
        return;
    case 4:
        small::sort(begin, begin + 1, begin + 2, begin + 3);
        return;
    }
    for (int* it = begin; ++it != end; )
        if (*it < *(it - 1)) {
            int* c = it, t = *c;
            *c = *(c - 1);
            while (begin != --c && t < *(c - 1))
                *c = *(c - 1);
            *c = t;
        }
    return;
}
void sort(int* begin, int* end) {
    std::size_t size(std::distance(begin, end)), max(0);
    while (size >>=1) { ++max; }
    sort(begin, end, 0, 2 * max);
}
```

Putting It All Together

```
bool incomplete_insertion(int* begin, int* end) {
    switch (std::distance(begin, end)) {
    case 0:
    case 1:
        return true;
    case 2:
        if (*--end < *begin) std::swap(*begin, *end);
        return true;
    case 3:
        small::sort(begin, begin + 1, begin + 2);
        return true;
    case 4:
        small::sort(begin, begin + 1, begin + 2, begin + 3);
        return true;
    case 5:
        small::sort(begin, begin + 1, begin + 2, begin + 3, begin
+ 4);
        return true;
    }
    constexpr int limit = 8;
    int  count(0);
    for (int* it = begin; ++it != end; )
        if (*it < *(it - 1)) {
            if (++count == limit) return false;
            int* c = it, t = *c;
            *c = *(c - 1);
            while (begin != --c && t < *(c - 1)) {
                *c = *(c - 1);
                if (++count == limit) {
                    *(c - 1) = t;
                    return false;
                }
            }
            *c = t;
        }
    return true;
}
void sort(int* begin, int* end, int depth, int max) {
    std::size_t len;
    while (20 < (len = std::distance(begin, end))) {
        if (++depth < max) {
            int* pivot = end - 1, * mid = begin + len / 2;
```

```
std::swap(*mid, *pivot);

unsigned swaps = 0;
int p = *pivot;
auto pred = [=](int arg){ return arg < p; };
int* pbegin = begin, * pend = end;
if (pbegin != pend && !pred(*pbegin)) {
    while (pbegin != pend && !pred(*--pend)) {
    }
    if (pbegin != pend) {
        std::iter_swap(pbegin, pend);
        ++swaps;
        ++pbegin;
    }
}
if (pbegin != pend) {
    while (true) {
        while (pred(*pbegin)) {
            ++pbegin;
        }
        while (!pred(*--pend)) {
        }
        if (pend <= pbegin) {
            break;
        }
        std::iter_swap(pbegin, pend);
        ++pbegin;
        ++swaps;
    }
}
mid = pbegin;
std::swap(*mid, *pivot);

if (swaps == 0
    && incomplete_insertion(begin, mid)
    && incomplete_insertion(mid + 1, end)) {
    return;
}
if (mid - begin < end - mid) {
    sort(begin, mid, depth, max);
    begin = mid + 1;
```

```
            }
            else {
                sort(mid + 1, end, depth, max);
                end = mid;
            }
        }
        else {
            std::stable_sort(begin, end);
            return;
        }
    }
    switch (std::distance(begin, end)) {
    case 0:
    case 1:
        return;
    case 2:
        if (*--end < *begin) std::swap(*begin, *end);
        return;
    case 3:
        small::sort(begin, begin + 1, begin + 2);
        return;
    case 4:
        small::sort(begin, begin + 1, begin + 2, begin + 3);
        return;
    }
    for (int* it = begin; ++it != end; )
        if (*it < *(it - 1)) {
            int* c = it, t = *c;
            *c = *(c - 1);
            while (begin != --c && t < *(c - 1))
                *c = *(c - 1);
            *c = t;
        }
    return;
}
void sort(int* begin, int* end) {
    std::size_t size(std::distance(begin, end)), max(0);
    while (size >>=1) { ++max; }
    sort(begin, end, 0, 2 * max);
}
```

Simple Implementation

```
void sort(int* begin, int* end) {
    std::size_t len = std::distance(begin, end);
    if (len <= 1) return;
    int* pivot = end - 1;
    int* mid = std::partition(begin, pivot,
                    [=](int value){ return value < *pivot; });
    swap(*mid, *pivot);
    sort(begin, mid);
    sort(mid + 1, end);
}
```

More Generic

```
void sort(int* begin, int* end) {
    std::size_t len = std::distance(begin, end);
    if (len <= 1) return;
    int* pivot = end - 1;
    int* mid = std::partition(begin, pivot,
                    [=](int value){ return value < *pivot; });
    swap(*mid, *pivot);
    sort(begin, mid);
    sort(mid + 1, end);
}
```

More Generic

```
template <typename T>
void sort(T * begin, T * end) {
    std::size_t len = std::distance(begin, end);
    if (len <= 1) return;
    T * pivot = end - 1;
    T * mid = std::partition(begin, pivot,
                        [=](T  value){ return value < *pivot; });
    swap(*mid, *pivot);
    sort(begin, mid);
    sort(mid + 1, end);
}
```

More Generic

```
template <typename Iter>
void sort(Iter begin, Iter end) {
    std::size_t len = std::distance(begin, end);
    if (len <= 1) return;
    Iter pivot = end - 1;
    Iter mid = std::partition(begin, pivot,
                [=](auto&& value){ return value < *pivot; });
    swap(*mid, *pivot);
    sort(begin, mid);
    sort(mid + 1, end);
}
```

More Generic

```
template <typename Iter>
void sort(Iter begin, Iter end) {
    std::size_t len = std::distance(begin, end);
    if (len <= 1) return;
    Iter pivot = end - 1;
    Iter mid = std::partition(begin, pivot,
                  [=](auto&& value){ return value < *pivot; });
    swap(*mid, *pivot);
    sort(begin, mid);
    sort(mid + 1, end);
}
```

More Generic

```
template <typename Iter, typename Comp>
void sort(Iter begin, Iter end, Comp c) {
    std::size_t len = std::distance(begin, end);
    if (len <= 1) return;
    Iter pivot = end - 1;
    Iter mid = std::partition(begin, pivot,
                  [=](auto&& value){ return c(value, *pivot); });
    swap(*mid, *pivot);
    sort(begin, mid, c);
    sort(mid + 1, end, c);
}
```

Summary

- quick sort is fast
  - when pulling a lot of tricks
  - for the expected cases
  - it is a hybrid using multiple algorithms
- => real-world algorithms need a bit of tuning

Questions?