

# Parallel Partition

Dietmar Kühl  
API Market Data London  
Bloomberg LP

# Partition

- input: a range and unary predicate
- output: the range with rearranged elements:  
all elements where the predicate is true come first

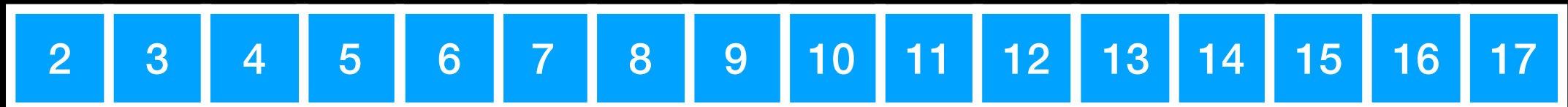
# Partition

- input: a range and unary predicate
- output: the range with rearranged elements:  
all elements where the predicate is true come first

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----

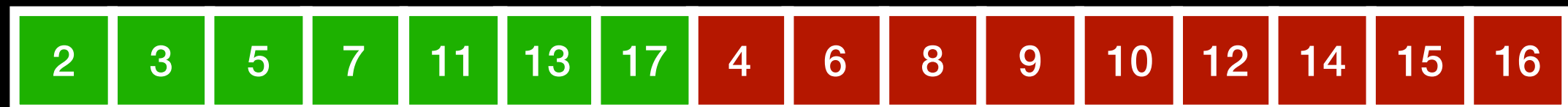
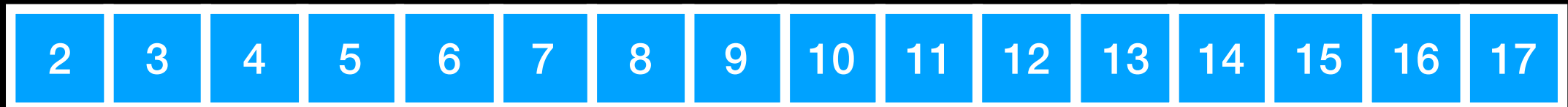
# Partition

- input: a range and unary predicate
- output: the range with rearranged elements:  
all elements where the predicate is true come first



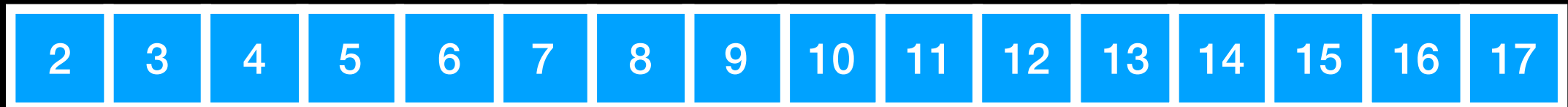
# Partition

- input: a range and unary predicate
- output: the range with rearranged elements:  
all elements where the predicate is true come first



# Partition

- input: a range and unary predicate
- output: the range with rearranged elements:  
all elements where the predicate is true come first



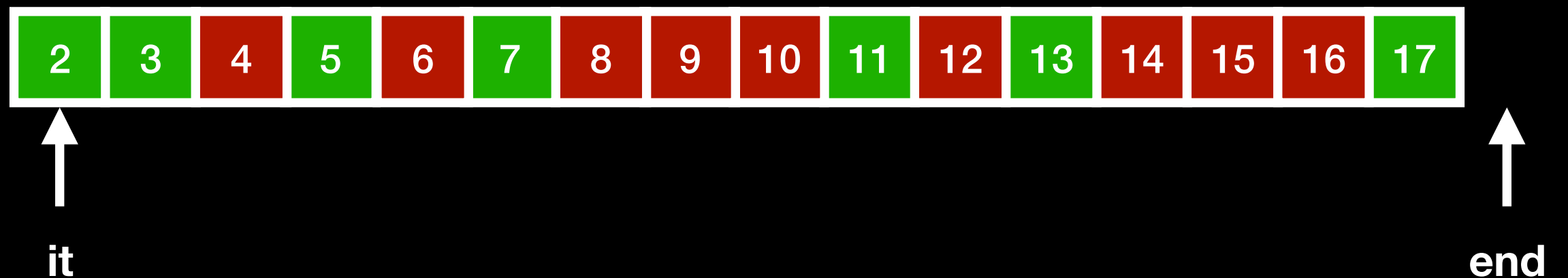
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----

# Lomuto's Partition

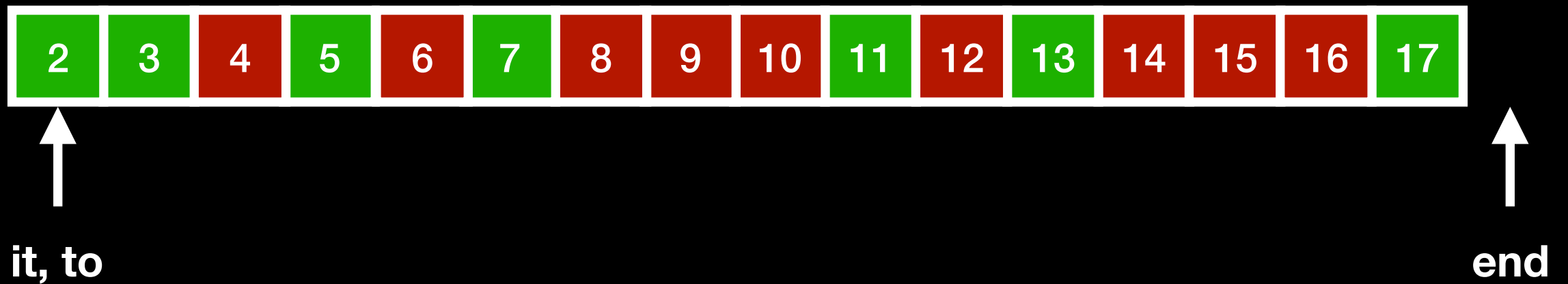
```
template <typename FwdIt, typename Predicate>  
> FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {  
    FwdIt to(it);  
    for (; it != end; ++it)  
        if (predicate(*it))  
            std::iter_swap(to++, it);  
    return to;  
}
```





# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    > for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



↑  
it, to

↑  
end

# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        > if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```

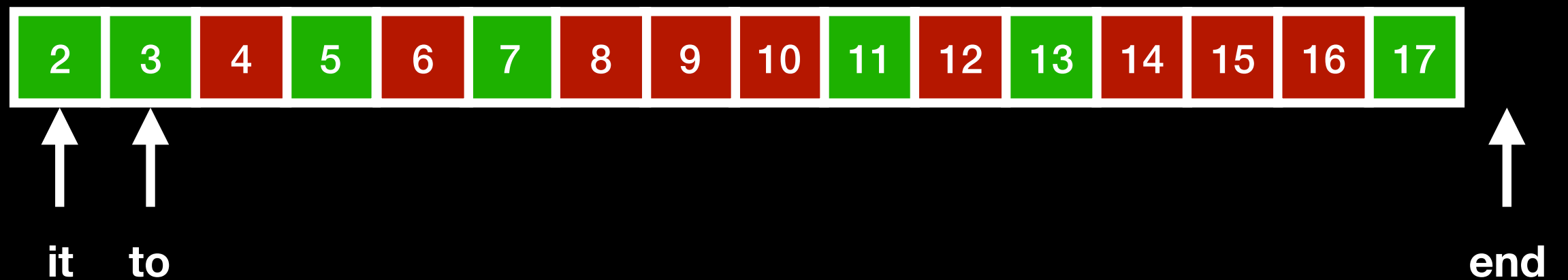


it, to

end

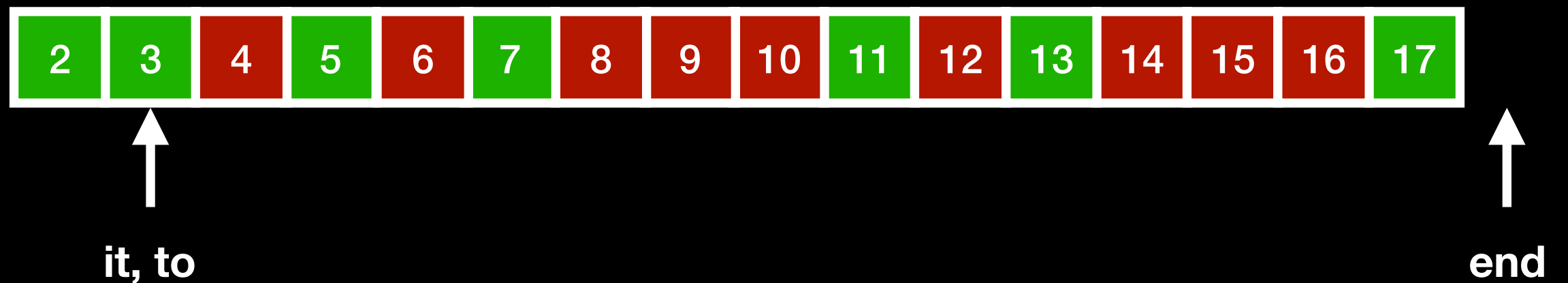
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        if (predicate(*it))
            > std::iter_swap(to++, it);
    return to;
}
```



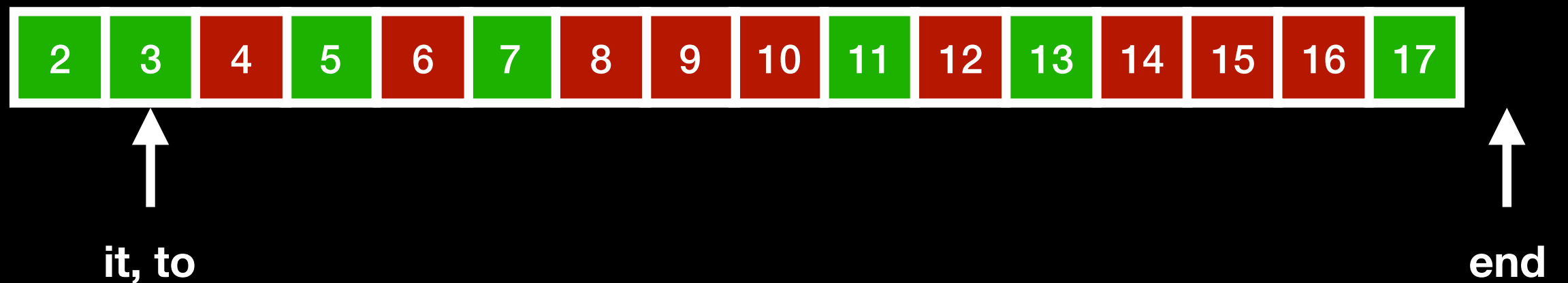
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    > for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



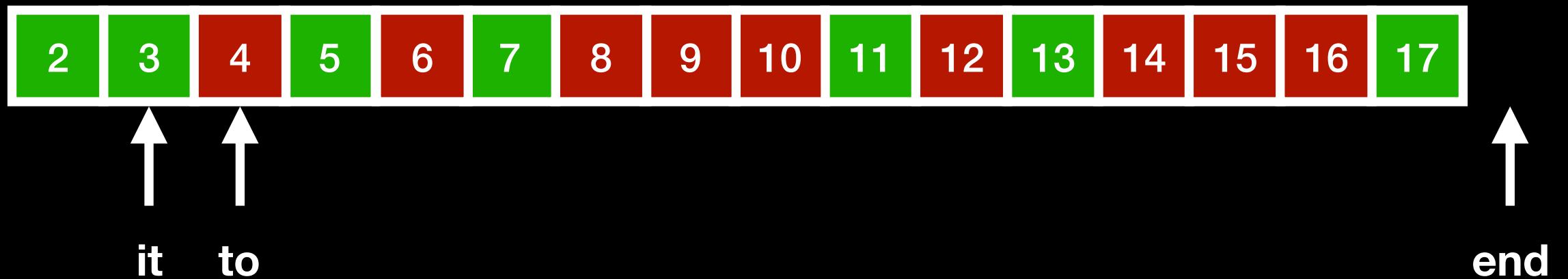
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        > if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



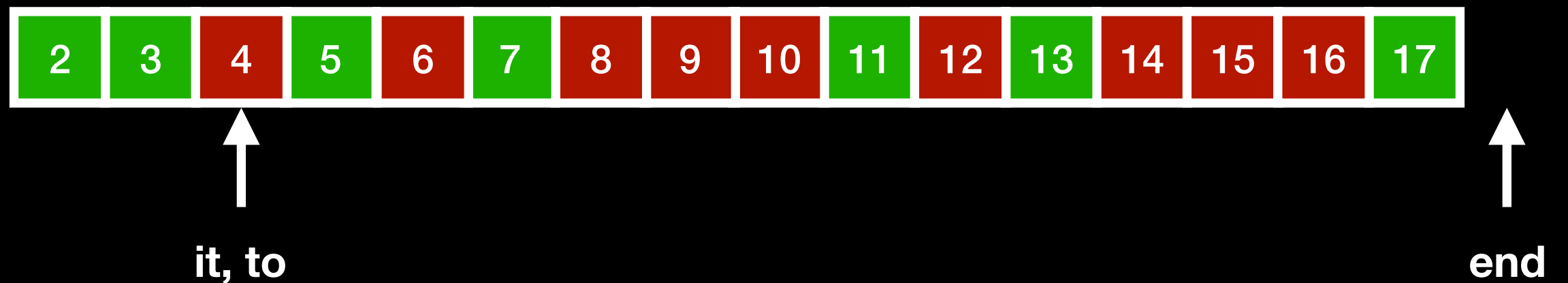
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



# Lomuto's Partition

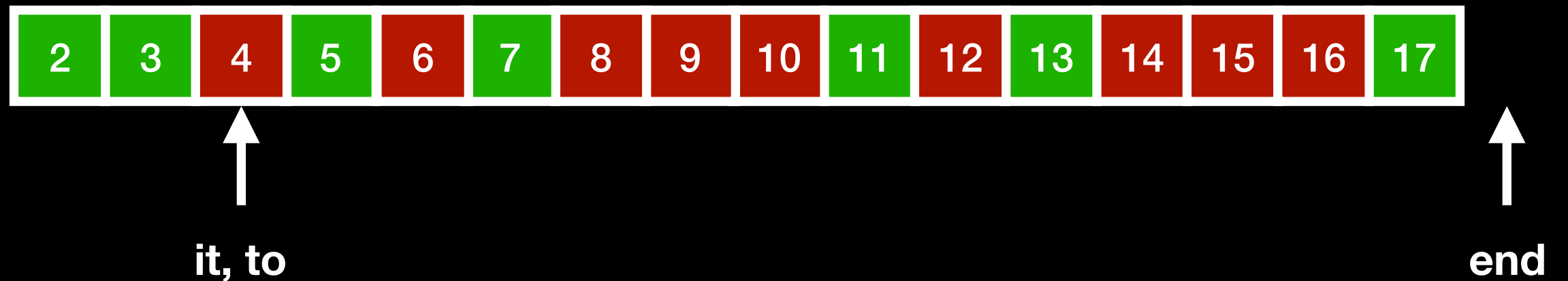
```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    > for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```





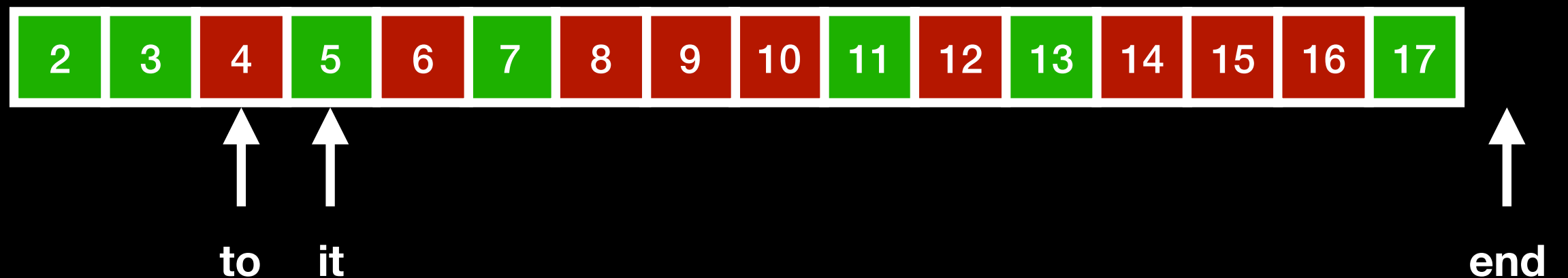
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        > if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



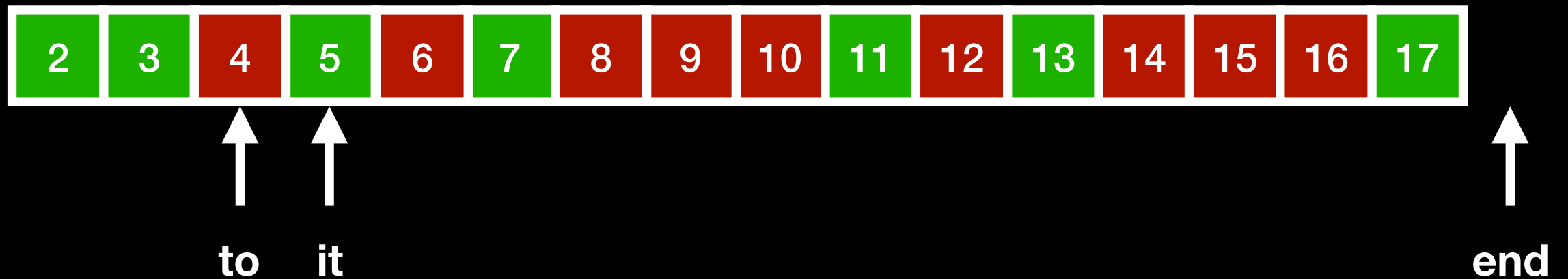
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    > for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



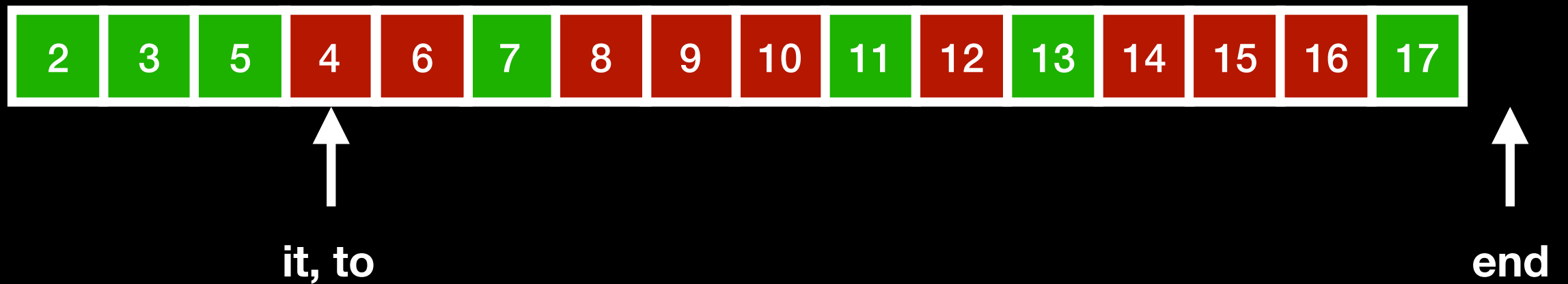
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



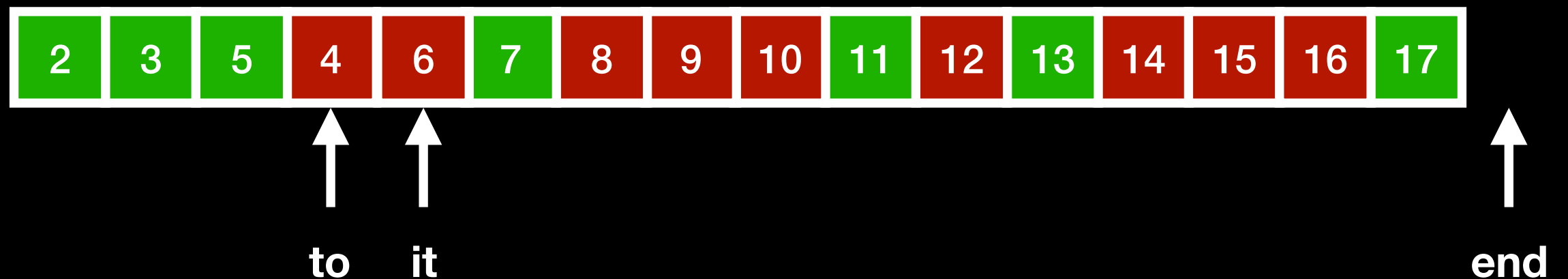
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



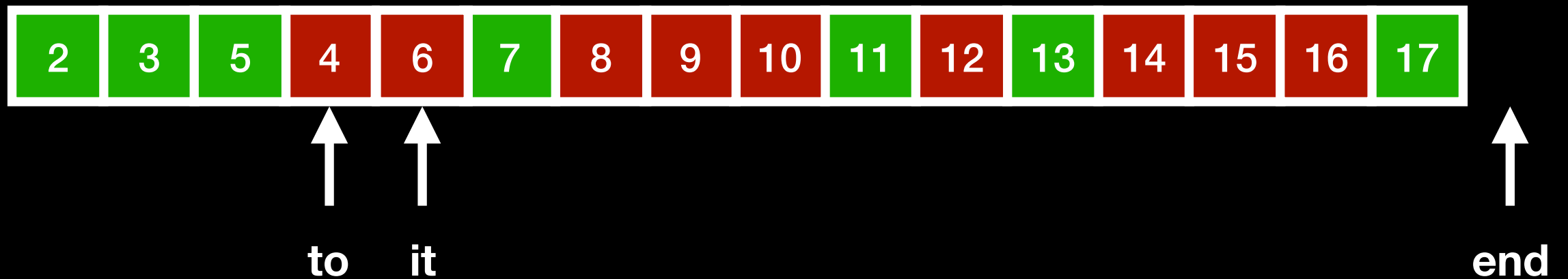
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    > for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



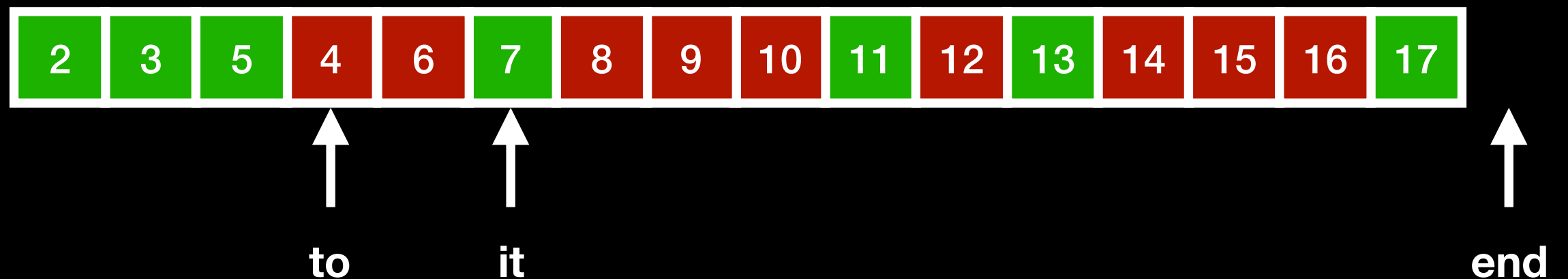
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



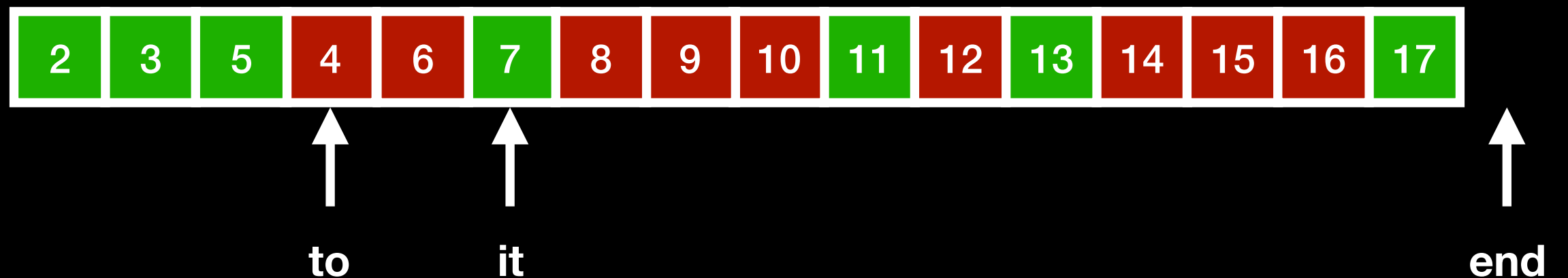
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    > for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



# Lomuto's Partition

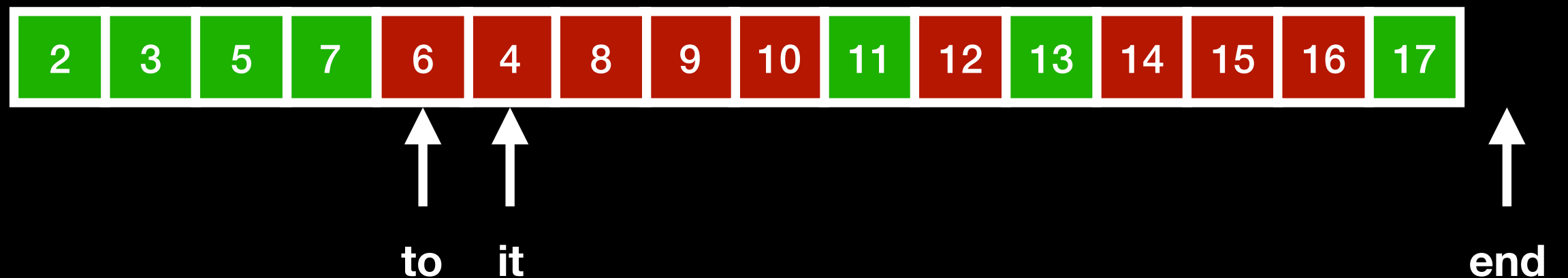
```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        > if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```





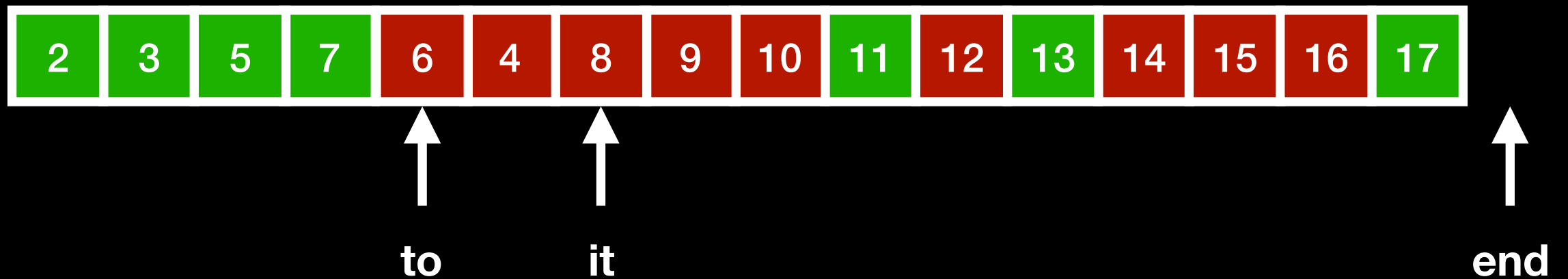
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        if (predicate(*it))
            > std::iter_swap(to++, it);
    return to;
}
```



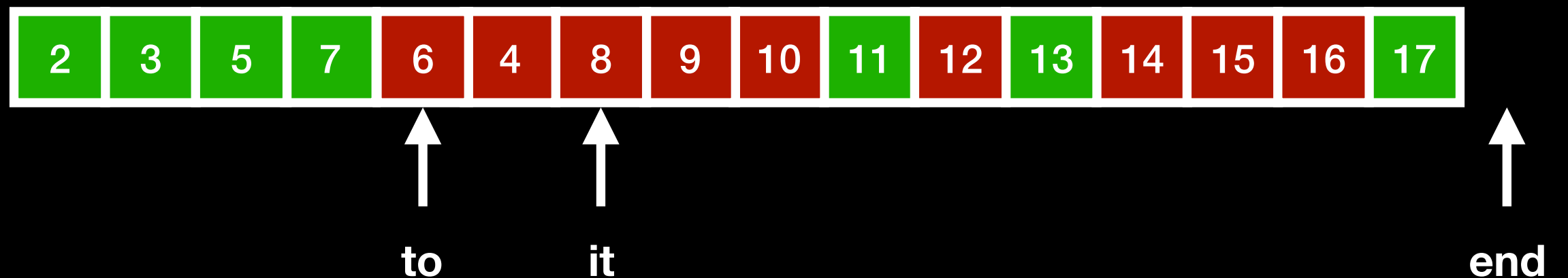
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    > for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



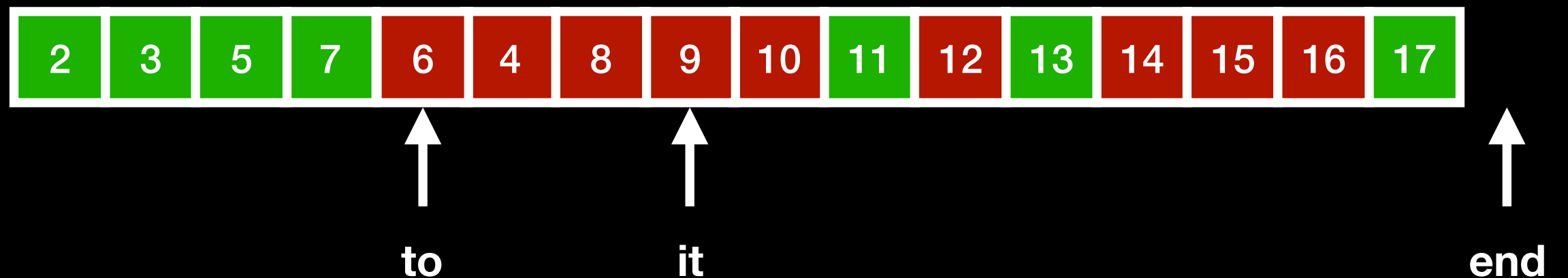
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        > if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



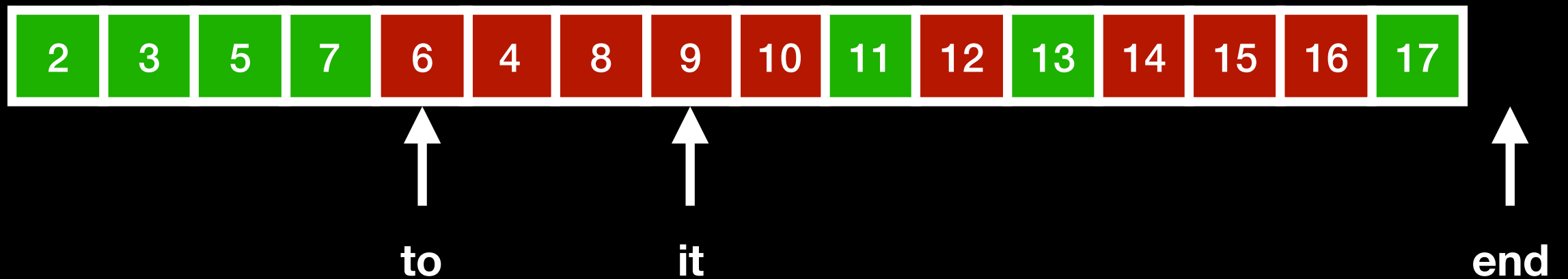
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    > for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



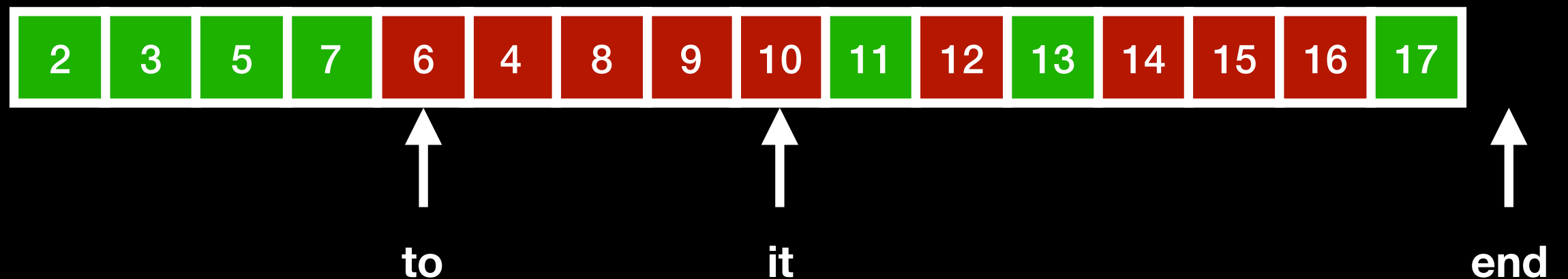
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



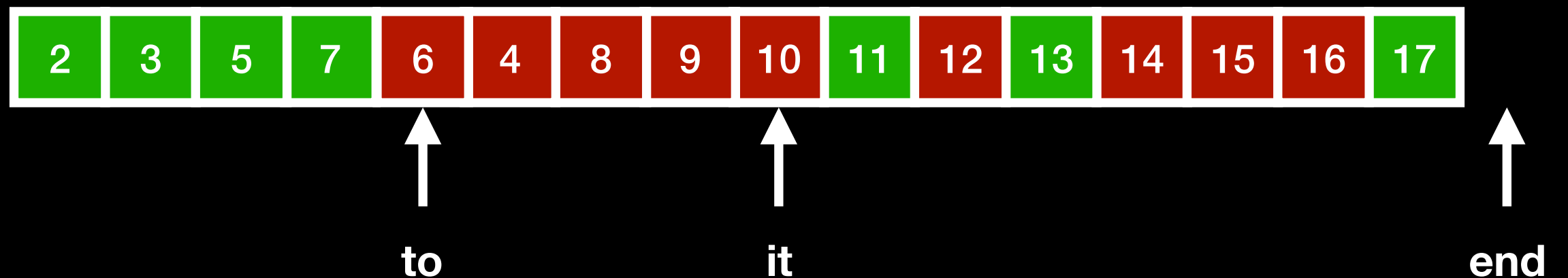
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    > for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



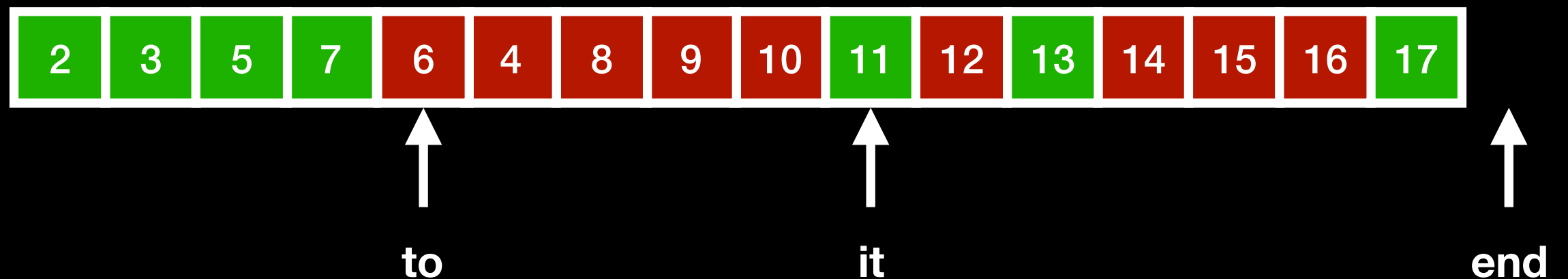
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        > if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



# Lomuto's Partition

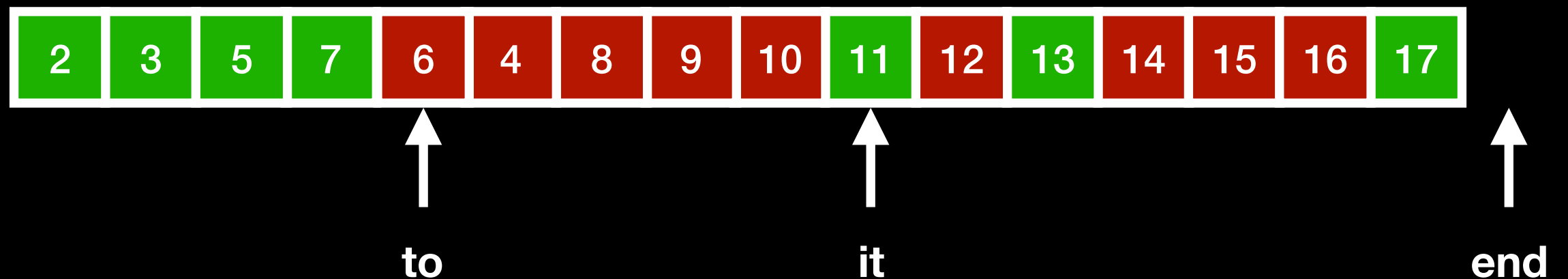
```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    > for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```





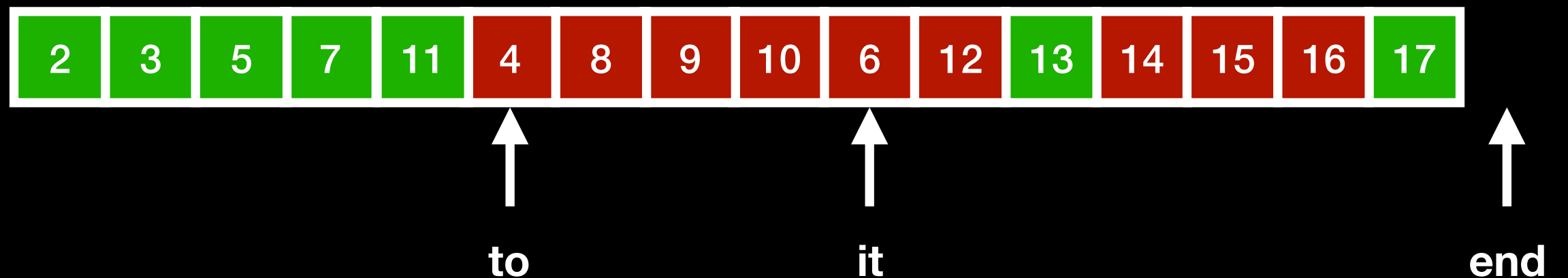
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        > if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



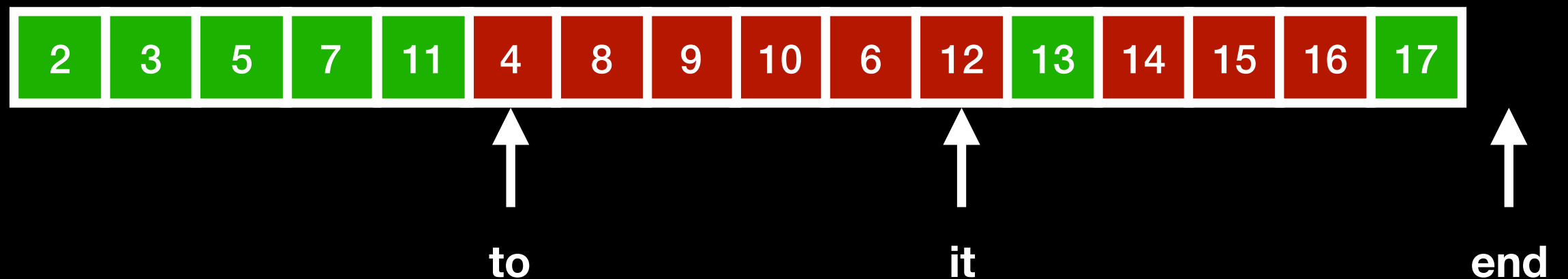
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        if (predicate(*it))
            > std::iter_swap(to++, it);
    return to;
}
```



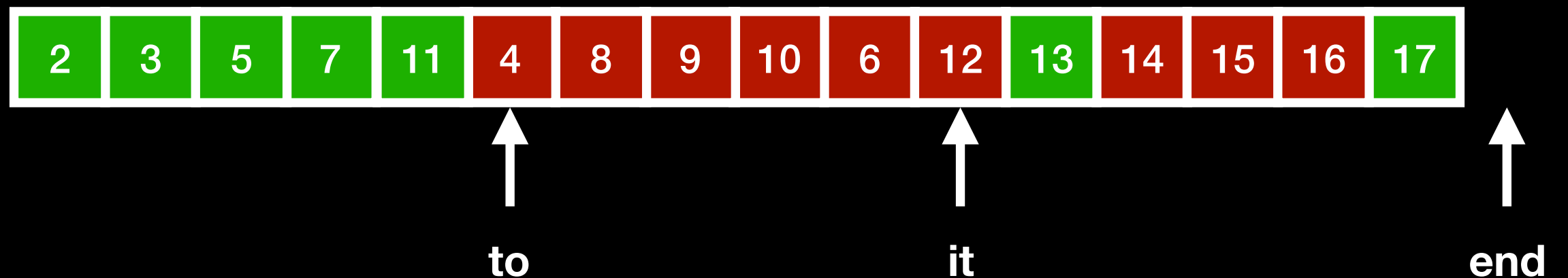
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    > for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



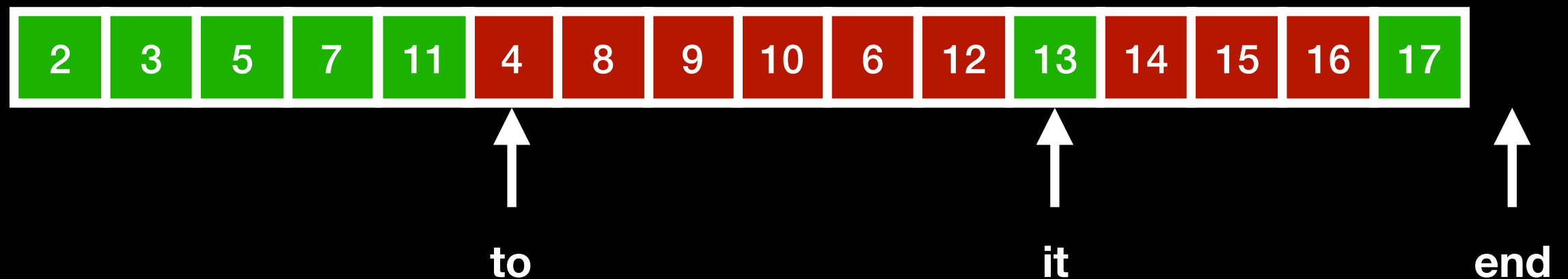
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        > if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



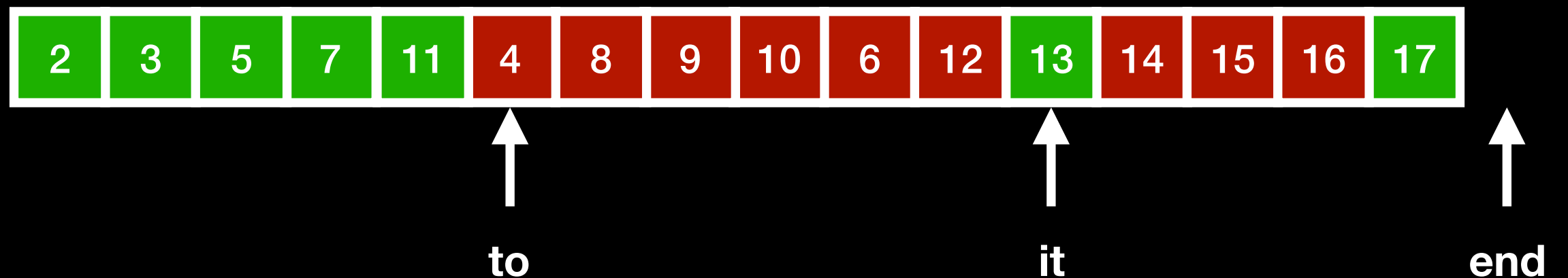
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    > for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



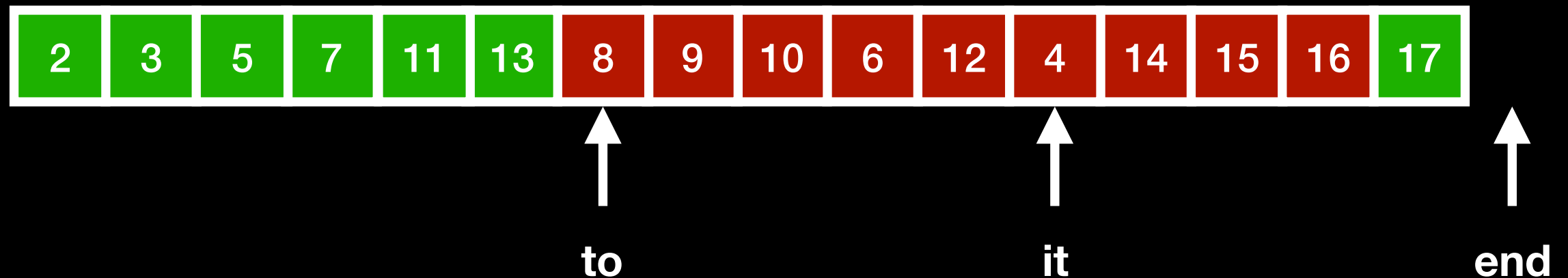
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        > if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



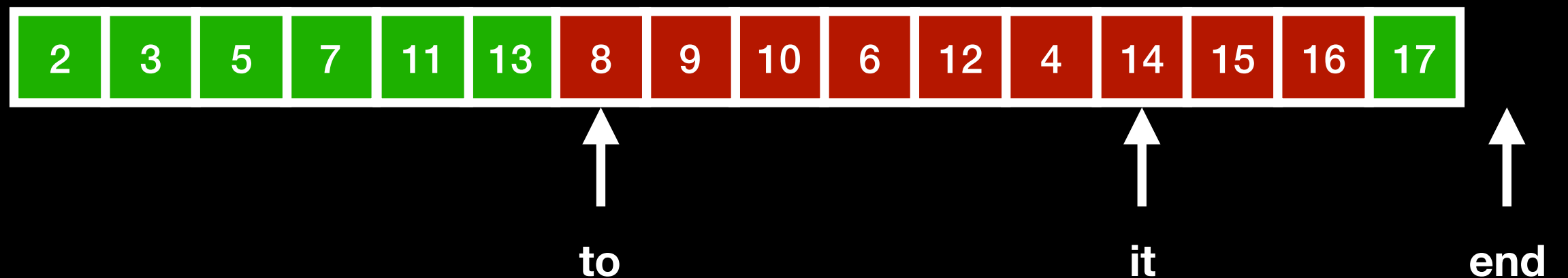
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



# Lomuto's Partition

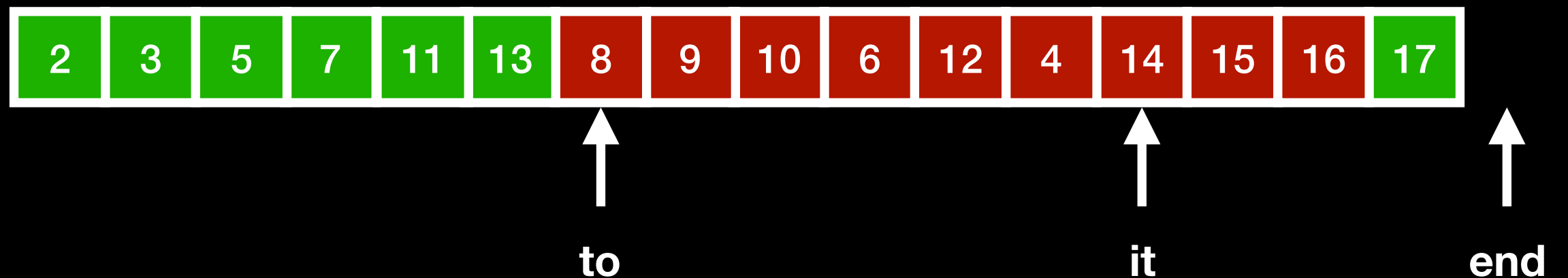
```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    > for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```





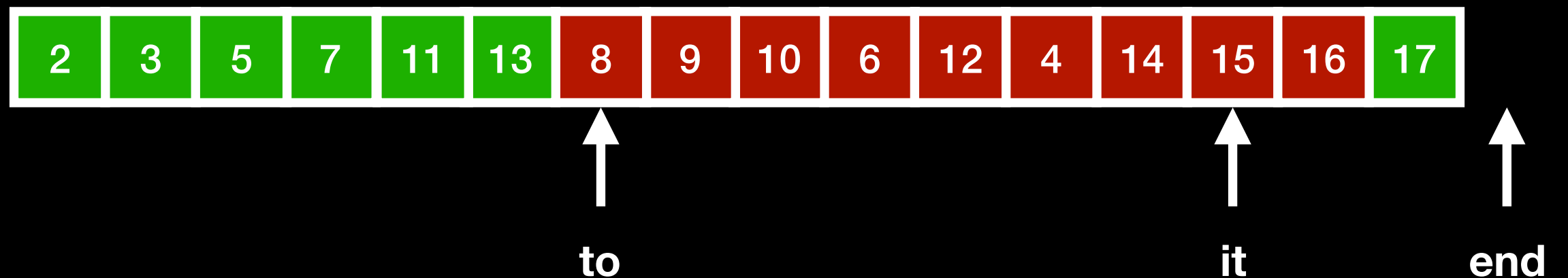
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        > if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



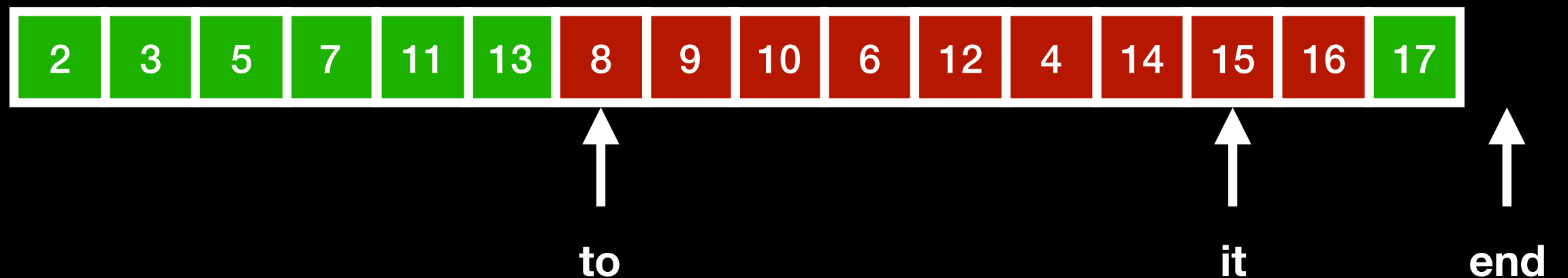
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    > for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



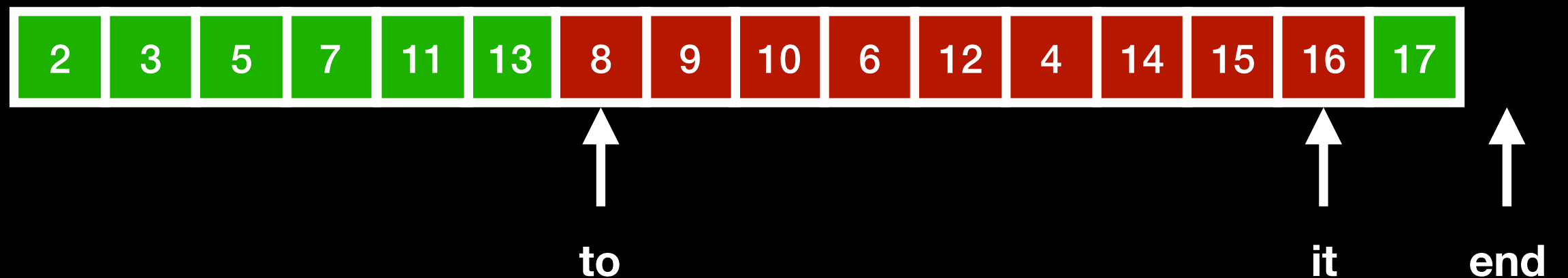
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        > if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



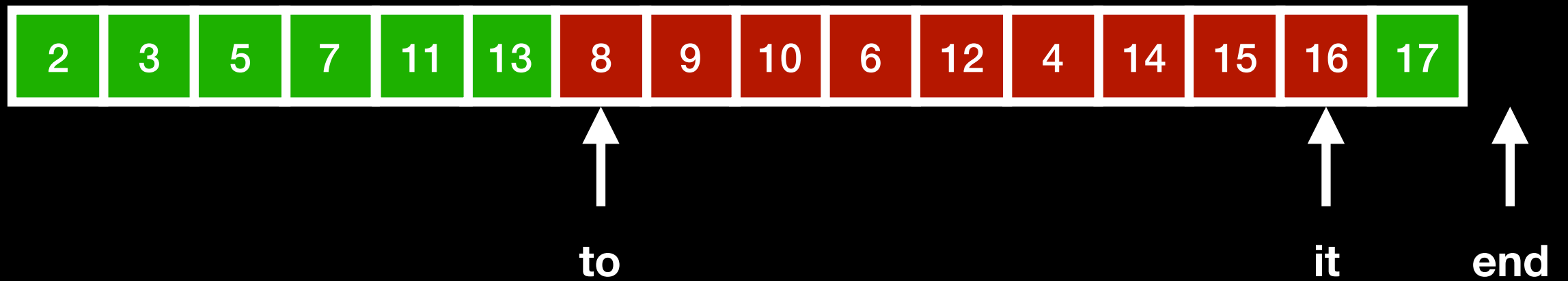
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    > for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



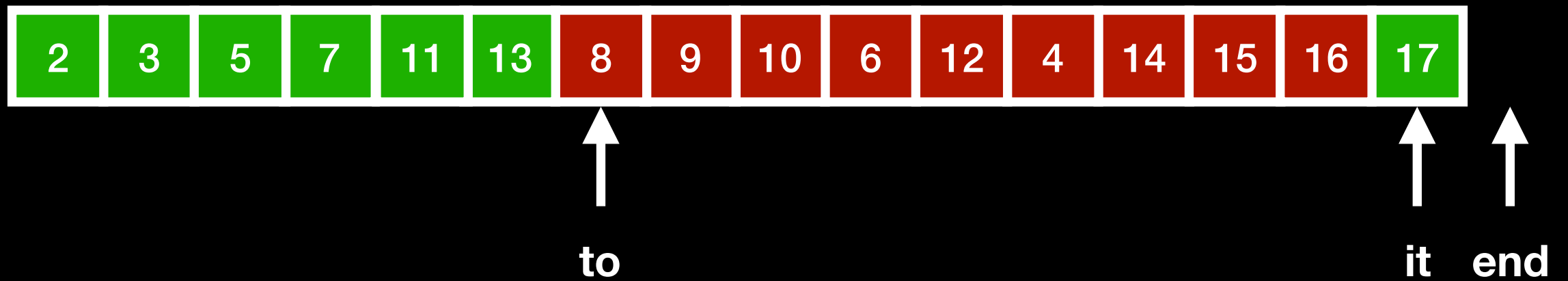
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



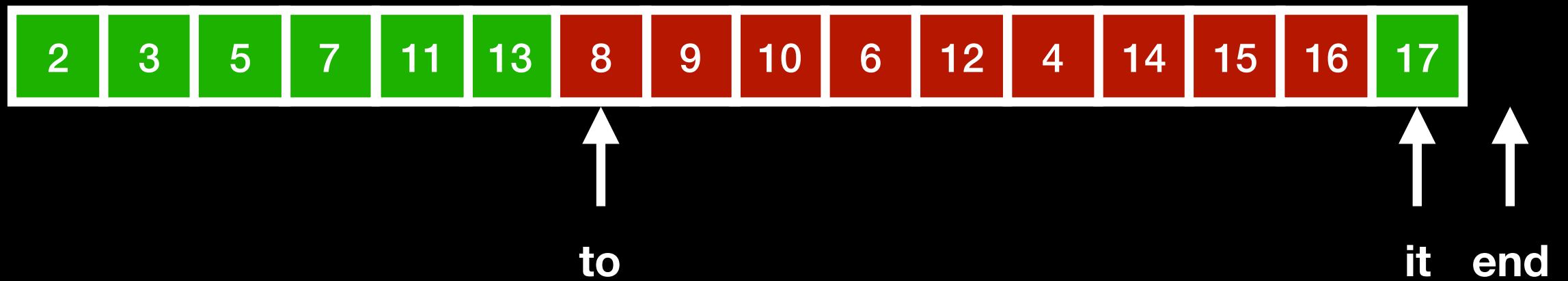
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
>   for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



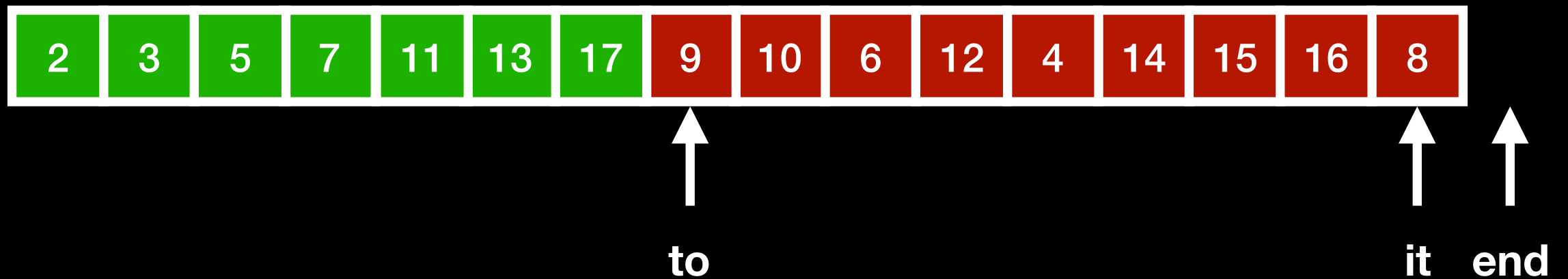
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



# Lomuto's Partition

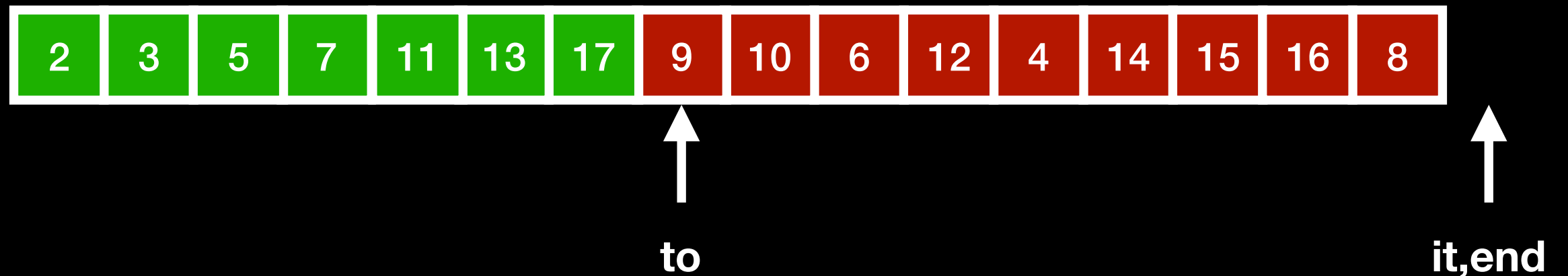
```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```





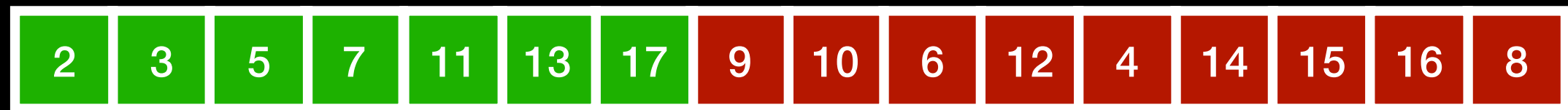
# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    > for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    > return to;
}
```

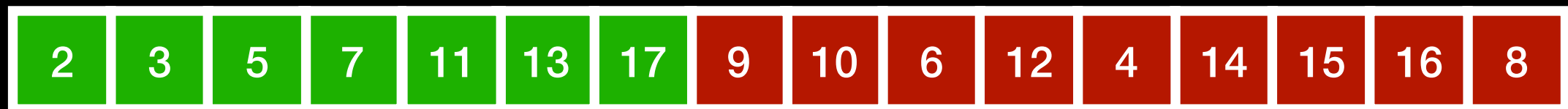


to

it,end

# Lomuto's Partition

```
template <typename FwdIt, typename Predicate>
FwdIt lomuto(FwdIt it, FwdIt end, Predicate predicate) {
    FwdIt to(it);
    for (; it != end; ++it)
        if (predicate(*it))
            std::iter_swap(to++, it);
    return to;
}
```



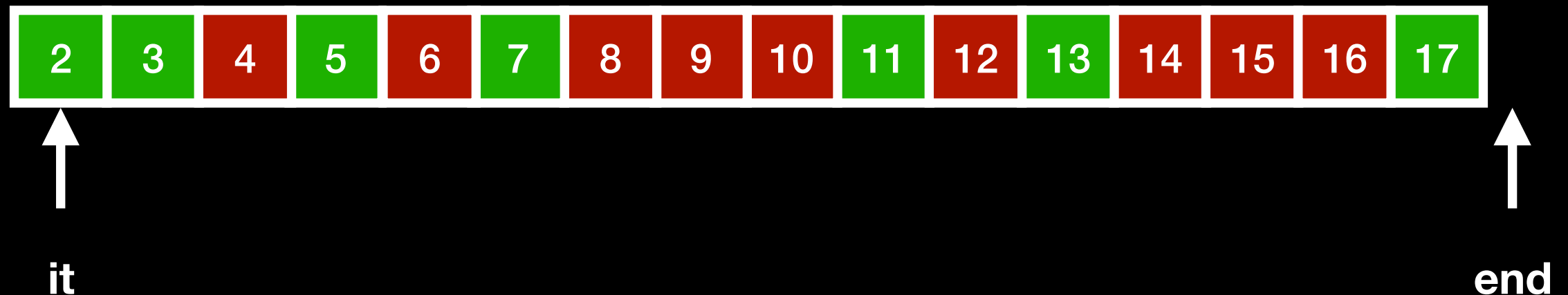
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----

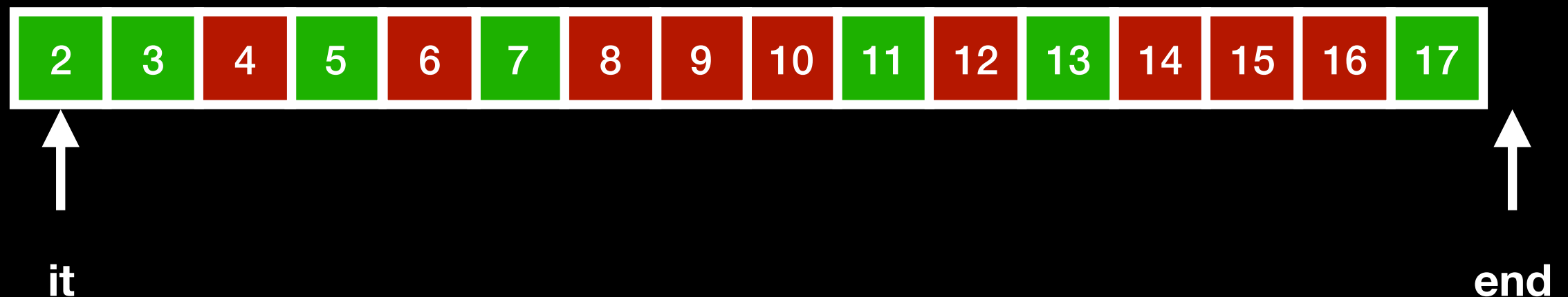
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
>   while (true) {
       while (it != end && predicate(*it)) { ++it; }
       while (it != end && !predicate(*--end)) {}
       if (it == end) { return it; }
       std::iter_swap(it++, end);
   }
}
```



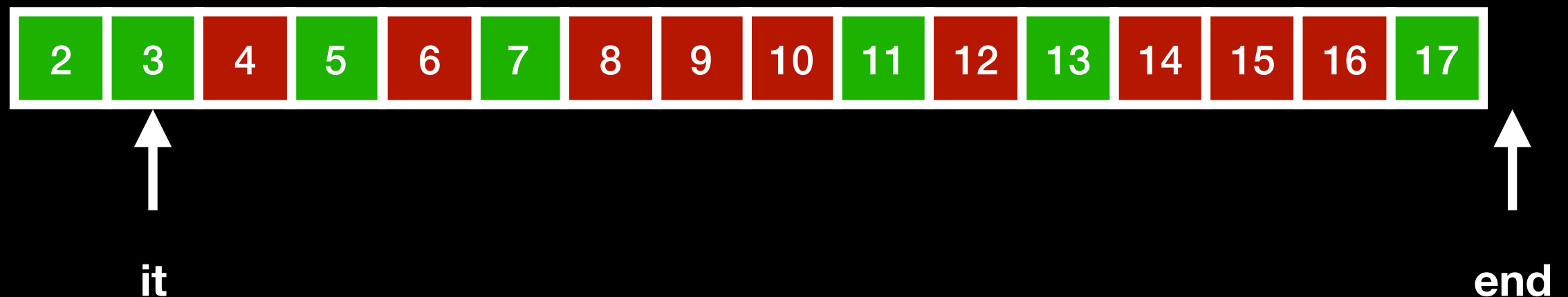
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        > while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



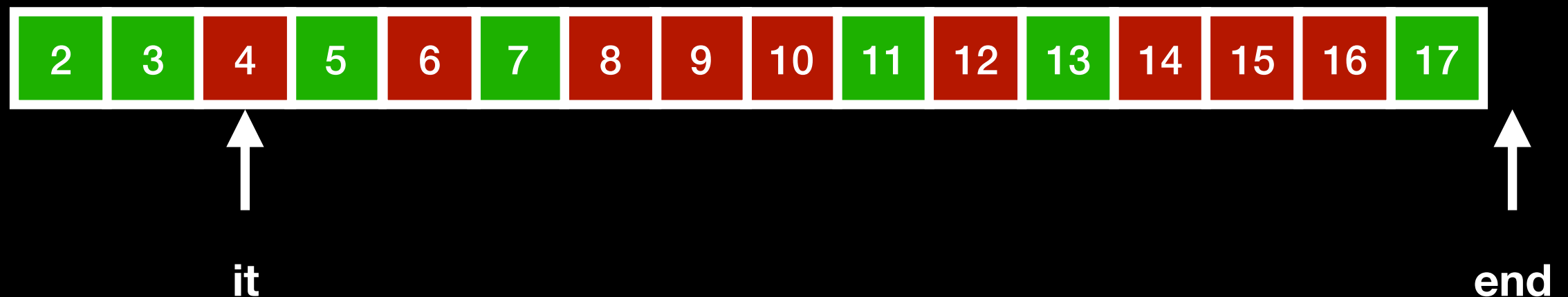
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        > while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



# Hoare's Partition

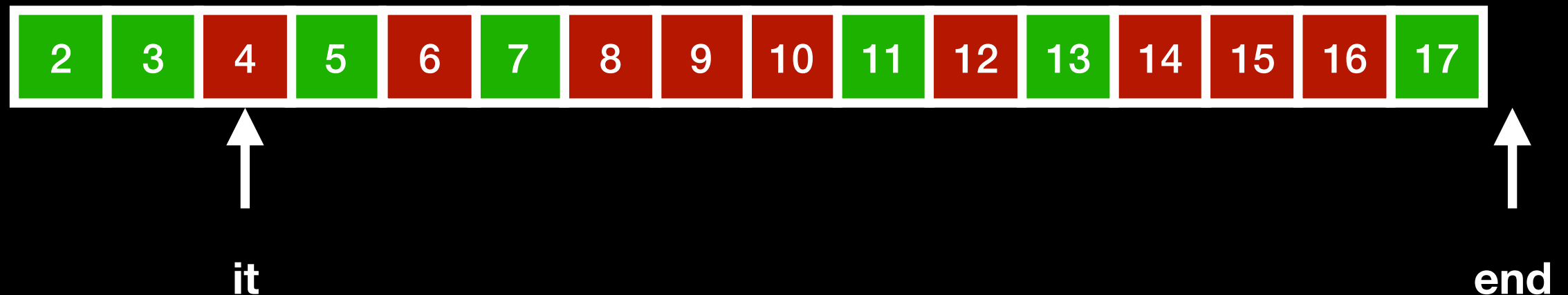
```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        > while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```





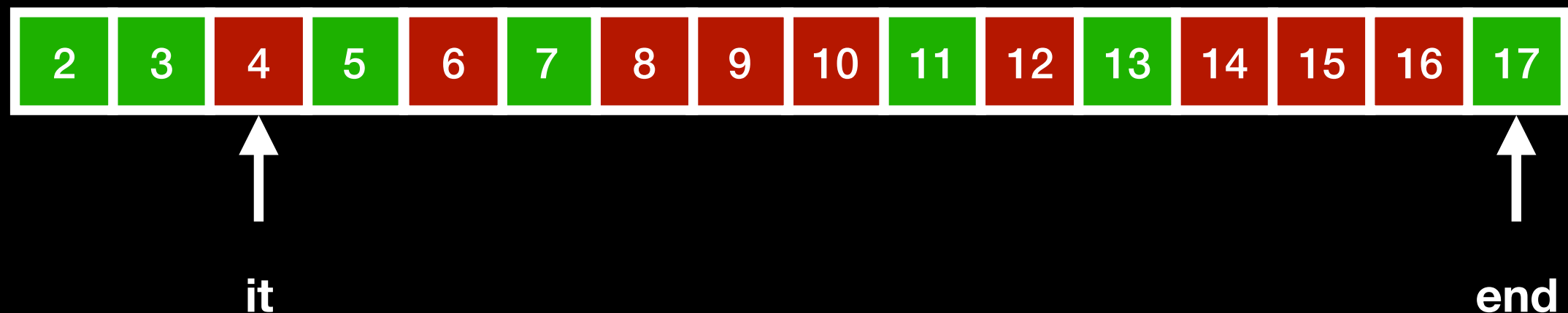
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        > while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



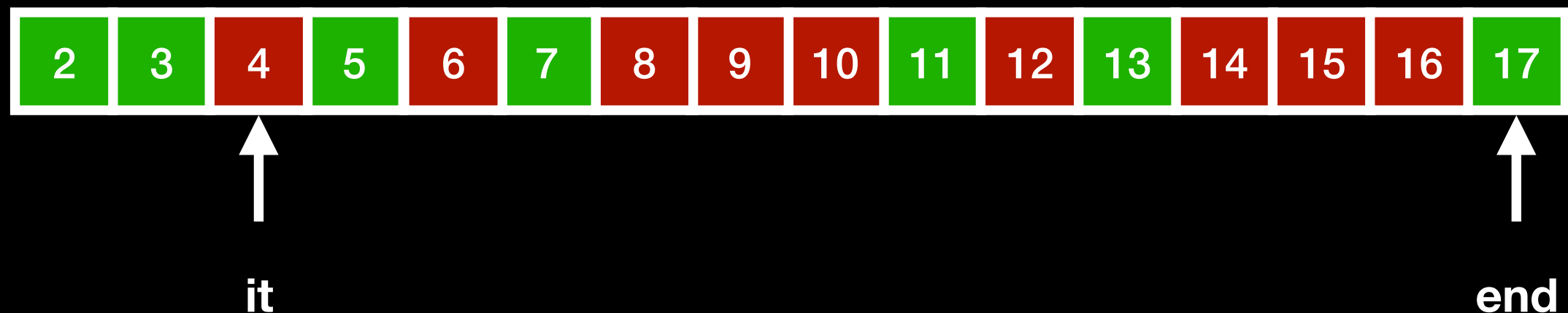
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
>       while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



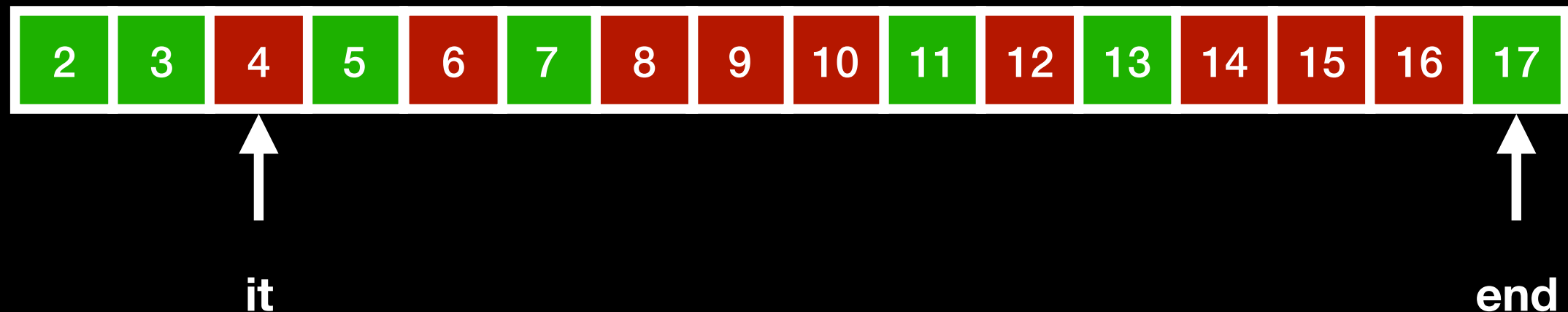
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
>    if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



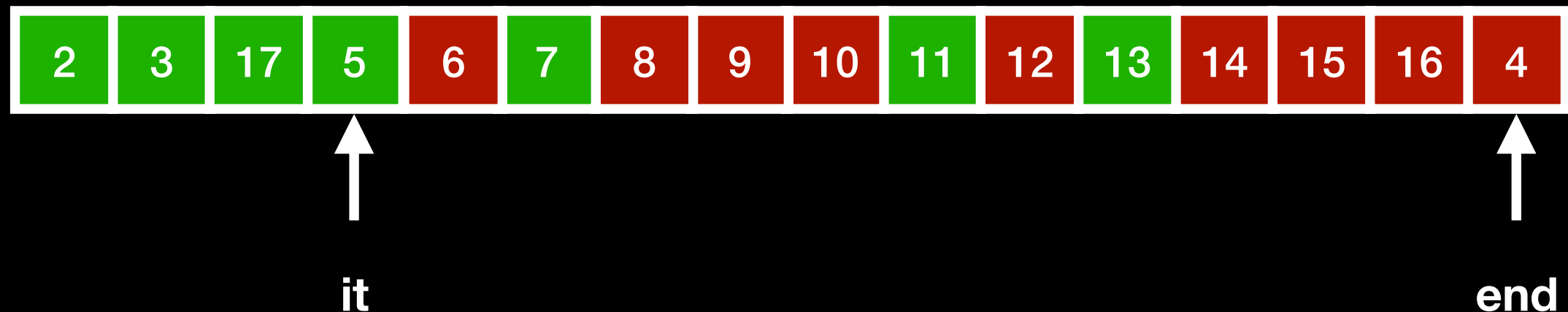
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



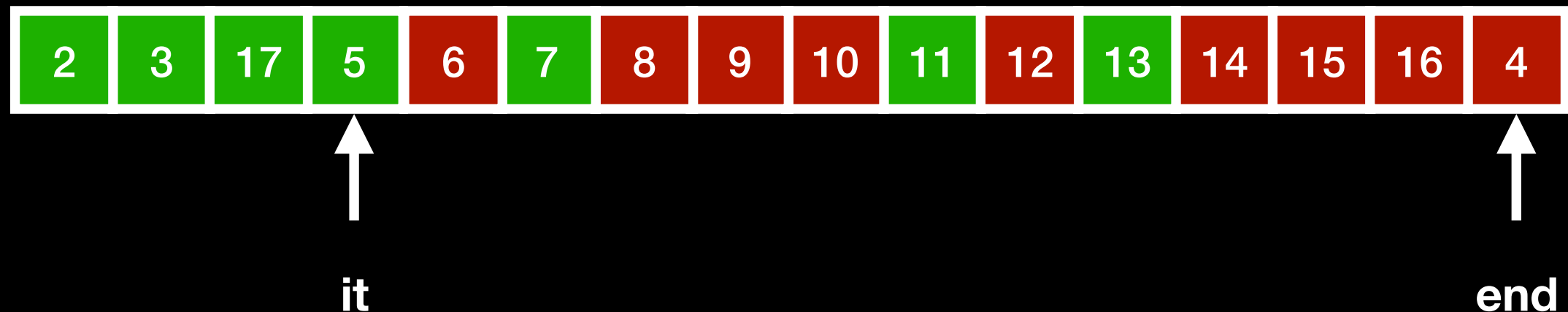
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



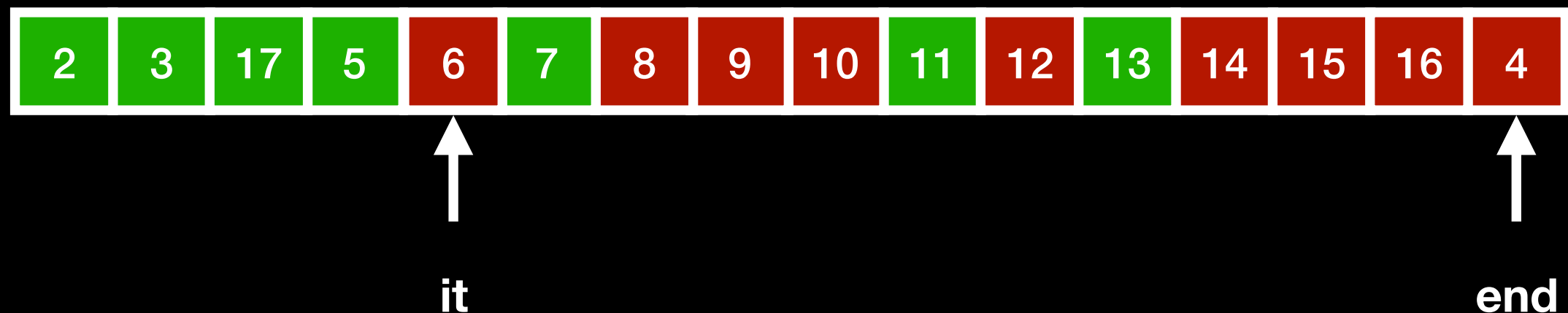
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        > while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



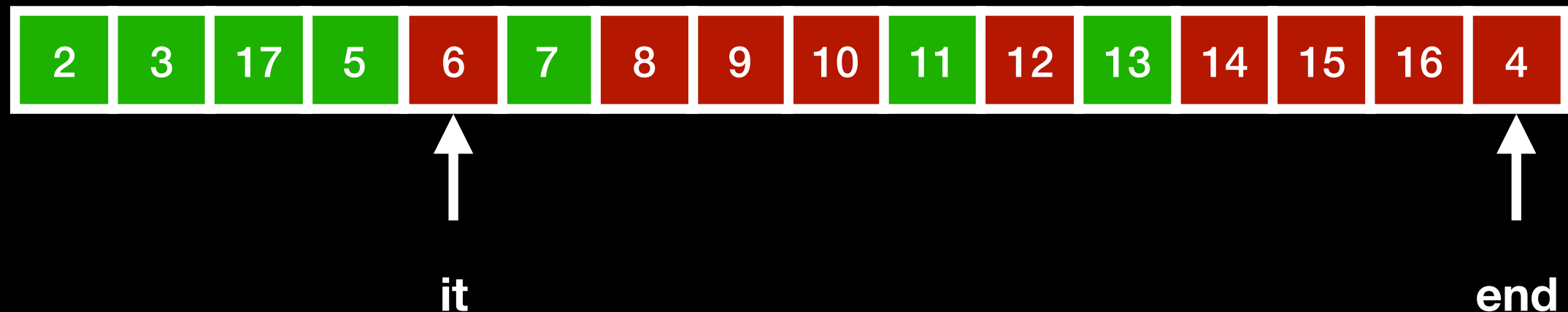
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        > while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



# Hoare's Partition

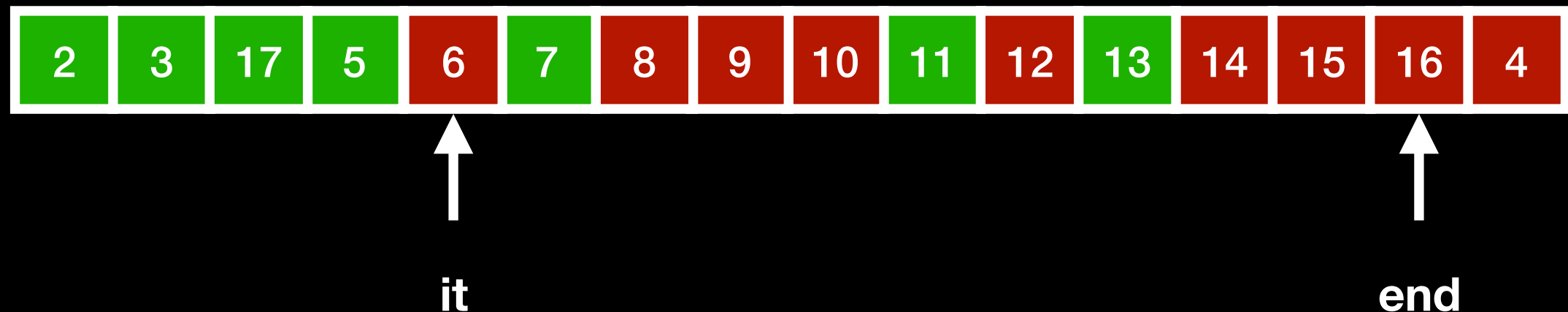
```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        > while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```





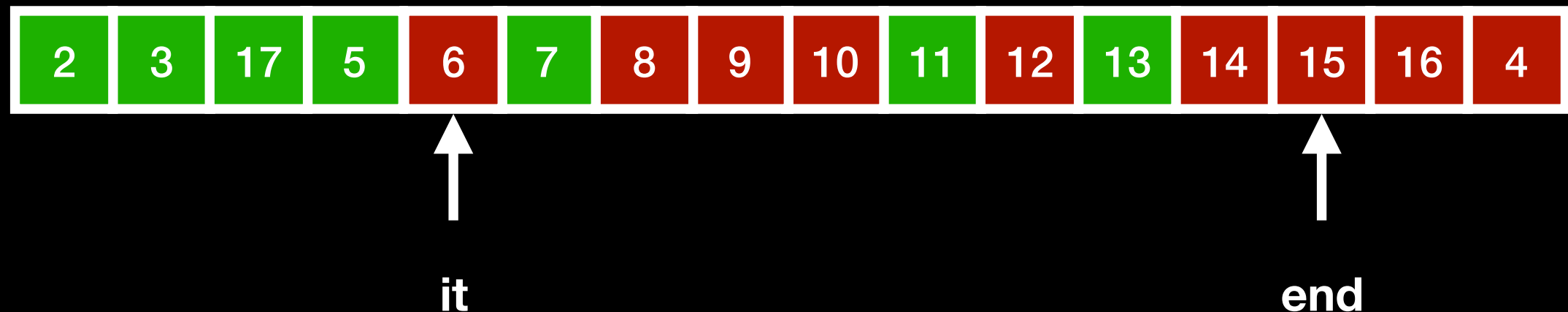
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        > while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



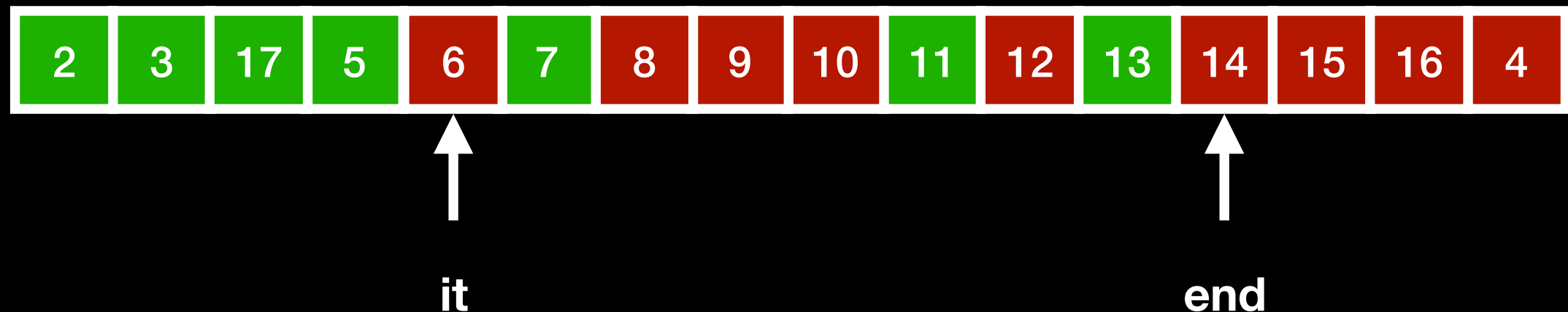
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        > while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



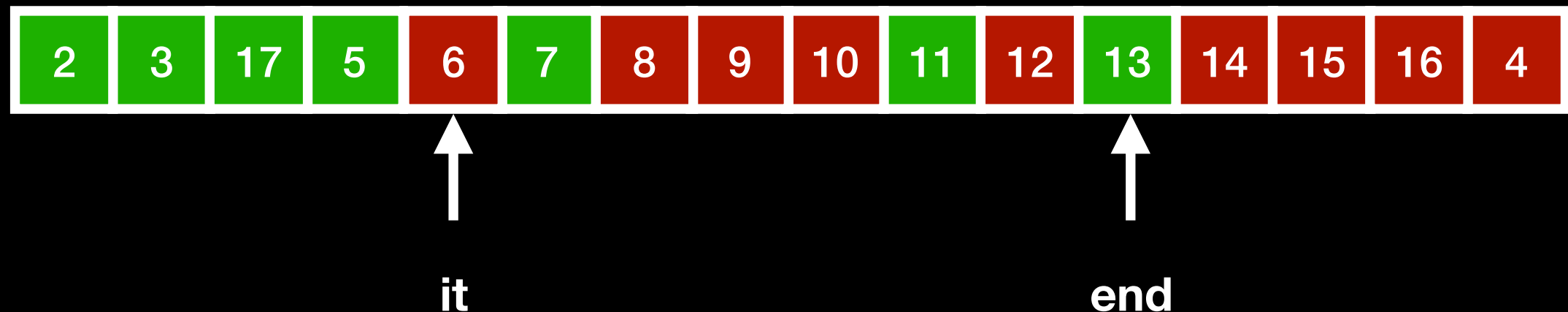
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        > while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



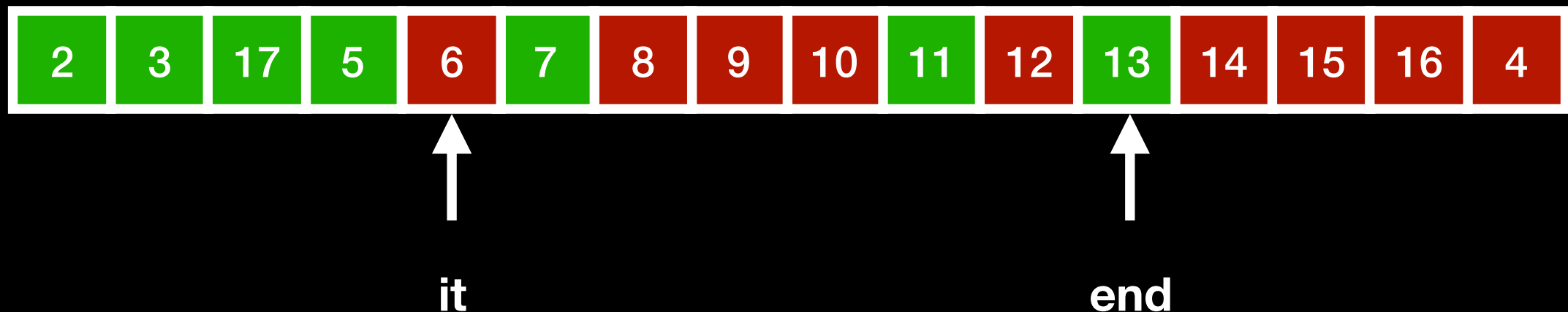
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        > while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



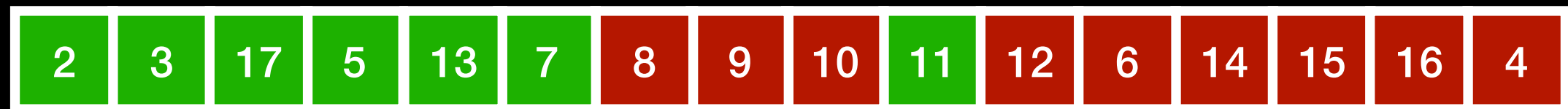
it



end

# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```

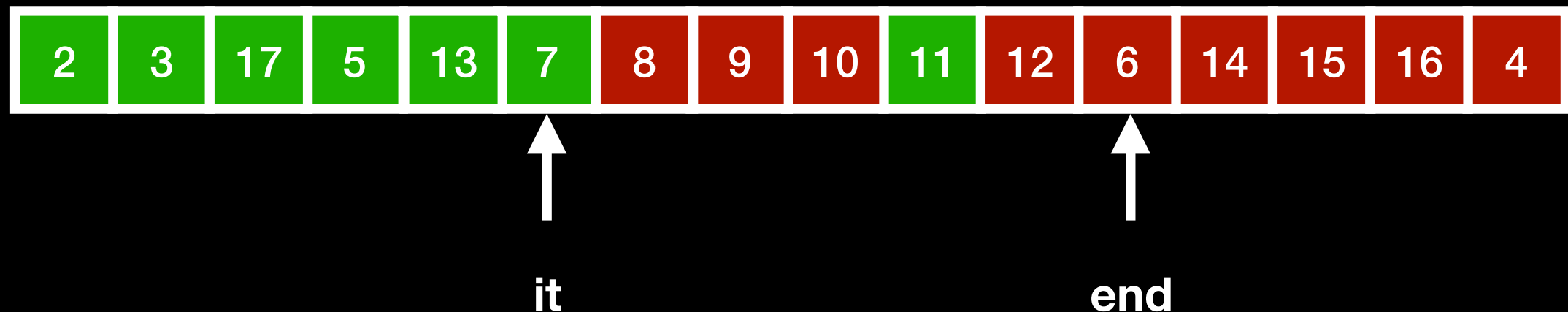


it

end

# Hoare's Partition

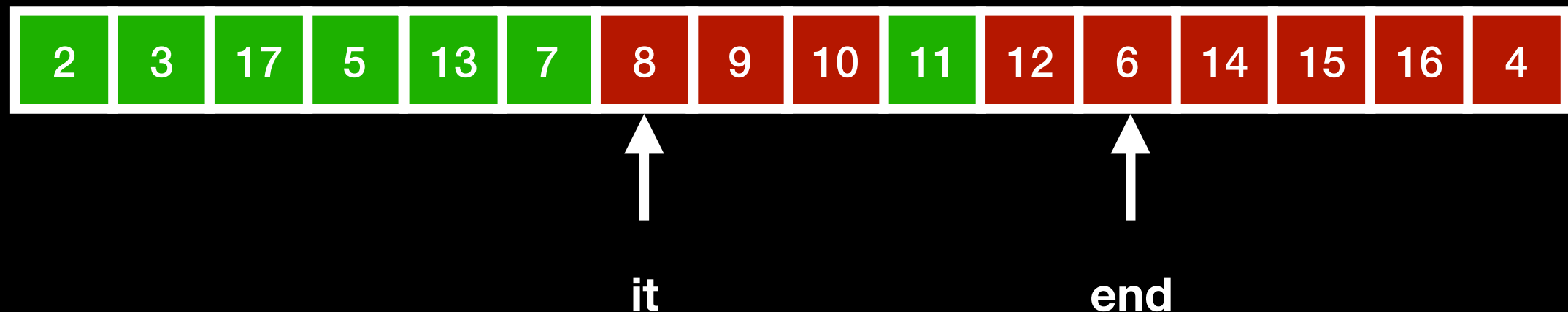
```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        > while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```





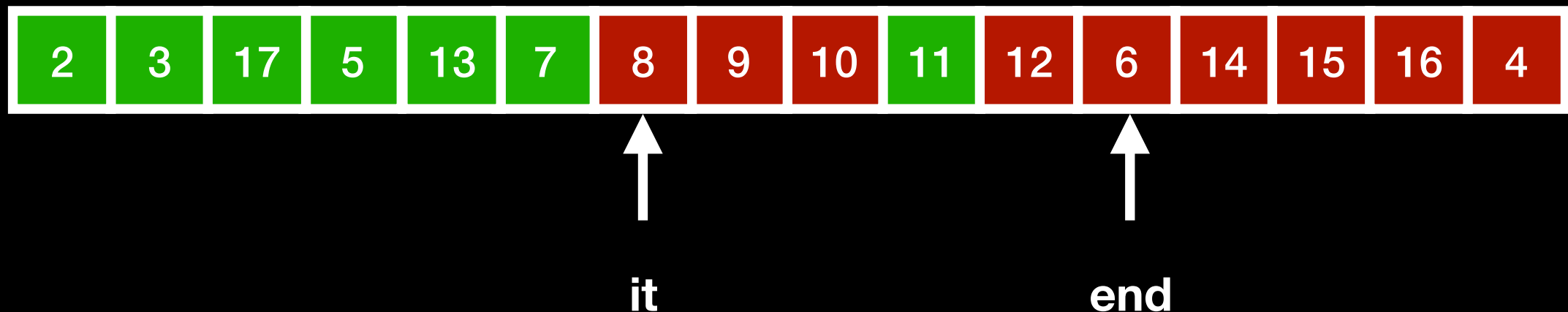
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        > while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



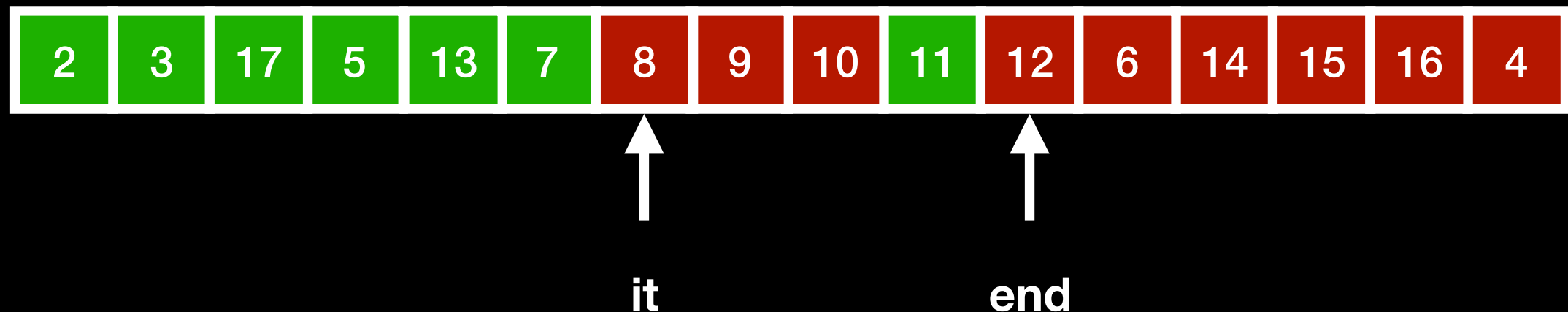
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        > while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



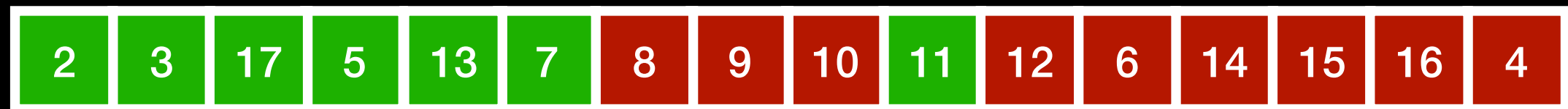
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
>       while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        > while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



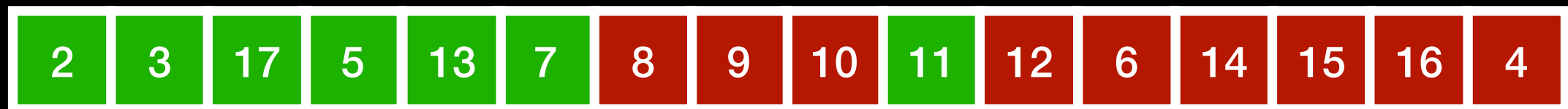
it



end

# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



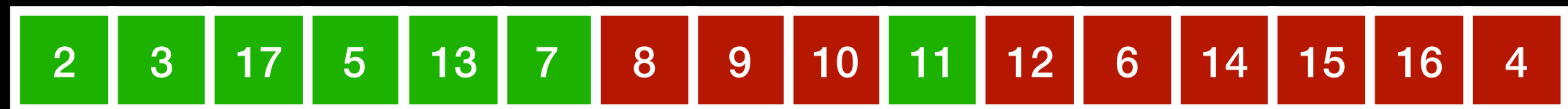
it



end

# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



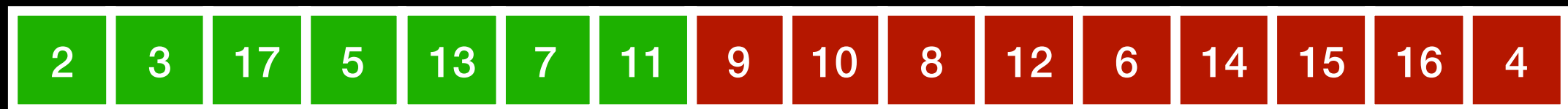
it



end

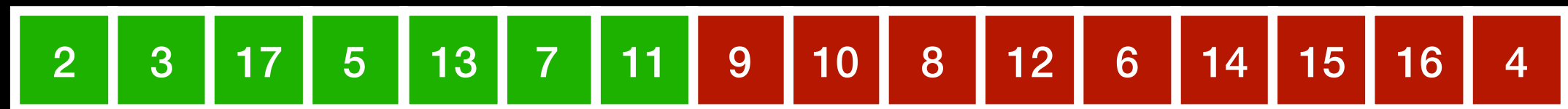
# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        > while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



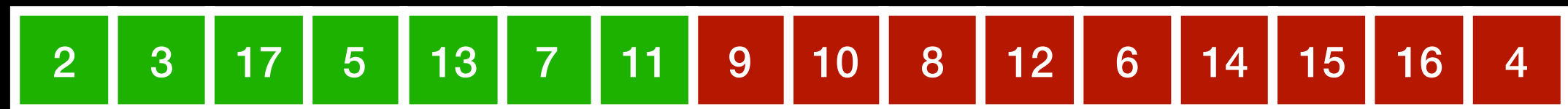
it

end



# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
>       while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```

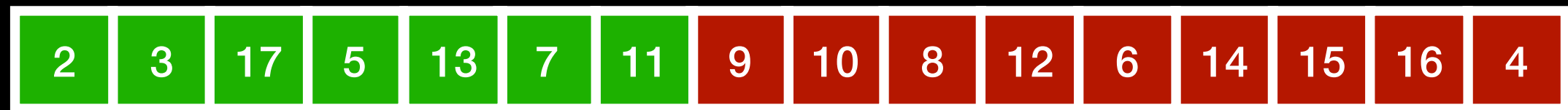


it

end

# Hoare's Partition

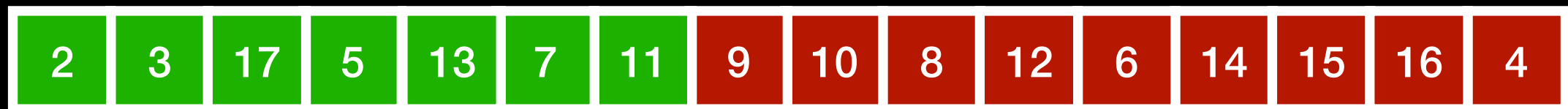
```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        > while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



it end

# Hoare's Partition

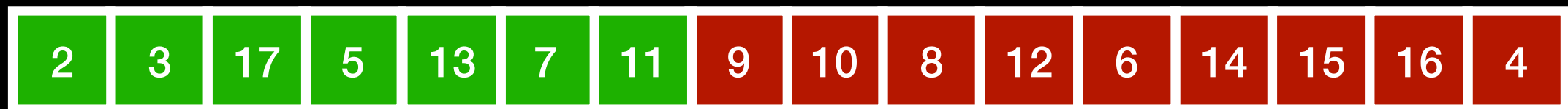
```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
>       while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



it, end

# Hoare's Partition

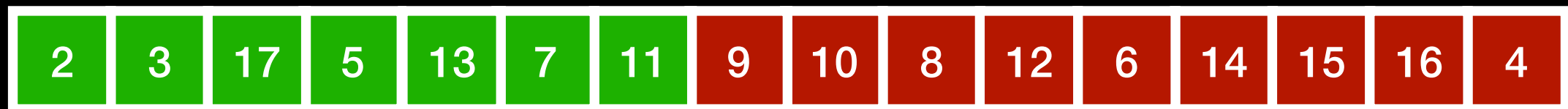
```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



it, end

# Hoare's Partition

```
template <typename BiDirlt, typename Predicate>
BiDirlt hoare(BiDirlt it, BiDirlt end, Predicate predicate) {
    while (true) {
        while (it != end && predicate(*it)) { ++it; }
        while (it != end && !predicate(*--end)) {}
        if (it == end) { return it; }
        std::iter_swap(it++, end);
    }
}
```



# Sequential

- Lomuto's and Hoare's scheme do not parallelise
  - Hoare's scheme is very effective when sequential
- something operating independently is needed:
  - do most work in parallel on parts of the range
  - clean up where things remain out of order

# Blocked Partition

```
template <typename RndIt, typename Predicate>
RndIt blocked(RndIt begin, RndIt end, Predicate pred) {
    BlockQueue<RndIt> q(begin, end); Block<RndIT> f, b;
    while (true) {
        if (f.empty() && (f = q.front()).empty()) { break; }
        if (b.empty() && (b = q.back()).empty()) { break; }
        tie(f, b) = block(f, b, pred);
    }
    return clean_up(f, b, pred);
}
```

# Blocked Partition

```
template <typename RndIt, typename Predicate>
RndIt blocked(RndIt begin, RndIt end, Predicate pred) {
    BlockQueue<RndIt> q(begin, end); Block<RndIT> f, b;
    while (true) {
        if (f.empty() && (f = q.front()).empty()) { break; }
        if (b.empty() && (b = q.back()).empty()) { break; }
        tie(f, b) = block(f, b, pred);
    }
    return clean_up(f, b, pred);
}
```



# Block Queue

```
template <typename RndIt> struct BlockQueue {  
    RndIt beg, end; int size; constexpr int bs = 123;  
    BlockQueue(RndIt b, RndIt e): beg(b), end(e), size(e-b) {}  
    Block front() {  
        auto s = min(size, bs); size -= s; beg += s;  
        return Block<RndIt>(beg - s, beg); }  
    Block back() {  
        auto s = min(size, bs); size -= s; end -= s;  
        return Block<RndIt>(end, end + s); }  
};
```

# Blocked Partition

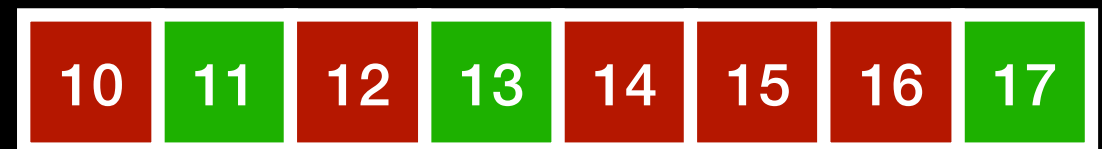
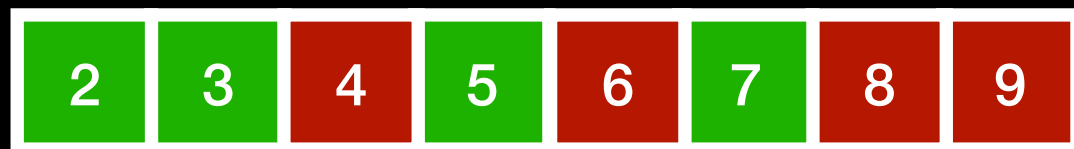
```
template <typename RndIt, typename Predicate>
RndIt blocked(RndIt begin, RndIt end, Predicate pred) {
    BlockQueue<RndIt> q(begin, end); Block<RndIT> f, b;
    while (true) {
        if (f.empty() && (f = q.front()).empty()) { break; }
        if (b.empty() && (b = q.back()).empty()) { break; }
        tie(f, b) = block(f, b, pred);
    }
    return clean_up(f, b, pred);
}
```

# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```

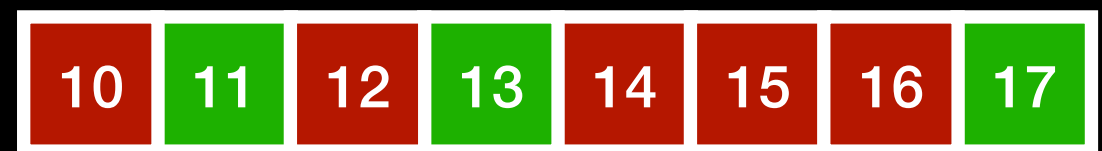
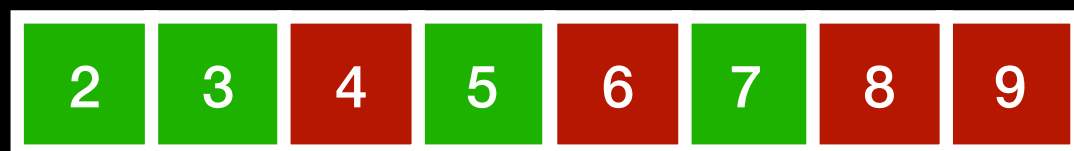
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



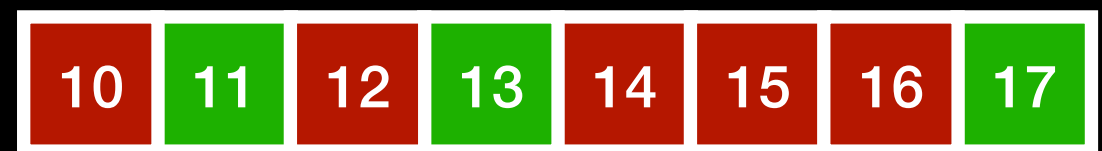
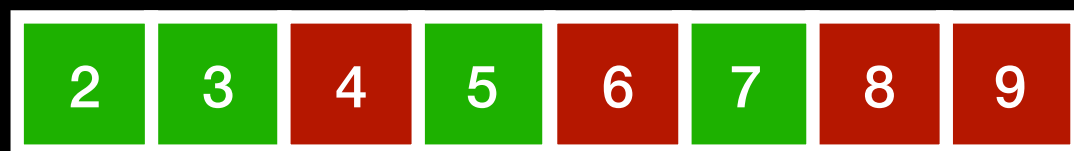
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
>   while (true) {
       while (!f.empty() && pred(*f)) { ++f; }
       while (!b.empty() && !pred(*b)) { ++b; }
       if (f.empty() || b.empty()) { return std::make_pair(f, b); }
       using std::swap; swap(*f, *b);
   } }
```



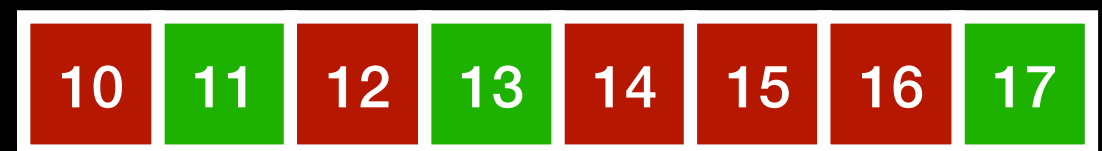
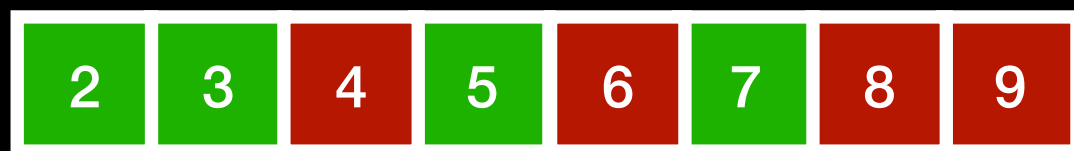
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        > while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



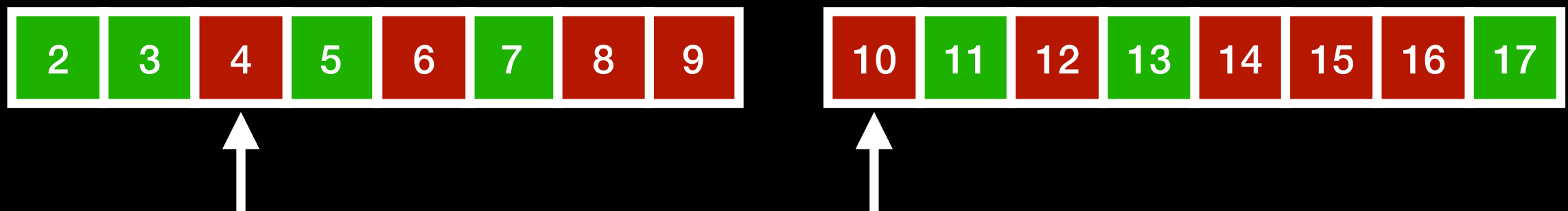
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        > while (!f.empty() && pred(*f)) { ++f; }
          while (!b.empty() && !pred(*b)) { ++b; }
          if (f.empty() || b.empty()) { return std::make_pair(f, b); }
          using std::swap; swap(*f, *b);
    } }
}
```



# Block Partition

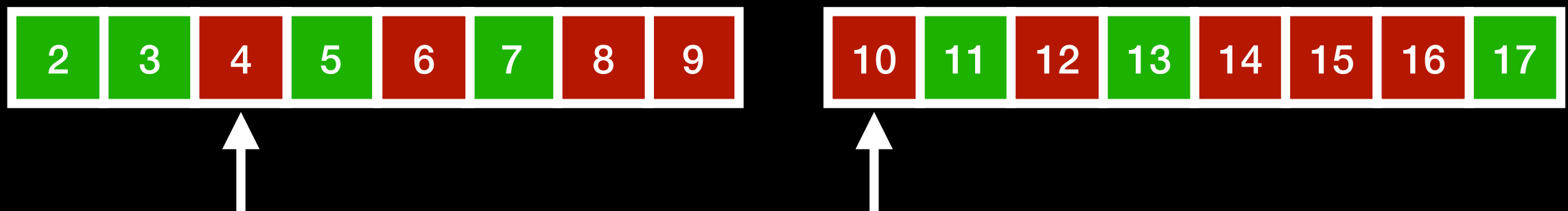
```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        > while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```





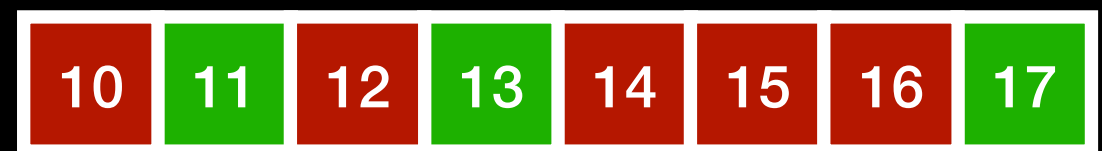
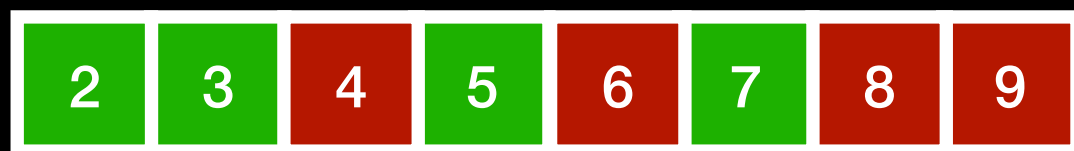
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



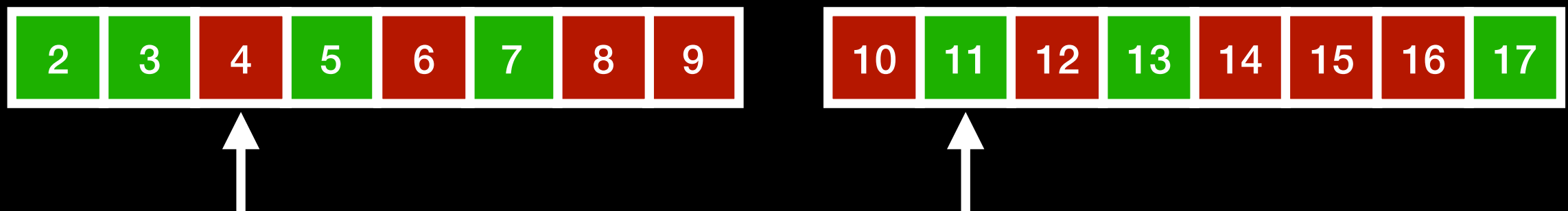
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



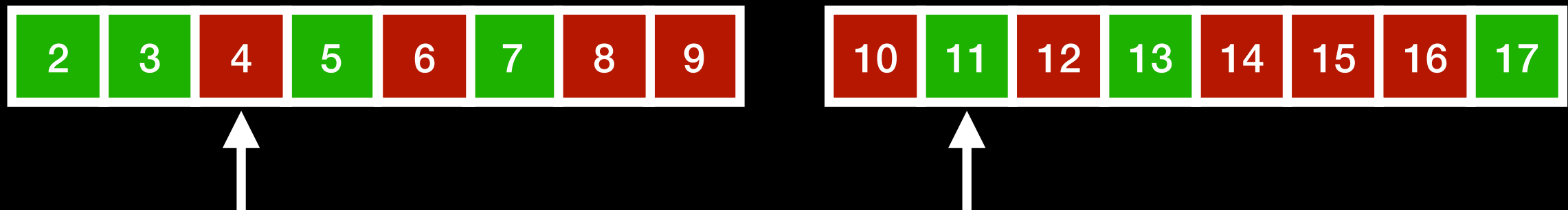
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



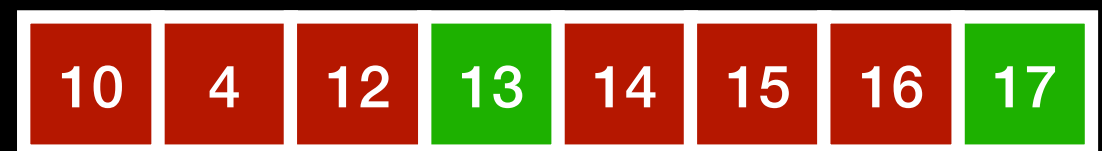
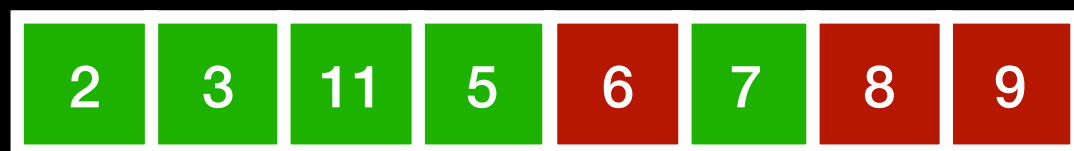
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



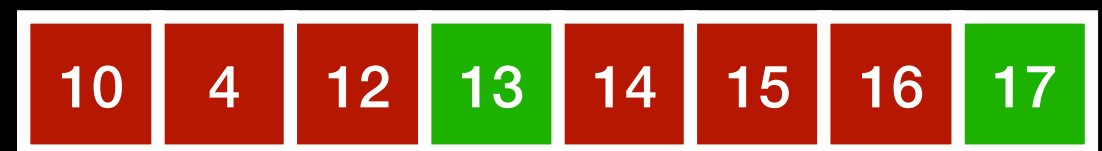
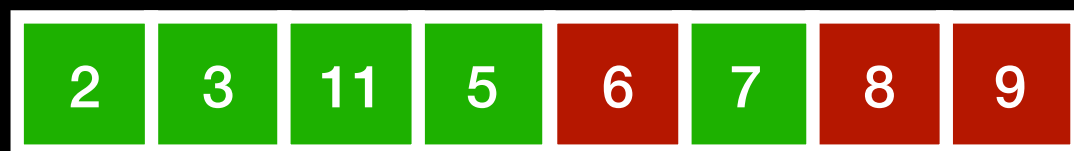
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



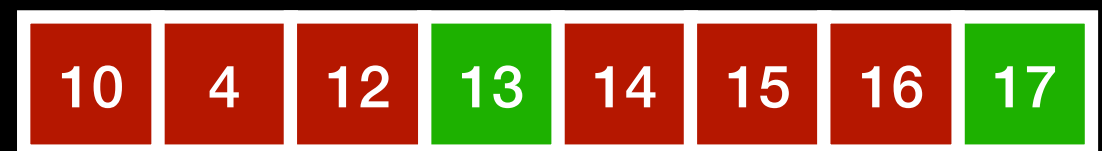
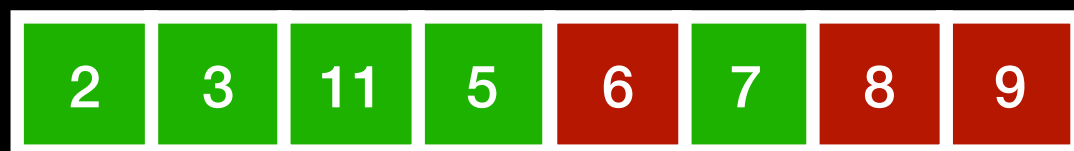
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
>   while (true) {
       while (!f.empty() && pred(*f)) { ++f; }
       while (!b.empty() && !pred(*b)) { ++b; }
       if (f.empty() || b.empty()) { return std::make_pair(f, b); }
       using std::swap; swap(*f, *b);
   } }
```



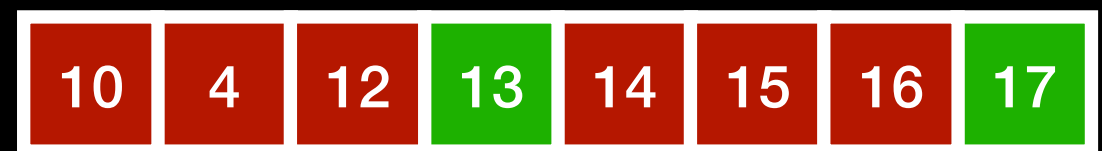
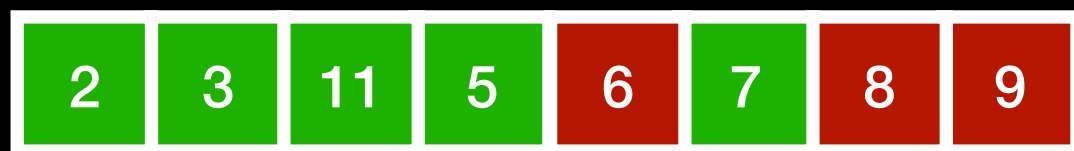
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        > while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



# Block Partition

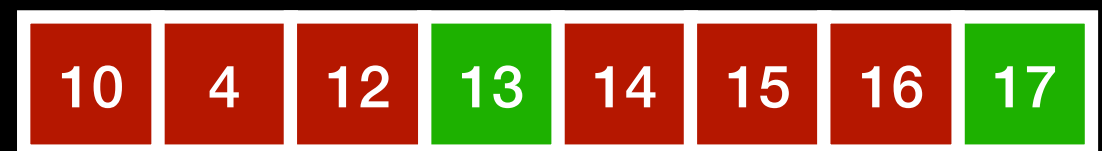
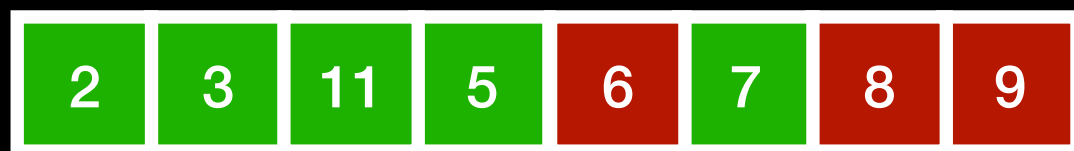
```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        > while (!f.empty() && pred(*f)) { ++f; }
          while (!b.empty() && !pred(*b)) { ++b; }
          if (f.empty() || b.empty()) { return std::make_pair(f, b); }
          using std::swap; swap(*f, *b);
    } }
}
```





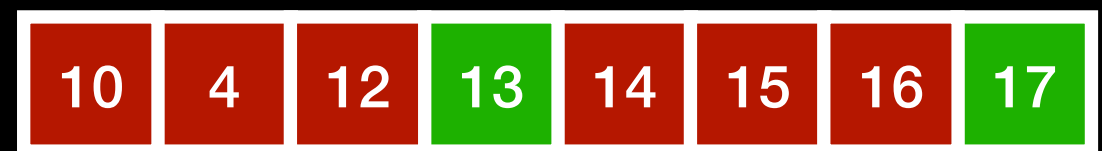
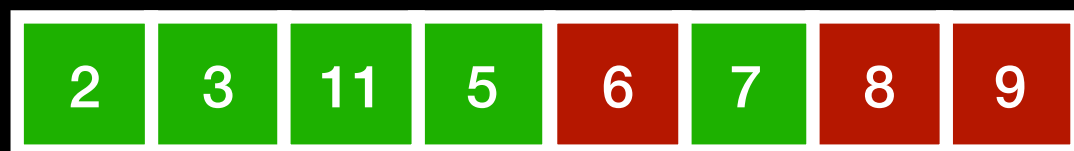
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        > while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



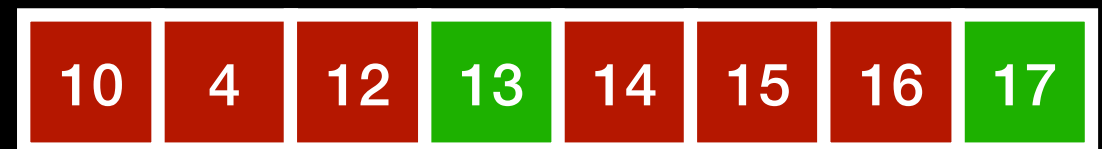
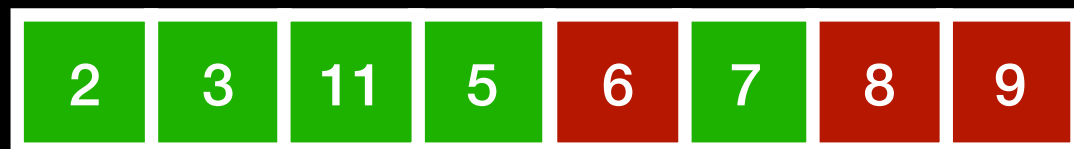
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



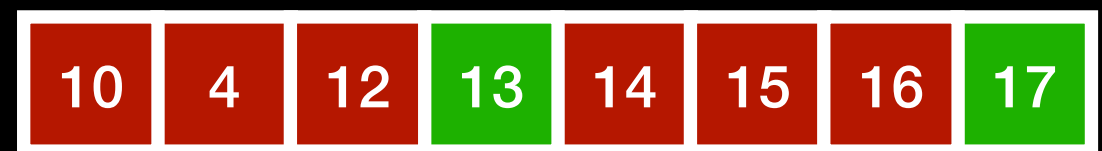
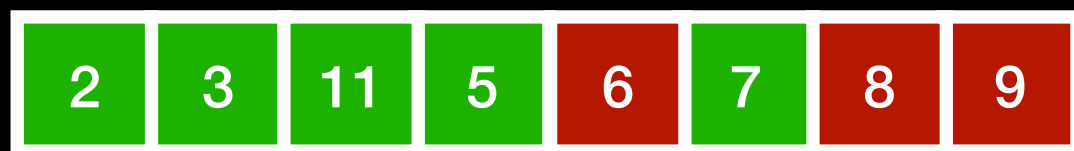
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



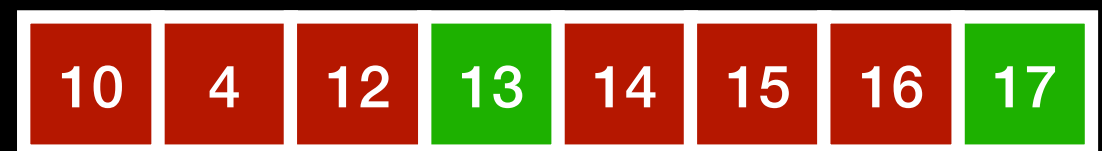
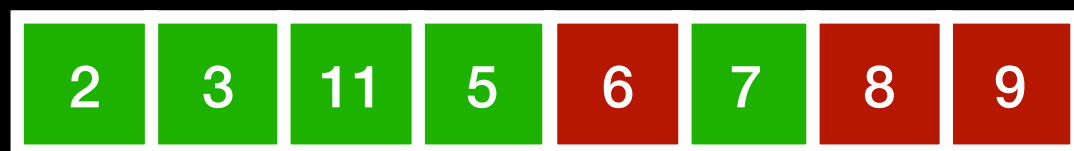
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



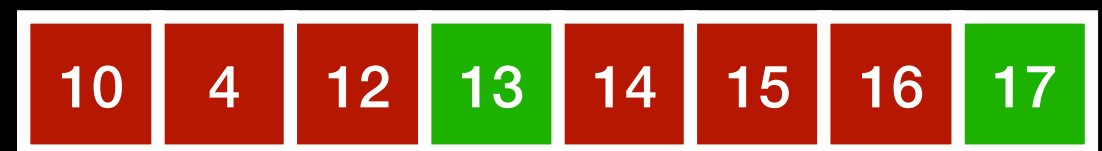
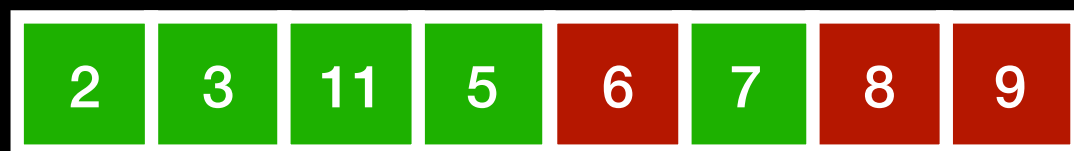
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



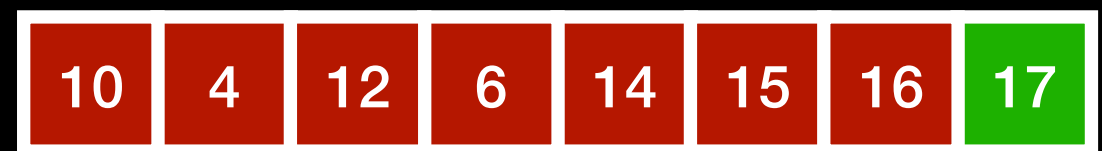
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



# Block Partition

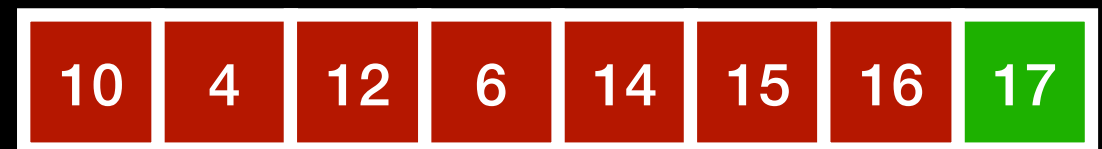
```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
>   while (true) {
       while (!f.empty() && pred(*f)) { ++f; }
       while (!b.empty() && !pred(*b)) { ++b; }
       if (f.empty() || b.empty()) { return std::make_pair(f, b); }
       using std::swap; swap(*f, *b);
   } }
```





# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        > while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



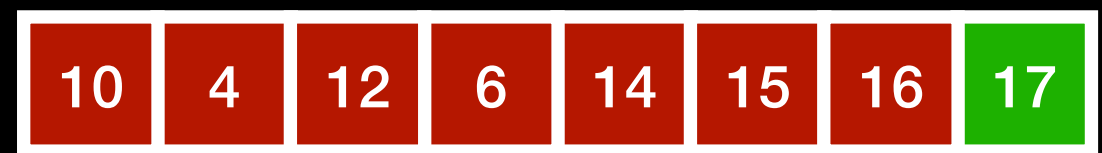
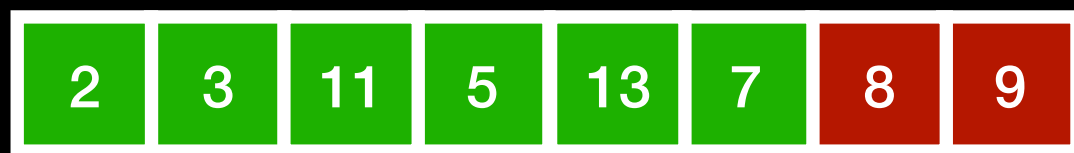
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        > while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
>        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



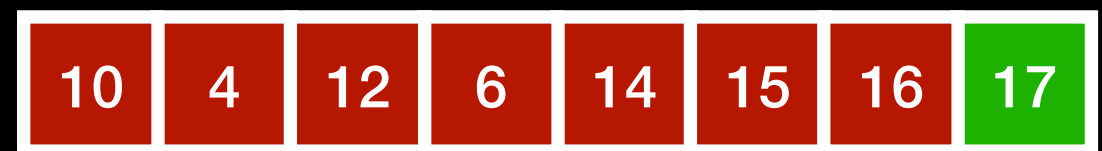
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



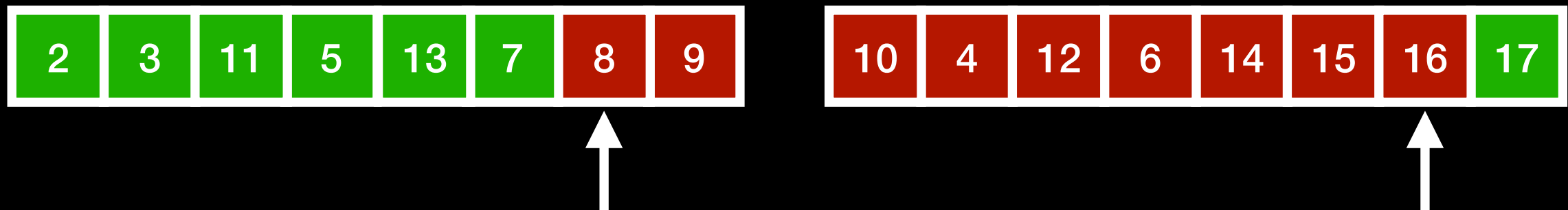
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



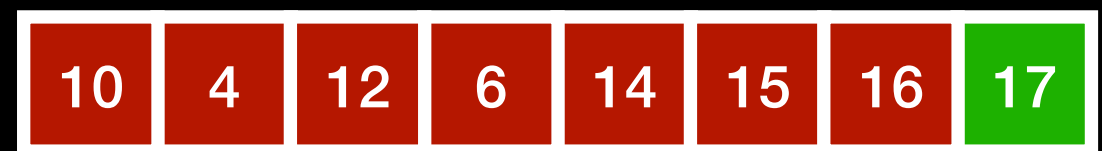
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```





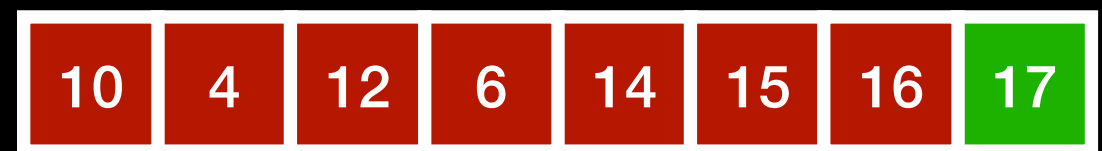
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



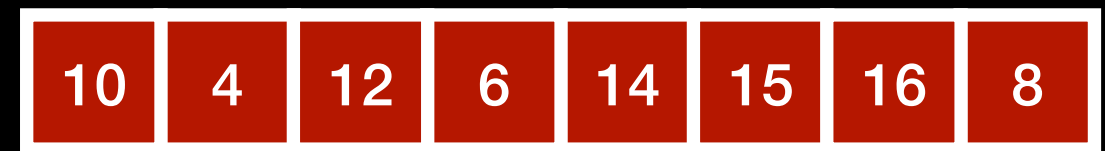
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



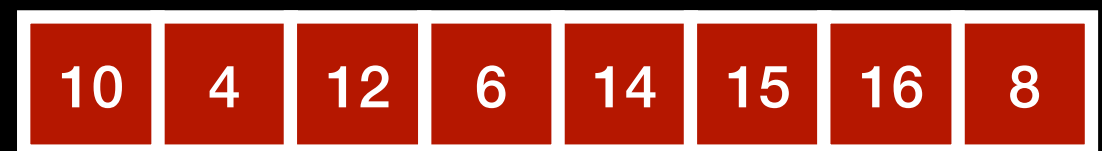
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
>   while (true) {
       while (!f.empty() && pred(*f)) { ++f; }
       while (!b.empty() && !pred(*b)) { ++b; }
       if (f.empty() || b.empty()) { return std::make_pair(f, b); }
       using std::swap; swap(*f, *b);
   } }
```



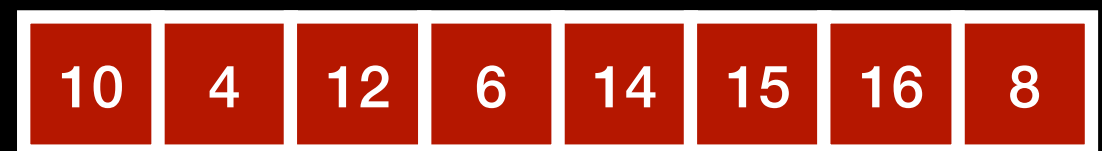
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        > while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



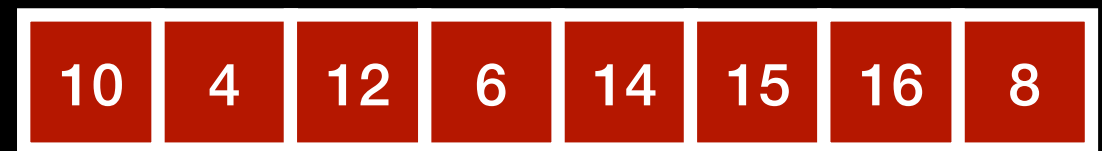
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        > while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



# Block Partition

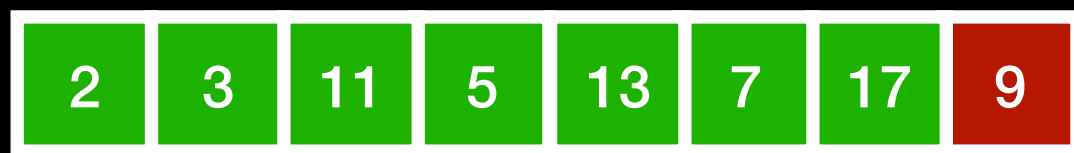
```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```





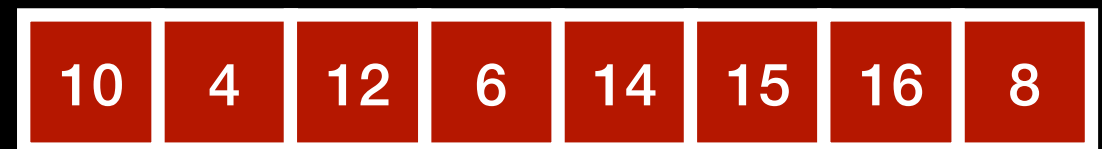
# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



# Block Partition

```
template <typename Blk, typename Predicate>
std::pair<Blk, Blk> block(Blk f, Blk b, Predicate pred) {
    while (true) {
        while (!f.empty() && pred(*f)) { ++f; }
        while (!b.empty() && !pred(*b)) { ++b; }
        if (f.empty() || b.empty()) { return std::make_pair(f, b); }
        using std::swap; swap(*f, *b);
    }
}
```



# Blocked Partition

```
template <typename RndIt, typename Predicate>
RndIt blocked(RndIt begin, RndIt end, Predicate pred) {
    BlockQueue<RndIt> q(begin, end); Block<RndIt> f, b;
    while (true) {
        if (f.empty() && (f = q.front()).empty()) { break; }
        if (b.empty() && (b = q.back()).empty()) { break; }
        tie(f, b) = block(f, b, pred);
    }
    return clean_up(f, b, pred);
}
```

# Clean Up

```
template <typename It, typename Pred>
It clean_up(Block<It> f, Block<It> b, Pred pred) {
    if (b.empty()) {
        return std::partition(f.current(), f.end(), pred);
    }
    auto s = b.current();
    auto p = std::partition(b.current(), b.end(), pred);
    std::swap_ranges(s, p, b.begin());
    return b.begin() + (p - s);
}
```

# Parallel Block Partition

- blocks can be processed in parallel
  - make the queue thread-safe to allow concurrent access
- running individual threads is not effective
  - instead schedule jobs with thread pool
- schedule processing blocks
- after partitioning blocks some clean-up is needed

# Parallel Block Partition

- blocks can be processed in parallel
  - make the queue thread-safe to allow concurrent access
- running individual threads is not effective
  - instead schedule jobs with thread pool
- schedule processing blocks
- after partitioning blocks some clean-up is needed

# Block Queue (retake)

```
template <typename RndIt> struct BlockQueue {  
    RndIt beg, end; static constexpr int bs = 123;  
    int size;  
    BlockQueue(RndIt b, RndIt e): beg(b), end(e), size(e-b) {}  
    Block<RndIt> front() {  
        auto s = min(size, bs);  
        size -= s;  
        beg += s;  
  
        return Block<RndIt>(beg - s, beg); }  
};
```

# Block Queue (retake)

```
template <typename RndIt> struct BlockQueue {  
    RndIt beg, end; static constexpr int bs = 123;  
    int size;  
    BlockQueue(RndIt b, RndIt e): beg(b), end(e), size(e-b) {}  
    Block<RndIt> front() {  
        auto s = min(size, bs);  
        size -= s;  
        beg += s;  
  
        return Block<RndIt>(beg - s, beg); }  
};
```



# Block Queue (retake)

```
template <typename RndIt> struct BlockQueue {  
    RndIt beg, end; static constexpr int bs = 123;  
    std::atomic<int> size;  
    BlockQueue(RndIt b, RndIt e): beg(b), end(e), size(e-b) {}  
    Block<RndIt> front() {  
        auto s = size.fetch_sub(bs);  
        s = min(max(0, s), bs);  
        beg += s;  
  
        return Block<RndIt>(beg - s, beg); }  
};
```

# Block Queue (retake)

```
template <typename RndIt> struct BlockQueue {  
    RndIt beg, end; static constexpr int bs = 123;  
    std::atomic<int> size;  
    BlockQueue(RndIt b, RndIt e): beg(b), end(e), size(e-b) {}  
    Block<RndIt> front() {  
        auto s = size.fetch_sub(bs);  
        s = min(max(0, s), bs);  
        beg += s;  
  
        return Block<RndIt>(beg - s, beg); }  
};
```

# Block Queue (retake)

```
template <typename RndIt> struct BlockQueue {  
    std::atomic<RndIt> beg, end; static constexpr int bs=123;  
    std::atomic<int> size;  
    BlockQueue(RndIt b, RndIt e): beg(b), end(e), size(e-b) {}  
    Block<RndIt> front() {  
        auto s = size.fetch_sub(bs);  
        s = min(max(0, s), bs);  
        beg += s;  
  
        return Block<RndIt>(beg - s, beg); }  
};
```

# Block Queue (retake)

```
template <typename RndIt> struct BlockQueue {  
    RndIt beg, end; static constexpr int bs = 123;  
    std::atomic<int> size, f_off{0}, b_off{0};  
    BlockQueue(RndIt b, RndIt e): beg(b), end(e), size(e-b) {}  
    Block<RndIt> front() {  
        auto s = size.fetch_sub(bs);  
        s = min(max(0, s), bs);  
        auto off = f_off.fetch_add(s);  
  
        return Block<RndIt>(beg + off, beg + off + s); }  
};
```

# Parallel Block Partition

- blocks can be processed in parallel
  - make the queue thread-safe to allow concurrent access
- running individual threads is not effective
  - instead schedule jobs with thread pool
- schedule processing blocks
- after partitioning blocks some clean-up is needed

# Thread Pool

```
class thread_pool {
```

```
public:
```

```
    explicit thread_pool(int count);
```

```
    ~thread_pool();
```

```
    void stop();
```

```
    template <typename Job> void enqueue(Job job);
```

```
};
```

# Thread Pool

```
class thread_pool {  
    std::mutex          mutex;  
  
public:  
    explicit thread_pool(int count);  
    ~thread_pool();  
    void stop();  
    template <typename Job> void enqueue(Job job);  
};
```

# Thread Pool

```
class thread_pool {  
    std::mutex                mutex;  
  
    void run();  
public:  
    explicit thread_pool(int count);  
    ~thread_pool();  
    void stop();  
    template <typename Job> void enqueue(Job job);  
};
```



# Thread Pool

```
class thread_pool {  
    std::mutex                mutex;  
    std::condition_variable   cond;  
  
    void run();  
public:  
    explicit thread_pool(int count);  
    ~thread_pool();  
    void stop();  
    template <typename Job> void enqueue(Job job);  
};
```

# Thread Pool

```
class thread_pool {  
    std::mutex                                mutex;  
    std::condition_variable                   cond;  
    std::deque<std::function<void()>> jobs;  
  
    void run();  
public:  
    explicit thread_pool(int count);  
    ~thread_pool();  
    void stop();  
    template <typename Job> void enqueue(Job job);  
};
```

# Thread Pool

```
class thread_pool {  
    std::mutex                                mutex;  
    std::condition_variable                   cond;  
    std::deque<std::function<void()>> jobs;  
    std::list<join_thread>                   threads;  
    void run();  
public:  
    explicit thread_pool(int count);  
    ~thread_pool();  
    void stop();  
    template <typename Job> void enqueue(Job job);  
};
```

# Thread Pool

```
class thread_pool {  
    std::mutex                mutex;  
    std::condition_variable   cond;  
    std::deque<std::function<void()>> jobs;  
    std::list<join_thread>    threads;  
    void run();  
public:  
    explicit thread_pool(int count);  
    ~thread_pool();  
    void stop();  
    template <typename Job> void enqueue(Job job);  
};
```

# Joining Threads

- destroying a non-joined, non-detect thread  $\Rightarrow$  terminate

```
struct join_thread  
    : std::thread {  
    using std::thread::thread;  
    ~join_thread() { this->join(); }  
};
```

# Thread Pool

```
class thread_pool {  
    std::mutex                                mutex;  
    std::condition_variable                   cond;  
    std::deque<std::function<void()>> jobs;  
    std::list<join_thread>                   threads;  
    void run();  
public:  
    explicit thread_pool(int count);  
    ~thread_pool();  
    void stop();  
    template <typename Job> void enqueue(Job job);  
};
```

# Thread Pool

```
class thread_pool {  
    std::mutex                                mutex;  
    std::condition_variable                  cond;  
    std::deque<std::function<void()>> jobs;  
    std::list<join_thread>                  threads;  
    void run();  
public:  
    explicit thread_pool(int count);  
    ~thread_pool();  
    void stop();  
    template <typename Job> void enqueue(Job job);  
};
```

# Thread Pool

```
void thread_pool::run() {  
    for (std::function<void()> fun({}); fun; ) {  
        fun();  
        std::unique_lock<std::mutex> lock(this->mutex);  
        this->cond.wait(lock,  
                        [this]{ return !this->jobs.empty(); });  
        fun = std::move(this->jobs.front());  
        this->jobs.pop_front();  
    }  
}
```



# Thread Pool

```
void thread_pool::run() {  
    for (std::function<void()> fun({}); fun; ) {  
        fun();  
        std::unique_lock<std::mutex> kerberos(this->mutex);  
        this->cond.wait(kerberos,  
                        [this]{ return !this->jobs.empty(); });  
        fun = std::move(this->jobs.front());  
        this->jobs.pop_front();  
    }  
}
```

# Thread Pool

```
class thread_pool {  
    std::mutex                mutex;  
    std::condition_variable   cond;  
    std::deque<std::function<void()>> jobs;  
    std::list<join_thread>    threads;  
    void run();  
public:  
    explicit thread_pool(int count);  
    ~thread_pool();  
    void stop();  
    template <typename Job> void enqueue(Job job);  
};
```

# Thread Pool

```
template <typename Job>
void thread_pool::enqueue(Job job) {
    {
        std::lock_guard<std::mutex> lock(this->mutex);
        this->jobs.emplace_back(std::move(job));
    }
    this->cond.notify_one();
}
```

# Thread Pool

```
class thread_pool {  
    std::mutex                                mutex;  
    std::condition_variable                  cond;  
    std::deque<std::function<void()>> jobs;  
    std::list<join_thread>                  threads;  
    void run();  
public:  
    explicit thread_pool(int count);  
    ~thread_pool();  
    void stop();  
    template <typename Job> void enqueue(Job job);  
};
```

# Thread Pool

```
thread_pool::thread_pool(int count) {  
    while (count—) {  
        threads.emplace_back([this]{ this->run(); });  
    }  
}  
thread_pool::~~thread_pool(){  
    this->stop();  
}
```

# Thread Pool

```
class thread_pool {  
    std::mutex                                mutex;  
    std::condition_variable                  cond;  
    std::deque<std::function<void()>> jobs;  
    std::list<join_thread>                  threads;  
    void run();  
public:  
    explicit thread_pool(int count);  
    ~thread_pool();  
    void stop();  
    template <typename Job> void enqueue(Job job);  
};
```

# Thread Pool

```
void thread_pool::stop(){
    {
        std::lock_guard<std::mutex> kerberos(this->mutex);
        for (std::size_t i(0); i != this->threads.size(); ++i) {
            this->jobs.emplace_back();
        }
    }
    this->cond.notify_all();
    this->jobs.clear();
}
```

# Thread Pool

```
void thread_pool::run() {  
    for (std::function<void()> fun({}); fun; ) {  
        fun();  
        std::unique_lock<std::mutex> lock(this->mutex);  
        this->cond.wait(lock,  
                        [this]{ return !this->jobs.empty(); });  
        fun = std::move(this->jobs.front());  
        this->threads.pop_front();  
    }  
}
```



# Parallel Block Partition

- blocks can be processed in parallel
  - make the queue thread-safe to allow concurrent access
- running individual threads is not effective
  - instead schedule jobs with thread pool
- schedule processing blocks
- after partitioning blocks some clean-up is needed

# Partition Job

```
BlockQueue<RndIt>      q(begin, end);
auto job = [&q, pred](auto& remain){
    remain = [&q, pred()->Block<RndIt>{
        for (Block<RndIt> f, b;;) {
            if (f.empty() && (f = q.front()).empty())
                return { b.cur(), std::partition(b.cur(), b.end(), pred) };
            if (b.empty() && (b = q.back()).empty())
                return { std::partition(f.cur(), f.end(), pred), f.end() };
            std::tie(f, b) = block(f, b, pred);
        }
    }();
};
```

# Partition Job

```
BlockQueue<RndIt>      q(begin, end);
auto job = [&q, pred](auto& remain){
    remain = [&q, pred]()->Block<RndIt>{
        for (Block<RndIt> f, b;;) {
            if (f.empty() && (f = q.front()).empty())
                return { b.cur(), std::partition(b.cur(), b.end(), pred) };
            if (b.empty() && (b = q.back()).empty())
                return { std::partition(f.cur(), f.end(), pred), f.end() };
            std::tie(f, b) = block(f, b, pred);
        }
    }();
};
```

# Partition Job

```
BlockQueue<RndIt>      q(begin, end);
auto job = [&q, pred](auto& remain){
    remain = [&q, pred()->Block<RndIt>{
        for (Block<RndIt> f, b;;) {
            if (f.empty() && (f = q.front()).empty())
                return { b.cur(), std::partition(b.cur(), b.end(), pred) };
            if (b.empty() && (b = q.back()).empty())
                return { std::partition(f.cur(), f.end(), pred), f.end() };
            std::tie(f, b) = block(f, b, pred);
        }
    }();
};
```

# Executing the Jobs

```
BlockQueue<RndIt>          q(begin, end);
std::vector<Block<RndIt>> remain(p.size());
latch                      l(remain.size());

auto job = [...](){ ... };

for (auto& block: remain) {
    p.enqueue_job([job, &l, &block](){ job(block); l.arrive(); });
}
l.wait();

// clean-up
```

# Executing the Jobs

```
BlockQueue<RndIt>          q(begin, end);
std::vector<Block<RndIt>> remain(p.size());
latch                      l(p.size());

auto job = [...](){ ... };

for (auto& block: remain) {
    p.enqueue_job([job, &l, &block](){ job(block); l.arrive(); });
}
l.wait();

// clean-up
```

# Latch

```
class latch {
    std::mutex          d_mutex;
    std::condition_variable d_condition; int d_await;
public:
    explicit latch(int await): d_await(await) {}
    void arrive() {
        std::lock_guard<std::mutex> kerberos(this->d_mutex);
        if (!--this->d_await) { this->d_condition.notify_one(); }
    }
    void wait() {
        std::unique_lock<std::mutex> kerberos(this->d_mutex);
        this->d_condition.wait(kerberos, [this]{return !this->d_await; });
    }
};
```

# Latch

```
class latch {
    std::mutex          d_mutex;
    std::condition_variable d_condition; int d_await;
public:
    explicit latch(int await): d_await(await) {}
    void arrive() {
        std::lock_guard<std::mutex> kerberos(this->d_mutex);
        if (!--this->d_await) { this->d_condition.notify_one(); }
    }
    void wait() {
        std::unique_lock<std::mutex> kerberos(this->d_mutex);
        this->d_condition.wait(kerberos, [this]{return !this->d_await; });
    }
};
```



# Latch

```
class latch {  
    std::mutex d_mutex;  
    std::condition_variable d_condition; int d_await;  
public:  
    explicit latch(int await): d_await(await) {}  
    void arrive() {  
        std::lock_guard<std::mutex> kerberos(this->d_mutex);  
        if (!--this->d_await) { this->d_condition.notify_one(); }  
    }  
    void wait() {  
        std::unique_lock<std::mutex> kerberos(this->d_mutex);  
        this->d_condition.wait(kerberos, [this]{return !this->d_await; });  
    }  
};
```

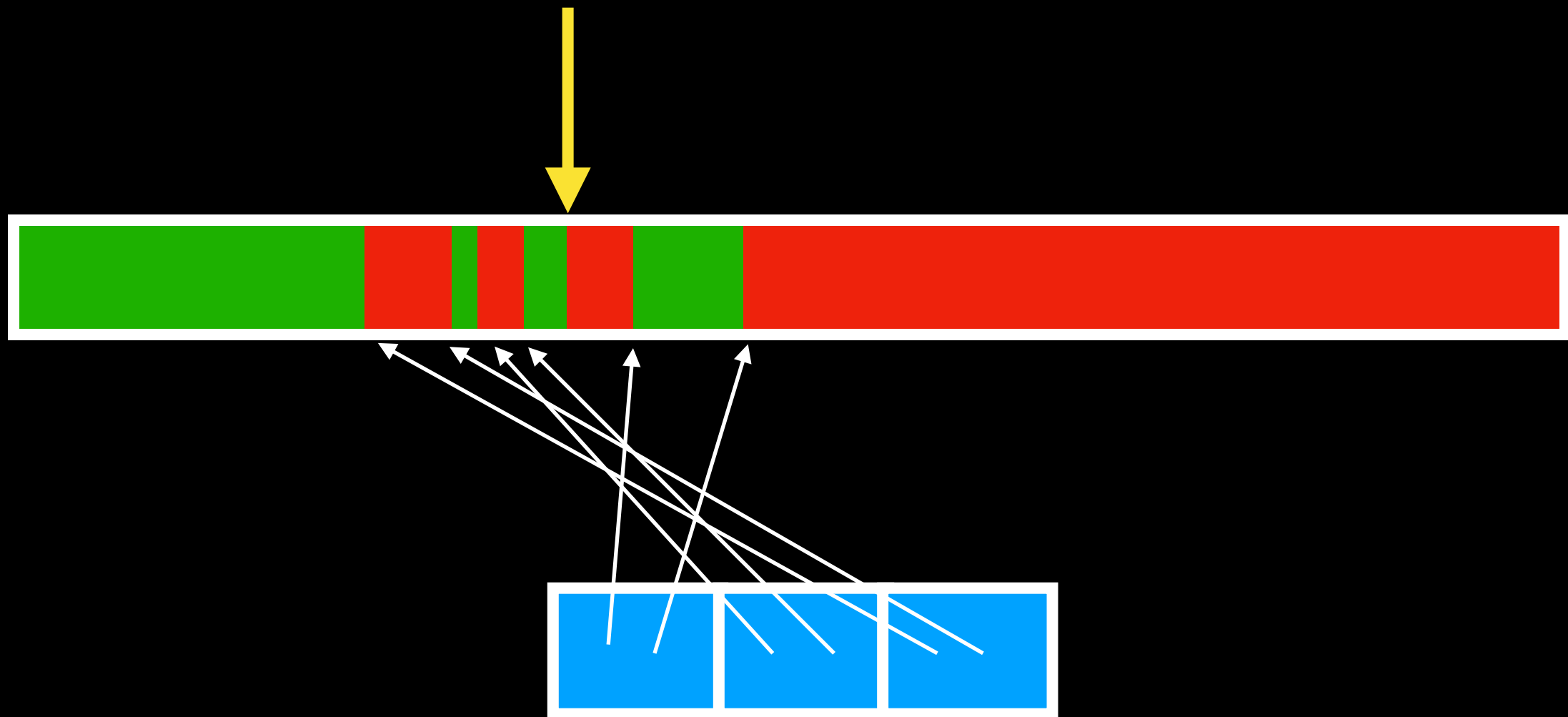
# Latch

```
class latch {
    std::mutex          d_mutex;
    std::condition_variable d_condition; int d_await;
public:
    explicit latch(int await): d_await(await) {}
    void arrive() {
        std::lock_guard<std::mutex> kerberos(this->d_mutex);
        if (!--this->d_await) { this->d_condition.notify_one(); }
    }
    void wait() {
        std::unique_lock<std::mutex> kerberos(this->d_mutex);
        this->d_condition.wait(kerberos, [this]{return !this->d_await; });
    }
};
```

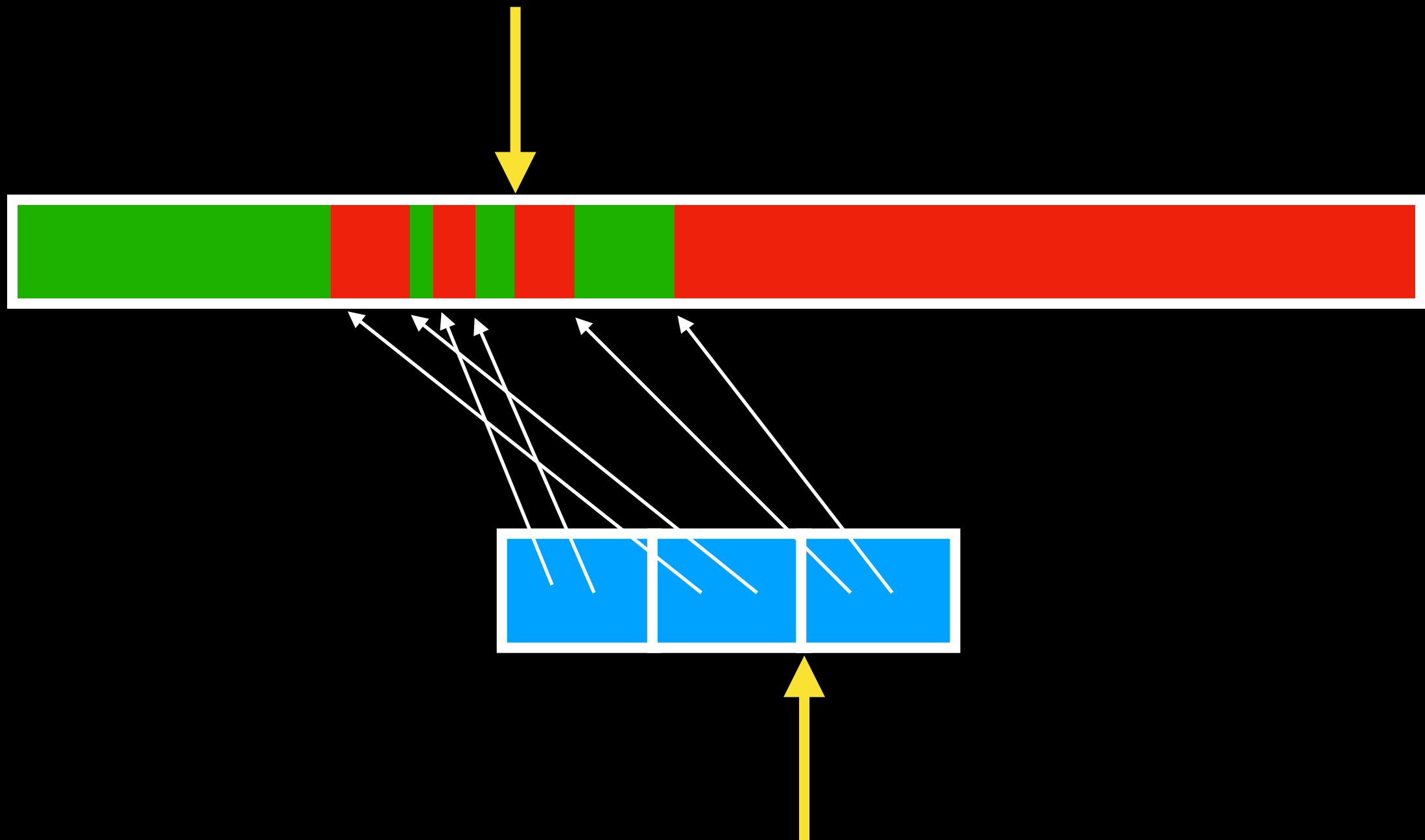
# Clean-up

```
RndIt mid = q.midpoint();
auto rp = std::partition(remain.begin(), remain.end(),
                        [mid](auto& b){ return b.begin() < mid; });
std::sort(remain.begin(), rp,
          [](auto& b0, auto& b1){ return b1.begin() < b0.begin(); });
for (auto it(remain.begin()); it != rp; ++it) {
    mid -= it->end() - it->begin();
    swap_ranges_helper(it->begin(), it->end(), mid);
}
std::sort(rp, remain.end(),
          [](auto& b0, auto& b1){ return b0.begin() < b1.begin(); });
for (auto it(rp); it != remain.end(); ++it) {
    swap_ranges_helper(mid, mid + (it->end() - it->begin()), it->begin());
    mid += it->end() - it->begin();
}
return mid;
```

# Clean-up



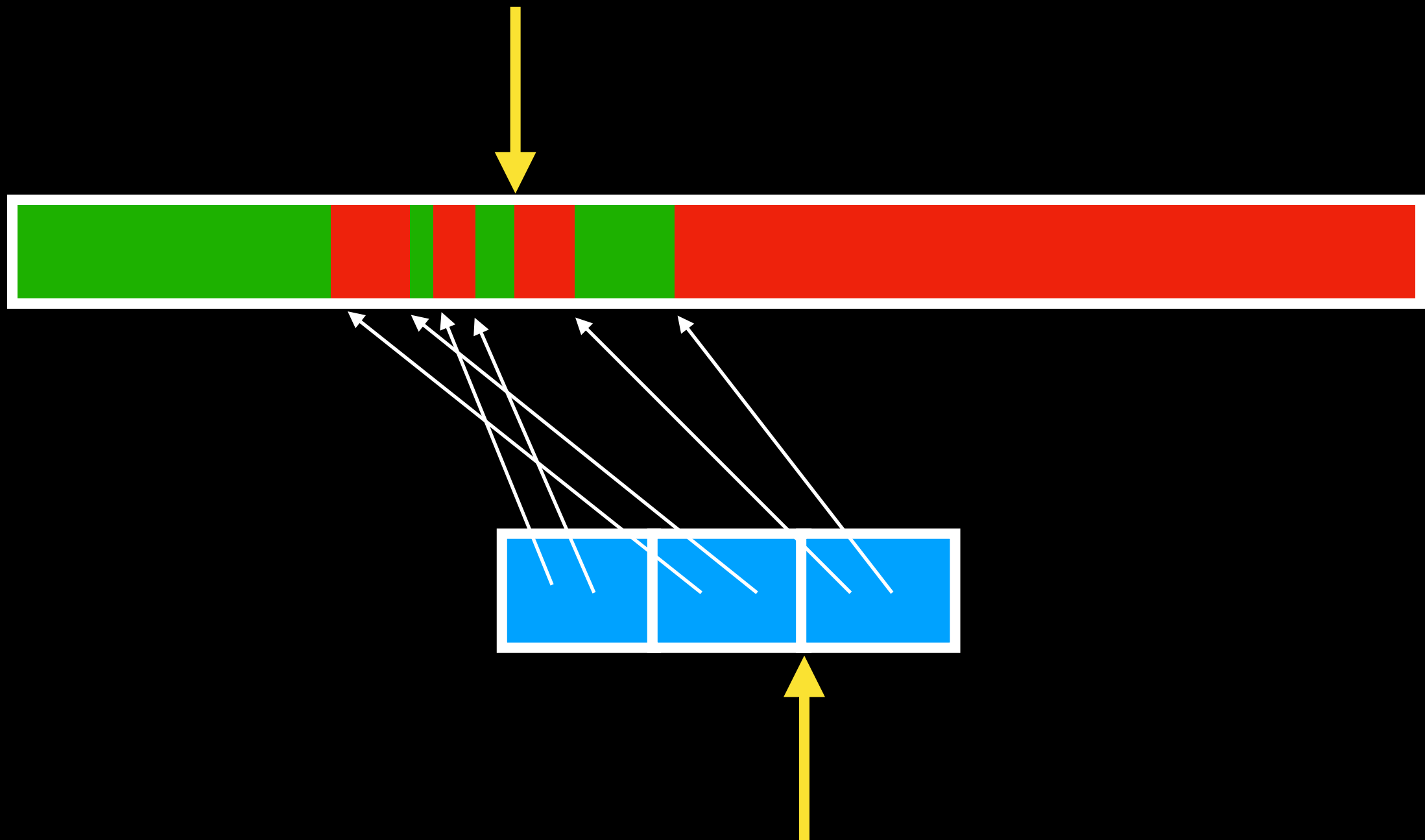
# Clean-up



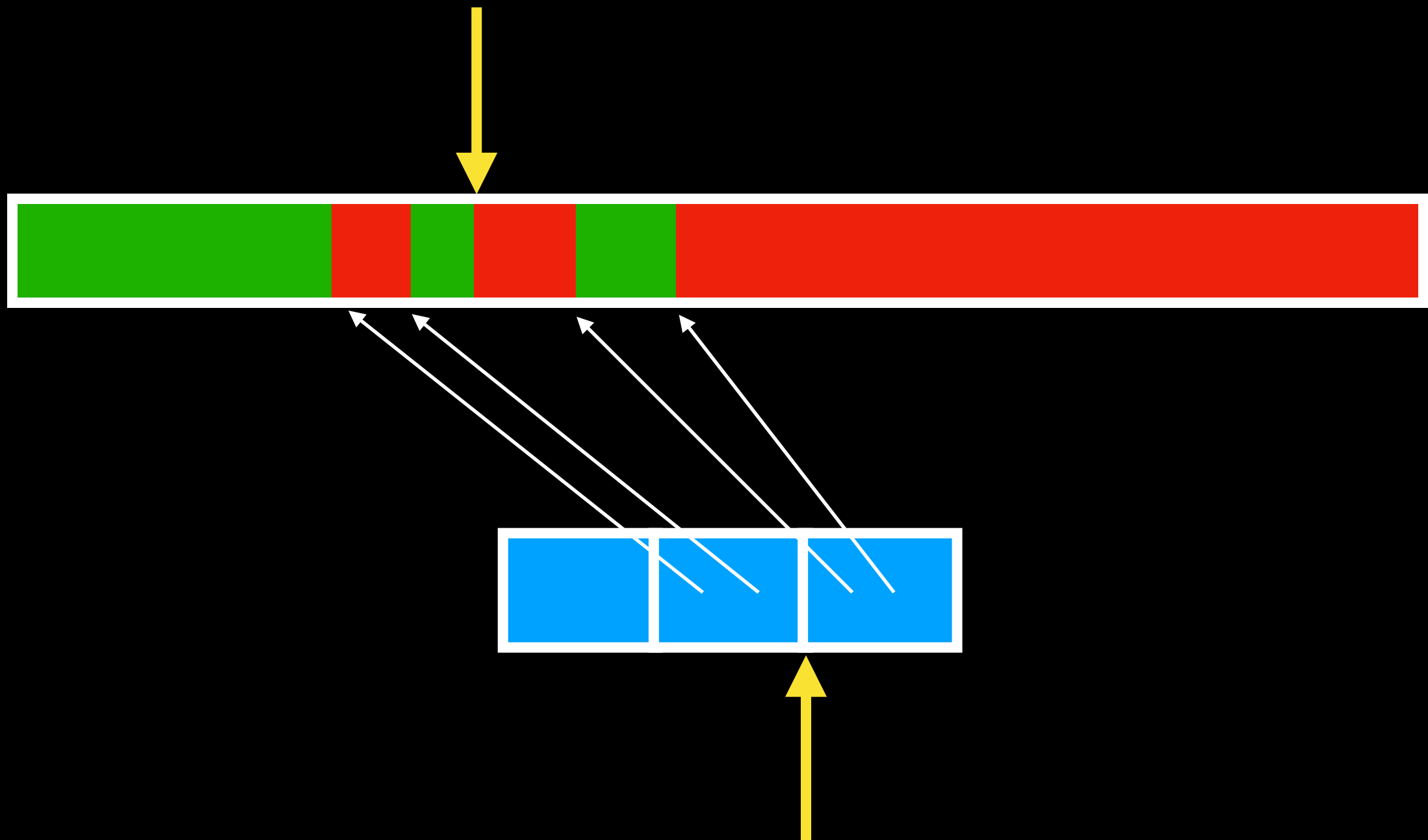
# Clean-up

```
RndIt mid = q.midpoint();
auto rp = std::partition(remain.begin(), remain.end(),
                        [mid](auto& b){ return b.begin() < mid; });
std::sort(remain.begin(), rp,
          [](auto& b0, auto& b1){ return b1.begin() < b0.begin(); });
for (auto it(remain.begin()); it != rp; ++it) {
    mid -= it->end() - it->begin();
    swap_ranges_helper(it->begin(), it->end(), mid);
}
std::sort(rp, remain.end(),
          [](auto& b0, auto& b1){ return b0.begin() < b1.begin(); });
for (auto it(rp); it != remain.end(); ++it) {
    swap_ranges_helper(mid, mid + (it->end() - it->begin()), it->begin());
    mid += it->end() - it->begin();
}
return mid;
```

# Clean-up

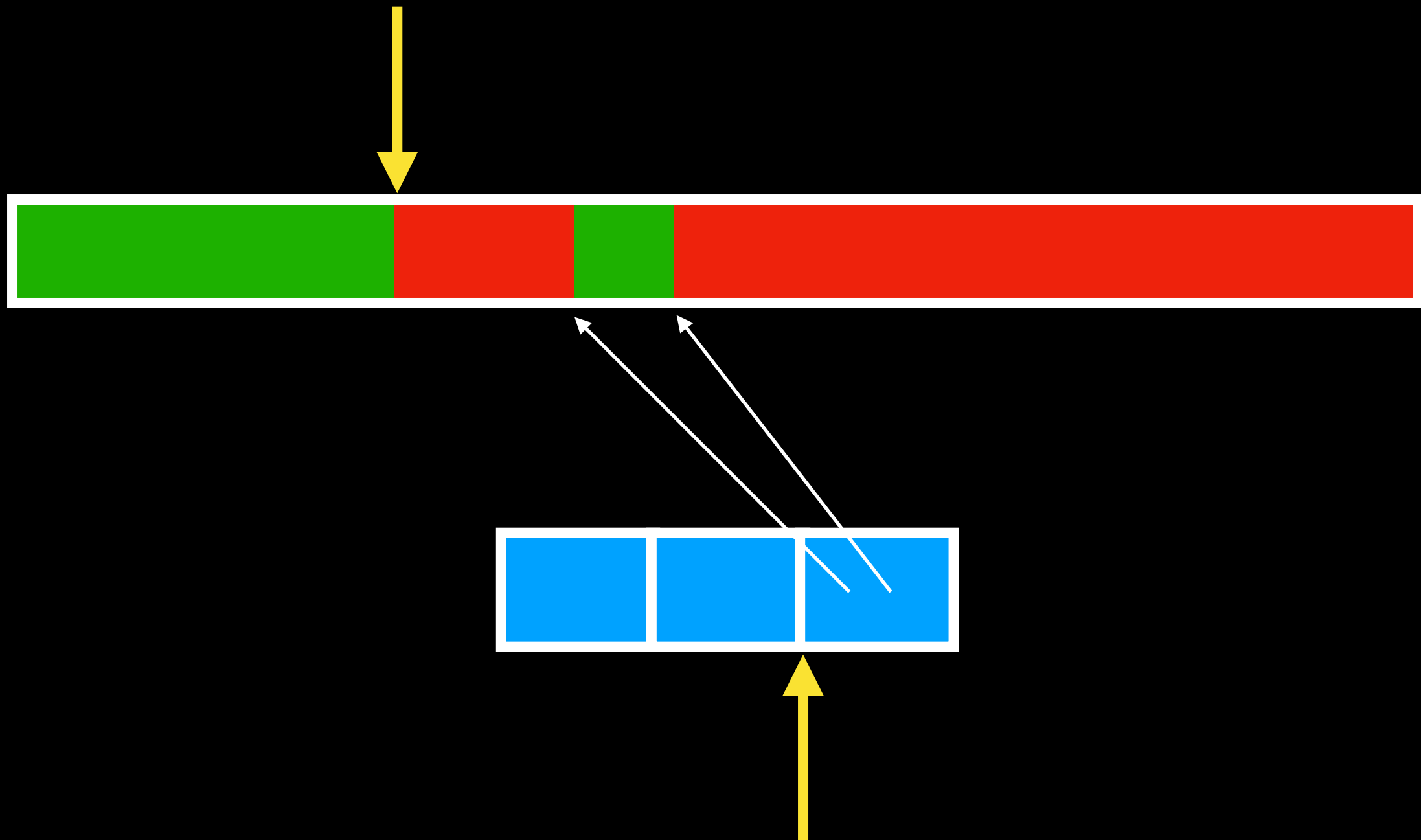


# Clean-up





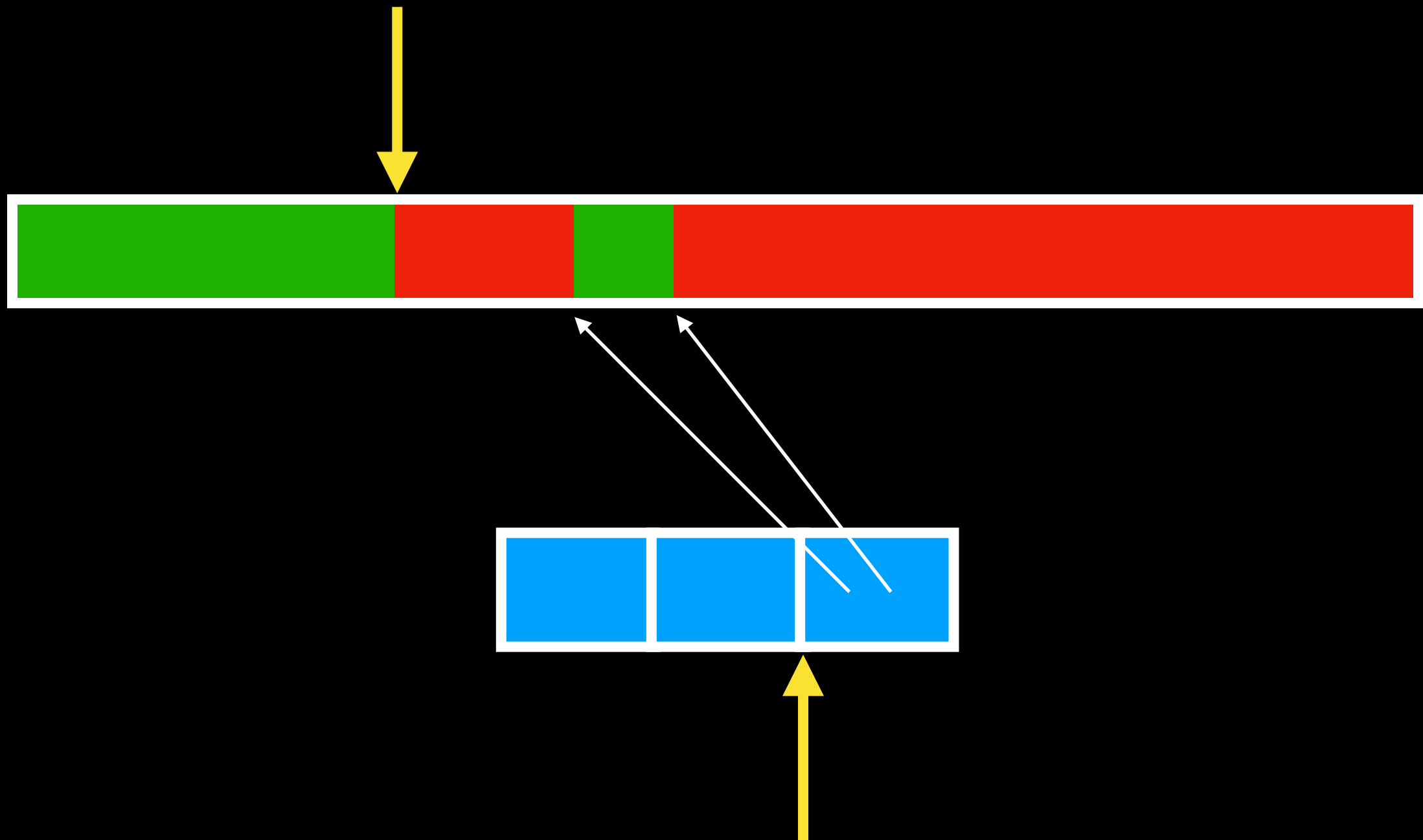
# Clean-up



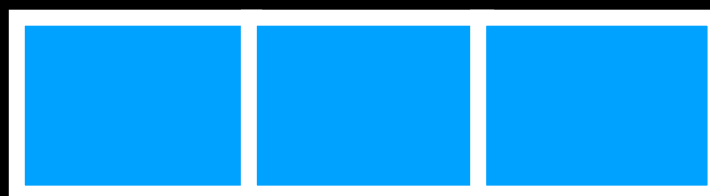
# Clean-up

```
RndIt mid = q.midpoint();
auto rp = std::partition(remain.begin(), remain.end(),
                        [mid](auto& b){ return b.begin() < mid; });
std::sort(remain.begin(), rp,
          [](auto& b0, auto& b1){ return b1.begin() < b0.begin(); });
for (auto it(remain.begin()); it != rp; ++it) {
    mid -= it->end() - it->begin();
    swap_ranges_helper(it->begin(), it->end(), mid);
}
std::sort(rp, remain.end(),
          [](auto& b0, auto& b1){ return b0.begin() < b1.begin(); });
for (auto it(rp); it != remain.end(); ++it) {
    swap_ranges_helper(mid, mid + (it->end() - it->begin()), it->begin());
    mid += it->end() - it->begin();
}
return mid;
```

# Clean-up



# Clean-up



# Clean-up

```
RndIt mid = q.midpoint();
auto rp = std::partition(remain.begin(), remain.end(),
                        [mid](auto& b){ return b.begin() < mid; });
std::sort(remain.begin(), rp,
          [](auto& b0, auto& b1){ return b1.begin() < b0.begin(); });
for (auto it(remain.begin()); it != rp; ++it) {
    mid -= it->end() - it->begin();
    swap_ranges_helper(it->begin(), it->end(), mid);
}
std::sort(rp, remain.end(),
          [](auto& b0, auto& b1){ return b0.begin() < b1.begin(); });
for (auto it(rp); it != remain.end(); ++it) {
    swap_ranges_helper(mid, mid + (it->end() - it->begin()), it->begin());
    mid += it->end() - it->begin();
}
return mid;
```

# Results

- relative time taken compared to a benchmark
  - benchmark: `std::partition` - the blue line
- high values  $\Rightarrow$  slow, low values  $\Rightarrow$  fast
- x-axis: roughly log scale
- machine: 64 cores, 96GB memory (+16GB on chip)  
Intel Knights Landing

# Results <sub>2</sub>

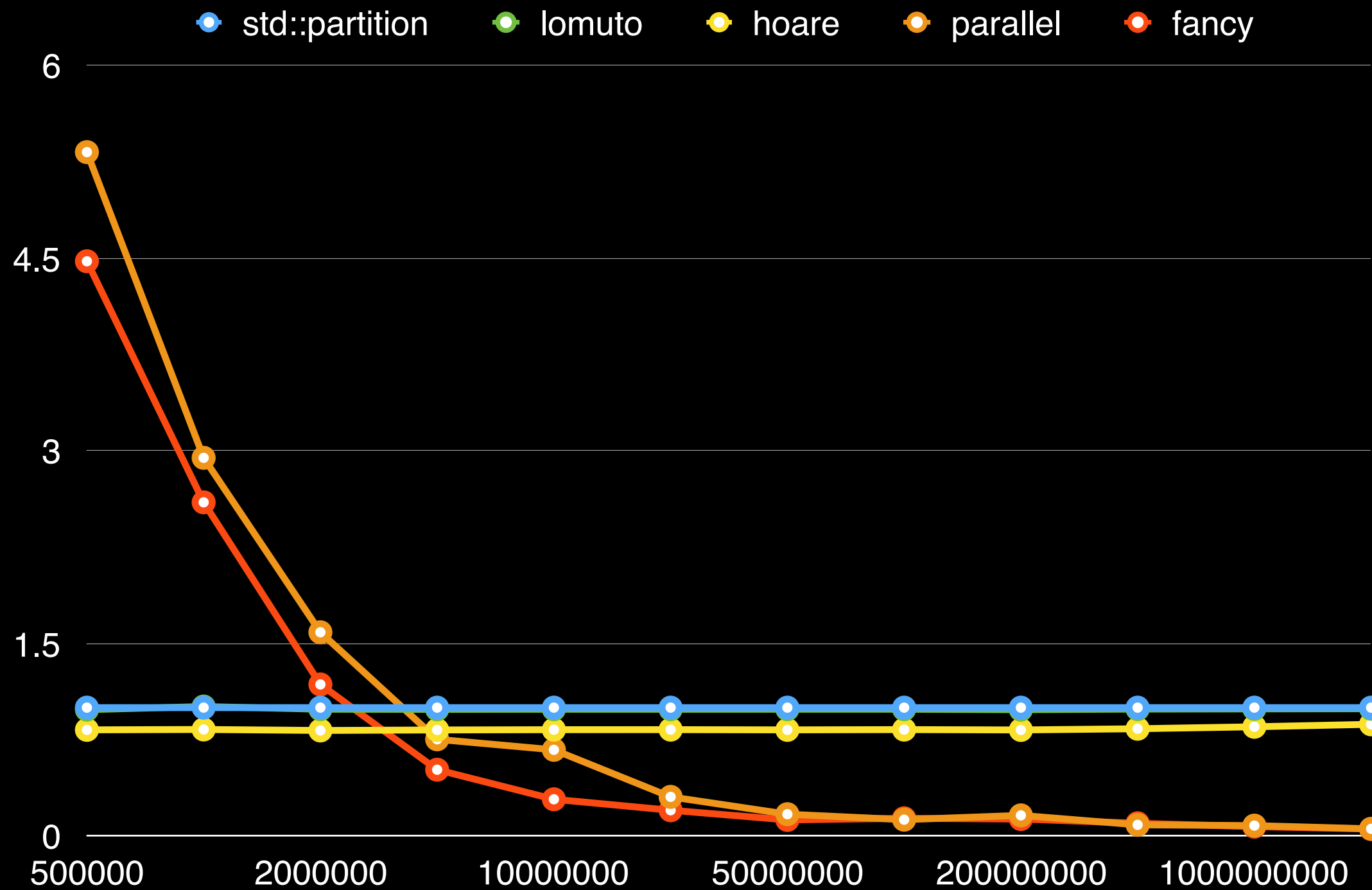


## 2

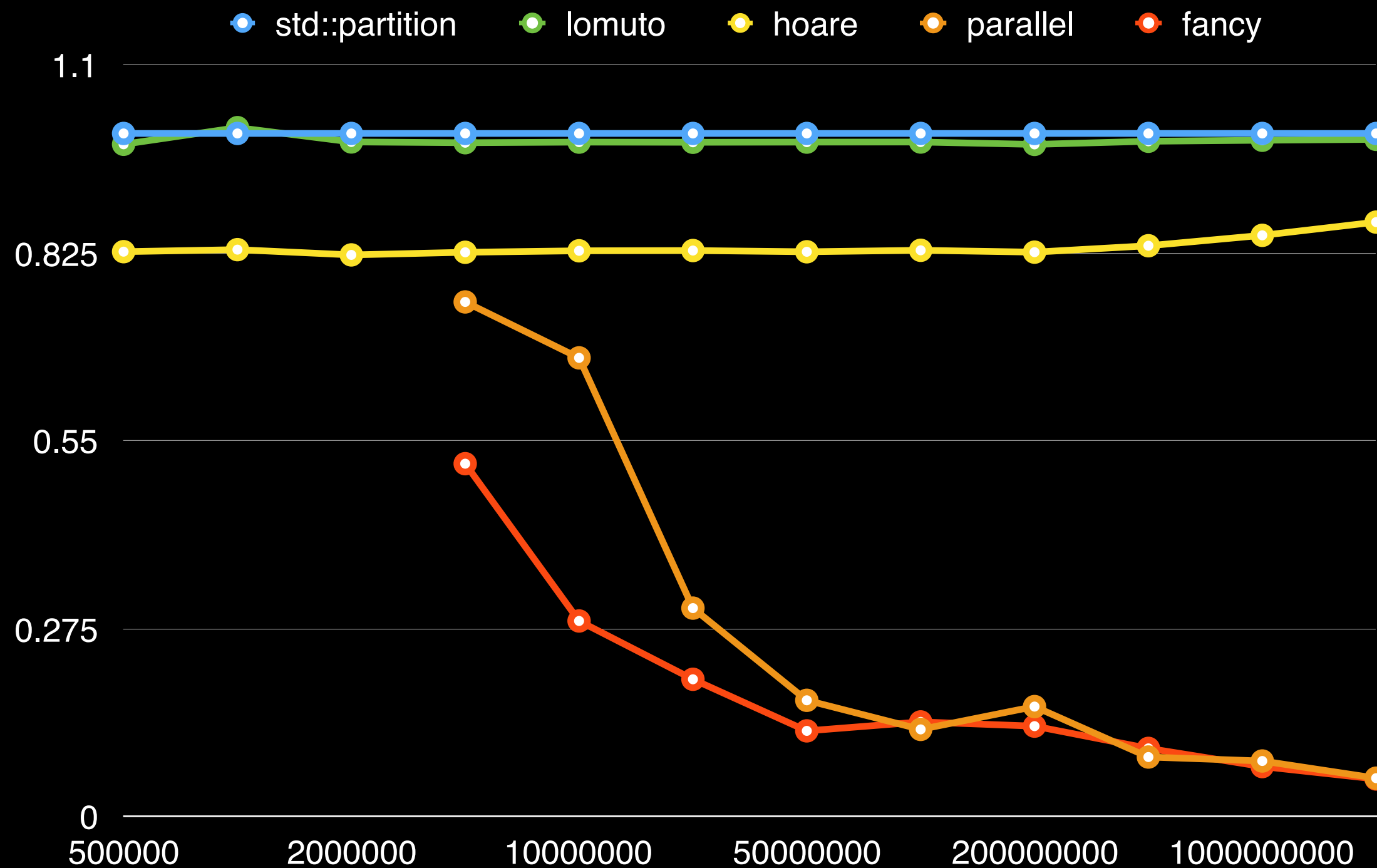




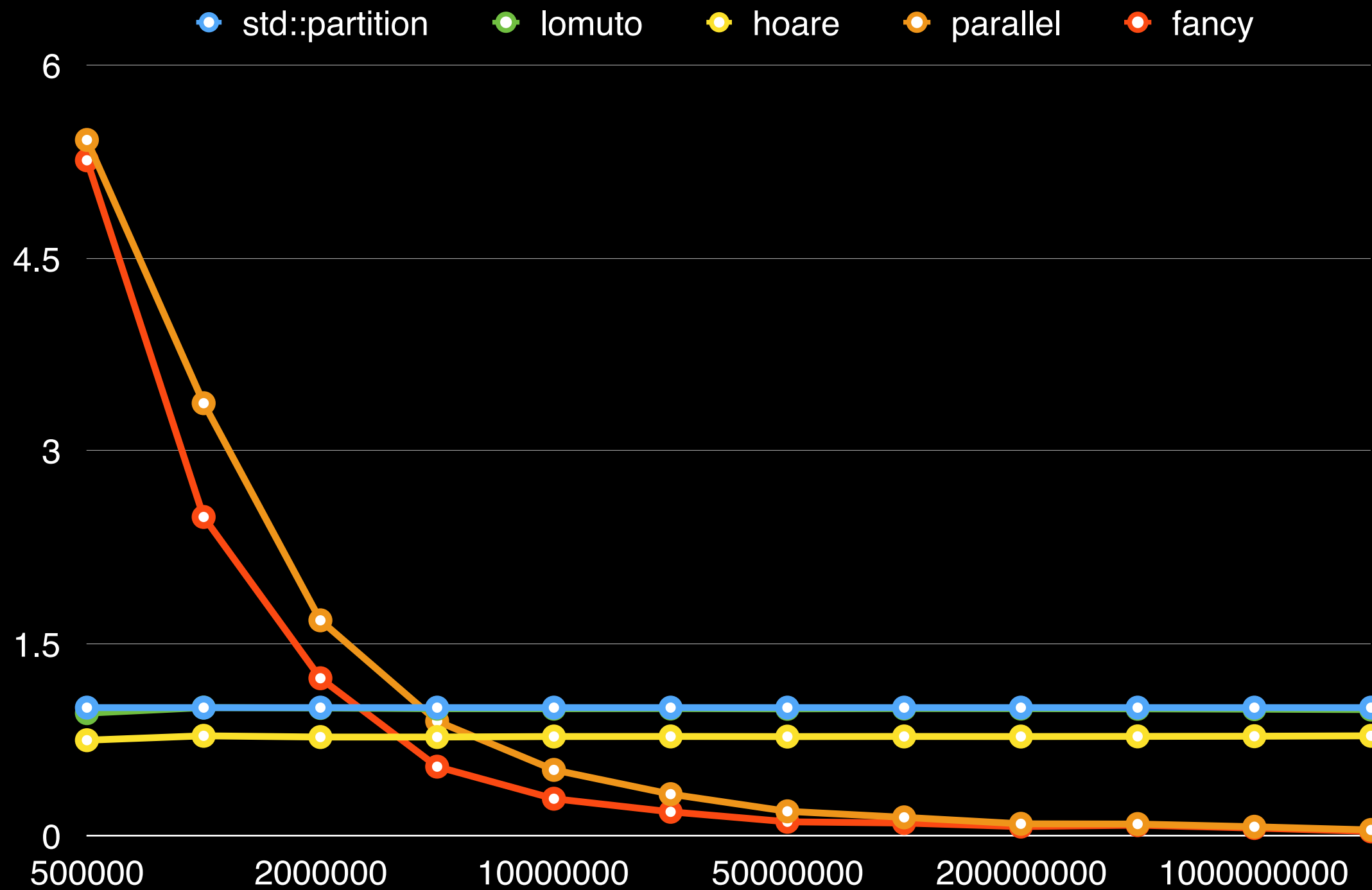
# Results <sup>20</sup>



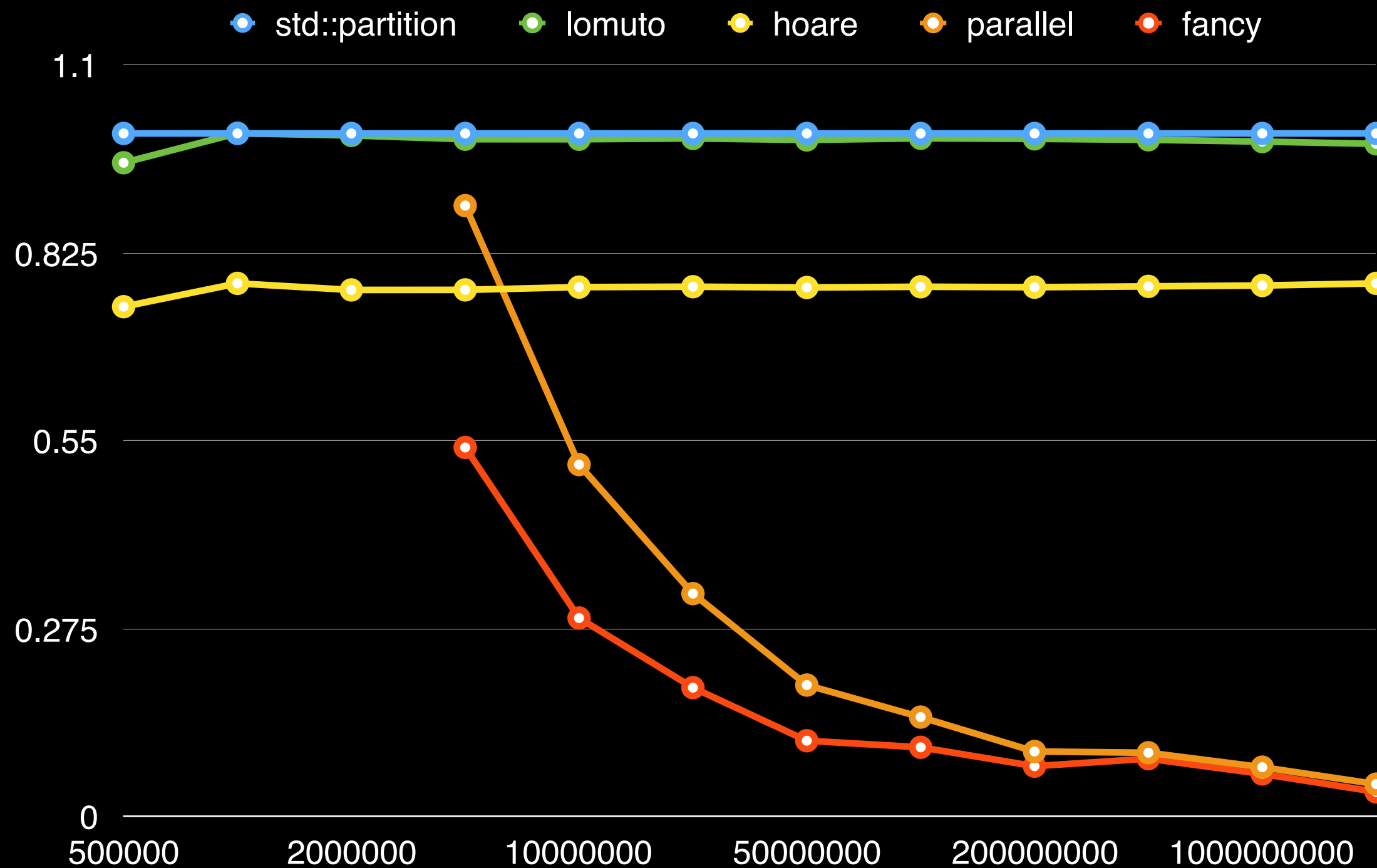
# Results <sup>20</sup>



# Results 100



# Results 100



# Improvements

- handling of smaller ranges
  - engage fewer threads with sufficiently sized blocks?
  - create tasks more clever?
- do clean-up from multiple threads
- extract a continuation/non-waiting form of the algorithm

# Summary

- quite a bit of administrative work around the algorithm
- need an idea how to separate the processing
- overall more total work than sequential algorithm
- on larger ranges faster by using multiple processors