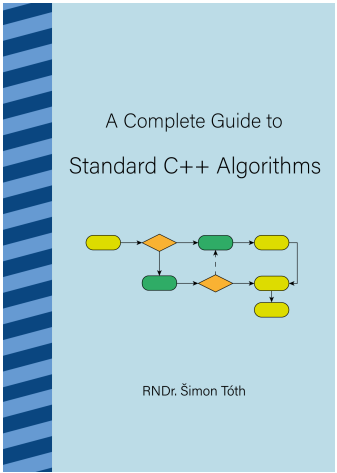
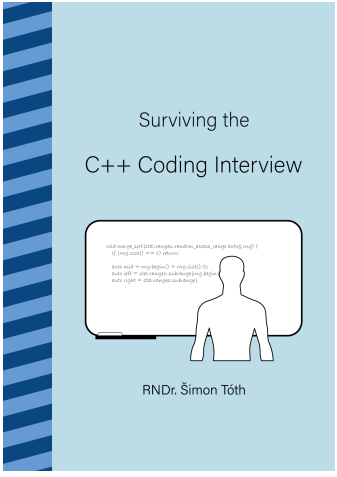
Patterns of interview solutions

Šimon Tóth

November 13, 2023



- Free on GitHub: HappyCerberus/book-cpp-algorithms
- Donate to EFF on LeanPub: leanpub.com/cpp-algorithms-guide



- Source on GitHub:
  HappyCerberus/cpp-coding-interview
- Community version free,
  or donate to EFF on LeanPub:
  leanpub.com/cpp-coding-interview

- linkedin.com/in/simontoth
- hachyderm.io/@simontoth
- simontoth.substack.com
- medium.com/@simontoth

Daily bit(e) of C++

- linkedin.com/in/simontoth
- hachyderm.io/@simontoth
- simontoth.substack.com
- medium.com/@simontoth

Starting 1st December, the series will switch to Advent of Code 2023 solutions.

What is the point of this problem?

What is the point of the solution?

Making good use of the standard library

Algorithms

Algorithms

- `all_of`
- `any_of`
- `none_of`
- `for_each`
- `for_each_n`
- `count`
- `count_if`
- `mismatch`
- `find`
- `find_if`
- `find_if_not`
- `ranges::find_last`
- `ranges::find_last_if`
- `ranges::find_last_if_not`
- `find_end`
- `find_first_of`
- `adjacent_find`
- `search`
- `search_n`
- `ranges::starts_with`
- `ranges::ends_with`
- `copy`
- `copy_if`
- `copy_n`
- `copy_backward`
- `move`
- `move_backward`
- `fill`
- `fill_n`
- `transform`
- `generate`
- `generate_n`
- `remove`
- `remove_if`
- `remove_copy`
- `remove_copy_if`
- `replace`
- `replace_if`
- `replace_copy`
- `replace_copy_if`
- `swap`
- `swap_ranges`
- `iter_swap`
- `reverse`
- `reverse_copy`
- `rotate`
- `rotate_copy`
- `shift_left`
- `shift_right`
- `shuffle`
- `sample`
- `unique`
- `unique_copy`
- `is_partitioned`
- `partition`
- `partition_copy`
- `stable_partition`

Algorithms

- partition_point
- is_sorted
- is_sorted_until
- sort
- partial_sort
- partial_sort_copy
- stable_sort
- nth_element
- lower_bound
- upper_bound
- binary_search
- equal_range
- merge
- inplace_merge
- includes
- set_difference
- set_intersection
- set_symmetric_difference
- set_union
- is_heap
- is_heap_until
- make_heap
- push_heap
- pop_heap
- sort_heap
- max
- max_element
- min
- min_element
- minmax
- minmax_element
- clamp
- equal
- lexicographical_compare
- lexicographical_compare_three_way
- is_permutation
- next_permutation
- prev_permutation
- iota
- accumulate
- inner_product
- adjacent_difference
- partial_sum
- reduce
- exclusive_scan
- inclusive_scan
- transform_reduce
- transform_exclusive_scan
- transform_inclusive_scan
- ranges::fold_left
- ranges::fold_left_first
- ranges::fold_right
- ranges::fold_right_last
- ranges::fold_left_with_iter
- ranges::fold_left_first_with_iter
- uninitialized_copy
- uninitialized_copy_n

Algorithms

- `uninitialized_fill`
- `uninitialized_fill_n`
- `uninitialized_move`
- `uninitialized_move_n`
- `uninitialized_default_construct`
- `uninitialized_default_construct_n`
- `uninitialized_value_construct`
- `uninitialized_value_construct_n`
- `destroy`
- `destroy_n`
- `destroy_at`
- `construct_at`

Iterators and ranges

Iterators and ranges

```
int data[] = {1,2,3,4,5};

int* begin = data;
int* end = begin + 5;

for (int* it = begin; it != end; ++it) {
}
```

Iterators and ranges

- contiguous blocks of memory
  `std::vector`, `std::array`
- chunks of contiguous memory
  `std::deque`
- linked-lists and trees
  `std::list`, `std::map`, `std::unordered_map`
- streams
  `std::ostream`, `std::istream`

The 126 algorithms

For-each 2/126

- `for_each`
- `for_each_n`

```cpp
std::vector<int> data{1,2,3,4,5};

std::for_each(data.begin(), data.end(),
    [](auto &e) { });

std::for_each_n(data.begin(), 3,
    [](auto &e) { });
```

Uninitialized memory algorithms 16/126

- `uninitialized_copy`
- `uninitialized_copy_n`
- `uninitialized_fill`
- `uninitialized_fill_n`
- `uninitialized_move`
- `uninitialized_move_n`
- `uninitialized_default_construct`
- `uninitialized_default_construct_n`
- `uninitialized_value_construct`
- `uninitialized_value_construct_n`
- `destroy`
- `destroy_n`

- `construct_at`
- `destroy_at`

Heap algorithms 22/126

- `is_heap`
- `is_heap_until`
- `make_heap`
- `push_heap`
- `pop_heap`
- `sort_heap`

Swaps 25/126

- `swap`
- `swap_ranges`
- `iter_swap`

```
void func(auto& a, auto& b) {
    using std::swap;
    swap(a,b);
}
```

```
void func(auto& a, auto& b) {
    using std::swap;
    swap(a,b);
}

void func_ranges(auto& a, auto& b) {
    std::ranges::swap(a,b);
}
```

- `sort`
- `stable_sort`

```cpp
struct Record {
    std::string name;
    uint64_t rank;
};

std::vector<Record> data{
    {"Banana", 2},
    {"Watermelon", 1},
    {"Apple", 1},
    {"Pear", 3}
};
```

```cpp
struct Record {
    std::string name;
    uint64_t rank;
};

std::vector<Record> data{
    {"Banana", 2},
    {"Watermelon", 1},
    {"Apple", 1},
    {"Pear", 3}
};
```

```cpp
std::stable_sort(data.begin(), data.end(),
    [](const auto& l, const auto& r) {
        return l.name < r.name;
    });
// Apple, Banana, Pear, Watermelon

std::stable_sort(data.begin(), data.end(),
    [](const auto& l, const auto& r) {
        return l.rank < r.rank;
    });
// Apple, Watermelon, Banana, Pear
```

```
std::stable_sort(data.begin(), data.end(),
    [](const auto& l, const auto& r) {
        return l.rank < r.rank;
    });
```

```
std::stable_sort(data.begin(), data.end(),
    [](const auto& l, const auto& r) {
        return l.rank < r.rank;
    });

std::ranges::stable_sort(data,
    [](const auto& l, const auto& r) {
        return l.rank < r.rank;
    });
```

```cpp
std::stable_sort(data.begin(), data.end(),
    [](const auto& l, const auto& r) {
        return l.rank < r.rank;
    });

std::ranges::stable_sort(data,
    [](const auto& l, const auto& r) {
        return l.rank < r.rank;
    });

std::ranges::stable_sort(data,
    std::less<>{}, &Record::rank);
```

- `partial_sort`
- `partial_sort_copy`

```cpp
std::list<int> data{4,1,5,3,2};
std::vector<int> out(3);

std::ranges::partial_sort_copy(data, out);
// out == {1,2,3}
```

```
std::list<int> data{4,1,5,3,2};
std::vector<int> out(3);

std::ranges::partial_sort_copy(data, out);
// out == {1,2,3}

const std::vector<int> immutable{4,1,5,3,2};
std::ranges::partial_sort_copy(immutable, out);
// out == {1,2,3}
```

- `is_sorted`
- `is_sorted_until`

- `partition`
- `partition_copy`
- `stable_partition`

Partitions 36/126

- `is_partitioned`
- `partition_point`

Nth element 37/126

- `nth_element`

```cpp
std::vector<int> data{8,5,7,4,2,1,9,3,6,};

std::ranges::nth_element(data, data.begin()+4);
// {2,1,3,4,5,7,9,8,6}
```

Fast operations on sorted ranges 41/126

- `lower_bound`
- `upper_bound`
- `equal_range`
- `binary_search`

Set operations 46/126

- `set_difference`
- `set_intersection`
- `set_symmetric_difference`
- `set_union`
- `includes`

Merging 48/126

- `merge`
- `inplace_merge`

```
void merge_sort(auto begin, auto end) {
    if (end-begin < 2) return;

    auto mid = begin + (end-begin)/2;
    merge_sort(begin, mid);
    merge_sort(mid, end);
    std::ranges::inplace_merge(begin, mid, end);
}
```

Generators 53/126

- `iota`
- `fill`
- `fill_n`
- `generate`
- `generate_n`

```cpp
std::vector<int> data;
std::fill_n(std::back_inserter(data), 7, 42);

std::multiset<int> set;
std::generate_n(std::inserter(set, set.end()), 13,
    [a=0,b=1] mutable {
        return std::exchange(a, std::exchange(b, a+b));
    });
```

```
std::multiset<int> set;
std::generate_n(std::inserter(set, set.end()), 13,
    [a=0,b=1] mutable {
        return std::exchange(a, std::exchange(b, a+b));
    });
```

```
std::multiset<int> set;
std::generate_n(std::inserter(set, set.end()), 13,
    [a=0,b=1] mutable {
        int result = a;
        a = std::exchange(b, a+b);
        return result;
    });
```

```
std::multiset<int> set;
std::generate_n(std::inserter(set, set.end()), 13,
    [a=0,b=1] mutable {
        int result = a;
        int prev = b;
        b = a+b;
        a = prev;
        return result;
    });
```

```
std::list<int> nodes{1,2,3,4,5,6,7,8,9};
std::vector<std::list<int>::iterator> indirect(nodes.size());

std::iota(indirect.begin(), indirect.end(), nodes.begin());
```

Reductions 63/126

- `accumulate`
- `inner_product`
- `reduce`
- `transform_reduce`
- `ranges::fold_left`
- `ranges::fold_left_first`
- `ranges::fold_right`
- `ranges::fold_right_last`
- `ranges::fold_left_with_iter`
- `ranges::fold_left_first_with_iter`

```cpp
std::vector<int> data{1,2,3,4,5,6};

int r1 = std::ranges::fold_left(data, 10, std::plus<>{});
// r1 == 31

std::optional<int> r2 = std::ranges::fold_left_first(
    data, std::plus<>{});
// r2.value() == 21
```

Partial reductions 68/126

- `partial_sum`
- `exclusive_scan`
- `inclusive_scan`
- `transform_exclusive_scan`
- `transform_inclusive_scan`

Boolean reductions 71/126

- `all_of`
- `any_of`
- `none_of`

- `max`
- `min`
- `minmax`
- `max_element`
- `min_element`
- `minmax_element`
- `clamp`

```cpp
int low = 0;
int high = 20;

int r1 = std::clamp(14, low, high);
// r1 == 14

int r2 = std::clamp(-10, low, high);
// r2 == 0
```

In-place mutations 83/126

- `remove`
- `remove_if`
- `replace`
- `replace_if`
- `unique`

```cpp
std::vector<int> data{1, 2, 3, 1, 2};

auto end = std::remove(data.begin(), data.end(), 1);
// {data.begin(), end} == {2,3,2}
```

Re-ordering 88/126

- `reverse`
- `rotate`
- `shift_left`
- `shift_right`
- `shuffle`

Comparisons 92/126

- `equal`
- `mismatch`
- `lexicographical_compare`
- `lexicographical_compare_three_way`

Permutations 95/126

- `next_permutation`
- `prev_permutation`
- `is_permutation`

Single-element search 103/126

- `find`
- `find_if`
- `find_if_not`
- `ranges::find_last`
- `ranges::find_last_if`
- `ranges::find_last_if_not`
- `count`
- `count_if`

Range search 109/126

- `find_first_of`
- `search`
- `find_end`
- `search_n`
- `ranges::starts_with`
- `ranges::ends_with`

```cpp
std::vector<int> data{1, 2, 3, 1, 2};
std::vector<int> needle{5, 4, 3};

auto it = std::ranges::find_first_of(data, needle);
// *it == 3
```

```cpp
std::vector<int> data{1,1,2,1,1,1,2,1,1,1};
auto it = std::search_n(data.begin(), data.end(),
    3, 1); // 3 copies of 1
// it == data.begin()+3
```

Copy 116/126

- `copy`
- `copy_if`
- `copy_n`
- `copy_backward`
- `move`
- `move_backward`
- `transform`

Copy 123/126

- `remove_copy`
- `remove_copy_if`
- `replace_copy`
- `replace_copy_if`
- `unique_copy`
- `reverse_copy`
- `rotate_copy`

Adjacent 125/126

- `adjacent_difference`
- `adjacent_find`

Sample 126/126

- `sample`

```cpp
std::vector<int> data{1,2,3,4,5,6,7,8,9};
std::vector<int> out;
std::ranges::sample(data, std::back_inserter(out),
    4, std::mt19937 {std::random_device{}()});
// out == random sample of 4 elements
```

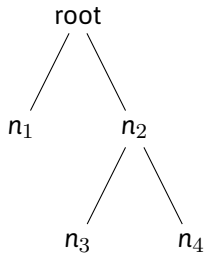
Algorithms

- 126 algorithms
- only 20 logical groups
- most have reasonable names

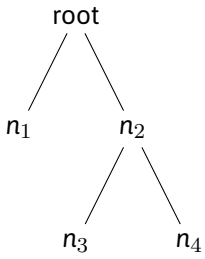
Analyzing a problem from a different perspective

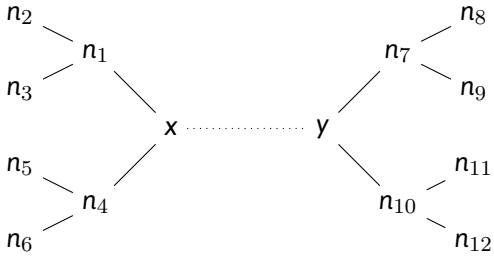
Sum of distances to all nodes

- Given a tree with n nodes, represented as a graph using a neighbourhood map, calculate the sum of distances to all nodes for each node.


root
n_1
n_2
n_3
n_4

```
void post_order(int node, int parent,
    const std::unordered_multimap<int,int>& neighbours,
    std::vector<int>& distances,
    std::vector<int>& node_count) {
    // If there are no children we have zero distance and one node.
    distances[node] = 0;
    node_count[node] = 1;

    auto [begin, end] = neighbours.equal_range(node);
    for (auto [from, to] : std::ranges::subrange(begin, end)) {
        // Avoid looping back to the node we came from.
        if (to == parent) continue;
        // post_order traversal, visit children first
        post_order(to, node, neighbours, distances, node_count);
        // accumulate number of nodes and distances
        distances[node] += distances[to] + node_count[to];
        node_count[node] += node_count[to];
    }
}
```


root
n_1
n_2
n_3
n_4


n_2
n_1
n_3
x
y
n_8
n_7
n_9
n_5
n_4
n_6
n_{11}
n_{10}
n_{12}

The formula

$$
\begin{aligned}
total(x) &= distance(x) + distance(y) + node\_count(y) \\
total(y) &= distance(y) + distance(x) + node\_count(x) \\
total(x) - total(y) &= node\_count(y) - node\_count(x)
\end{aligned}
$$

The formula

$$
\begin{aligned}
total(x) &= distance(x) + distance(y) + node\_count(y) \\
total(y) &= distance(y) + distance(x) + node\_count(x) \\
total(x) - total(y) &= node\_count(y) - node\_count(x)
\end{aligned}
$$

$$
\begin{aligned}
total(child) &= total(parent) + node\_count(parent) - node\_count(child) \\
total(child) &= total(parent) + (nodes - node\_count(child)) - node\_count(child) \\
total(child) &= total(parent) + nodes - 2 * node\_count(child)
\end{aligned}
$$

```
void pre_order(int node, int parent,
    const std::unordered_multimap<int,int>& neighbours,
    const std::vector<int>& distances,
    const std::vector<int>& node_count,
    std::vector<int>& result) {
    // For the root node the we have already calculated the value.
    if (parent == -1) {
        result[node] = distances[node];
    } else {
        // Otherwise, we can calculate the result from the parent,
        // because in pre-order we visit the parent before the children.
        result[node] = result[parent] + result.size() - 2*node_count[node];
    }
    // Now visit any children.
    auto [begin, end] = neighbours.equal_range(node);
    for (auto [from, to] : std::ranges::subrange(begin, end)) {
        if (to == parent) continue;
        pre_order(to, node, neighbours, distances, node_count, result);
    }
}
```

Analyzing a problem from a different perspective

- always consider other angles that can give you a solution
- get a different perspective from a colleague (or a rubber duck)
- AI chatbots are fairly good rubber ducks

Optimizing code by removing duplicate work

Longest palindromic substring

- given a string as `std::string_view`
- find the length of the longest palindromic substring

Longest palindromic substring

- given a string as `std::string_view`
- find the length of the longest palindromic substring

Examples:

- `longest_palindrome("") == 0`
- `longest_palindrome("a") == 1`
- `longest_palindrome("aba") == 3`
- `longest_palindrome("abba") == 4`
- `longest_palindrome("ababc") == 3`

```
int64_t longest_palindrome(std::string_view text) {
  int64_t max = 0;
  for (int64_t i = 0; i < std::ssize(text); ++i) {
```

```
int64_t longest_palindrome(std::string_view text) {
  int64_t max = 0;
  for (int64_t i = 0; i < std::ssize(text); ++i) {
    int64_t odd = 0;
    while (i - odd >= 0 && i + odd < std::ssize(text) &&
           text[i - odd] == text[i + odd])
      ++odd;
```

```
int64_t longest_palindrome(std::string_view text) {
  int64_t max = 0;
  for (int64_t i = 0; i < std::ssize(text); ++i) {
    int64_t odd = 0;
    while (i - odd >= 0 && i + odd < std::ssize(text) &&
           text[i - odd] == text[i + odd])
      ++odd;
    int64_t even = 0;
    while (i - even >= 0 && i + 1 + even < std::ssize(text) &&
           text[i - even] == text[i + 1 + even])
      ++even;
```

```
int64_t longest_palindrome(std::string_view text) {
  int64_t max = 0;
  for (int64_t i = 0; i < std::ssize(text); ++i) {
    int64_t odd = 0;
    while (i - odd >= 0 && i + odd < std::ssize(text) &&
           text[i - odd] == text[i + odd])
      ++odd;
    int64_t even = 0;
    while (i - even >= 0 && i + 1 + even < std::ssize(text) &&
           text[i - even] == text[i + 1 + even])
      ++even;
    max = std::max(max, std::max(odd * 2 - 1, even * 2));
  }
  return max;
}
```

```
int64_t longest_palindrome(std::string_view text) {
  int64_t max = 0;
  for (auto it = text.begin(); it != text.end(); ++it) {
```

```
int64_t longest_palindrome(std::string_view text) {
  int64_t max = 0;
  for (auto it = text.begin(); it != text.end(); ++it) {
    auto rev = std::reverse_iterator(it);
    auto next = std::next(it);
```

```
int64_t longest_palindrome(std::string_view text) {
  int64_t max = 0;
  for (auto it = text.begin(); it != text.end(); ++it) {
    auto rev = std::reverse_iterator(it);
    auto next = std::next(it);

    auto [l_odd, r_odd] = std::mismatch(rev, text.rend(), next, text.end());
    auto [l_even, r_even] = std::mismatch(rev, text.rend(), it, text.end());
```

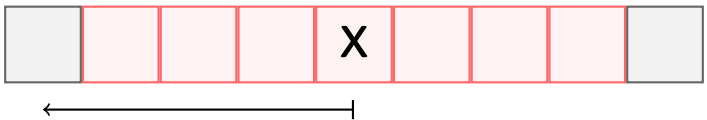
```
int64_t longest_palindrome(std::string_view text) {
  int64_t max = 0;
  for (auto it = text.begin(); it != text.end(); ++it) {
    auto rev = std::reverse_iterator(it);
    auto next = std::next(it);

    auto [l_odd, r_odd] = std::mismatch(rev, text.rend(), next, text.end());
    auto [l_even, r_even] = std::mismatch(rev, text.rend(), it, text.end());
    max = std::max(max, std::max(std::distance(l_odd.base(), r_odd),
                                 std::distance(l_even.base(), r_even)));
  }
  return max;
}
```

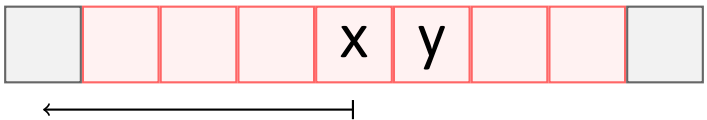
```
int64_t longest_palindrome(std::string_view text) {
  int64_t max = 0;
  for (int64_t i = 0; i < std::ssize(text); ++i) {
    int64_t odd = 0;
    while (i - odd >= 0 && i + odd < std::ssize(text) &&
           text[i - odd] == text[i + odd])
      ++odd;
    int64_t even = 0;
    while (i - even >= 0 && i + 1 + even < std::ssize(text) &&
           text[i - even] == text[i + 1 + even])
      ++even;
    max = std::max(max, std::max(odd * 2 - 1, even * 2));
  }
  return max;
}
```

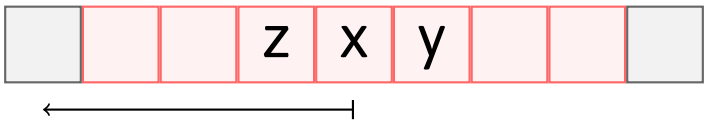
```
auto all_splits(std::string_view &text) {
  return std::views::iota(text.begin(), text.end()) |
    std::views::transform([&text](auto iter) {
      return std::pair{std::ranges::subrange(text.begin(), iter),
                       std::ranges::subrange(iter, text.end())};
    });
};
```

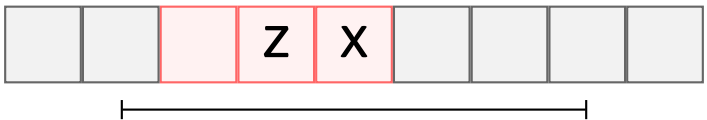
```
int64_t longest_palindrome(std::string_view text) {
  int64_t max = 0;
  auto distance = [](auto result) {
    return std::distance(result.in1.base(), result.in2);
  };
  for (auto [prefix, suffix] : all_splits(text)) {
    auto odd = std::ranges::mismatch(
        prefix | std::views::reverse,
        suffix | std::views::drop(1));
    auto even = std::ranges::mismatch(
        prefix | std::views::reverse,
        suffix);
    max = std::ranges::max({max, distance(odd), distance(even)});
  }
  return max;
}
```

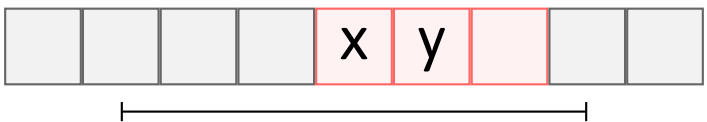

X

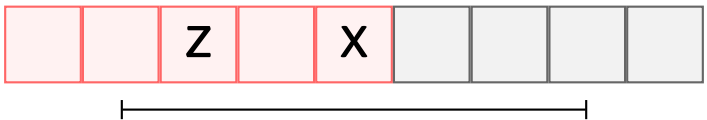

X

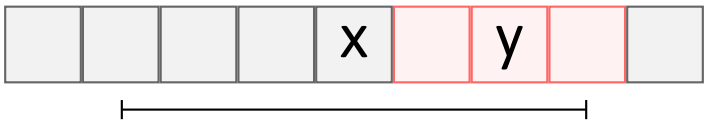

x
y

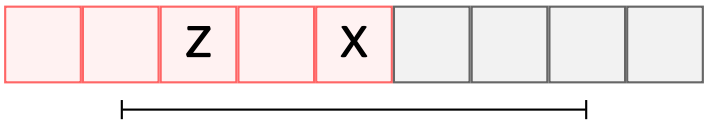

z
x
y

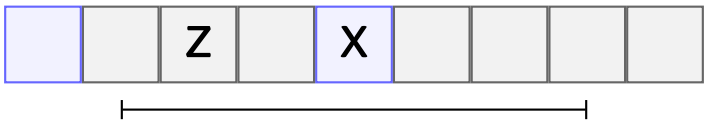

Z
X

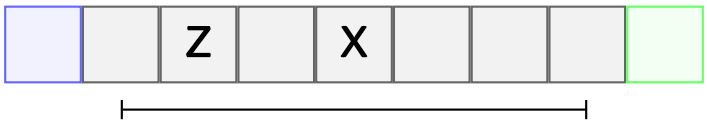

x
y

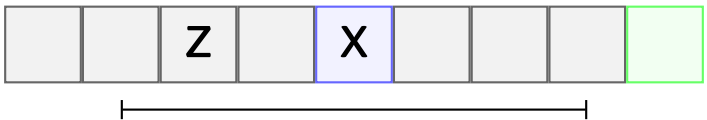

Z
X

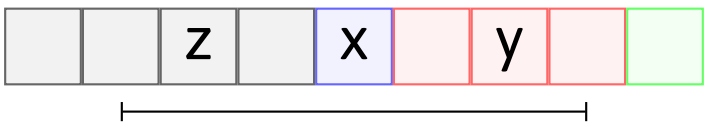

x
y

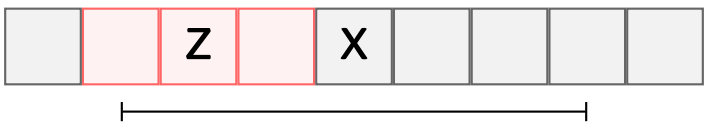

Z
X

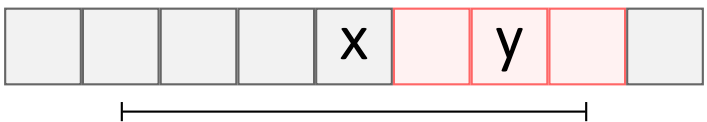

Z
X

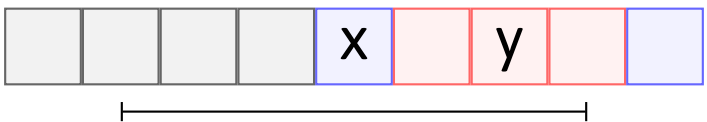

Z
X


Z
X


z
x
y


Z
X


x
y


x
y

Manacher's Algorithm

- expand current palindrome candidate center
- re-use information for reflected palindromes
- if one of the reflected palindromes ends at the boundary, it is the new candidate center
- otherwise, the next candidate center is the first element after the current palindrome

Manacher's Algorithm

- expand current palindrome candidate center
- re-use information for reflected palindromes
- if one of the reflected palindromes ends at the boundary, it is the new candidate center
- otherwise, the next candidate center is the first element after the current palindrome

$O(n)$ time and $O(n)$ space complexity, however, can only deal with odd-length palindromes.

```
std::vector<int> lengths(s.length(), 0);
int64_t radius = 0;
int64_t c = 0;
while (c < std::ssize(s)) {
    // Expand from the current centre until we find non-matching characters
    while (c - (radius+1) >= 0 &&
            c + (radius+1) < std::ssize(s) &&
            s[c - (radius+1)] == s[c + (radius+1)])
        ++radius;
    lengths[c] = radius; // Record the radius

    mirror_information();
}
```

```
    int64_t curr_c = c;
    int64_t curr_r = radius;

    // Precalculate minimum radius for the next center(s)
    ++c;
    radius = 0;
    while (c <= curr_c + curr_r) {
        int64_t mirror = curr_c - (c - curr_c);
        int64_t max_radius = curr_c + curr_r - c;

        // Completely mirrored palindrome
        if (lengths[mirror] < max_radius) {
            lengths[c] = lengths[mirror]; // Reuse
            ++c;
        // Palindrome that extends beyond current palindrome
        } else if (lengths[mirror] > max_radius) {
            lengths[c] = max_radius; // Truncate
            ++c;
        // Palindrome that fits exactly into the boundary
        } else {
            // Can expand but we know that max_radius
            // is already mirrored, i.e. no point in rechecking above
            radius = max_radius;
            break;
        }
    }
```

Reducing duplicate work

- identify where you are processing the same data, or repeating the same operations
- figure out how to re-use existing results/information

Patterns of interview solutions

- know what tools are available in the standard library
- consider different angles when solving problems
- look for redundant work to optimize your algorithm

Thank you

Links

- leanpub.com/cpp-algorithms-guide
- leanpub.com/cpp-coding-interview
- linkedin.com/in/simontoth
- hachyderm.io/@simontoth
- simontoth.substack.com
- medium.com/@simontoth