Finding tandem repeats in genomic data

Projects in Bioinformatics, 10 ECTS
Astrid Christiansen, 201404423
Supervised by Thomas Mailund
Aarhus University

June 2022

Table of Contents

Table of Contents

Introduction

Tandem repeat: $ATCTG \; ATCTG$.

Introduction

Tandem repeat: $ATCTG\ ATCTG$.

- Find all TRs in string x of length n.

Introduction

Tandem repeat: $ATCTG\ \ ATCTG$.

- Find all TRs in string x of length n.
- Suffix tree- vs. suffix array algorithm.

Introduction

Tandem repeat: $ATCTG\ \ ATCTG$.

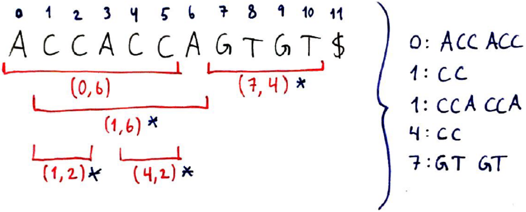
- Find all TRs in string x of length n.
- Suffix tree- vs. suffix array algorithm.
- Both time $O(n \log n + z)$.

Tandem repeats and branching tandem repeats

- Non-branching vs. branching TRs.

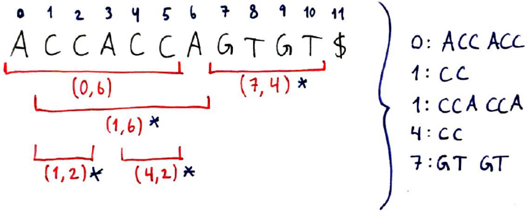
Tandem repeats and branching tandem repeats

- Non-branching vs. branching TRs.



Tandem repeats and branching tandem repeats

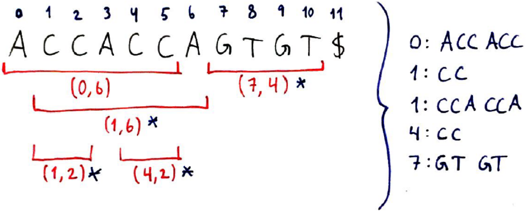
- Non-branching vs. branching TRs.



- Every *non-branching* TR is a *left-rotation* of another TR that starts one place to its right.

Tandem repeats and branching tandem repeats

- Non-branching vs. branching TRs.



- Every *non-branching* TR is a *left-rotation* of another TR that starts one place to its right.
- Find all TRs from branching ones in time $O(z)$.

Table of Contents

Suffix tree

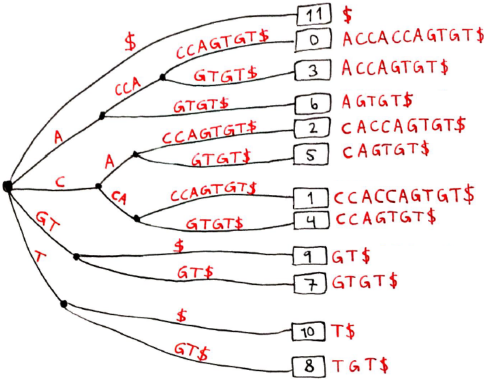
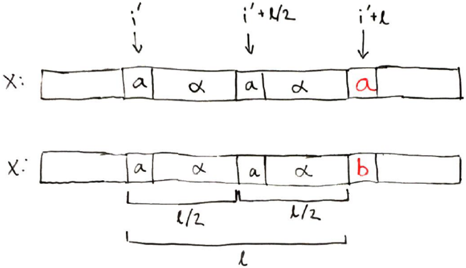
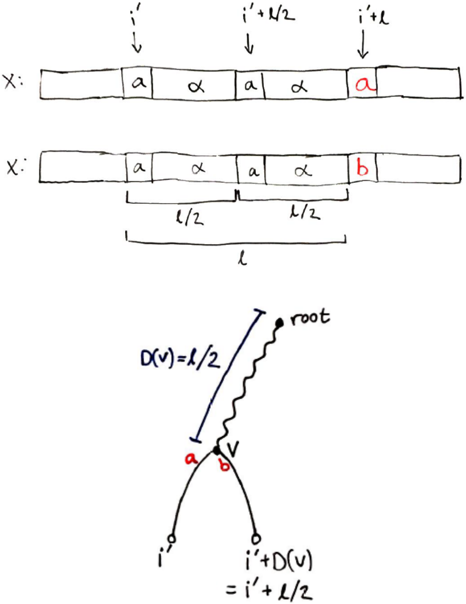


Figure: Suffix tree for string $x = ACCACCAGTGT\$$.

Finding branching TRs



Finding branching TRs



Finding branching TRs

Algorithm BRANCHING-REPEATS(T)

```
for each inner node, v, in T do
    for each leaf, i', in sub-tree of v do
        if leaf i' + D(v) is also in sub-tree of v and x[i'] ≠ x[i' + 2D(v)] then
            report (i', 2D(v))
        end if
    end for
end for
```

Finding branching TRs

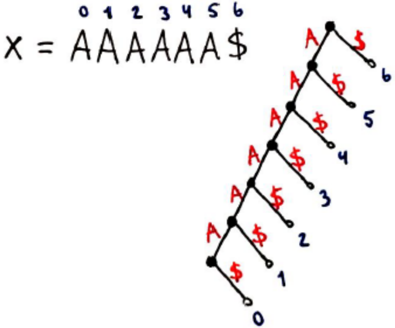
Algorithm BRANCHING-REPEATS(T)

```
for each inner node, v, in T do
    for each leaf, i', in sub-tree of v do
        if leaf i' + D(v) is also in sub-tree of v and x[i'] ≠ x[i' + 2D(v)] then
            report (i', 2D(v))
        end if
    end for
end for
```

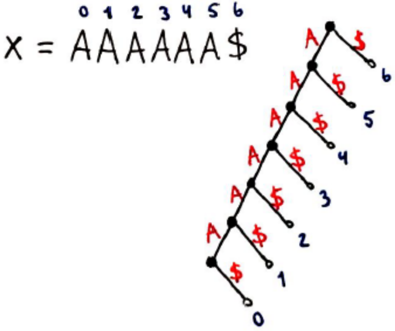
How to determine if leaf is in sub-tree of node v?

- DFS numbering $\Rightarrow$ constant query time.

Running time



Running time



Worst case: $\Theta(n^2)$.

The smaller half trick

Algorithm BRANCHING-REPEATS-SMALLER-HALF(T)

for each inner node, v, in T **do**
    for each leaf, i', in "sub-tree of v except widest(v)" **do**
        if leaf $i' + D(v)$ is also in sub-tree of v **and** $x[i'] \neq x[i' + 2D(v)]$ **then**
            report $(i', 2D(v))$
        end if
        $j' \leftarrow i' - D(v)$
        if leaf j' is in sub-tree of *widest*(v) **and** $x[j'] \neq x[j' + 2D(v)]$ **then**
            report $(j', 2D(v))$
        end if
    end for
end for

The smaller half trick

Algorithm BRANCHING-REPEATS-SMALLER-HALF(T)

```
for each inner node, v, in T do
    for each leaf, i', in "sub-tree of v except widest(v)" do
        if leaf i' + D(v) is also in sub-tree of v and x[i'] ≠ x[i' + 2D(v)] then
            report (i', 2D(v))
        end if
        j' ← i' − D(v)
        if leaf j' is in sub-tree of widest(v) and x[j'] ≠ x[j' + 2D(v)] then
            report (j', 2D(v))
        end if
    end for
end for
```

Smaller half trick: $O(n \log n)$.

The smaller half trick

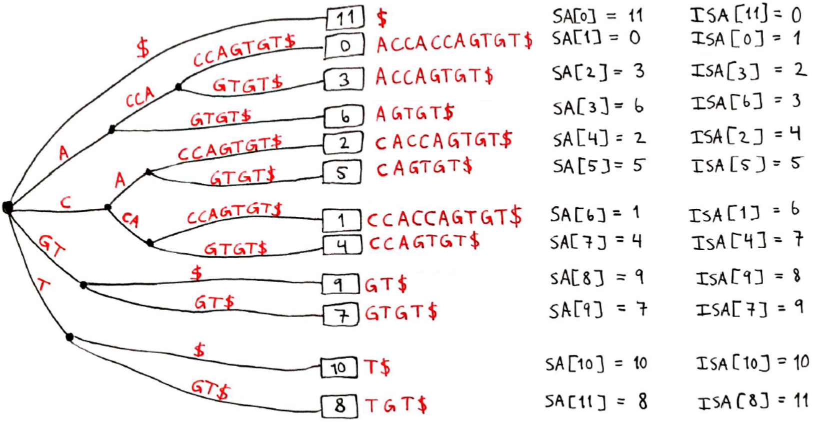
Algorithm BRANCHING-REPEATS-SMALLER-HALF(T)

for each inner node, v, in T **do**
    for each leaf, i', in "sub-tree of v except widest(v)" **do**
        if leaf $i' + D(v)$ is also in sub-tree of v **and** $x[i'] \neq x[i' + 2D(v)]$ **then**
            report $(i', 2D(v))$
        end if
        $j' \leftarrow i' - D(v)$
        if leaf j' is in sub-tree of *widest*(v) **and** $x[j'] \neq x[j' + 2D(v)]$ **then**
            report $(j', 2D(v))$
        end if
    end for
end for

Smaller half trick: $O(n \log n)$.
Total time: $O(n \log n + z)$.

Table of Contents

Suffix array



ℓ-intervals

SA:	LCP:
$	0
ACCACCAGTGT$	0
ACCAGTGT$	4
AGTGT$	1
CACCAGTGT$	0
CAGTGT$	2
CCACCAGTGT$	1
CCAGTGT$	3
GT$	0
GTGT$	2
T$	0
TGT$	1

ℓ-intervals

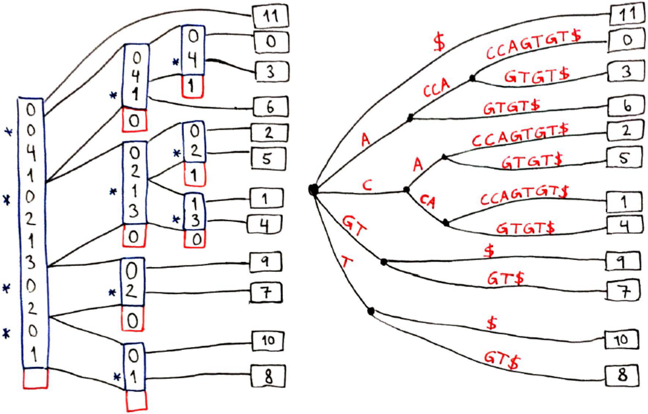
SA:	LCP:
$	0
ACCACCAGTGT$	0
ACCAGTGT$	$\boxed{4}$
AGTGT$	1
CACCAGTGT$	0
CAGTGT$	2
CCACCAGTGT$	1
CCAGTGT$	3
GT$	0
GTGT$	2
T$	0
TGT$	1

$\ell = min_{k \in (i,j)} LCP[k]$ and either

$i = 0$ or $LCP[i] < \ell$ and either

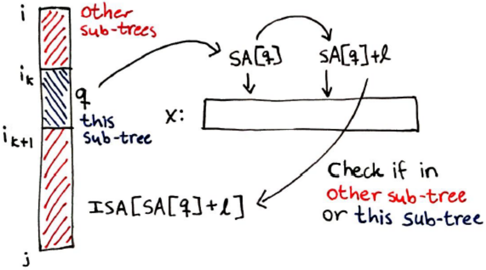
$j = n$ or $LCP[j] < \ell$

ℓ-intervals

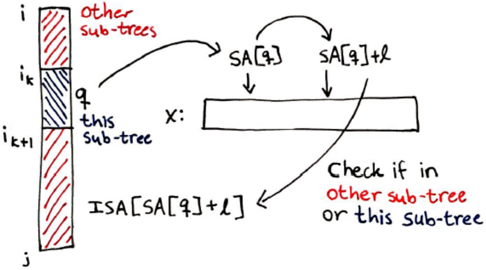


- Use RMQ to find ℓ.

Find TRs using suffix array

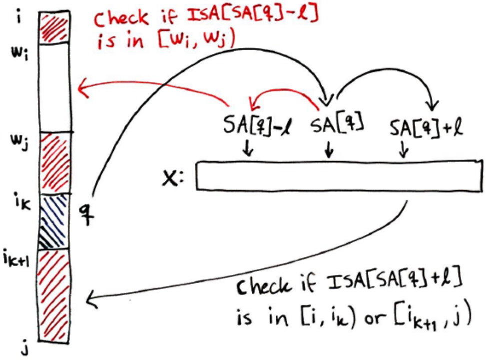


Find TRs using suffix array

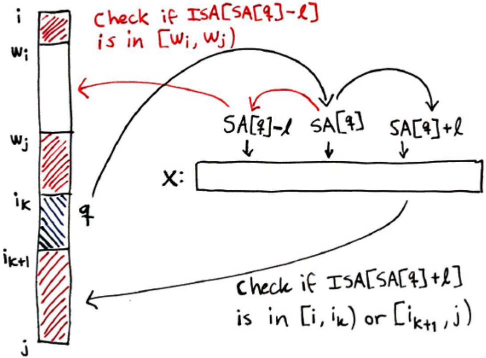


- Again time $\Theta(n^2) \Rightarrow$ use smaller half trick!

Find TRs using suffix array - smaller half trick

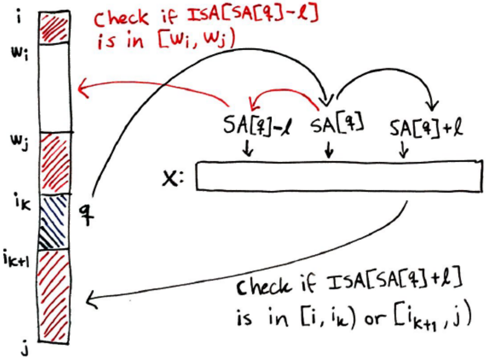


Find TRs using suffix array - smaller half trick



- Time $O(n \log n)$.

Find TRs using suffix array - smaller half trick



- Time $O(n \log n)$.
- Total time $O(n \log n + z)$.

Table of Contents

Implementation: Finding branching TRs

```
def branching_TR_smaller_half(x, sa, lcp):
    isa = construct_isa(sa)
    M = RMQ_preprocess(lcp)
    for (i, j) in get_inner_nodes(lcp, M, 0, len(x)):
        child_nodes = list(get_child_nodes(lcp, M, i, j))
        (w_i, w_j) = widest(child_nodes)
        (_, L) = RMQ(lcp, M, i + 1, j)
        for (ii, jj) in child_nodes:
            if (ii, jj) == (w_i, w_j):
                continue
            for q in valid_isa_index(sa, ii, jj, +L):
                r = isa[sa[q] + L]
                if (i <= r < j) and not (ii <= r < jj):
                    yield (sa[q], 2*L)
            for q in valid_isa_index(sa, ii, jj, -L):
                r = isa[sa[q] - L]
                if w_i <= r < w_j:
                    yield (sa[r], 2*L)
```

Implementation: Finding all TRs

```
def find_all_tandem_repeats(x, branching_TRs):
    for (i, L) in branching_TRs:
        yield (i, L)
        while can_rotate(x, i, L):
            yield (i-1, L)
            i -= 1

def can_rotate(x, i, L):
    return i > 0 and x[i - 1] == x[i + L - 1]
```

Table of Contents

Correctness

```
**********************
TANDEM REPEATS, (index, length), for string:
x: ACCACCAGTGT$
(0, 6): ACC ACC
(1, 2): C C
(1, 6): CCA CCA
(4, 2): C C
(7, 4): GT GT
**********************
```

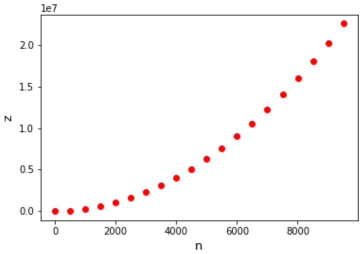
```
**********************
TANDEM REPEATS, (index, length), for string:
x: aaaaaa$
(0, 2): a a
(0, 4): aa aa
(0, 6): aaa aaa
(1, 2): a a
(1, 4): aa aa
(2, 2): a a
(2, 4): aa aa
(3, 2): a a
(4, 2): a a
**********************
```

Correctness

```
*********************
TANDEM REPEATS, (index, length), for string:
x: ACCACCAGTGT$
(0, 6): ACC ACC
(1, 2): C C
(1, 6): CCA CCA
(4, 2): C C
(7, 4): GT GT
*********************
```

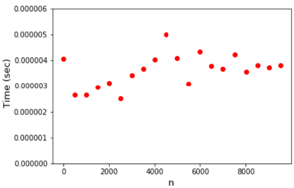
```
*********************
TANDEM REPEATS, (index, length), for string:
x: aaaaaa$
(0, 2): a a
(0, 4): aa aa
(0, 6): aaa aaa
(1, 2): a a
(1, 4): aa aa
(2, 2): a a
(2, 4): aa aa
(3, 2): a a
(4, 2): a a
*********************
```

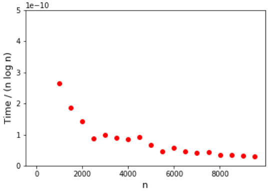
Number of found TRs for $A^n\$$:



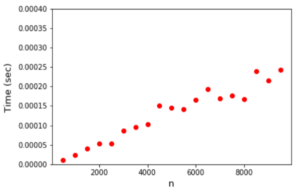
Running time: Worst case input

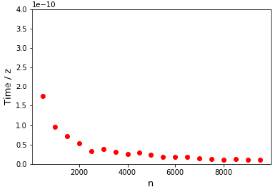
Finding branching tandem repeats:





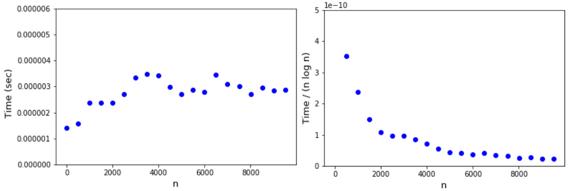
Finding all tandem repeats:



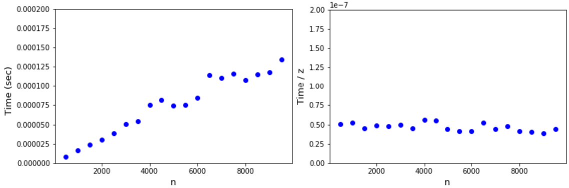


Running time: Random input

Finding branching tandem repeats:



Finding all tandem repeats:



Discussion and conclusion

- TRs can be found using ST or SA.
- Same time complexity, $O(n \log n + z)$.
- ST better for visualising algorithm.
- SA more space efficient and simpler data structures.

Finding all TRs from the branching TRs

Every *non-branching* TR is a *left-rotation* of another TR that starts one place to its right.

Finding all TRs from the branching TRs

Every *non-branching* TR is a *left-rotation* of another TR that starts one place to its right.

- (i', ℓ) where $x[i', i' + \ell) = \alpha\alpha = a\beta a\beta$.

Finding all TRs from the branching TRs

Every *non-branching* TR is a *left-rotation* of another TR that starts one place to its right.

- (i', ℓ) where $x[i', i' + \ell) = \alpha\alpha = a\beta a\beta$.
- Non-branching $\Rightarrow x[i', i' + \ell] = a\beta a\beta a$.

Finding all TRs from the branching TRs

Every *non-branching* TR is a *left-rotation* of another TR that starts one place to its right.

- (i', ℓ) where $x[i', i' + \ell) = \alpha\alpha = a\beta a\beta$.
- Non-branching $\Rightarrow x[i', i' + \ell] = a\beta a\beta a$.
- (i', ℓ) is a left-rotation of $(i' + 1, \ell)$.

Finding all TRs from the branching TRs

Every *non-branching* TR is a *left-rotation* of another TR that starts one place to its right.

- (i', ℓ) where $x[i', i' + \ell) = \alpha\alpha = a\beta a\beta$.
- Non-branching $\Rightarrow x[i', i' + \ell] = a\beta a\beta a$.
- (i', ℓ) is a left-rotation of $(i' + 1, \ell)$.

We can find all the TRs by repeated left-rotations from the *branching* TRs! Time $O(z)$.