Suffix arrays

Disadvantage of suffix tree

- Suffix tree is space inefficient. It requires O(n|Σ|log n) bits.
- Manber and Myers (SIAM J. Comp 1993) proposes a new data structure, called suffix array, which has a similar functionality as suffix tree. Moreover, it only requires O(n log n) bits.

Suffix Array (I)

- It is just sorted suffixes.
- E.g. consider S = acacag$

Suffix	Position
acacag$	1
cacag$	2
acag$	3
cag$	4
ag$	5
g$	6
$	7

=>

Sort

SA[i]	Suffix
7	$
1	acacag$
3	acag$
5	ag$
2	cacag$
4	cag$
6	g$

- Suffix array is an array of n indices. Thus, it takes O(n log n) bits.

Suffix array

```
T = abaaba$
    0123456
```

$\leftarrow$ As with suffix tree, *T* is part of index

SA(*T*) =

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

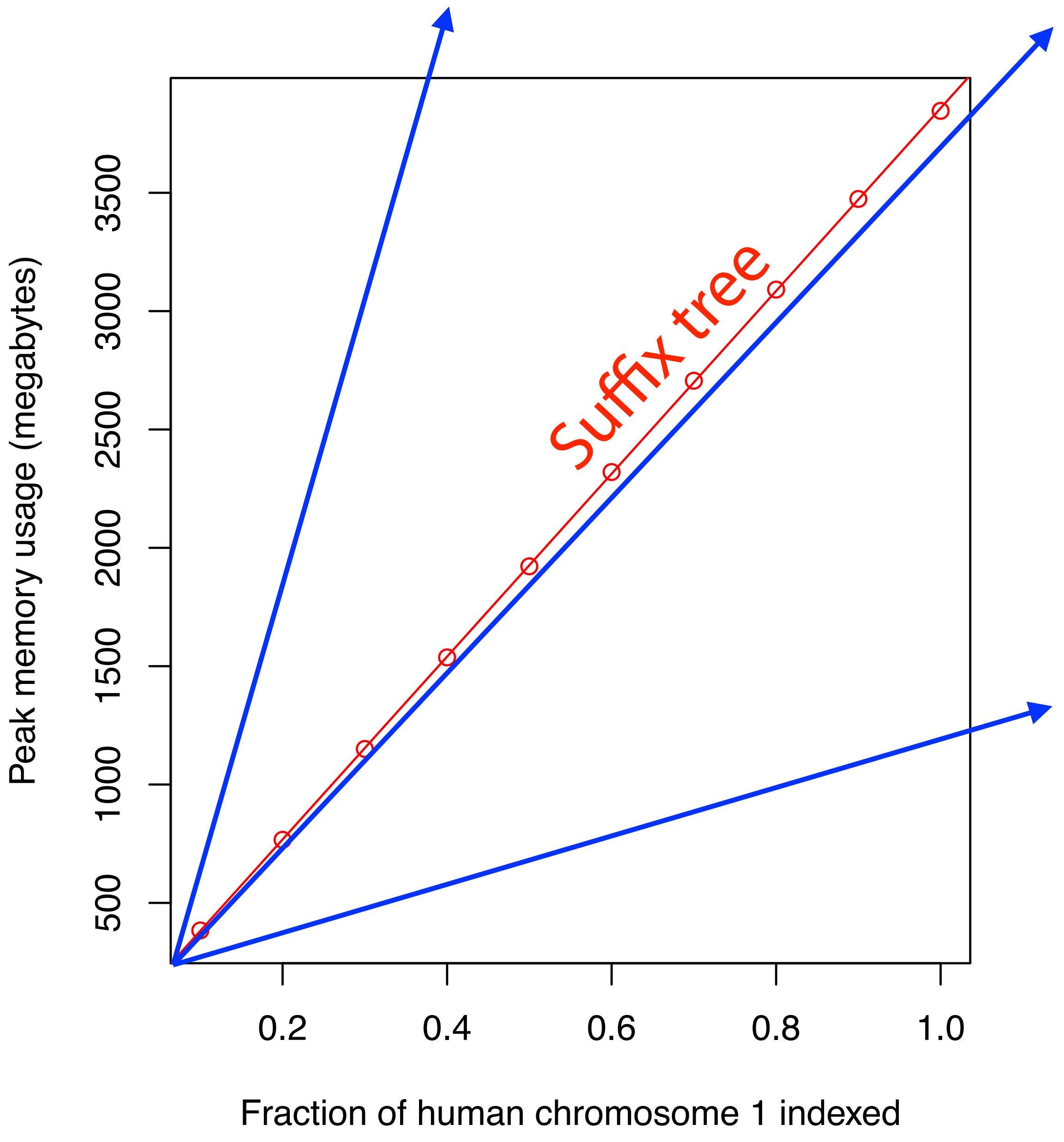
m integers

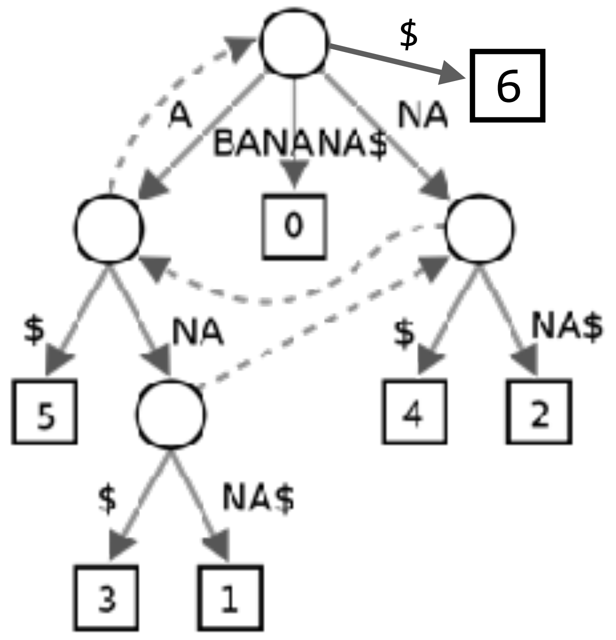
Suffix array of *T* is an array of integers in [0, *m*) specifying lexicographic (alphabetical) order of *T*'s suffixes

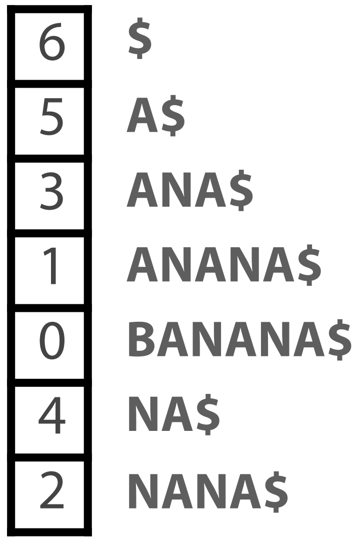
Suffix array

O(*m*) space, like suffix tree

Is "constant factor" worse, better, same?



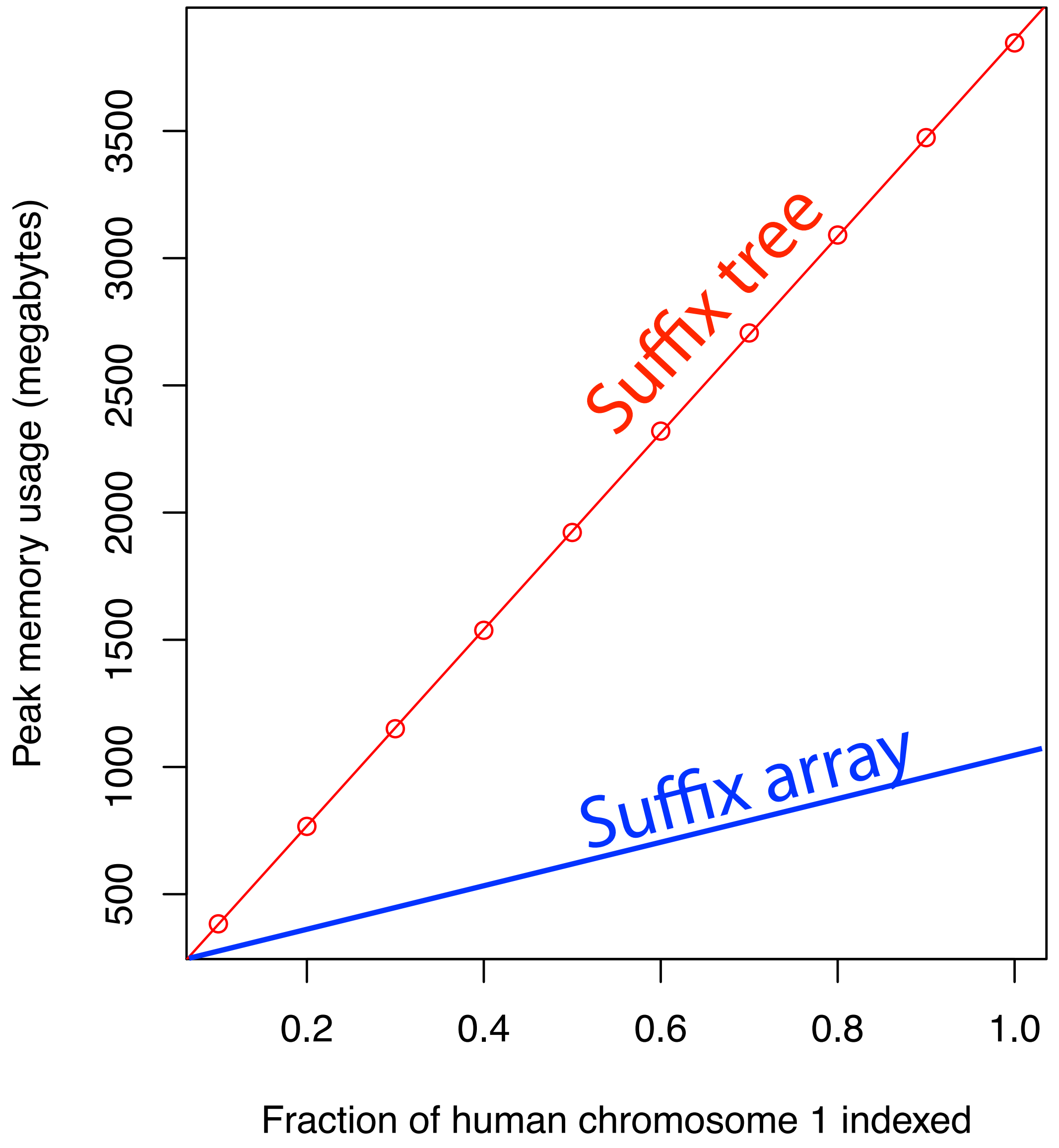


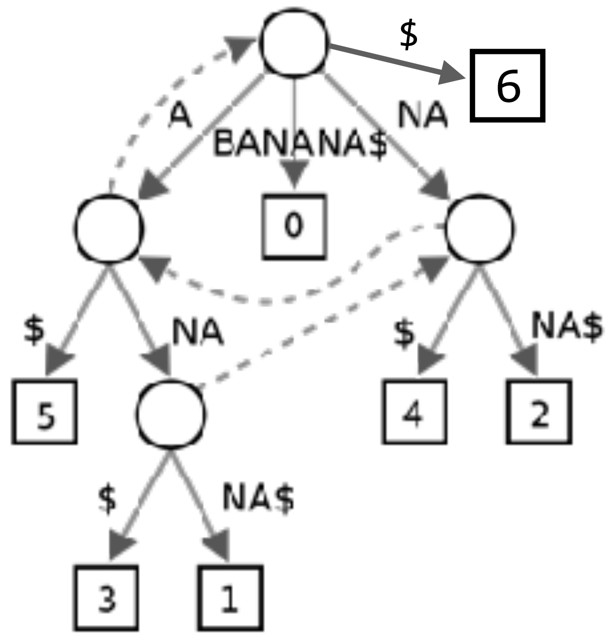


6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

Suffix array

32-bit integers sufficient for human genome, so fits in ~4 bytes/base $\times$ 3 billion bases $\approx$ 12 GB. Suffix tree is >45 GB.





6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

Suffix array: querying

Is *P* a substring of *T*?

T = abaaba$

1. For *P* to be a substring, it must be a prefix of ≥1 of *T*'s suffixes
2. Suffixes sharing a prefix are consecutive in the suffix array

Use binary search

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

Suffix array: querying

Is *P* a substring of *T*?

Do binary search, check whether *P* is a prefix of the suffix there

Query time is *O*(?)...

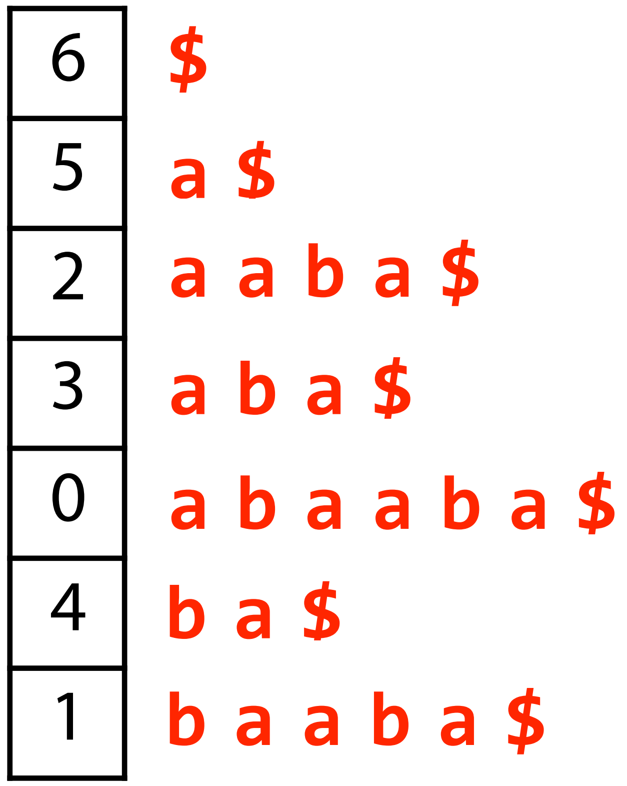
...$O(\log_2 m)$ bisections, $O(n)$ comparisons per bisection, so $O(n \log m)$

T = abaaba$

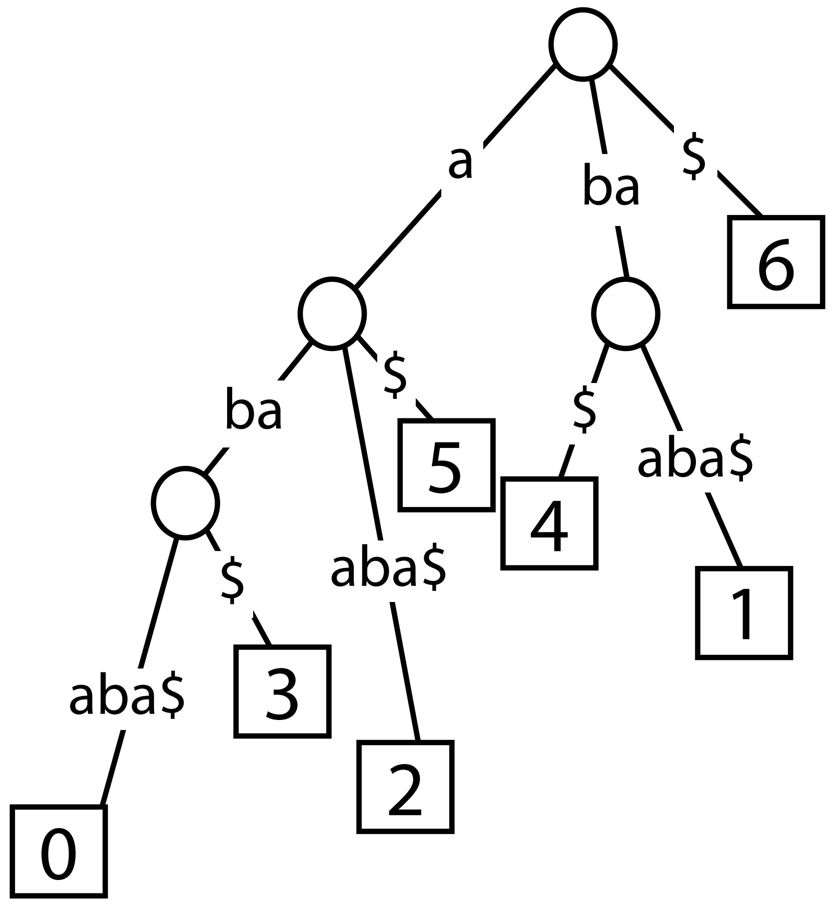
6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

Suffix array: querying

Contrast suffix array query time: O(*n* log *m*) with suffix tree: O(*n*)



6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $



Time can be improved to O(*n* + log *m*), but we won't discuss here (See Gusfield 7.17.4). For this class, we'll consider it O(*n* log *m*).

Suffix array: sorting suffixes

Use your favorite sort, e.g., quicksort

0	a b a a b a $
1	b a a b a $
2	a a b a $
3	a b a $
4	b a $
5	a $
6	$

```
def quicksort(q):
    lt, gt = [], []
    if len(q) <= 1:
        return q
    for x in q[1:]:
        if x < q[0]:
            lt.append(x)
        else:
            gt.append(x)
    return quicksort(lt) + q[0:1] + quicksort(gt)
```

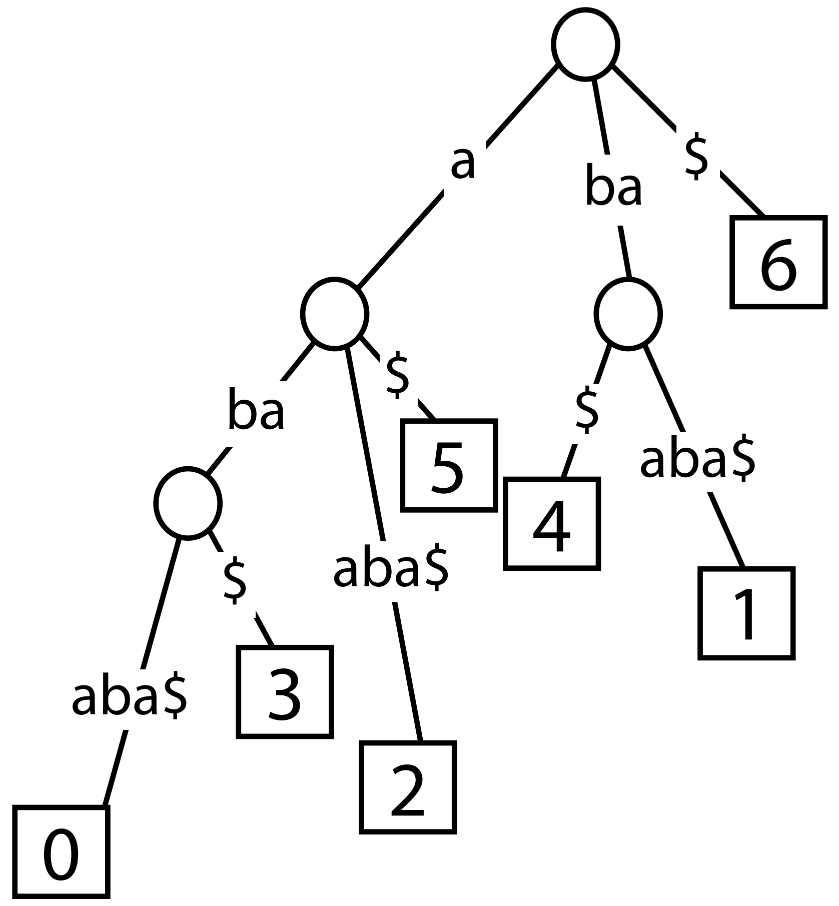
Expected time: O($m^2 \log m$)

Not $O(m \log m)$ because a suffix comparison is O(m) time

Suffix array: building

How to build a suffix array?

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $



(a) Build suffix tree, (b) traverse in alphabetical order,
(c) upon reaching leaf, append suffix to array

Suffix Array (I)

- It is just sorted suffixes.
- E.g. consider S = acacag$

Suffix	Position
acacag$	1
cacag$	2
acag$	3
cag$	4
ag$	5
g$	6
$	7

=>

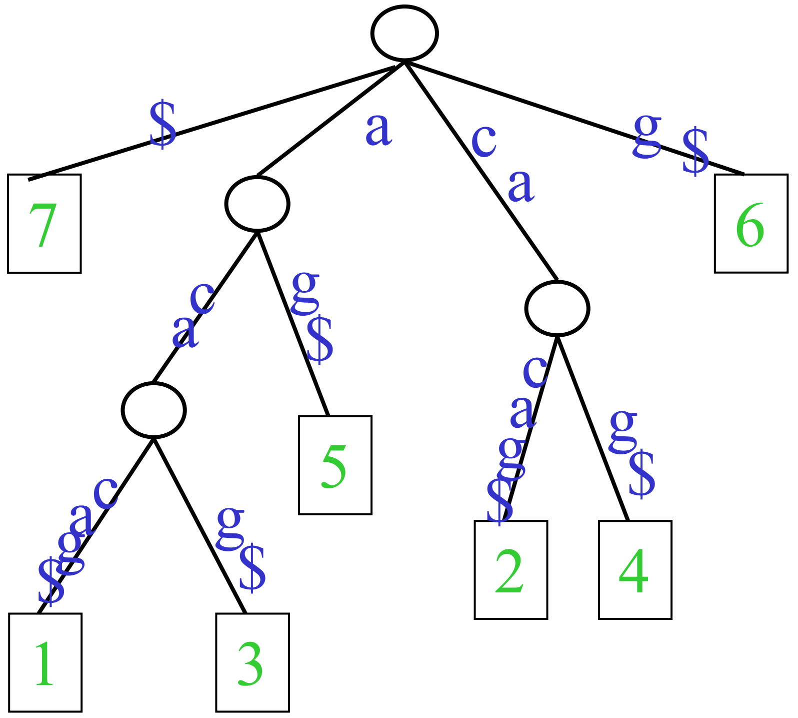
Sort

SA[i]	Suffix
7	$
1	acacag$
3	acag$
5	ag$
2	cacag$
4	cag$
6	g$

- Suffix array is an array of n indices. Thus, it takes O(n log n) bits.

Observation

- The leaves of a suffix tree is in suffix array order.



SA[i]	Suffix
7	$
1	acacag$
3	acag$
5	ag$
2	cacag$
4	cag$
6	g$

Linear time construction of suffix array from suffix tree

- Recall that the suffix tree *T* of S[1..n] can be constructed in O(n) time.
- Then, by "lexical" depth-first traversal of *T*, the suffix array of S is obtained.
  - This takes O(n) time.
- However, the space used during construction is the same as that for suffix tree! This defeats the purpose of suffix array.
- Today, we can build suffix array using O(n) bit space and O(n) time.

Suffix array: sorting suffixes

Another idea: Use a sort algorithm that's aware that the items being sorted are all suffixes of the same string

Original suffix array paper suggested an $O(m \log m)$ algorithm

Manber U, Myers G. "Suffix arrays: a new method for on-line string searches." SIAM Journal on Computing 22.5 (1993): 935-948.

Other popular $O(m \log m)$ algorithms have been suggested

Larsson NJ, Sadakane K. Faster suffix sorting. Technical Report LU-CS-TR: 99-214, LUNDFD6/(NFCS-3140)/1-43/(1999), Department of Computer Science, Lund University, Sweden, 1999.

More recently $O(m)$ algorithms have been demonstrated!

Kärkkäinen J, Sanders P. "Simple linear work suffix array construction." Automata, Languages and Programming (2003): 187-187.

Ko P, Aluru S. "Space efficient linear time construction of suffix arrays." *Combinatorial Pattern Matching*. Springer Berlin Heidelberg, 2003.

range(T,Q)

- For a pattern Q, its occurrences in T form a consecutive SA range.
- Example: For T= acacag$, ca occurs in SA[5] and SA[6].
- Definition:
  - We called range(T,Q)=[st..ed] if
    - Q is a prefix of every T_j for j=SA[st], SA[st+1], …, SA[ed]
    - where T_j = j suffix of T = T[j..n].
  - Example: range(T,ca)=[5..6]

	SA[i]	Suffix
1	7	$
2	1	acacag$
3	3	acag$
4	5	ag$
→ 5	2	cacag$
→ 6	4	cag$
7	6	g$

Find occurrence of query Q in a string S using suffix array

- Input: (1) the suffix array of a string T of length n and (2) a query Q of length m
- Aim: check if Q occurs in T

- Idea: binary search!

Algorithm

```
SA_binary_search($Q$)
1: Let $L = 1$ and $R = n$;
2: loop
3:     Let $M = (L + R)/2$.
4:     Find the length $m$ of the longest common prefix of $Q$ and suffix $SA[M]$;
5:     if $m = |Q|$ then
6:         Report suffixes $SA[M]$ contain the occurrence of $Q$;
7:     else if suffix $SA[M] > Q$ then
8:         set $R = M$;
9:     else
10:        if $L = R$ then
11:            Report $Q$ does not exist in $S$;
12:        else
13:            set $L = M$ if $L + 1 \neq R$; and set $L = R$, otherwise;
14:        end if
15:    end if
16: end loop
```

Example

- Consider T = acacag$
- Pattern Q = acag

- L=1
- R=7
- M=(L+R)/2=4

	i	SA[i]	Suffix
→	1	7	$
	2	1	acacag$
	3	3	acag$
	4	5	ag$
	5	2	cacag$
	6	4	cag$
→	7	6	g$

Example

- Consider T = acacag$
- Pattern Q = acag

- L=1
- R=7
- M=(L+R)/2=4

- suffix-SA[M] > Q.
- Set R=M=4.

	i	SA[i]	Suffix
→	1	7	$
	2	1	acacag$
	3	3	acag$
→	4	5	ag$
	5	2	cacag$
	6	4	cag$
→	7	6	g$

Example

- Consider T = acacag$
- Pattern Q = acag

- L=1
- R=4
- M=(L+R)/2=2

- suffix-SA[M] < Q.
- Set L=M=2.

	i	SA[i]	Suffix
→	1	7	$
→	2	1	acacag$
	3	3	acag$
→	4	5	ag$
	5	2	cacag$
	6	4	cag$
	7	6	g$

Example

- Consider T = acacag$
- Pattern Q = acag

- L=2
- R=4
- M=(L+R)/2=3

- The pattern Q is found at SA[M]=3.

	i	SA[i]	Suffix
	1	7	$
→	2	1	acacag$
→	3	3	acag$
→	4	5	ag$
	5	2	cacag$
	6	4	cag$
	7	6	g$

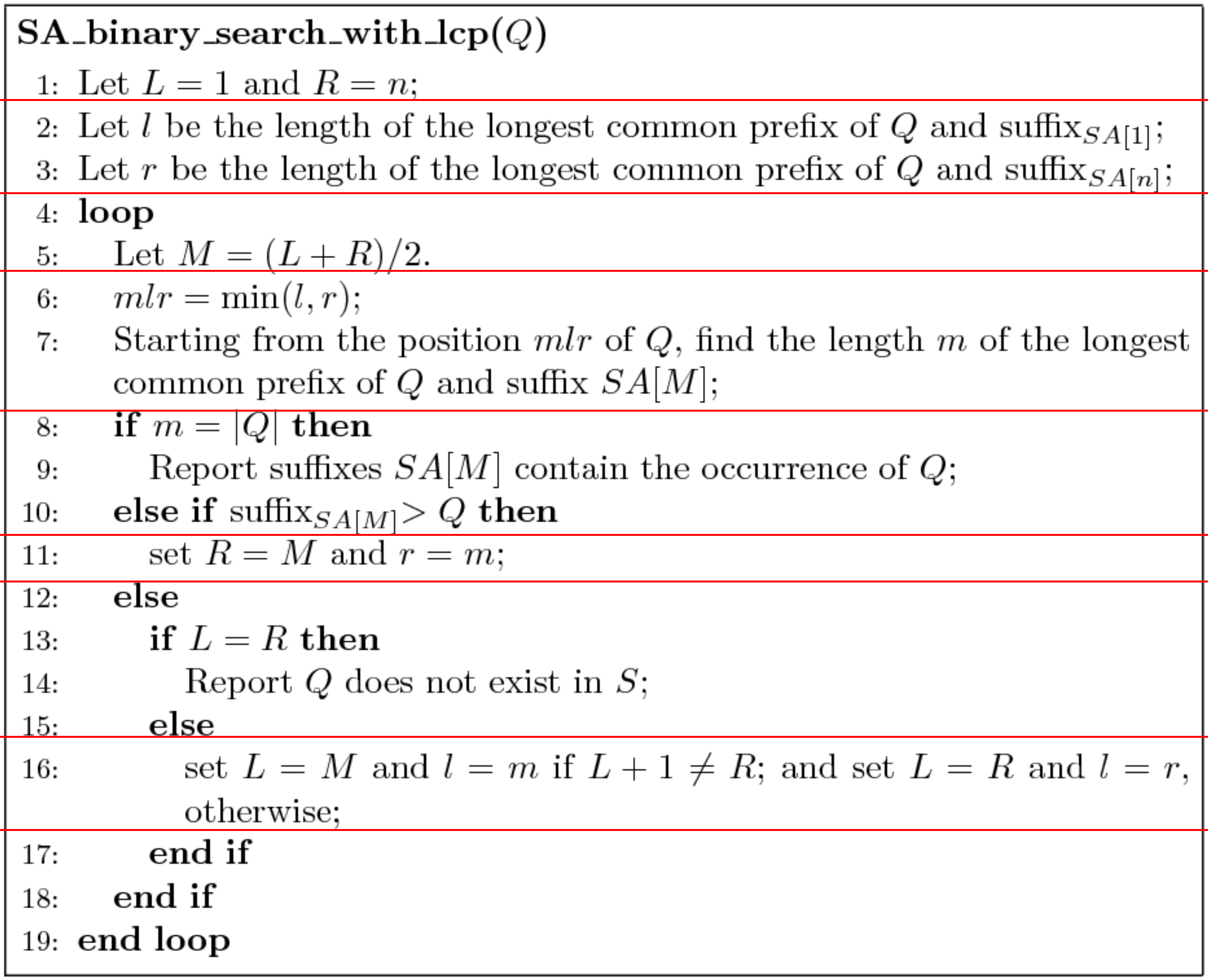
Can we do better?

- During each step of binary search,
  - we need to compare Q with a suffix using O(m) time, which is time consuming.
- Can we do better?

- We have the following observation.
  - Suppose LCP(Q, suffix-SA[L]) is l and LCP(Q, suffix-SA[R]) is r.
  - Then, LCP(Q, suffix-SA[M]) > min{l,r}.

- Below, we describe how to utilize this observation to speedup the computation.

Algorithm



```
SA_binary_search_with_lcp($Q$)
Let $L = 1$ and $R = n$;
Let $l$ be the length of the longest common prefix of $Q$ and suffix$_{SA[1]}$;
Let $r$ be the length of the longest common prefix of $Q$ and suffix$_{SA[n]}$;
loop
    Let $M = (L + R)/2$.
    $mlr = \min(l, r)$;
    Starting from the position $mlr$ of $Q$, find the length $m$ of the longest
    common prefix of $Q$ and suffix $SA[M]$;
    if $m = |Q|$ then
        Report suffixes $SA[M]$ contain the occurrence of $Q$;
    else if suffix$_{SA[M]} > Q$ then
        set $R = M$ and $r = m$;
    else
        if $L = R$ then
            Report $Q$ does not exist in $S$;
        else
            set $L = M$ and $l = m$ if $L + 1 \neq R$; and set $L = R$ and $l = r$,
            otherwise;
        end if
    end if
end loop
```

Example

- Consider T = acacag$
- Pattern Q = acag

- L=1, l=0
- R=7, r=0
- mlr = min(l,r)=0
- M=(L+R)/2=4

	i	SA[i]	Suffix
→	1	7	$
	2	1	acacag$
	3	3	acag$
	4	5	ag$
	5	2	cacag$
	6	4	cag$
→	7	6	g$

Example

- Consider T = acacag$
- Pattern Q = acag

- L=1, l=0
- R=7, r=0
- mlr = min(l,r)=0
- M=(L+R)/2=4, m=1

- The (m+1) char of suffix-SA[M] is g.
- The (m+1) char of Q is c.
- So, suffix-SA[M] > Q.
- Set R=M=4 and r=m=1.

i	SA[i]	Suffix
1	7	$
2	1	acacag$
3	3	acag$
4	5	ag$
5	2	cacag$
6	4	cag$
7	6	g$

Example

- Consider T = acacag$
- Pattern Q = acag

- L=1, l=0
- R=4, r=1
- mlr = min(l,r)=0
- M=(L+R)/2=2, m=3

- The (m+1) char of suffix-SA[M] is c.
- The (m+1) char of Q is g.
- So, suffix-SA[M] < Q.
- Set L=M=2 and l=m=3.

	i	SA[i]	Suffix
→	1	7	$
→	2	1	acacag$
	3	3	acag$
→	4	5	ag$
	5	2	cacag$
	6	4	cag$
	7	6	g$

Example

- Consider T = acacag$
- Pattern Q = acag

- L=2, l=3
- R=4, r=1
- mlr = min(l,r)=1
- M=(L+R)/2=3, m=4

	i	SA[i]	Suffix
	1	7	$
→	2	1	acacag$
→	3	3	acag$
→	4	5	ag$
	5	2	cacag$
	6	4	cag$
	7	6	g$

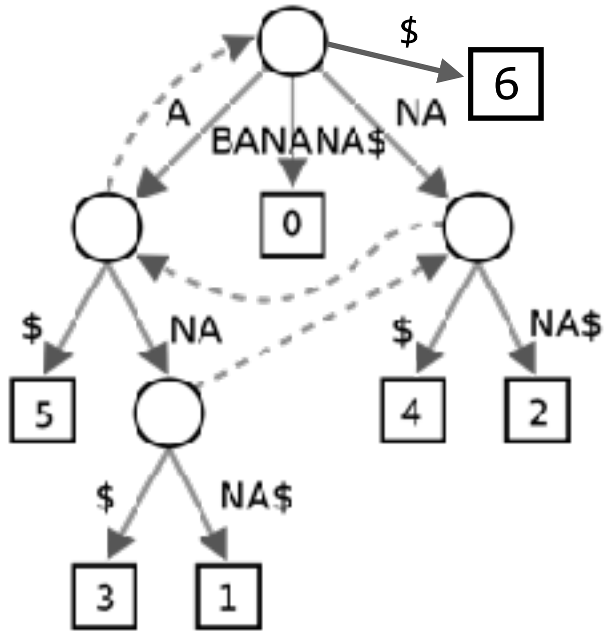
- The pattern Q is found at SA[M]=3.

Time analysis

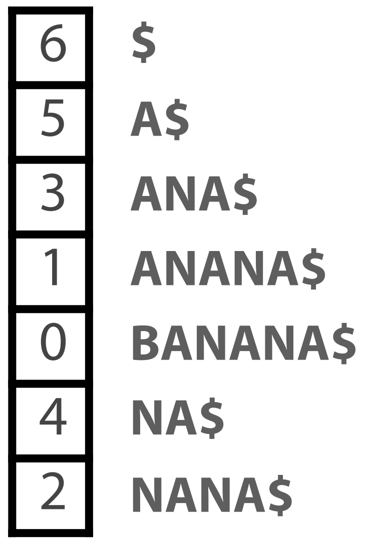
- Binary search will perform log n comparisons
- Each comparison takes at most O(m) time
- In the worst case, O(m log n) time.
- Myers and Manber report that, in practice, the time is O(m + log n).

Suffix array: summary

Just *m* integers, with O(*n* log *m*) query time



Suffix Tree



6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

Suffix Array

Constant factor greatly reduced compared to suffix tree:
human genome index fits in ~12 GB instead of > 45 GB

Suffix array and suffix tree

- We show one example of replacing suffix tree by suffix array
- Note that most applications related to suffix tree can be solved using suffix array with some time blow up!
- When space is limited, replacing suffix tree by suffix array is a good choice.

The size is still too big!

- Why?
- DNA sequences can be very long!
  E.g. Fly: ~100M bases, Human: ~3G bases, Tree: ~9G bases
- Storage to store indexing data structure for human genome
  Suffix Tree: ~40G bytes
  Suffix Array: ~13G bytes
- Can we further reduce the space?

Solution

- Grossi, Vitter (STOC2000)
  - Compressed suffix array (CSA)
- Ferragine, Manzini (FOCS2000)
  - FM-index
- Both of them can be stored in O(n) bit space
- For Human Genome
  - Both CSA and FM-index can be stored within 2G bytes.