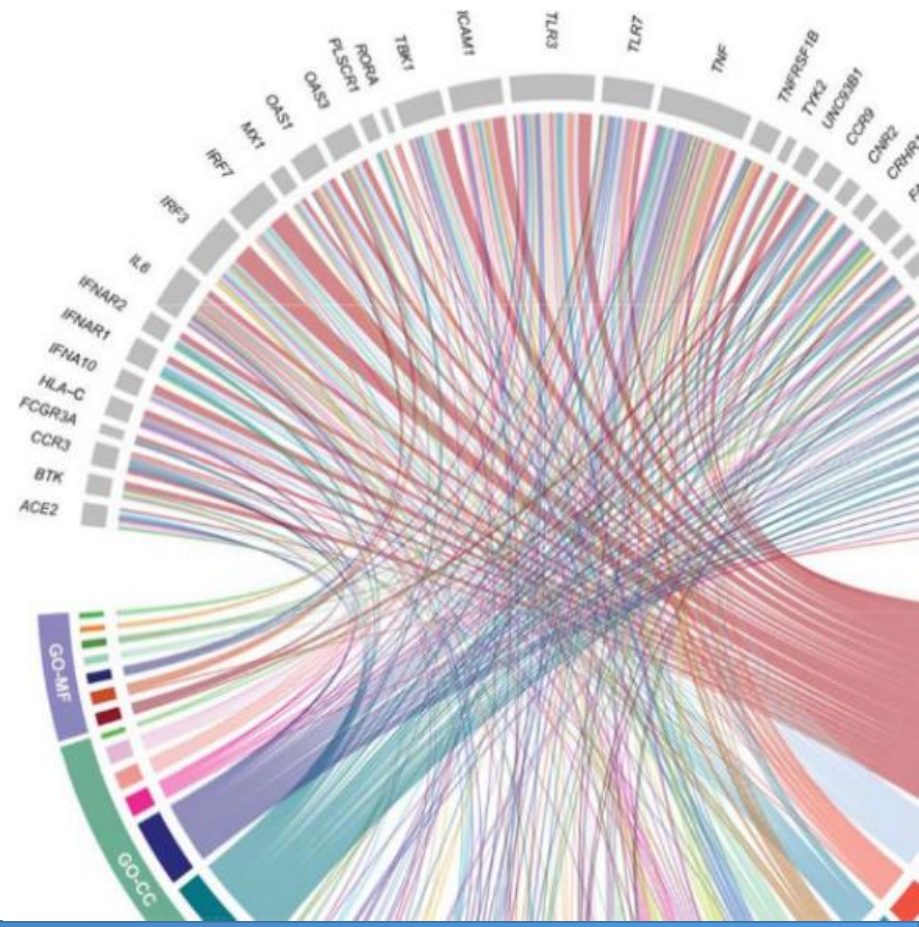
Tècniques i Eines Bioinformàtiques

Suffix Arrays

Santiago Marco-Sola (santiago.marco@upc.edu)

Màster en Enginyeria Informàtica, UPC
Departament of Computer Science
Facultat d'Informàtica de Barcelona (FIB), UPC



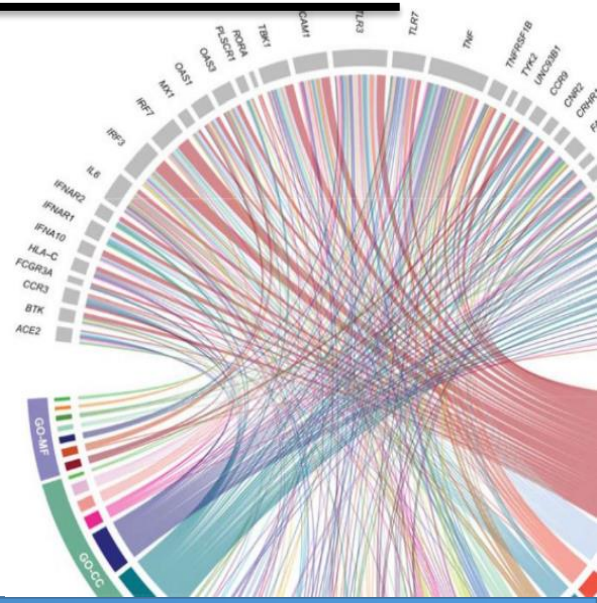
Acknowledgements

Many pictures and materials are taken from **Ben Langmead's course**.

Course heavily inspired in:

- **Genome-Scale Algorithm Design**. Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, Alexandru I. Tomescu. Cambridge University Press.
- **Algorithms on Strings, Trees, and Sequences**. Dan Gusfield. Cambridge University Press.
- **An Introduction to Bioinformatics Algorithms.** Neil C. Jones, Pavel A. Pevzner. MIT Press.

1 Suffix Arrays



Suffix array

```
T = abaaba$
    0123456
```

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 $

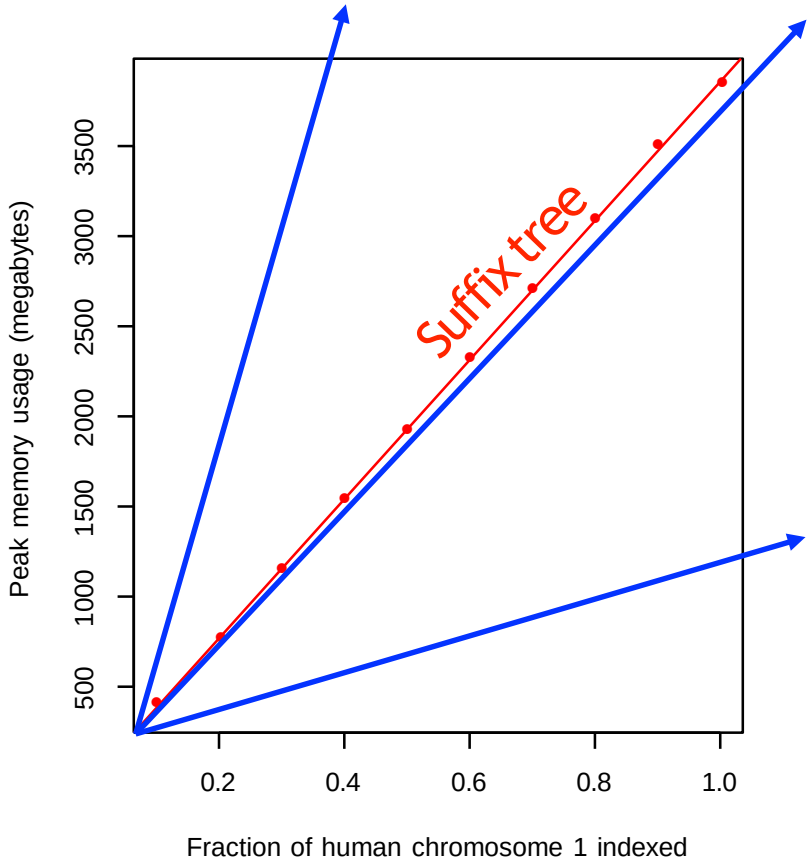
m integers

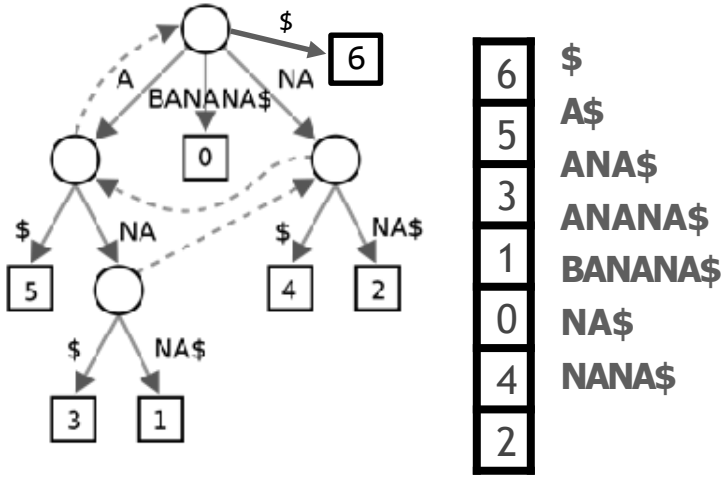
Suffix array of *T* is an array of integers in [0, *m*) specifying lexicographic (alphabetical) order of *T*'s suffixes

Suffix array

$O(m)$ space, like suffix tree

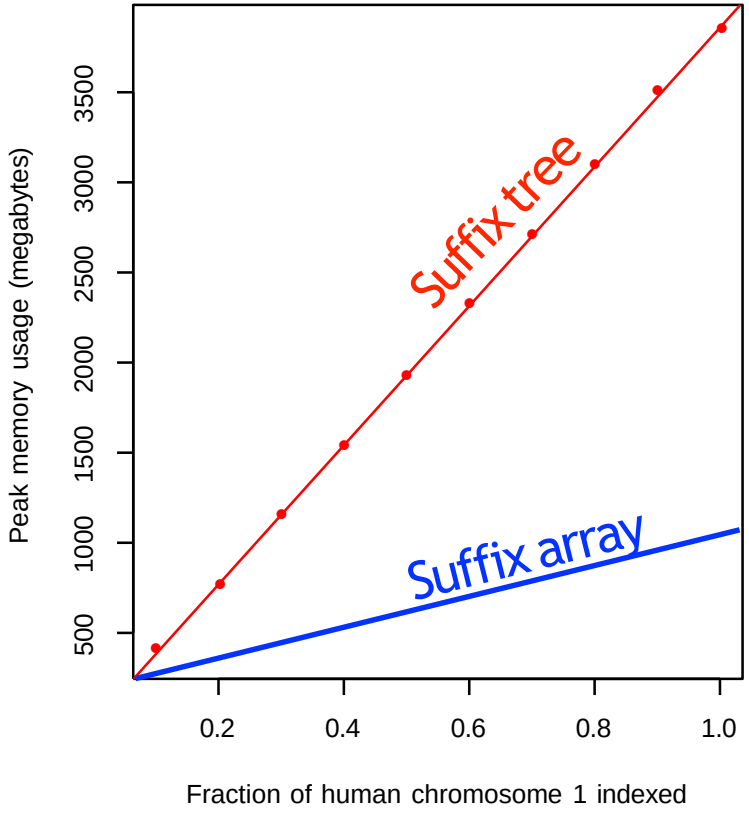
Its "constant factor" is worse, better, same?

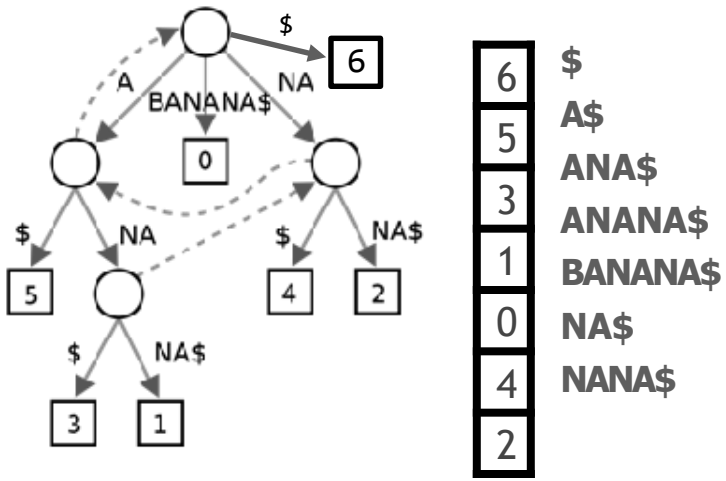




Suffix array

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





Suffix array: querying

Is *P* a substring of *T*?

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

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

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 $O(n \log m)$

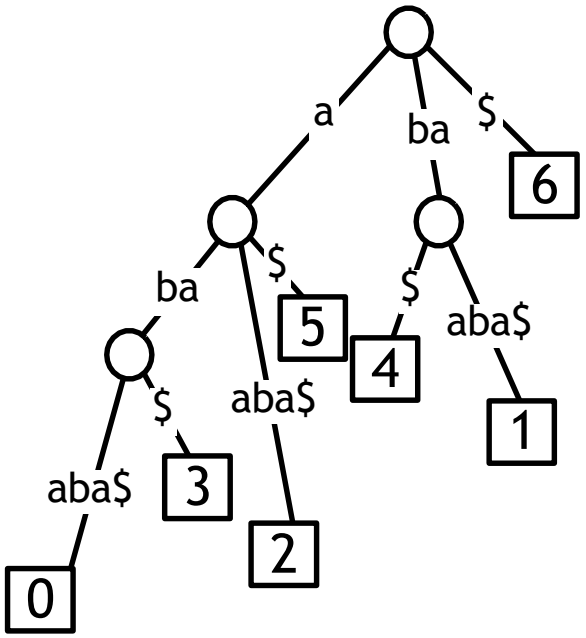
T = abaaba$

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	$

Code

```
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)
```

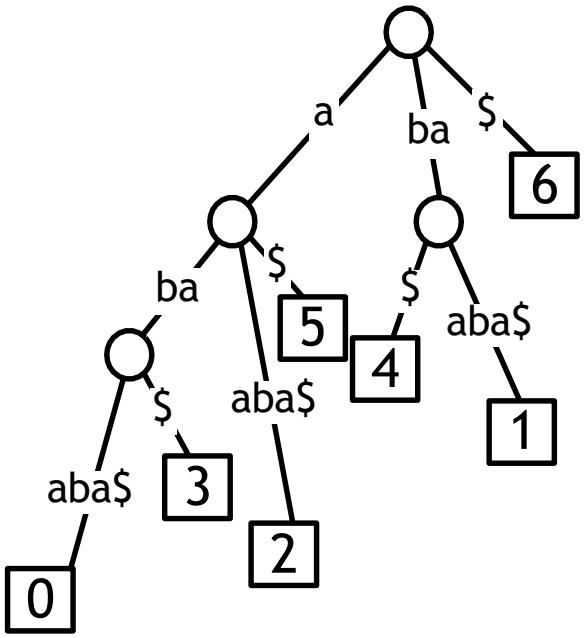
Expected time: O($m^2 \log m$)

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: 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.

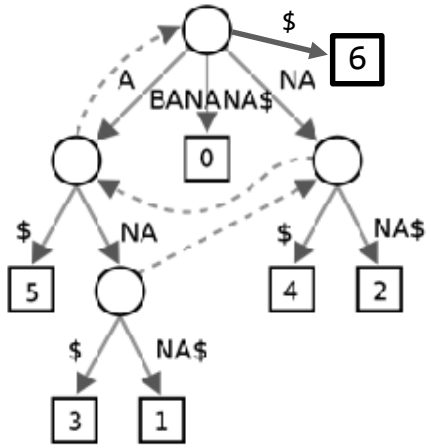
- More recently $O(m)$ algorithms have been demonstrated!

  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.

Suffix array: summary

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



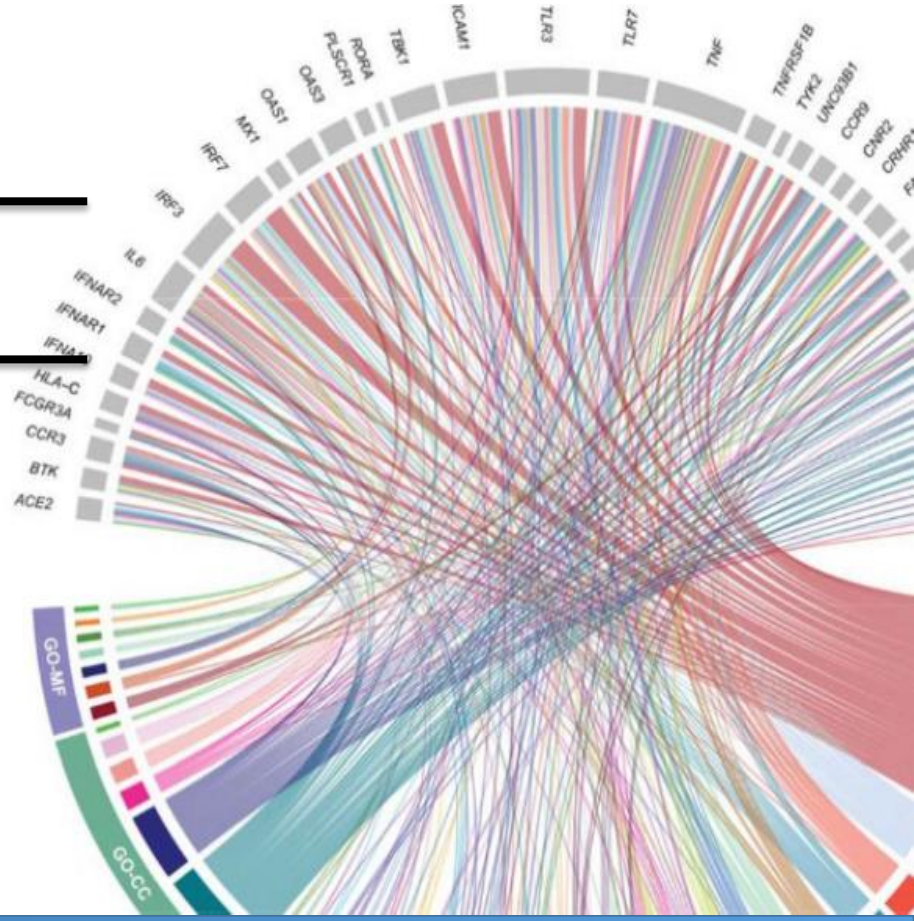
Suffix Tree

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

Exercises


ACE2
BTK
CCR3
FCGR3A
HLA-C
IFNAR1
IFNAR2
IL6
IRF3
IRF7
MX1
OAS1
OAS3
PLSCR1
RORA
TBK1
ICAM1
TLR3
TLR7
TNF
TNFRSF1B
TYK2
UNC93B1
CCR9
CNR2
GO-MF
GO-CC

Exercises

Build a Suffix Array

- Implement a Python function that builds the suffix array of a given input reference. Display the SA in the screen.

Exact Search on the Suffix Array (existence)

- Implement a binary search-based function to determine if a pattern exists within a given text, using a suffix array. That is, exact search of a pattern into the Suffix Array and reports whether it exists.

Exact Search on the Suffix Array (report all occurrences)

- Extend the previous solution to report all the occurrences found

Exercises

Build a Suffix Array

- Implement a Python function that builds the suffix array of a given input reference. Display the SA in the screen.

Code

```python
def build_suffix_array(text):
  text += "$"
  suffixes = [(text[i:], i) for i in range(len(text))]
  suffixes.sort()
  suffix_array = [idx for (suffix, idx) in suffixes]
  return suffix_array

def display_suffix_array(text,suffix_array):
  print("Suffix Array:")
  for idx in suffix_array:
    print(f"{idx}: {text[idx:]}$")

# Example usage
if __name__ == "__main__":
  input_text = "ACGTACGT"
  suffix_array = build_suffix_array(input_text)
  display_suffix_array(input_text,suffix_array)
```

Exercises

Exact Search on the Suffix Array (existence)

- Implement a binary search-based function to determine if a pattern exists in the suffix array.

Code

```
def pattern_exists(text, pattern, sa):
  left, right = 0, len(sa) - 1
  while left <= right:
    mid = (left + right) // 2
    suffix = text[sa[mid]:]
    if suffix.startswith(pattern):
      return True
    elif pattern > suffix:
      left = mid + 1
    else:
      right = mid - 1
  return False

text = "ACGTACGTACGT$"
suffix_array = build_suffix_array(text)
patterns = ["TAC", "CGT", "AAA"]
for pattern in patterns:
  exists = pattern_exists(text, pattern, suffix_array)
  print(f"Pattern '{pattern}' exists: {exists}")
```

Exercises

Exact Search on the Suffix Array (report all occurrences)

```
def find_all_occurrences(text, pattern, sa):
  occurrences = []
  left, right = 0, len(sa) - 1
  while left <= right:
    mid = (left + right) // 2
    suffix = text[sa[mid]:]
    if suffix.startswith(pattern):
      # Explore neighbors for all matches
      occurrences.append(sa[mid])
      # Check left neighbors
      l_it = mid - 1
      while l_it >= left and text[sa[l_it]:].startswith(pattern):
        occurrences.append(sa[l_it])
        l_it -= 1
      # Check right neighbors
      r_it = mid + 1
      while r_it <= right and text[sa[r_it]:].startswith(pattern):
        occurrences.append(sa[r_it])
        r_it += 1
      break
    elif pattern > suffix:
      left = mid + 1
    else:
      right = mid - 1
  return sorted(occurrences)
```

Exercise

Approximate String Matching on a Suffix Array.

- Using a suffix array, implement an approximated search function that allows 2 mismatches at most.