**Suffix array**

$T$ = abaaba$

As with suffix tree, $T$ is part of index

\[
\begin{array}{c|c}
6 & $ \\
5 & a$ \\
4 & b_a$ \\
3 & a_b_a$ \\
2 & a_a_b$ \\
1 & b_a_b$ \\
0 & a_b_a_b_a$ \\
\end{array}
\]

$SA(T) =$

(SA = “Suffix Array”)

Suffix array of $T$ is an array of integers in $[0, m]$ specifying the lexicographic order of $T$’s suffixes

$m + 1$ integers
Another Example Suffix Array

$s = \text{cattcat}$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

<table>
<thead>
<tr>
<th>index of suffix</th>
<th>suffix of s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cattcat$</td>
</tr>
<tr>
<td>2</td>
<td>attcat$</td>
</tr>
<tr>
<td>3</td>
<td>ttcat$</td>
</tr>
<tr>
<td>4</td>
<td>tcat$</td>
</tr>
<tr>
<td>5</td>
<td>cat$</td>
</tr>
<tr>
<td>6</td>
<td>at$</td>
</tr>
<tr>
<td>7</td>
<td>t$</td>
</tr>
<tr>
<td>8</td>
<td>$</td>
</tr>
</tbody>
</table>

sort the suffixes alphabetically

the indices just “come along for the ride”

8 | $  
6 | at$  
5 | cat$  
1 | cattcat$  
7 | t$  
4 | tcat$  
3 | ttcat$  

slide courtesy of Carl Kingsford
Another Example Suffix Array

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

$s = \text{cattcat}$

<table>
<thead>
<tr>
<th>Index</th>
<th>Suffix of $s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cattcat$</td>
</tr>
<tr>
<td>2</td>
<td>attcat$</td>
</tr>
<tr>
<td>3</td>
<td>ttcat$</td>
</tr>
<tr>
<td>4</td>
<td>tcat$</td>
</tr>
<tr>
<td>5</td>
<td>cat$</td>
</tr>
<tr>
<td>6</td>
<td>at$</td>
</tr>
<tr>
<td>7</td>
<td>t$</td>
</tr>
<tr>
<td>8</td>
<td>$</td>
</tr>
</tbody>
</table>

Sort the suffixes alphabetically, the indices just “come along for the ride”.

8 6 2 5 1 7 4 3
Suffix array

\(O(m)\) space, same as suffix tree. Is constant factor smaller?

32-bit integer can distinguish characters in the human genome, so suffix array is \(~12\) GB, smaller than MUMmer’s 47 GB suffix tree.
Relationship Between Suffix Trees & Suffix Arrays

\[ \Sigma = \{\$, a, c, t\} \]

\[ s = \text{cattcat}\$ \]

12345678

Red \#s = starting position of the suffix ending at that leaf

Edges leaving each node are sorted by label (left-to-right).

Leaf labels left to right: 86251743

*slide courtesy of Carl Kingsford*
Relationship Between Suffix Trees & Suffix Arrays

$\Sigma = \{\$,a,c,t\}$
$s = \text{cattcat}$

Red #s = starting position of the suffix ending at that leaf

Edges leaving each node are sorted by label (left-to-right).

Leaf labels left to right: 86251743

$s = \text{cattcat}$

*slide courtesy of Carl Kingsford*
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst Case</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix-Doubling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MM [Manber and Myers 1993]</td>
<td>$O(n \log n)$</td>
<td>30</td>
<td>$8n$</td>
</tr>
<tr>
<td>LS [Larsson and Sadakane 1999]</td>
<td>$O(n \log n)$</td>
<td>3</td>
<td>$8n$</td>
</tr>
<tr>
<td>Recursive</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KA [Ko and Aluru 2003]</td>
<td>$O(n)$</td>
<td>2.5</td>
<td>7–10n</td>
</tr>
<tr>
<td>KS [Kärkkäinen and Sanders 2003]</td>
<td>$O(n)$</td>
<td>4.7</td>
<td>10–13n</td>
</tr>
<tr>
<td>KSPP [Kim et al. 2003]</td>
<td>$O(n)$</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>HSS [Hon et al. 2003]</td>
<td>$O(n)$</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>KJP [Kim et al. 2004]</td>
<td>$O(n \log \log n)$</td>
<td>3.5</td>
<td>13–16n</td>
</tr>
<tr>
<td>N [Na 2005]</td>
<td>$O(n)$</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Induced Copying</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT [Itoh and Tanaka 1999]</td>
<td>$O(n^2 \log n)$</td>
<td>6.5</td>
<td>$5n$</td>
</tr>
<tr>
<td>S [Seward 2000]</td>
<td>$O(n^2 \log n)$</td>
<td>3.5</td>
<td>$5n$</td>
</tr>
<tr>
<td>BK [Burkhardt and Kärkkäinen 2003]</td>
<td>$O(n \log n)$</td>
<td>3.5</td>
<td>5–6n</td>
</tr>
<tr>
<td>MF [Manzini and Ferragina 2004]</td>
<td>$O(n^2 \log n)$</td>
<td>1.7</td>
<td>$5n$</td>
</tr>
<tr>
<td>SS [Schürmann and Stoye 2005]</td>
<td>$O(n^2)$</td>
<td>1.8</td>
<td>9–10n</td>
</tr>
<tr>
<td>BB [Baron and Bresler 2005]</td>
<td>$O(n \sqrt{\log n})$</td>
<td>2.1</td>
<td>18n</td>
</tr>
<tr>
<td>M [Maniscalco and Puglisi 2007]</td>
<td>$O(n^2 \log n)$</td>
<td>1.3</td>
<td>5–6n</td>
</tr>
<tr>
<td>MP [Maniscalco and Puglisi 2006]</td>
<td>$O(n^2 \log n)$</td>
<td>1</td>
<td>5–6n</td>
</tr>
<tr>
<td>Hybrid</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT+KA</td>
<td>$O(n^2 \log n)$</td>
<td>4.8</td>
<td>$5n$</td>
</tr>
<tr>
<td>BK+IT+KA</td>
<td>$O(n \log n)$</td>
<td>2.3</td>
<td>5–6n</td>
</tr>
<tr>
<td>BK+S</td>
<td>$O(n \log n)$</td>
<td>2.8</td>
<td>5–6n</td>
</tr>
<tr>
<td>Suffix Tree</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K [Kurtz 1999]</td>
<td>$O(n \log \sigma)$</td>
<td>6.3</td>
<td>13–15n</td>
</tr>
</tbody>
</table>

Time is relative to MP, the fastest in our experiments. Memory is given in bytes including space required for the suffix array and input string and is the average space required in our experiments. Algorithms HSS and N are included, even though to our knowledge they have not been implemented. The time for algorithm MM is estimated from experiments in Larsson and Sadakane [1999].

*slide courtesy of Carl Kingsford
Suffix array: querying

Is $P$ a substring of $T$?

1. For $P$ to be a substring, it must be a prefix of $\geq 1$ of $T$'s suffixes

2. Suffixes sharing a prefix are consecutive in the suffix array

Use binary search
Suffix array: binary search

Python has `bisect` module for binary search

`bisect.bisect_left(a, x)`: Leftmost offset where we can insert `x` into `a` to maintain sorted order. `a` is already sorted!

`bisect.bisect_right(a, x)`: Like `bisect_left`, but returning *rightmost* instead of leftmost offset

```python
from bisect import bisect_left, bisect_right

a = [1, 2, 3, 3, 3, 4, 5]
print(bisect_left(a, 3), bisect_right(a, 3))  # output: (2, 5)

a = [2, 4, 6, 8, 10]
print(bisect_left(a, 5), bisect_right(a, 5))  # output: (2, 2)
```

Python example: http://nbviewer.ipython.org/6753277
Suffix array: binary search

We can straightforwardly use binary search to find a range of elements in a sorted list that equal some query:

```python
from bisect import bisect_left, bisect_right
strls = ['a', 'awkward', 'aw1', 'awls', 'axe', 'axes', 'bee']

# Get range of elements that equal query string 'aw1'
st, en = bisect_left(strls, 'aw1'), bisect_right(strls, 'aw1')
print(st, en) # output: (2, 3)
```

Python example: http://nbviewer.ipython.org/6753277
Suffix array: binary search

Can also use binary search to find a range of elements in a sorted list with some query as a prefix:

```python
from bisect import bisect_left, bisect_right
strls = ['a', 'awkward', 'awl', 'awls', 'axe', 'axes', 'bee']

# Get range of elements with 'aw' as a prefix
st, en = bisect_left(strls, 'aw'), bisect_left(strls, 'ax')
print(st, en)  # output: (1, 4)
```

Python example: http://nbviewer.ipython.org/6753277
Suffix array: binary search

We can do the same thing for a sorted list of suffixes:

```python
from bisect import bisect_left, bisect_right
t = 'abaaba$
suffixes = sorted([t[i:] for i in xrange(len(t))])

st, en = bisect_left(suffixes, 'aba'),
        bisect_left(suffixes, 'abb')

print(st, en) # output: (3, 5)
```

Python example: http://nbviewer.ipython.org/6753277
Suffix array: querying

Is $P$ a substring of $T$?

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

How many times does $P$ occur in $T$?

Two binary searches yield the range of suffixes with $P$ as prefix; size of range equals # times $P$ occurs in $T$

Worst-case time bound?

$O(\log_2 m)$ bisections, $O(n)$ comparisons per bisection, so $O(n \log m)$
Suffix array: querying

Contrast suffix array: $O(n \log m)$ with suffix tree: $O(n)$

But we can improve bound for suffix array...
Suffix array: querying

Consider further: binary search for suffixes with $P$ as a prefix

Assume there’s no $\$ \text{ in } P$. So $P$ can’t be equal to a suffix.

Initialize $l = 0$, $c = \text{floor}(m/2)$ and $r = m$ (just past last elt of SA)

"left" "center" "right"

Notation: We’ll use use $\text{SA}[l]$ to refer to the suffix corresponding to suffix-array element $l$. We could write $T[\text{SA}[l]:]$, but that’s too verbose.

Throughout the search, invariant is maintained:

$\text{SA}[l] < P < \text{SA}[r]$
Suffix array: querying

Throughout search, invariant is maintained:

\[ SA[l] < P < SA[r] \]

What do we do at each iteration?

Let \( c = \lfloor (r + l) / 2 \rfloor \)

If \( P < SA[c] \), either stop or let \( r = c \) and iterate

If \( P > SA[c] \), either stop or let \( l = c \) and iterate

When to stop?

\[ P < SA[c] \text{ and } c = l + 1 \quad \text{answer is } c \]
\[ P > SA[c] \text{ and } c = r - 1 \quad \text{answer is } r \]
def binarySearchSA(t, sa, p):
    assert t[-1] == '$'  # t already has terminator
    assert len(t) == len(sa)  # sa is the suffix array for t
    if len(t) == 1: return 1
    l, r = 0, len(sa)  # invariant: sa[l] < p < sa[r]
    while True:
        c = (l + r) // 2
        # determine whether p < T[sa[c]:] by doing comparisons
        # starting from left-hand sides of p and T[sa[c]:]
        plt = True  # assume p < T[sa[c]:] until proven otherwise
        i = 0
        while i < len(p) and sa[c]+i < len(t):
            if p[i] < t[sa[c]+i]:
                break  # p < T[sa[c]:]
            elif p[i] > t[sa[c]+i]:
                plt = False
                break  # p > T[sa[c]:]
            i += 1  # tied so far
        if plt:
            if c == l + 1: return c
            r = c
        else:
            if c == r - 1: return r
            l = c

Python example: http://nbviewer.ipython.org/6765182
Suffix array: querying

Say we’re comparing $P$ to $SA[c]$ and we’ve already compared $P$ to $SA[l]$ and $SA[r]$ in previous iterations.

$LCP(P, SA[l]) = 3$

More generally:

$LCP(P, SA[c]) \geq \min(LCP(P, SA[l]), LCP(P, SA[r]))$

We can skip character comparisons
Prefix array: querying

```python
def binarySearchSA_lcp1(t, sa, p):
    if len(t) == 1: return 1

    l, r = 0, len(sa) # invariant: sa[l] < p < sa[r]

    lcp lp, lcp rp = 0, 0

    while True:
        c = (l + r) // 2
        plt = True
        i = min(lcp lp, lcp rp)
        while i < len(p) and sa[c]+i < len(t):
            if p[i] < t[sa[c]+i]:
                break # p < T[sa[c]:]
            elif p[i] > t[sa[c]+i]:
                plt = False
                break # p > T[sa[c]:]
            i += 1 # tied so far

        if plt:
            if c == l + 1: return c
            r = c
            lcp rp = i
        else:
            if c == r - 1: return r
            l = c
            lcp lp = i
```

Worst-case time bound is still $O(n \log m)$, but we’re closer

Python example: [http://nbviewer.ipython.org/6765182](http://nbviewer.ipython.org/6765182)
Suffix array: querying

Take an iteration of binary search:

Assume, wlog, that $U = \text{LCP}(\text{SA}[l], \text{SA}[c]) \geq D = \text{LCP}(\text{SA}[c], \text{SA}[r])$ otherwise there are symmetric cases.

key: $u$ has already been computed by previous iterations, and $U$ can be looked-up in constant
Suffix array: querying

Three cases: or, if $\text{LCP}(P, \text{SA}[r])$ is larger, 3 symmetric cases.

$LCP(\text{SA}[c], \text{SA}[l]) > LCP(P, \text{SA}[l])$

$LCP(\text{SA}[c], \text{SA}[l]) < LCP(P, \text{SA}[l])$

$LCP(\text{SA}[c], \text{SA}[l]) = LCP(P, \text{SA}[l])$
Suffix array: querying

Case 1:

Next char of $P$ after the $\text{LCP}(P, \text{SA}[l])$ must be greater than corresponding char of $\text{SA}[c]$

$P > \text{SA}[c]$

In this case, we compute $\text{LCP}(P[u:], \text{SA}[c][u:])$. $c$ becomes our new $l$, and now we know that $\text{LCP}(P, \text{SA}[l])$, b/c we just computed it!
Suffix array: querying

Case 2:

Next char of $SA[c]$ after $LCP(SA[c], SA[l])$ must be greater than corresponding char of $P$

$P < SA[c]$

In this case, we compute $LCP(P[u:], SA[c][u:])$. $c$ becomes our new $r$, and now we know that $LCP(P, SA[r])$, b/c we just computed it!
Suffix array: querying

Case 3:

Must do further character comparisons between \( P \) and \( SA[c] \)

Each such comparison either:

(a) mismatches, leading to a bisection

(b) matches, in which case \( LCP(P, SA[c]) \) grows

\[ LCP(SA[c], SA[l]) = LCP(P, SA[l]) \]
Suffix array: querying

We improved binary search on suffix array from $O(n \log m)$ to $O(n + \log m)$ using information about Longest Common Prefixes (LCPs).

LCPs between $P$ and suffixes of $T$ computed during search, LCPs among suffixes of $T$ computed offline

LCP($SA[c], SA[l]$) $>$ LCP($P, SA[l]$)  
Bisect right!

LCP($SA[c], SA[l]$) $<$ LCP($P, SA[l]$)  
Bisect left!

LCP($SA[c], SA[l]$) $=$ LCP($P, SA[l]$)  
Compare some characters, then bisect!
Sketch of Running Time

**Thm.** Given the LCP($X, Y$) values, searching for a string $P$ in a suffix array of length $m$ now takes $O(|P| + \log m)$ time.

In case 1 & 2, we make $O(1)$ comparisons and bisect left or right — there are at most $O(\log m)$ bisections.

In case 3 we try to match characters starting at some offset between $SA[c]$ and $P$. If they match, those characters will never be compared again, so there are at most $O(|P|)$ such comparisons.

Mismatching characters may be compared more than once.

**But** there can be only 1 mismatch / bisection. There are $O(\log m)$ bisections, so there are at most $O(\log m)$ mismatches.

\[\therefore \text{Total \# of comparisons} = O(|P| + \log m).\]
Suffix array: LCPs

How to pre-calculate LCPs for every \((l, c)\) and \((c, r)\) pair in the search tree?

Triples are \((l, c, r)\) triples

Example where \(m = 16\) (incl. \$)  # search tree nodes = \(m - 1\)
Suffix array: LCPs

Suffix Array (SA) has $m$ elements

Define LCP1 array with $m - 1$ elements such that $LCP[i] = LCP(SA[i], SA[i+1])$
Suffix array: LCPs

LCP2\[i\] = LCP(SA\[i\], SA\[i+1\], SA\[i+2\])

In fact, LCP of a range of consecutive suffixes in SA equals the minimum LCP1 among adjacent pairs in the range

LCP1 is a building block for other useful LCPs
Suffix array: LCPs

Good time to calculate LCP1 it is *at the same time* as we *build* the suffix array, since putting the suffixes in order involves breaking ties after common prefixes.

<table>
<thead>
<tr>
<th>SA(T):</th>
<th>LCP1(T):</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>$</td>
</tr>
<tr>
<td>5</td>
<td>a $</td>
</tr>
<tr>
<td>2</td>
<td>a a b a $</td>
</tr>
<tr>
<td>3</td>
<td>a b a $</td>
</tr>
<tr>
<td>0</td>
<td>a b a a b a $</td>
</tr>
<tr>
<td>4</td>
<td>b a $</td>
</tr>
<tr>
<td>1</td>
<td>b a a b a $</td>
</tr>
</tbody>
</table>

$\text{SA}(T)$:

$\text{LCP1}(T)$:
Suffix array: LCPs

T = abracadabracada
Suffix array: LCPs

$T = \text{abracadabra}\text{cada}$

SA(T): 15 14 7 0 10 3 12 5 8 1 11 4 13 6 9 2

LCP1(T): 0 1 8 1 5 1 3 0 7 0 4 0 2 0 6
Suffix array: LCPs

\( T = \text{abracadabracada}\$ 

\[
\begin{array}{cccccccccccccccc}
15 & 14 & 7 & 0 & 10 & 3 & 12 & 5 & 8 & 1 & 11 & 4 & 13 & 6 & 9 & 2 \\
0 & 1 & 8 & 1 & 5 & 1 & 3 & 0 & 7 & 0 & 4 & 0 & 2 & 0 & 6 \\
\end{array}
\]
Suffix array: LCPs

T = abracadabracada$
Suffix array: LCPs

$T = \text{abracadabra\textasciicircum{c}cada}$
Suffix array: LCPs

T = abracadabracada$
Suffix array: LCPs

$T = \text{abracadabra}cada$

NOTE: These arrays are “shifted” by 1 — the value in LCP\_LC corresponds to (0, 1, 2), which is at LCP\_LC[0], not LCP\_LC[1]. So, to look up LCP(SA[i], SA[c]) we look at LCP\_LC[c-1]
Suffix array: LCPs

# Calculates (l, c) LCPs and (c, r) LCPs from LCP1 array. Returns
# pair where first element is list of LCPs for (l, c) combos and
# second is LCPs for (c, r) combos.
def precomputeLcps(lcp1):
    llcp, rlcp = [None] * len(lcp1), [None] * len(lcp1)
    lcp1 += [0]
    def precomputeLcpsHelper(l, r):
        if l == r-1: return lcp1[l]
        c = (l + r) // 2
        llcp[c-1] = precomputeLcpsHelper(l, c)
        rlcp[c-1] = precomputeLcpsHelper(c, r)
        return min(llcp[c-1], rlcp[c-1])
    precomputeLcpsHelper(0, len(lcp1))
    return llcp, rlcp

O(m) time and space

Python example: http://nbviewer.ipython.org/6783863
We saw 3 ways to query (binary search) the suffix array:

1. Typical binary search. Ignores LCPs. $O(n \log m)$.
2. Binary search with some skipping using LCPs between $P$ and $T$’s suffixes. Still $O(n \log m)$, but it can be argued it’s near $O(n + \log m)$ in practice. [Gusfield: “Simple Accelerant”]
3. Binary search with skipping using all LCPs, including LCPs among T’s suffixes. $O(n + \log m)$. [Gusfield: “Super Accelerant”]

How much space do they require?

1. $\sim m$ integers (SA)
2. $\sim m$ integers (SA)
3. $\sim 3m$ integers (SA, LCP_LC, LCP_CR)
### Suffix array: performance comparison

<table>
<thead>
<tr>
<th></th>
<th>Super accelerant</th>
<th>Simple accelerant</th>
<th>No accelerant</th>
</tr>
</thead>
<tbody>
<tr>
<td>python -O</td>
<td>68.78 s</td>
<td>69.80 s</td>
<td>102.71 s</td>
</tr>
<tr>
<td>pypy -O</td>
<td>5.37 s</td>
<td>5.21 s</td>
<td>8.74 s</td>
</tr>
<tr>
<td># character comparisons</td>
<td>99.5 M</td>
<td>117 M</td>
<td>235 M</td>
</tr>
</tbody>
</table>

Matching 500K 100-nt substrings to the ~ 5 million nt-long *E. coli* genome. Substrings drawn randomly from the genome.

Index building time not included
Suffix array: building

Given $T$, how to we efficiently build $T$'s suffix array?

<table>
<thead>
<tr>
<th></th>
<th>Suffix Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>$</td>
</tr>
<tr>
<td>5</td>
<td>a $</td>
</tr>
<tr>
<td>2</td>
<td>a a b a $</td>
</tr>
<tr>
<td>3</td>
<td>a b a $</td>
</tr>
<tr>
<td>0</td>
<td>a b a a b a $</td>
</tr>
<tr>
<td>4</td>
<td>b a $</td>
</tr>
<tr>
<td>1</td>
<td>b a a b a $</td>
</tr>
</tbody>
</table>
Suffix array: building SA

Idea: Build suffix tree, do a lexicographic depth-first traversal reporting leaf offsets as we go.

Traverse $O(m)$ nodes and emit $m$ integers, so $O(m)$ time assuming edges are already ordered.
Suffix array: building LCP1

Can calculate LCP1 at the same time

Yes: on our way from one leaf to the next, record the shallowest “label depth” observed
def saLcp(self):
    # Return suffix array and an LCP1 array corresponding to this suffix tree. self.root is root, self.t is the text.
    self.minSinceLeaf = 0
    sa, lcp1 = [], []
    def __visit(n):
        if len(n.out) == 0:
            # leaf node, record offset and LCP1 with previous leaf
            sa.append(len(self.t) - n.depth)
            lcp1.append(self.minSinceLeaf)
            # reset LCP1 to depth of this leaf
            self.minSinceLeaf = n.depth
        # visit children in lexicographical order
        for c, child in sorted(n.out.items()):
            __visit(child)
            # after each child visit, perhaps decrease
            # minimum-depth-since-last-leaf value
            self.minSinceLeaf = min(self.minSinceLeaf, n.depth)
    __visit(self.root)
    return sa, lcp1[1:]

This is a member function from a SuffixTree class, the rest of which isn’t shown

Python example: http://nbviewer.ipython.org/6796858
Suffix array: building

Suffix trees are big. Given $T$, how do we efficiently build $T$'s suffix array \textit{without} first building a suffix tree?

\begin{array}{|c|c|}
\hline
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 $ \\
\hline
\end{array}
Suffix array: sorting suffixes

One idea: Use your favorite sort, e.g., quicksort

<table>
<thead>
<tr>
<th></th>
<th>a b a a b a $</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b a a b a $</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>a a b a $</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>a b a $</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>b a $</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>a $</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>$</td>
<td>6</td>
</tr>
</tbody>
</table>

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: sorting suffixes

One idea: Use a sort algorithm that’s aware that the items being sorted are strings, e.g. “multikey quicksort”

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a b a a b a $</td>
</tr>
<tr>
<td>1</td>
<td>b a a b a $</td>
</tr>
<tr>
<td>2</td>
<td>a a b a $</td>
</tr>
<tr>
<td>3</td>
<td>a b a $</td>
</tr>
<tr>
<td>4</td>
<td>b a $</td>
</tr>
<tr>
<td>5</td>
<td>a $</td>
</tr>
<tr>
<td>6</td>
<td>$</td>
</tr>
</tbody>
</table>

Essentially $O(m^2)$ time

---

The Skew Algorithm
Kärkkäinen & Sanders, 2003

- Main idea: Divide suffixes into 3 groups:
  - Those starting at positions $i=0,3,6,9,...$ (i mod 3 = 0)
  - Those starting at positions 1,4,7,10,... (i mod 3 = 1)
  - Those starting at positions 2,5,8,11,... (i mod 3 = 2)

- For simplicity, assume text length is a multiple of 3 after padding with a special character.

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
T[0, n) = y & a & b & b & a & d & a & b & b & a & d & o \end{array}
\]

\[
SA = (12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0)
\]

Basic Outline:
- Recursively handle suffixes from the $i$ mod 3 = 1 and $i$ mod 3 = 2 groups.
- Merge the $i$ mod 3 = 0 group at the end.

*slide courtesy of Carl Kingsford
Step 0 — Constructing a sample
These are called the “sample suffixes”

Step 0: Construct a sample. For $k = 0, 1, 2$, define

$$B_k = \{i \in [0, n] \mid i \mod 3 = k\}.$$ 

Let $C = B_1 \cup B_2$ be the set of sample positions and $S_C$ the set of sample suffixes.

Example. $B_1 = \{1, 4, 7, 10\}$, $B_2 = \{2, 5, 8, 11\}$, i.e., $C = \{1, 4, 7, 10, 2, 5, 8, 11\}$.

Step 1 — Sorting the sample

Step 1: Sort sample suffixes. For $k = 1, 2$, construct the strings

$$R_k = [t_k t_{k+1} t_{k+2}][t_{k+3} t_{k+4} t_{k+5}] \ldots [t_{\max B_k} t_{\max B_k+1} t_{\max B_k+2}]$$

whose characters are triples $[t_i t_{i+1} t_{i+2}]$. Note that the last character of $R_k$ is always unique because $t_{\max B_k+2} = 0$. Let $R = R_1 \odot R_2$ be the concatenation of $R_1$ and $R_2$. Then the (nonempty) suffixes of $R$ correspond to the set $S_C$ of sample suffixes: $[t_i t_{i+1} t_{i+2}] [t_i+3 t_i+4 t_i+5] \ldots$ corresponds to $S_i$. The correspondence is order preserving, i.e., by sorting the suffixes of $R$ we get the order of the sample suffixes $S_C$.

Example. $R = [abb][ada][bba][do0][bba][dab][bad][000]$. 

Step 1 — Sorting the sample

To sort the suffixes of $R$, first radix sort the characters of $R$ and rename them with their ranks to obtain the string $R'$. If all characters are different, the order of characters gives directly the order of suffixes. Otherwise, sort the suffixes of $R'$ using Algorithm DC3.

Example. $R' = (1, 2, 4, 6, 4, 5, 3, 7)$ and $SA_{R'} = (8, 0, 1, 6, 4, 2, 5, 3, 7)$.
Interlude: Radix Sort

- $O(n)$-time sort for $n$ items when items can be divided into constant # of digits.
- Put into buckets based on least-significant digit, flatten, repeat with next-most significant digit, etc.
- Example items: 100 123 042 333 777 892 236

- # of passes = # of digits
- Each pass goes through the numbers once.

*slide courtesy of Carl Kingsford*
Step 1.5 — Sorting the sample

Example. \( R = [abb][ada][bba][do0][bba][dab][bad][000] \).

Once the sample suffixes are sorted, assign a rank to each suffix. For \( i \in C \), let \( rank(S_i) \) denote the rank of \( S_i \) in the sample set \( S_C \). Additionally, define \( rank(S_{n+1}) = rank(S_{n+2}) = 0 \). For \( i \in B_0 \), \( rank(S_i) \) is undefined.

\[
\begin{array}{cccccccccccccc}
 i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
 rank(S_i) & \perp & 1 & 4 & \perp & 2 & 6 & \perp & 5 & 3 & \perp & 7 & 8 & \perp & 0 & 0 \\
\end{array}
\]
Step 2 — Sorting the non-sample suffixes

Step 2: Sort nonsample suffixes. Represent each nonsample suffix $S_i \in S_{B_0}$ with the pair $(t_i, \text{rank}(S_{i+1}))$. Note that $\text{rank}(S_{i+1})$ is always defined for $i \in B_0$. Clearly we have, for all $i, j \in B_0$,

$$S_i \leq S_j \iff (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1})).$$

The pairs $(t_i, \text{rank}(S_{i+1}))$ are then radix sorted.

Example. $S_{12} < S_6 < S_9 < S_3 < S_0$ because $(0, 0) < (a, 5) < (a, 7) < (b, 2) < (y, 1)$. 

Step 2 — Sorting the non-sample suffixes

Step 3: Merge. The two sorted sets of suffixes are merged using a standard comparison-based merging. To compare suffix $S_i \in S_C$ with $S_j \in S_{B_0}$, we distinguish two cases:

$$i \in B_1 : \quad S_i \leq S_j \iff (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1}))$$

$$i \in B_2 : \quad S_i \leq S_j \iff (t_i, t_{i+1}, \text{rank}(S_{i+2})) \leq (t_j, t_{j+1}, \text{rank}(S_{j+2}))$$

Note that the ranks are defined in all cases.

Example. $S_1 < S_6$ because $(a, 4) < (a, 5)$ and $S_3 < S_8$ because $(b, a, 6) < (b, a, 7)$.

Running Time

\[ T(n) = O(n) + T(2n/3) \]

- time to sort and merge
- array in recursive calls is 2/3rds the size of starting array

Solves to \( T(n) = O(n) \):

- Expand big-O notation: \( T(n) \leq cn + T(2n/3) \) for some \( c \).
- Guess: \( T(n) \leq 3cn \)
- Induction step: assume that is true for all \( i < n \).
- \( T(n) \leq cn + 3c(2n/3) = cn + 2cn = 3cn \) \( \square \)

*slide courtesy of Carl Kingsford*
Handing the 1 and 2 groups

These are called the “sample suffixes”

\[ s = \text{mississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]

\[ \text{ississippi} \]
Key Point #1: The lexicographical order of the suffixes of t is the same as the order of the group 1 & 2 suffixes of s.

Why?

Every suffix of t corresponds to some suffix of s (perhaps with some extra letters at the end of it --- in this case EED)

Because the tokens are sorted in the same order as the triples, the sort order of the suffix of t matches that of s.

So: The recursive computational of the suffix array for t gives you the ordering of the group 1 and group 2 suffixes.
Radix Sort

• O(n)-time sort for n items when items can be divided into constant # of digits.

• Put into buckets based on least-significant digit, flatten, repeat with next-most significant digit, etc.

• Example items: 100 123 042 333 777 892 236

• # of passes = # of digits

• Each pass goes through the numbers once.

*slide courtesy of Carl Kingsford
Handling 0 Suffixes

- First: sort the group 0 suffixes, using the representation \((s[i], S_{i+1})\)
- Since the \(S_{i+1}\) suffixes are already in the array sorted, we can just *stably* sort them with respect to \(s[i]\), using radix sort.

1,2-array: ipp iiss iiss i$$ ppi ssi ssi

0-array: mis pi$ sip sis

- We have to merge the group 0 suffixes into the suffix array for group 1 and 2.
- Given suffix \(S_i\) and \(S_j\), need to decide which should come first.
  - If \(S_i\) and \(S_j\) are both either group 1 or group 2, then the recursively computed suffix array gives the order.
  - If one of \(i\) or \(j\) is 0 \(\pmod{3}\), see next slide.

*slide courtesy of Carl Kingsford*
Comparing 0 suffix $S_j$ with 1 or 2 suffix $S_i$

Represent $S_i$ and $S_j$ using subsequent suffixes:

\[
\begin{align*}
&i \pmod{3} = 1: & i \pmod{3} = 2: \\
&(s[i], S_{i+1}) < (s[j], S_{j+1}) & (s[i], s[i+1], S_{i+2}) < (s[j], s[j+1], S_{j+2}) \\
&\equiv 2 \pmod{3} & \equiv 1 \pmod{3} \\
&\equiv 1 \pmod{3} & \equiv 2 \pmod{3}
\end{align*}
\]

⇒ the suffixes can be compared quickly because the relative order of $S_{i+1}$, $S_{j+1}$ or $S_{i+2}$, $S_{j+2}$ is known from the 1,2-array we already computed.

*slide courtesy of Carl Kingsford*
Running Time

\[ T(n) = O(n) + T(2n/3) \]

- Expand big-O notation: \( T(n) \leq cn + T(2n/3) \) for some \( c \).
- Guess: \( T(n) \leq 3cn \)
- Induction step: assume that is true for all \( i < n \).
- \( T(n) \leq cn + 3c(2n/3) = cn + 2cn = 3cn \)

Solves to \( T(n) = O(n) \):

- time to sort and merge
- array in recursive calls is 2/3rds the size of starting array

*slide courtesy of Carl Kingsford*
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


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


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


And there are comparable advances with respect to LCP1
### Suffix array: summary

Suffix array gives us index that is:

(a) Just $m$ integers, with $O(n \log m)$ worst-case query time, but close to $O(n + \log m)$ in practice

or (b) $3m$ integers, with $O(n + \log m)$ worst case

(a) will often be preferable: index for entire human genome fits in ~12 GB instead of > 45 GB
Enhanced Suffix Arrays


Can restore the **full** asymptotic efficiency of suffix trees with a small number of auxiliary tables.

<table>
<thead>
<tr>
<th>Application</th>
<th>Enhanced suffix array</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>suftab</td>
</tr>
<tr>
<td></td>
<td>4$n$ bytes</td>
</tr>
<tr>
<td>esasupermax</td>
<td>✓</td>
</tr>
<tr>
<td>esamum</td>
<td>✓</td>
</tr>
<tr>
<td>esarep</td>
<td>✓</td>
</tr>
<tr>
<td>Ziv–Lempel</td>
<td>✓</td>
</tr>
<tr>
<td>esamatch</td>
<td>✓</td>
</tr>
<tr>
<td>shortest unique sub.</td>
<td>✓</td>
</tr>
<tr>
<td>esams</td>
<td>✓</td>
</tr>
</tbody>
</table>

The operations that can be done optimally in an enhanced suffix array (esa), and the aux. tables required for them.
Many Suffix Array Variants

**Compressed suffix arrays** — require even less space

**Compressed enhanced suffix arrays** — strive for the best of both worlds and allow interesting query times like $O(n \log |\Sigma| + k)$ for finding $k$ occurrences of a pattern, where $|\Sigma|$ is the size of the alphabet (independent of $m$).
