Indexing with substrings

Ben Langmead

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Department of Computer Science

You are free to use these slides. If you do, please sign the guestbook (www.langmead-lab.org/teaching-materials), or email me (ben.langmead@gmail.com) and tell me briefly how you're using them. For original Keynote files, email me.

Preprocessing

We saw the naive algorithm and Boyer-Moore. These and other algorithms can be distinguished by the kind of *preprocessing* they do.

Naive algorithm does no preprocessing

Nothing for algorithm to do until it is given both pattern P and text T

Boyer-Moore preprocesses the pattern P

If P is provided first, we can build lookup tables for the bad character and good suffix rules

If T_1 is provided later on, we use the already-built tables to match P to T_1

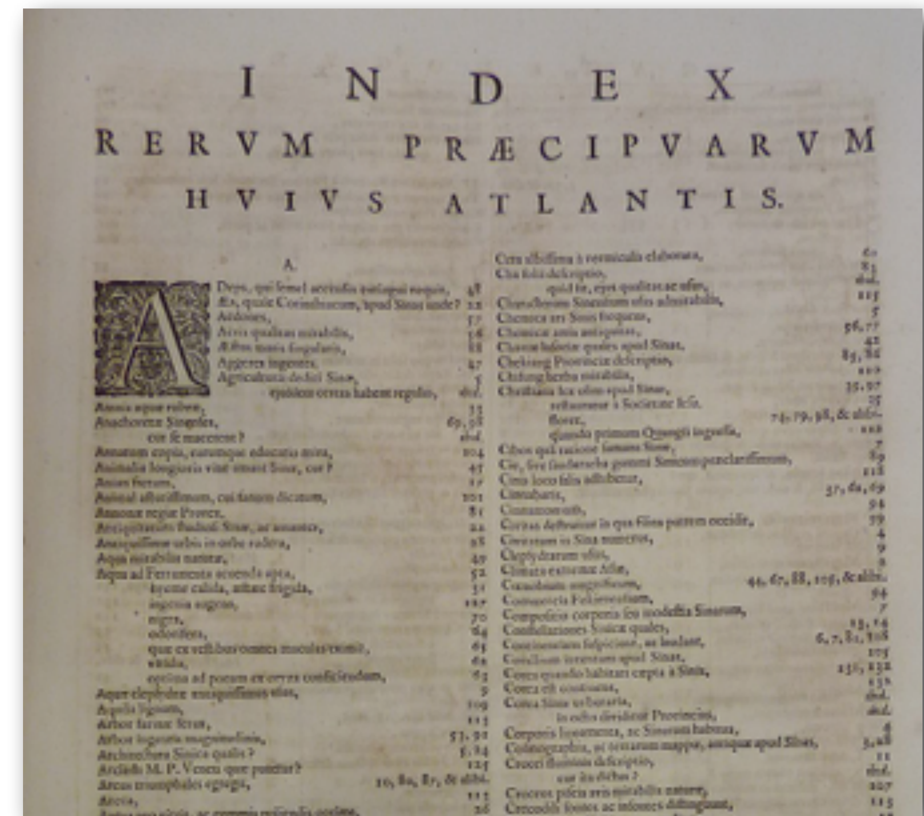
If T_2 is provided later on, we use the already-built tables to match P to T_2

...

Indexing

Can preprocess *P*, *T* or both. When preprocessing *T*, we say we are making an *index* of *T*, like the index of a book.

Sometimes called *inverted indexing,* since it inverts the word/page relationship


I N D E X
R E R V M P R Æ C I P V A R V M
H V I V S A T L A N T I S.

http://en.wikipedia.org/wiki/Index_(publishing)#mediaviewer/File:Atlas_maior_1655_-_vol_10_-_Novus_Atlas_Sinensis_-_index_-_P1080375.JPG

Web seach engines also use inverted indexing

Documents:

```
T[0] = "it is what it is"
T[1] = "what is it"
T[2] = "it is a banana"
```

http://en.wikipedia.org/wiki/Inverted_index

Index:

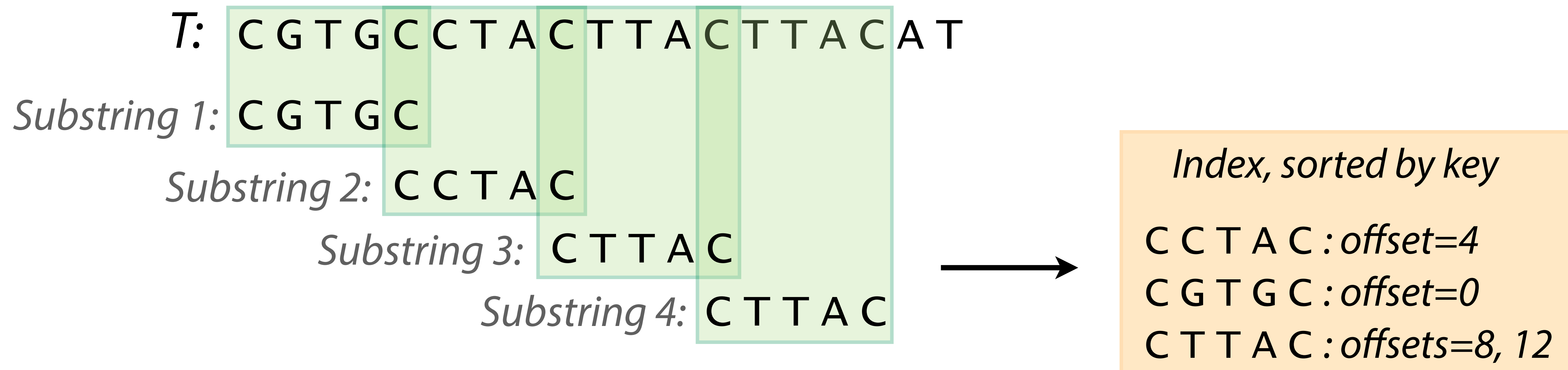
```
"a":      [2]
"banana": [2]
"is":     [0, 1, 2]
"it":     [0, 1, 2]
"what":   [0, 1]
```

Substring index

To index *T*:

Extract sequences from *T* (usually substrings), along with pointers back to where they occurred

Organize pieces and pointers into a *map* data structure. Various map structures are possible, all involving some mix of grouping / sorting.

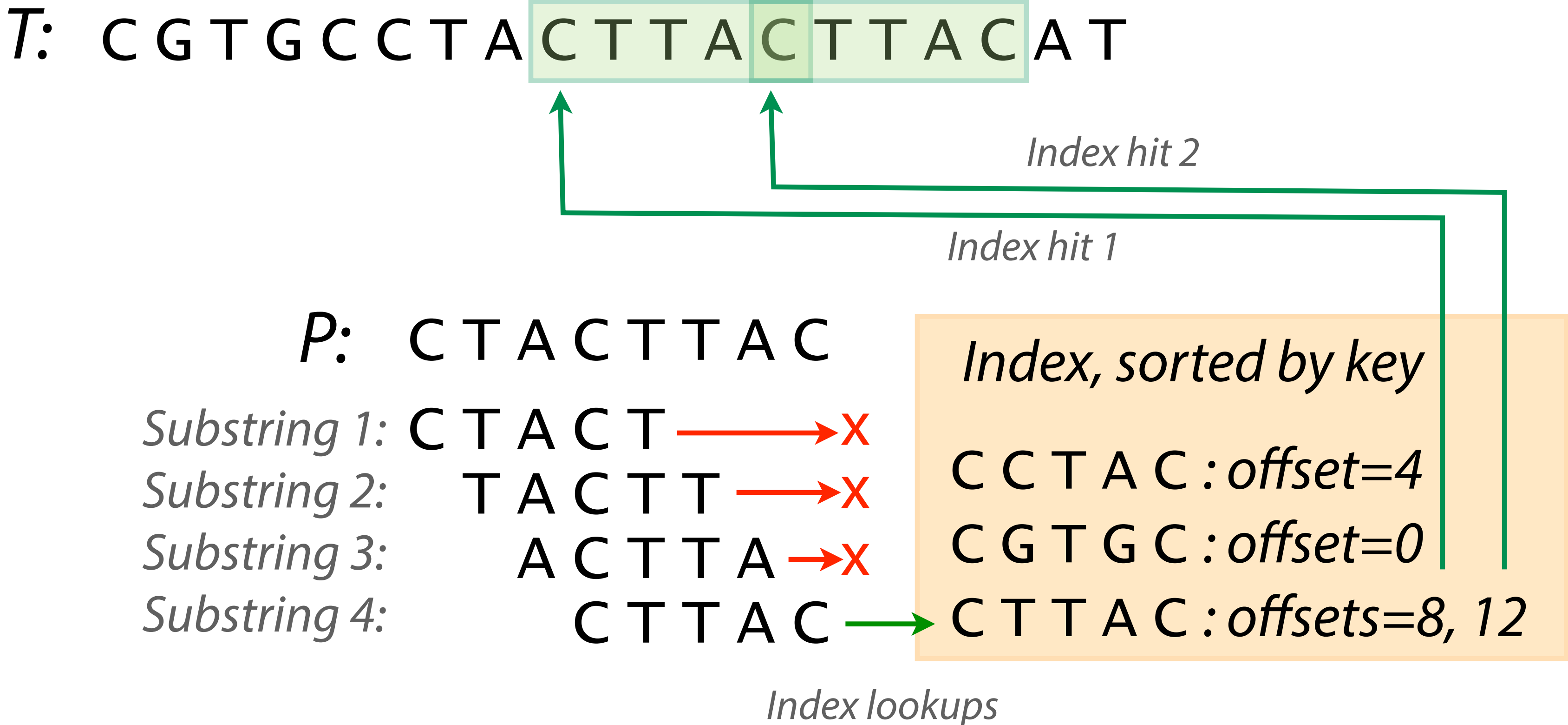


Substring index

To query index with *P*:

Extract substrings from *P*. Use them to query the index. Index tells us where they occur in *T*.

For each occurrence, check if there is a match in that vicinity



See Python example: http://nbviewer.ipython.org/6582444

Substring index

Index for *T*: sorted table of all length-2 substrings, and their offsets

To query: extract length-2 prefix of *P*, look up in index, investigate all hits

P: ord

Two hits to check:

```
T: time_for_such_a_word
P:       ord         ord
```

Hit at offset 6 but no match

Hit at offset 17 matches

T: time_for_such_a_word

Substring	Offset
_a	13
_f	4
_s	8
_w	15
a_	14
ch	11
e_	3
fo	5
h_	12
im	1
me	2
or	6
or	17
r_	7
rd	18
su	9
ti	0
uc	10
wo	16

Substring index: one implementation

```
import bisect
import sys

class Index(object):

    def __init__(self, t, ln=2):
        """ Create index, extracting substrings of length 'ln' """
        self.ln = ln
        self.index = []
        for i in xrange(0, len(t)-ln+1):
            self.index.append((t[i:i+ln], i)) # add <substr, offset> pair
        self.index.sort() # sort pairs

    def query(self, p):
        """ Return candidate alignments for p """
        st = bisect.bisect_left(self.index, (p[:self.ln], -1))
        en = bisect.bisect_right(self.index, (p[:self.ln], sys.maxint))
        hits = self.index[st:en]
        return [ h[1] for h in hits ] # return just the offsets

def queryIndex(p, t, index):
    """ Look for occurrences of p in t with help of index """
  ln = index.ln
    occurrences = []
    for i in index.query(p):
        if t[i+ln:i+len(p)] == p[ln:]:
            occurrences.append(i)
    return occurrences
```

bisect module implements binary search for us

Extract <substring, offset> pairs, put in list, sort list

Binary search for first & last entries matching substring

Get index hits, check for complete matches

```
>>> t = "time for such a word"
>>> ind = Index(t, ln=2, interval=2)
>>> queryIndex("ord", t, ind)
[17]
```

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Substring index: comparison, part 1

Comparing simple Python implementations Boyer-Moore exact matching and an index like on previous slide, using length-4 substrings:

	Boyer-Moore		Sorted index of length-4 substrings				
	# character comparisons	wall clock time	# character comparisons	# index hits	wall clock time (query)	wall clock time (indexing)	
P: "tomorrow" **T**: Shakespeare's complete works	785,855	1.91s	68	17	0.00 s	10.22 s	17 matches $\|T\| = 5.59$ M
P: 50 nt string from Alu repeat* **T**: Human reference (hg19) chromosome 1	32,495,111	67.21 s	*Very slow, took >12 GB of memory*				336 matches $\|T\| = 249$ M

* GCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGG

Substring index: every other substring

Can we fix the large memory footprint?

Idea: instead of extracting every length-2 substring, skip some. E.g. skip every other.

To query *T*: extract leftmost 2 length-2 substrings of *P*, look up in index, try all candidates. First lookup will find matches at even offsets, second at odd offsets.

T: `time_for_such_a_word`

Index'(T):

Substring	Offset
`_f`	4
`_s`	8
`a_`	14
`h_`	12
`me`	2
`or`	6
`rd`	18
`ti`	0
`uc`	10
`wo`	16

Pieces in Index(T):
```
  me fo _s ch a_ or
 im _f r_ uc _a wo
ti e_ or su h_ _w rd
```

Pieces of P used to query Index(T): `or` ← all hits

T: `time_for_such_a_word` P: `ord`

Substrings:
```
ti  _f  _s  h_  wo
  me  or  uc  a_  rd
```

Pieces of P used to query Index'(T): `or` ← just even; `rd` ← just odd

Substring index: every *N*th substring

...and just as we can take every other substring, we can also take every 3rd, 4th, 5th, etc

If we take every *N*th substring, we must use each of the first *N* substrings of P to query the index

First query finds index hits corresponding to matches at offsets $\equiv 0 \pmod{N}$, second finds hits corresponding to match offsets $\equiv 1 \pmod{N}$, etc.

We'll call *N* the substring *interval*

Substring index: new implementation

```
import bisect
import sys

class Index2(object):

    def __init__(self, t, ln=2, interval=2):
        """ Create index, extracting substrings of length 'ln' every
            'interval' positions """
        self.ln = ln
        self.interval = interval
        self.index = []
        for i in xrange(0, len(t)-ln+1, interval):
            self.index.append((t[i:i+ln], i)) # add <substr, offset> pair
        self.index.sort() # sort pairs

    def query(self, p):
        """ Return candidate alignments for p """
        st = bisect.bisect_left(self.index, (p[:self.ln], -1))
        en = bisect.bisect_right(self.index, (p[:self.ln], sys.maxint))
        hits = self.index[st:en]
        return [ h[1] for h in hits ] # return just the offsets

def queryIndex2(p, t, index):
    """ Look for occurrences of p in t with help of index """
    ln, interval = index.ln, index.interval
    occurrences = []
    for k in xrange(0, interval): # For each offset into interval
        for i in index.query(p[k:]): # For each index hit
            # Test for match
            if t[i-k:i] == p[:k] and t[i+ln:i-k+len(p)] == p[k+ln:]:
                occurrences.append(i-k)
    return sorted(occurrences)
```

Configurable "interval" between substrings extracted from reference

Loop stride

When interval = *x*, extract first *x* substrings from *P* and do lookup for each

```
>>> t = "time for such a word"
>>> ind = Index2(t, ln=2, interval=2)
>>> queryIndex2("ord", t, ind)
[17]
```

Substring index

Python demos for those examples here:
http://nbviewer.ipython.org/6584538

Substring index: comparison, part 2

Comparing simple Python implementations Boyer-Moore exact matching and an index like on previous slide, using length-4 substrings extracted every 4 positions of *T*:

	Boyer-Moore		Sorted index of length-4 substrings, **interval=4**					
	# character comparisons	wall clock time	# character comparisons	# index hits	wall clock time (query)	wall clock time (indexing)	Peak memory footprint	
P: "tomorrow" **T**: Shakespeare's complete works	786 K	1.91s	34	17	0.00 s	13.69 s	150 MB	17 matches | *T* | = 5.59 M
P: 50 nt string from Alu repeat* **T**: Human reference (hg19) chromosome 1	32.5 M	67.21 s	445 K	277 K	0.40 s	59.31 s	7.6 GB	336 matches | *T* | = 249 M

* GCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGG

Inverted indexing: map data structures

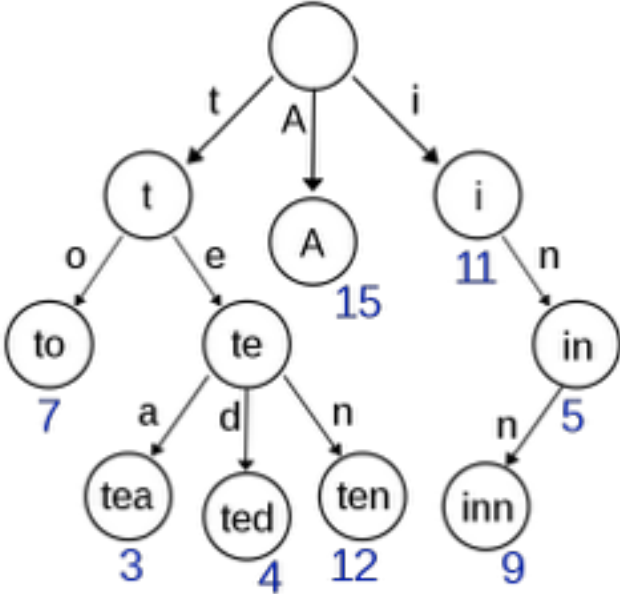
An inverted index maps substrings or subsequences to offsets where they occur

There are many candidate map data structures:

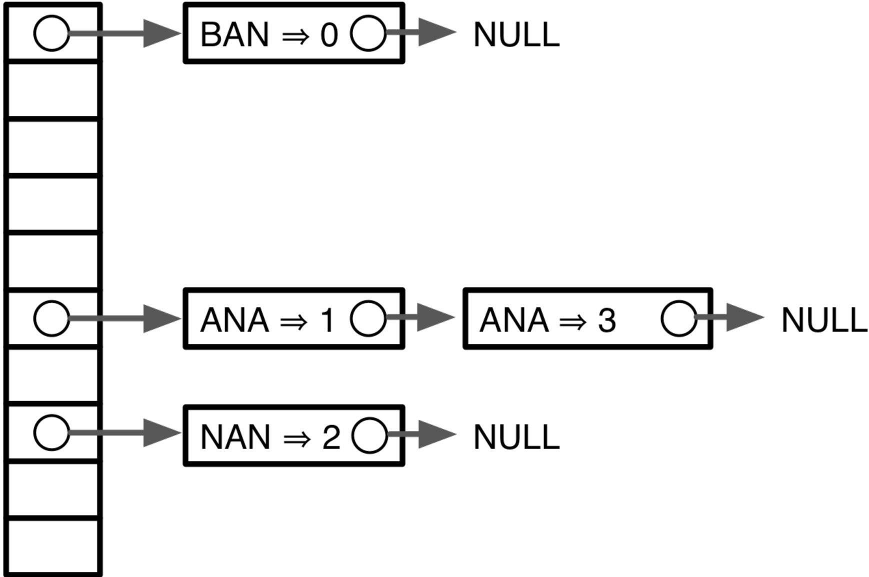
Sorted list:

_f	4
_s	8
a_	14
h_	12
me	2
or	6
rd	18
ti	0
uc	10
wo	16

Trie:

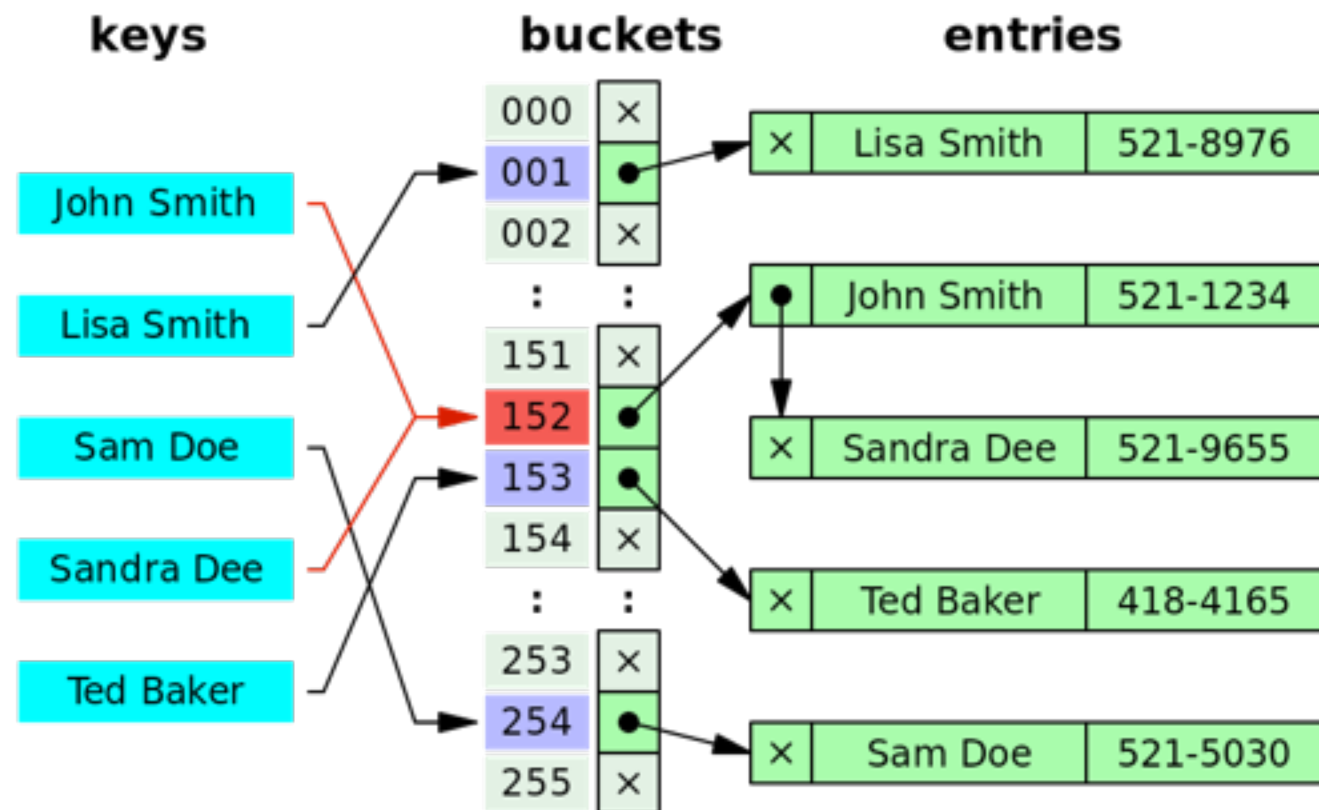


Hash table:



Indexing: quick hash table review

This hash table encodes a map from names to phone numbers:



Hash function maps a name to a *bucket id*: integers in [0, 256) in this case

A bucket is a linked list of <key, value> pairs for keys that fell into that bucket

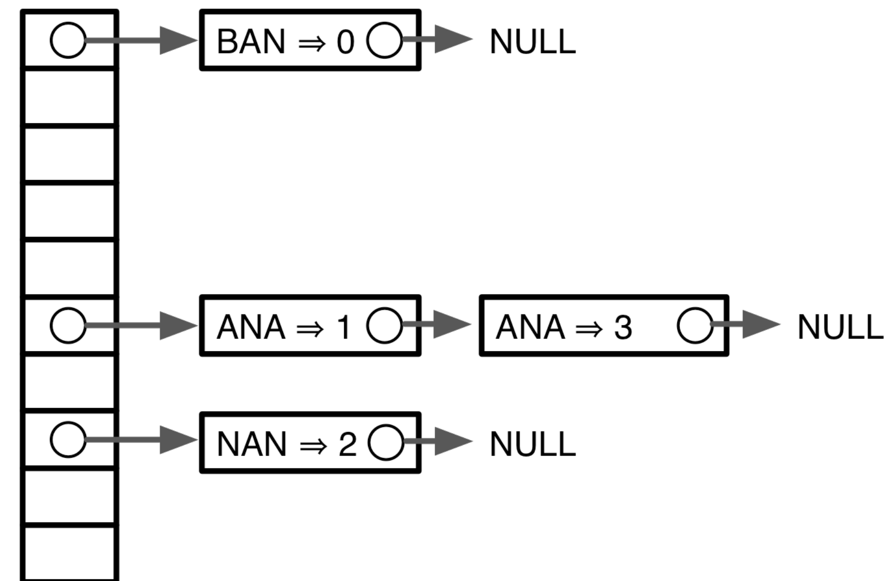
Involves grouping but no sorting. Many, many variations on this theme.

Queries are O(1) time "on average"

http://en.wikipedia.org/wiki/Hash_table

Indexing: hash versus sorted list

_f	4
_s	8
a_	14
h_	12
me	2
or	6
rd	18
ti	0
uc	10
wo	16



O(log *m*) query time

Just stores keys and values

O(1) query time on average

Must store keys, values, bucket array, pointers

Python examples using both: http://nbviewer.ipython.org/6582836

Indexing: specificity

P: ord

Two candidate alignments to check:

```
T: time_for_such_a_word
P:       ord         ord
```

Some index hits are fruitless; i.e. don't correspond to matches of P

Substring	Offset
_a	13
_f	4
_s	8
_w	15
a_	14
ch	11
e_	3
fo	5
h_	12
im	1
me	2
or	6
or	17
r_	7
rd	18
su	9
ti	0
uc	10
wo	16

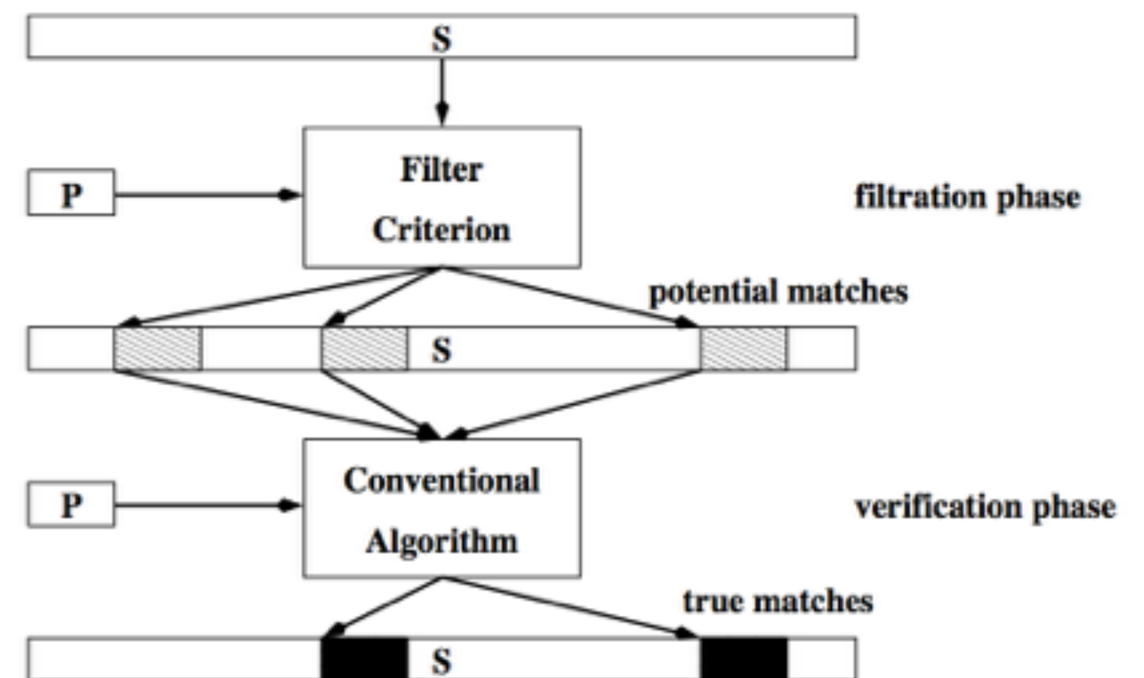
Indexing: specificity

Index-assisted method proceeds in two phases:

1. Index is queried to produce list of *candidate* loci (offsets)
2. Neighborhood around each candidate is checked for complete match

These are sometimes called *filter algorithms*, phase 1 being the filter

Specificity refers to the fraction of candidates from phase 1 that yield matches in phase 2. Higher specificity saves effort spent fruitlessly checking neighborhoods.



From: Burkhardt, Stefan. *Filter algorithms for approximate string matching*. Diss. Universitätsbibliothek, 2002.

Indexing: specificity comparison

Comparing specificities for several combinations of substring-length and interval settings. *P* & *T* are from the human chromosome 1 example.

Substring length	Interval	# character comparisons	# index hits	specificity	wall clock time (query)	wall clock time (indexing)	Peak memory footprint
4	4	445 K	277 K	0.12%	0.40 s	59.31 s	~7.6 GB
7	7	105 K	27 K	1.26%	0.06 s	63.74 s	~5.0 GB
10	10	62 K	8.2 K	4.11%	0.02 s	72.52 s	~3.6 GB
18	18	49 K	7.7 K	4.37%	0.02 s	40.20 s	~2.1 GB
30	30	13 K	1.9 K	11.72%	0.00 s	19.78 s	~1.6 GB

Increasing substring and interval lengths increases specificity, which improves query time on balance. In many cases, the increasing interval improves index size and building time.