Basic notions of algorithms

Mihaela Zavolan

Programming for Life Sciences 2024

Why do we need to talk about algorithms?

Algorithm = specification (‘recipe’) for solving a certain type of problem

- What’s the need?
    - ”Not reinvent the wheel” – once we have an optimal solution to a problem we can keep applying it in whatever context
    - Solutions to many and many different types of problem can be specified in terms of combinations of steps that come from a limited repertoire – e.g. parse a sentence into its parts vs. find secondary structures in RNAs vs. find binding sites for transcription factors in a genome. If we have a solution to one problem and we specify it in abstract terms, then we can apply it to other problems, e.g. by changing the type of variables/entities that we operate with.

What properties should an algorithm have?

- Unambiguous – should be interpretable by a machine in only one way
- General – applicable to a class of problems, should handle all possible parameter values
- Efficient – execution time should be as short as possible
- Memory efficient – should not require more memory than available

Key algorithm analyses:

- Correctness - Invariants
- Resource bounds – Scaling of memory and computing time

Estimating the resource requirements of a program

When we write a program to solve a certain problem we expect to it to return a solution on any specific set of inputs. But will it?

Analysis of algorithms is a branch of computer science that concerns itself with determining the resources (time and memory) that are needed by a specific algorithm to arrive at an answer.

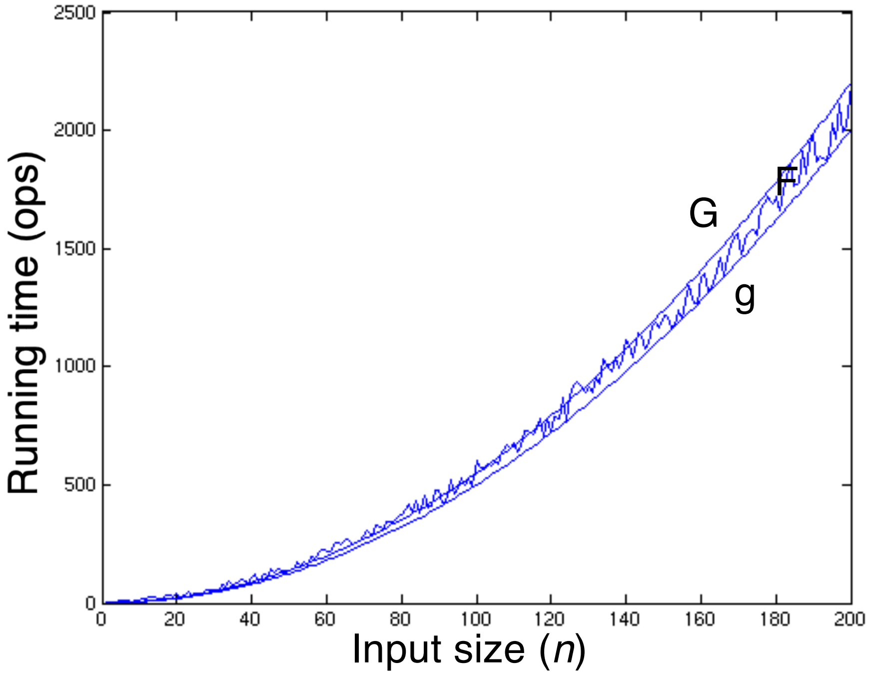
Important are not only (or less so) the absolute estimates of time and memory that the program needs when running on a specific input, but rather how these quantities scale with the size of the input.

Let’s say we want to find out whether a specific sequence has been deposited in the NCBI sequence database. How long do we expect this to take?

Similarly, let’s say that we want to find out whether there is any peptide annotated in a given genome that has a predicted mass equal to a mass that we measured experimentally. How long would this take?

Time complexity of a program: Big-O/little-o notation

Assume that the running time of a program defined as the number of ‘atomic’ operations (additions/substractions/comparisons/etc) varies with the size of the input as shown in the figure, and let this empirical function be F(n).



We say that

G(n) is O(F(n)) if F(n) ≤ G(n)
g(n) is o(F(n)) if F(n) ≥ g(n)

for all n sufficiently large.

What we are interested in is the functions G and g that satisfy the properties above, and are as close as possible to F.

These will have the form

a * F + b

with a and b constants.

Time complexity of a program: Big-O/little-o notation

Let’s take an example: a program that reads in a sequence of numbers and then computes their mean and variance. In Python, the program would look like this:

```
import re

p = re.compile(’[^\d\.\-\+]’, re.IGNORECASE)
inputSeq = []

input_var = ‘yes’
while(input_var.lower() != 'quit' ):
          input_var = input("Enter a number or 'quit' to stop: ")
          if(input_var.lower() != 'quit' and not p.search(input_var)):
                    print("Adding " + str(input_var) + " to the list")
                    inputSeq.append(float(input_var))

mean = 0.0
var = 0.0
num_elem = len(inputSeq)

for i in range(num_elem):
          mean += inputSeq[i]
          var += (inputSeq[i] * inputSeq[i])

mean /= num_elem
var /= num_elem

print("Mean is " + str(mean) + " and variance is " + str(var - mean * mean))
```

How does the run time of this program scales with the size of the input?

Ubiquitous computational tasks: sort and search

Let’s take another toy example: we get as input a list of numbers and we want to output it sorted.

A simple way to do it:

```
define the start index of the unsorted part of the list to be 0 (first element)
iterate until start index < L
    find the smallest element in the list that starts at start index start and ends at L
    swap this element with the element at start index
```

12	4	5	18	9	7	11
0						L-1

start = 0

min = 12	min = 4	min = 4	min = 4	min = 4	min = 4	min = 4
12	4	5	18	9	7	11

start = 1

	min = 12	min = 5	min = 5	min = 5	min = 5	min = 5
4	12	5	18	9	7	11

start = 2

		min = 12	min = 12	min = 9	min = 7	min = 7
4	5	12	18	9	7	11

How does the running time scale with the length of the list?

Ubiquitous computational tasks: sort and search

Iteration 1: L-1 comparisons
Iteration 2: L-2 comparisons
…
Iteration L-1: 1 comparison

$$(L-1)+(L-2)+\ ...+1=\frac{L(L-1)}{2}=O(L^2)$$

	min = 12	min = 4	min = 4	min = 4	min = 4	min = 4	min = 4
start = 0	12	4	5	18	9	7	11

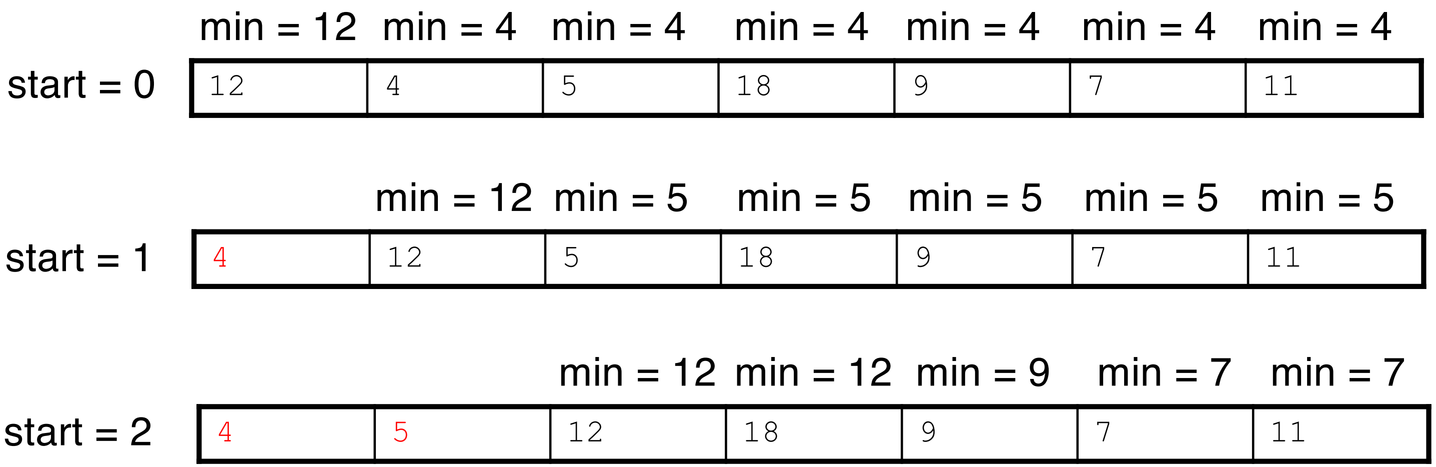
		min = 12	min = 5	min = 5	min = 5	min = 5	min = 5
start = 1	4	12	5	18	9	7	11

			min = 12	min = 12	min = 9	min = 7	min = 7
start = 2	4	5	12	18	9	7	11

How does the running time scale with the length of the list?

Ubiquitous computational tasks: sort and search

How would you prove the correctness of this algorithm?



start = 0

min = 12	min = 4	min = 4	min = 4	min = 4	min = 4	min = 4
12	4	5	18	9	7	11

start = 1

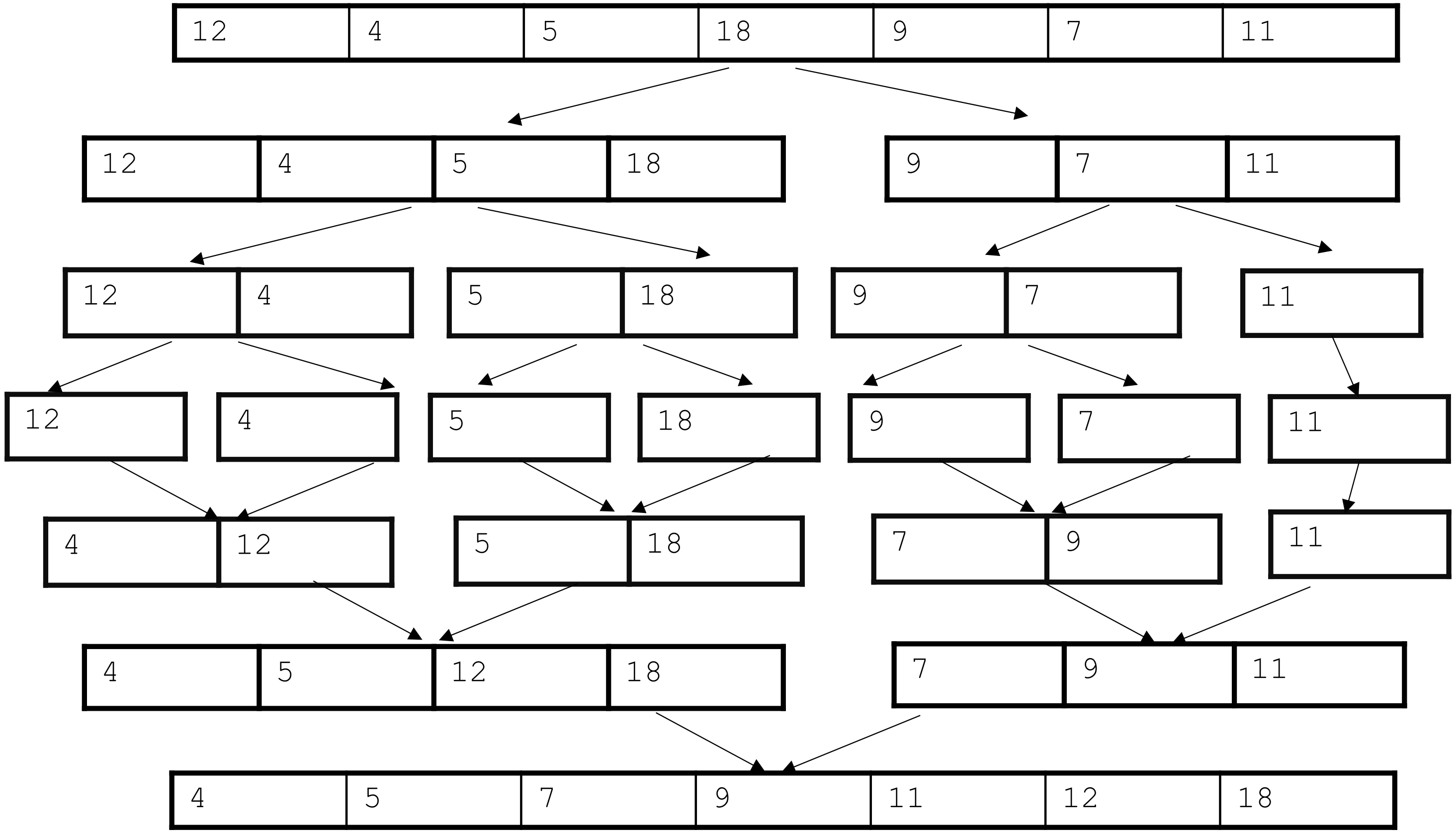
	min = 12	min = 5	min = 5	min = 5	min = 5	min = 5
4	12	5	18	9	7	11

start = 2

		min = 12	min = 12	min = 9	min = 7	min = 7
4	5	12	18	9	7	11

Can we do better than quadratic time in sorting an array?

Merge sort


12 4 5 18 9 7 11
12 4 5 18
9 7 11
12 4
5 18
9 7
11
12
4
5
18
9
7
11
4 12
5 18
7 9
11
4 5 12 18
7 9 11
4 5 7 9 11 12 18

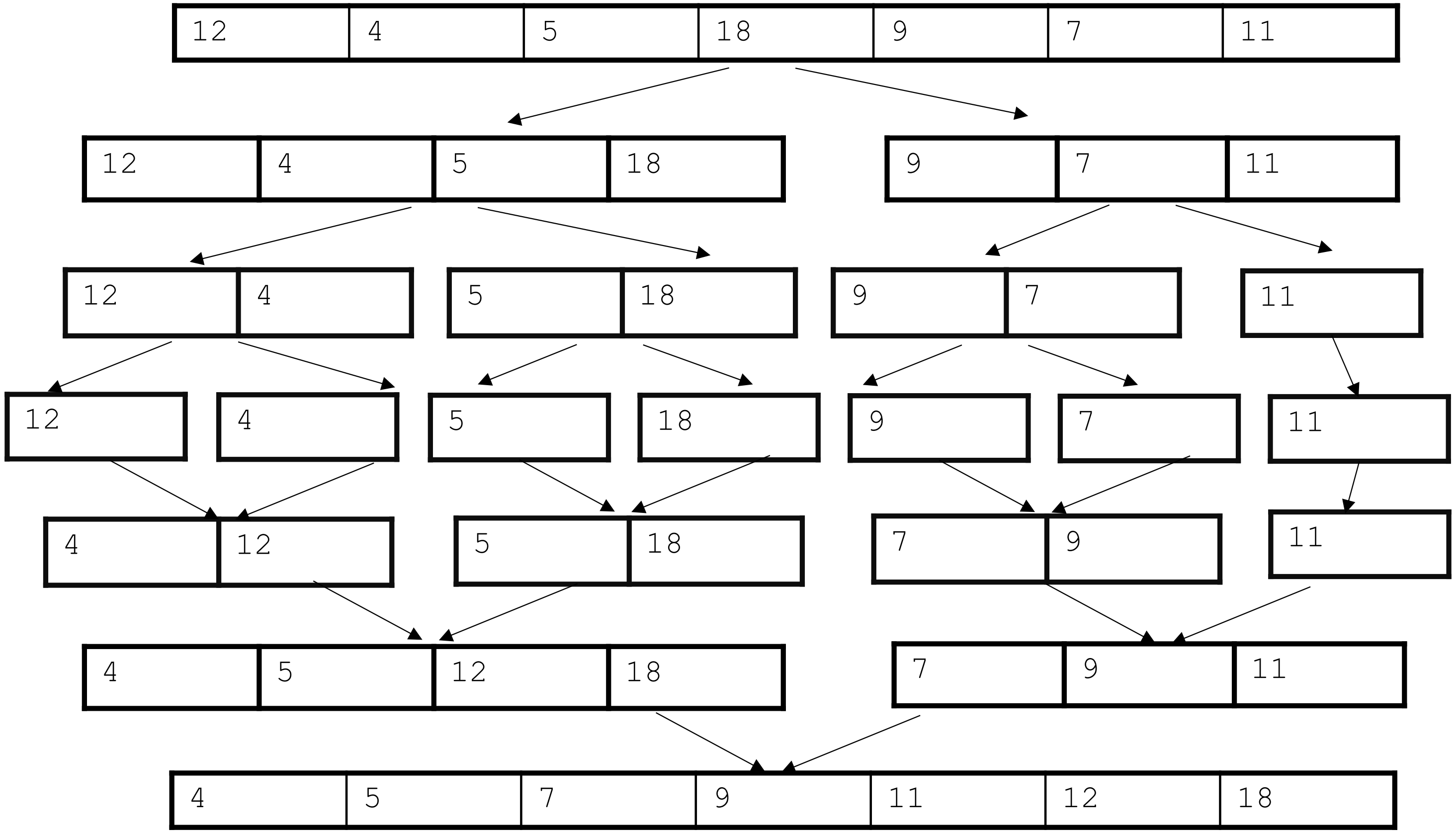
Merge sort

```
def mergeSort(A):
        if len(A) < 2:
                return
        left, right = A[ : len(A) // 2], A[len(A) // 2 : ]
        mergeSort(left)
        mergeSort(right)
        merge(left, right, A)

def merge(left, right, A):
        i, j, k = 0, 0, 0
        while i < len(left) and j < len(right):
                if left[i] <= right[j]:
                        A[k], i, k = left[i], i + 1, k + 1
                 else:
                        A[k], j, k = right[j], j + 1, k + 1
        while i < len(left):
                A[k], i, k = left[i], i + 1, k + 1
        while j < len(right):
                A[k], j, k = right[j], j + 1, k + 1
```

Merge sort

How many operations?



Copy	Compare
L	
L	
L	
L	L
L	L
L	L

Roughly c L steps per recursion level. How many recursion levels?

Merge sort

How many operations?

More generally, we want to stimate the time complexity of an algorithm that, at each step, divides the problem into 2, solves the subproblems and then it takes linear time to merge the solutions of the two subproblems.

Without worrying about integer divisions, this means: $T(N) = 2\ T(N/2) + N$.

Here we have to use a trick: ignoring that we deal with discrete steps, and writing N as 2^n we have

$$T(2^n) = 2 * T(2^{n-1}) + 2^n$$

Dividing both sides by 2^n, we obtain

$T(2^n)/2^n = T(2^{n-1})/2^{n-1} + 1$ and telescoping further we get

$$\begin{aligned} T(2^n)/2^n &= T(2^{n-1})/2^{n-1} + 1 = (2 * T(2^{n-2}) + 2^{n-1})/\ 2^{n-1} + 1 \\ &= T(2^{n-2})/2^{n-2} + 1 + 1 \\ &\dots \\ &= 1 + 1 + \dots + 1 \ (n \text{ times}) \end{aligned}$$

Thus $T(2^n)/2^n = n$, and if we make the reverse substition, $2^n = N$, we get $T(N) = N * \log_2(N)$.

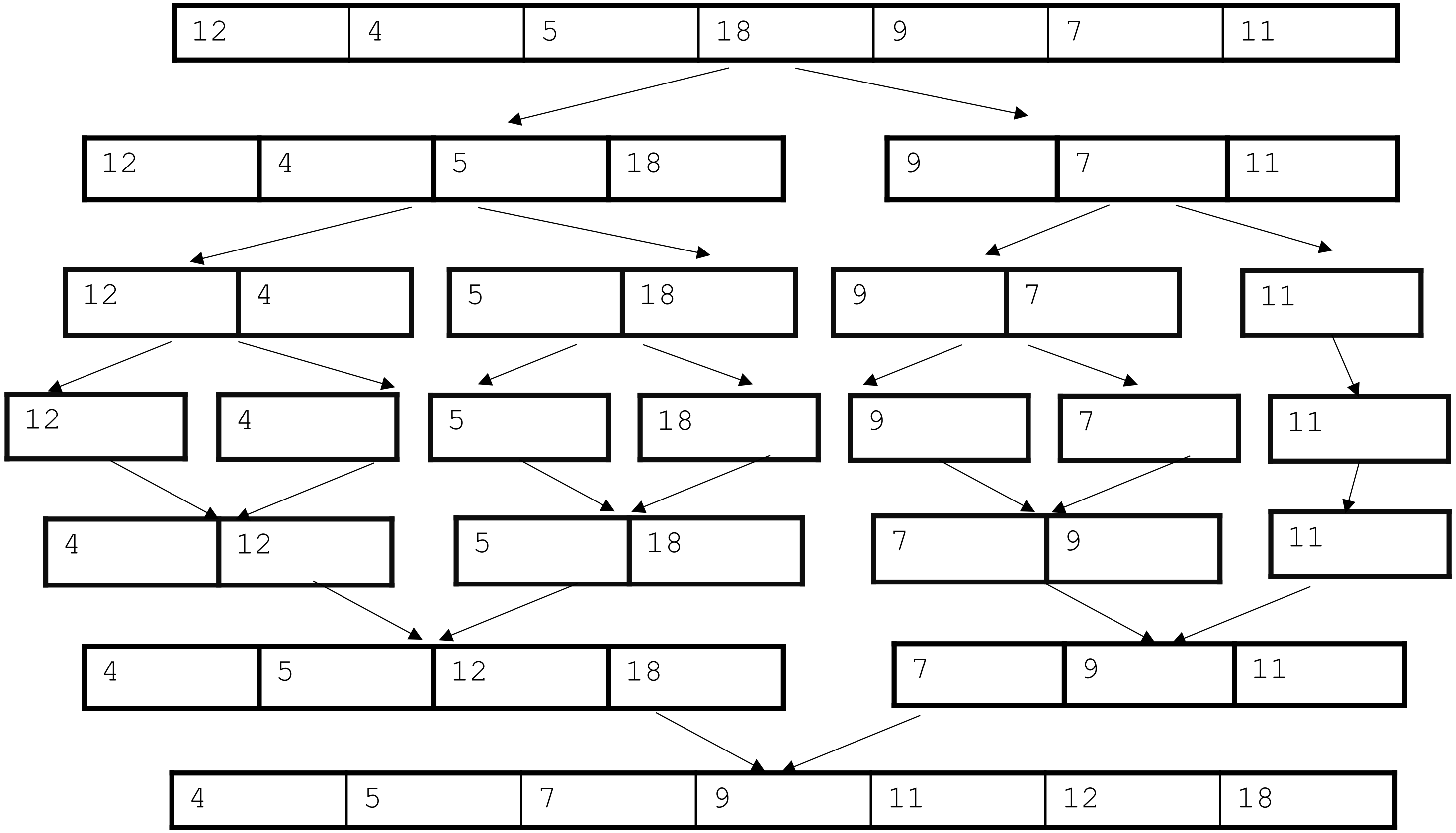
This is the complexity of the merge sort algorithm.

Trading space for time

12	4	5	18	9	7	11

Bubble sort – in place

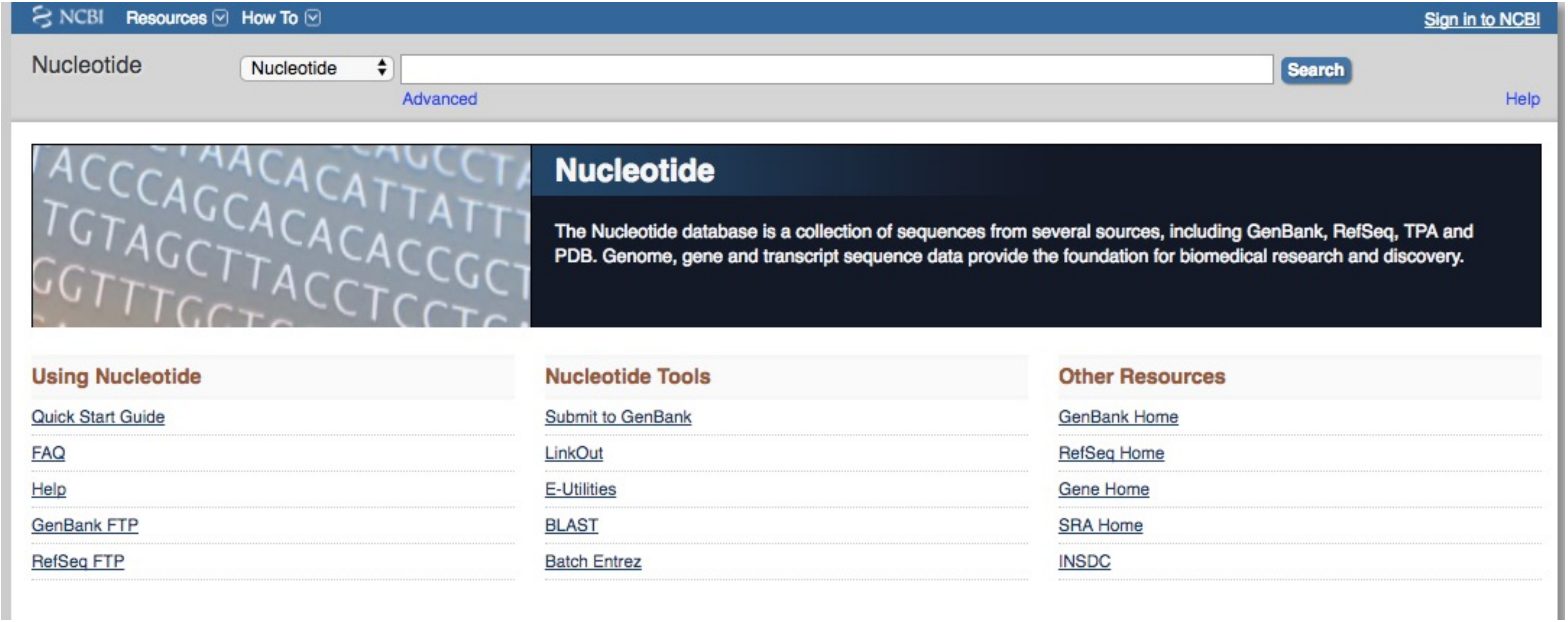
Merge sort – multiple copies of the list



Database searching

NCBI's 'Nucleotide' repository of sequence information (https://www.ncbi.nlm.nih.gov/genbank/statistics/)

Release	Date	Bases (GenBank)	Sequences	Bases (WGS)	Sequences
262	Aug 2024	3'675'462'701'077	251'998'350	29'643'594'176'326	3'569'715'357



The Nucleotide database is a collection of sequences from several sources, including GenBank, RefSeq, TPA and PDB. Genome, gene and transcript sequence data provide the foundation for biomedical research and discovery.

How can we search this large database efficiently?

Upper bound: linear traversal of the sequences in the DB, at each position looking for a perfect match

DB organization

Time to find the perfect match: logarithmic in the size of the database

Lower bound: constant time identification of sequence-positions where a perfect match is to be found

Binary search

Let’s say that we have a list of objects that is sorted by some measure.

We are given a new object and we want to find out whether it is already present in the list.

Binary search

A concrete example: we have a list of ions that can be obtained in a mass spectrometry experiment, sorted by their mass/charge ratio.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
m/z	15	16	17	18	19	26	27	28	29	30	31	32	33	34	35	36	39	41	42	43	44	45	46
Ions	CH_3	O	OH	H_2O	F	CN	C_2H_3	C_2H_4 CO N_2	C_2H_5 CHO	CH_2NH_2 NO	CH_2OH OCH_3	O_2	SH	H_2S	Cl	HCl	C_3H_3	C_2H_5	C_2H_6	C_2H_7	CO_2	COOH	NO_2

We are given a new m/z (say 31) and we want to find out whether it (and its associated ion) is already in the list.

The way we will do it is by repeatedly halving the interval in which the number can possibly be.

Binary search

```
def binarySearch (num, numList):
        begin = 0
        end = len(numList)-1
        while(end > begin):
                mid = int((begin+end)/2)
                if(num == numList[mid]):
                        return True
                elif(num > numList[mid]):
                        begin = mid + 1
                elif (num < numList[mid]):
                        end = mid
        if(end == begin):
                if(num == numList[end]):
                        return True
        return False
```

Binary search

Let’s take a simple example: we have a list of ions that can be obtained in a mass spectrometry experiment, sorted by their mass/charge ratio.

i	**0**	1	2	3	4	5	6	7	8	9	10	**11**	12	13	14	15	16	17	18	19	20	21	**22**
m/z	15	16	17	18	19	26	27	28	29	30	31	32	33	34	35	36	39	41	42	43	44	45	46
Ions	CH_3	O	OH	H_2O	F	CN	C_2H_3	C_2H_4 CO N_2	C_2H_5 CHO	CH_2NH_2 NO	CH_2OH OCH_3	O_2	SH	H_2S	Cl	HCl	C_3H_3	C_2H_5	C_2H_6	C_2H_7	CO_2	COOH	NO_2

We are given a new m/z (say 31) and we want to find out whether it (and its associated ion) is already in the list.

Binary search

Let’s take a simple example: we have a list of ions that can be obtained in a mass spectrometry experiment, sorted by their mass/charge ratio.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
m/z	15	16	17	18	19	26	27	28	29	30	31	32	33	34	35	36	39	41	42	43	44	45	46
Ions	CH_3	O	OH	H_2O	F	CN	C_2H_3	C_2H_4 CO N_2	C_2H_5 CHO	CH_2NH_2 NO	CH_2OH OCH_3	O_2	SH	H_2S	Cl	HCl	C_3H_3	C_2H_5	C_2H_6	C_2H_7	CO_2	COOH	NO_2

We are given a new m/z (say 31) and we want to find out whether it (and its associated ion) is already in the list.

Binary search

Let’s take a simple example: we have a list of ions that can be obtained in a mass spectrometry experiment, sorted by their mass/charge ratio.

i	0	1	2	3	4	5	**6**	7	**8**	9	10	**11**	12	13	14	15	16	17	18	19	20	21	22
m/z	15	16	17	18	19	26	27	28	29	30	31	32	33	34	35	36	39	41	42	43	44	45	46
Ions	CH_3	O	OH	H_2O	F	CN	C_2H_3	C_2H_4 CO N_2	C_2H_5 CHO	CH_2NH_2 NO	CH_2OH OCH_3	O_2	SH	H_2S	Cl	HCl	C_3H_3	C_2H_5	C_2H_6	C_2H_7	CO_2	COOH	NO_2

We are given a new m/z (say 31) and we want to find out whether it (and its associated ion) is already in the list.

Binary search

Let’s take a simple example: we have a list of ions that can be obtained in a mass spectrometry experiment, sorted by their mass/charge ratio.

i	0	1	2	3	4	5	6	7	8	**9**	**10**	**11**	12	13	14	15	16	17	18	19	20	21	22
m/z	15	16	17	18	19	26	27	28	29	30	31	32	33	34	35	36	39	41	42	43	44	45	46
Ions	CH_3	O	OH	H_2O	F	CN	C_2H_3	C_2H_4 CO N_2	C_2H_5 CHO	CH_2NH_2 NO	CH_2OH OCH_3	O_2	SH	H_2S	Cl	HCl	C_3H_3	C_2H_5	C_2H_6	C_2H_7	CO_2	COOH	NO_2

We are given a new m/z (say 31) and we want to find out whether it (and its associated ion) is already in the list.

Binary search

Let’s take a simple example: we have a list of ions that can be obtained in a mass spectrometry experiment, sorted by their mass/charge ratio.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	**15**	16	17	18	19	20	21	22
m/z	15	16	17	18	19	26	27	28	29	30	31	32	33	34	35	36	39	41	42	43	44	45	46
Ions	CH_3	O	OH	H_2O	F	CN	C_2H_3	C_2H_4 CO N_2	C_2H_5 CHO	CH_2NH_2 NO	CH_2OH OCH_3	O_2	SH	H_2S	Cl	HCl	C_3H_3	C_2H_5	C_2H_6	C_2H_7	CO_2	COOH	NO_2

We are given a new m/z (say 38) and we want to find out whether it (and its associated ion) is already in the list.

Binary search

How does the running time of binary search scale with the size of the list?

Analogous to merge sort, at every iteration we reduce the size of the interval in which we have to look.

$$T(N) = T\left(\frac{N}{2}\right) + c = T\left(\frac{N}{4}\right) + c + c = \cdots = T(1) + c \ldots + c$$

And with the same substitution trick

$$T(2^n) = T(2^{n-1}) + c = T(2^{n-2}) + 2c = \cdots = nc$$

$$T(N) = c\ log_2(N)$$

Trees

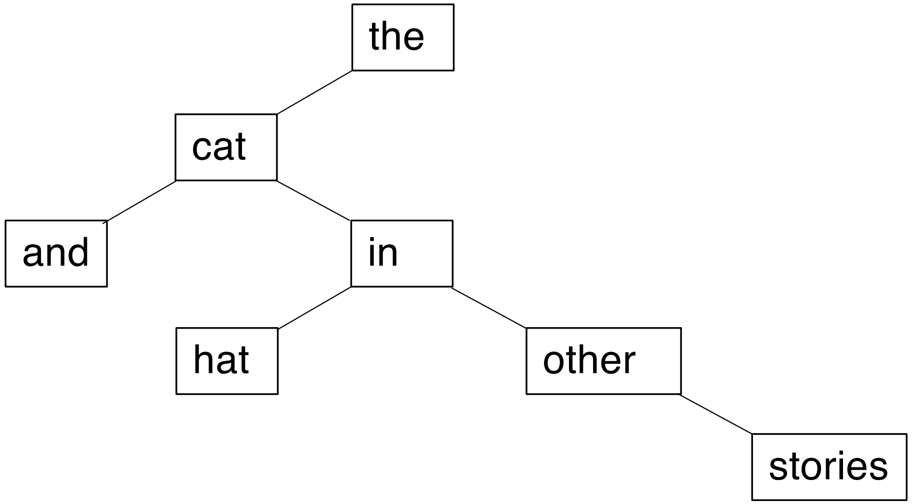
Tree data structures support searches in $O(\log(n))$ time, n being the number of items in the tree.

They exploit some underlying order in the data.

For example, let's say that we have some message, composed of words (could be the genome and the genes in it) and we want to find out if arbitrary input words occur in that message.

the cat in the hat and other stories

We organize the message in a binary search tree, which satisfies the condition that all nodes in the left subtree rooted at a specific node have 'smaller' key values than the key of the root, and all nodes in the right subtree have higher key values.



Note: here we compare key values lexicographically.

Difference with respect to array?

Importance of tree structure

Given the list

L = [‘the’, ‘cat’, ‘in’, ‘the’, ‘hat’, ’and’, ’other’, ‘stories’]

Construct the binary tree of the words that occur in the list, inserting all elements of the list into the binary tree, one at a time, from left-most to right-most.

What does the tree that you get look like?

How many steps do you need, on average, to find one of these words in the constructed binary tree?

Now do the same for the list

L’ = [‘and’, ‘cat’, ‘hat’, ‘in’, ‘other’, ‘stories’, ’the’]

What do you conclude?

Estimating time/memory requirements

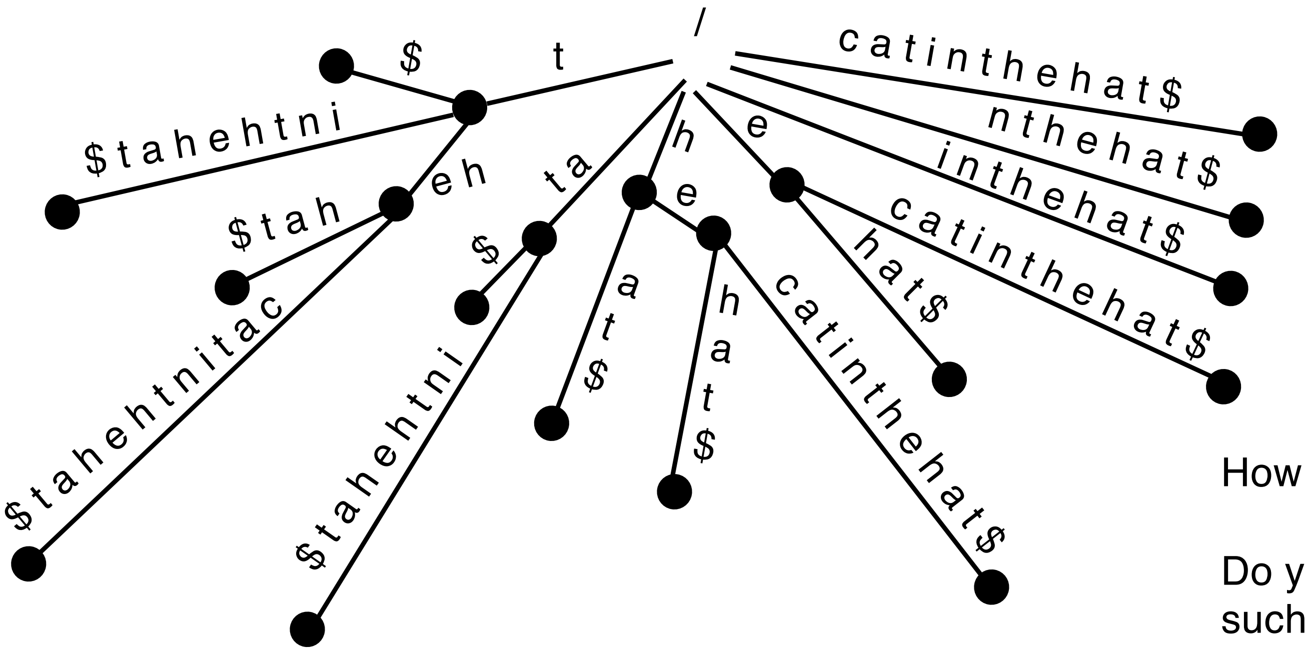
1. Looping through a list – searching, copying, passing list as argument to function
2. Nested for loops – traversing all elements of a matrix, searching multiple elements in a database
3. Order of magnitude calculation – determine the number of digits of a number, search in a *n*-ary tree

Suffix trees

For large sequence data such as genomes, a special kind of tree structure is frequently used, the suffix tree.

This exploits more of the structure in the underlying data (keeping track of the sequence in which letters occur).

thecatinthehat$



How would you prove this?

Do you see any catch in having such an efficient search?

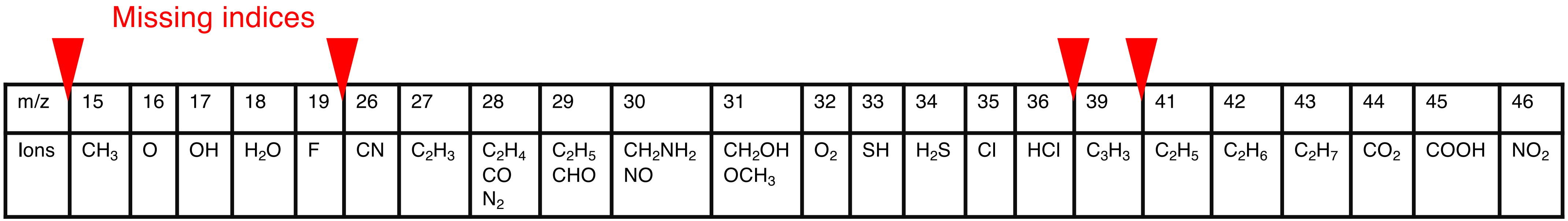
Search time with a suffix tree is linear in the query length.

Hashing

Supports search operations in constant time, O(1).

Idea: store elements in a table, at indices that can be computed directly from the elements in constant time.

An example we have seen before:



Missing indices

m/z	15	16	17	18	19	26	27	28	29	30	31	32	33	34	35	36	39	41	42	43	44	45	46
Ions	CH_3	O	OH	H_2O	F	CN	C_2H_3	C_2H_4 CO N_2	C_2H_5 CHO	CH_2NH_2 NO	CH_2OH OCH_3	O_2	SH	H_2S	Cl	HCl	C_3H_3	C_2H_5	C_2H_6	C_2H_7	CO_2	COOH	NO_2

Hashing

Supports search operations in constant time, O(1).

Idea: store elements in a table, at indices that can be computed directly from the elements in constant time.

An example we have seen before:

m/z	15	16	17	18	19	26	27	28	29	30	31	32	33	34	35	36	39	41	42	43	44	45	46
Ions	CH_3	O	OH	H_2O	F	CN	C_2H_3	C_2H_4 CO N_2	C_2H_5 CHO	CH_2NH_2 NO	CH_2OH OCH_3	O_2	SH	H_2S	Cl	HCl	C_3H_3	C_2H_5	C_2H_6	C_2H_7	CO_2	COOH	NO_2

Difference from the sorted list: we could not drop indices. The hash table would have 46 entries, only 23 populated with values, the others empty (or 23 populated with the value True and the rest with the value False).

Main challenges with hash tables: ensure that they are
densely populated
without clashes (multiple elements hashing to the same index).

Applications of hashing

Dictionaries: indexing by strings

```
#string argument has to be a sequence of bytes
#b’stringValue’ #default representation utf-8
def hash(string):
       x = string[0] << 7
       for chr in string[1:]:
              x = ((1000003 * x) ^ chr) & (1<<32)
       return x
```

Applications of hashing

Data transfer, computer security etc.

SHA – Secure Hash Algorithm

Name	Input (message) size (bits)	Output size (bits)
MD5	Unlimited	128
SHA-1	2^{64}-1	160
SHA-2	2^{64}-1/2^{128}-1	224-512 (depending on variant)
SHA-3	Unlimited	224-arbitrary (depending on variant)

Standard data structures and their methods

1. Queue
2. Stack
3. Tree
4. Heap
5. Graphs

- What operations (methods) are defined on these data structures?
- How do they scale with the size of the data structure?
- What are shallow and deep copies?
- What are stable vs unstable algorithms?

Algorithms in bioinformatics

- **Global Alignment :**
  - Needleman-Wunsch
- **Local Alignment :**
  - Smith-Waterman
  - BLAST
- **Multiple Sequence Alignment:**
  - ProbCons
- **Motif finding**
  - GibbsSampler
  - MEME
- **Assembly :**
  - Debrujin Graph Assembly
  - Euler Assembly
- **Read Mapping :**
  - Suffix Trees
  - Burrows Wheeler Alignment

Improving efficiency – Burroughs-Wheeler transform

Suffix trees take a lot of space to save time. We can construct a much more space-efficient data structure.

```
thecatinthehat$
012345678911111
          01234
```

Make all circular permutations		Sort lexicographically		Lexicographically-sorted suffices		BWT	
0	thecatinthehat$	14	$thecatinthehat	14	$thecatinthehat	14	$thecatintheha**t**
14	$thecatinthehat	12	at$thecatintheh	12	at$thecatintheh	12	at$thecatinthe**h**
13	t$thecatintheha	4	atinthehat$thec	4	atinthehat$thec	4	atinthehat$the**c**
12	at$thecatintheh	3	catinthehat$the	3	catinthehat$the	3	catinthehat$th**e**
11	hat$thecatinthe	2	ecatinthehat$th	2	ecatinthehat$th	2	ecatinthehat$t**h**
10	ehat$thecatinth	10	ehat$thecatinth	10	ehat$thecatinth	10	ehat$thecatint**h**
9	hehat$thecatint	11	hat$thecatinthe	11	hat$thecatinthe	11	hat$thecatinth**e**
8	thehat$thecatin	1	hecatinthehat$t	1	hecatinthehat$t	1	hecatinthehat$**t**
7	nthehat$thecati	9	hehat$thecatint	9	hehat$thecatint	9	hehat$thecatin**t**
6	inthehat$thecat	6	inthehat$thecat	6	inthehat$thecat	6	inthehat$theca**t**
5	tinthehat$theca	7	nthehat$thecati	7	nthehat$thecati	7	nthehat$thecat**i**
4	atinthehat$thec	13	t$thecatintheha	13	t$thecatintheha	13	t$thecatinthe**ha**
3	catinthehat$the	0	thecatinthehat$	0	thecatinthehat$	0	thecatinthehat**$**
2	ecatinthehat$th	8	thehat$thecatin	8	thehat$thecatin	8	thehat$thecati**n**
1	hecatinthehat$t	5	tinthehat$theca	5	tinthehat$theca	5	tinthehat$thec**a**

Improving efficiency – Burroughs-Wheeler transform

$t_1h_1e_1c_1a_1t_2i_1n_1t_3h_2e_2h_3a_2t_4\$$
0 1 2 3 4 5 6 7 8 9 1 1 1 1 1
0 1 2 3 4

$t_2h_2e_1c_1a_2t_4i_1n_1t_3h_3e_2h_1a_1t_1\$$
0 1 2 3 4 5 6 7 8 9 1 1 1 1 1
0 1 2 3 4

BWT		F L				
14	$thecatinthehat	**$**thecatinthehat	14	**$** thecatinthehat**t**$_4$	14	**$** thecatinthehat**t**$_1$
12	at$thecatintheh	**a**t$thecatinthe**h**	12	**a**$_2$t$thecatinthe**h**$_3$	12	**a**$_1$t$thecatinthe**h**$_1$
4	atinthehat$thec	**a**tinthehat$the**c**	4	**a**$_1$tinthehat$the**c**$_1$	4	**a**$_2$tinthehat$the**c**$_1$
3	catinthehat$the	**c**atinthehat$th**e**	3	**c**$_1$atinthehat$th**e**$_1$	3	**c**$_1$atinthehat$th**e**$_1$
2	ecatinthehat$th	**e**catinthehat$t**h**	2	**e**$_1$catinthehat$t**h**$_1$	2	**e**$_1$catinthehat$t**h**$_2$
10	ehat$thecatinth	**e**hat$thecatint**h**	10	**e**$_2$hat$thecatint**h**$_2$	10	**e**$_2$hat$thecatint**h**$_3$
11	hat$thecatinthe	**h**at$thecatinth**e**	11	**h**$_3$at$thecatinth**e**$_2$	11	**h**$_1$at$thecatinth**e**$_2$
1	hecatinthehat$t	**h**ecatinthehat$**t**	1	**h**$_1$ecatinthehat$**t**$_1$	1	**h**$_2$ecatinthehat$**t**$_2$
9	hehat$thecatint	**h**ehat$thecatin**t**	9	**h**$_2$ehat$thecatin**t**$_3$	9	**h**$_3$ehat$thecatin**t**$_3$
6	inthehat$thecat	**i**nthehat$theca**t**	6	**i**$_1$nthehat$theca**t**$_2$	6	**i**$_1$nthehat$theca**t**$_4$
7	nthehat$thecati	**n**thehat$thecat**i**	7	**n**$_1$thehat$thecat**i**$_1$	7	**n**$_1$thehat$thecat**i**$_1$
13	t$thecatintheha	**t**$thecatintheh**a**	13	**t**$_4$$thecatintheh**a**$_2$	13	**t**$_1$$thecatintheh**a**$_1$
0	thecatinthehat$	**t**hecatinthehat**$**	0	**t**$_1$hecatinthehat**$**	0	**t**$_2$hecatinthehat**$**
8	thehat$thecatin	**t**hehat$thecati**n**	8	**t**$_3$hehat$thecati**n**$_1$	8	**t**$_3$hehat$thecati**n**$_1$
5	tinthehat$theca	**t**inthehat$thec**a**	5	**t**$_2$inthehat$thec**a**$_2$	5	**t**$_4$inthehat$thec**a**$_2$

Can be stored efficiently:
$1a2c1e2h3i1n1t4

sorted

preceding char

Order of character occurrence is preserved in L column

Relabel by rank in F

Which row begins with the second e-starting suffix?

0 $\mathbf{\$}$ thecatintheha$\mathbf{t}_1$
1 $\mathbf{a}_1$t\$thecatinthe$\mathbf{h}_1$
2 $\mathbf{a}_2$tinthehat\$the$\mathbf{c}_1$
3 $\mathbf{c}_1$atinthehat\$th$\mathbf{e}_1$
4 $\mathbf{e}_1$catinthehat\$t$\mathbf{h}_2$
5 $\mathbf{e}_2$hat\$thecatint$\mathbf{h}_3$
6 $\mathbf{h}_1$at\$thecatinth$\mathbf{e}_2$
7 $\mathbf{h}_2$ecatinthehat\$$\mathbf{t}_2$
8 $\mathbf{h}_3$ehat\$thecatin$\mathbf{t}_3$
9 $\mathbf{i}_1$nthehat\$theca$\mathbf{t}_4$
10 $\mathbf{n}_1$thehat\$thecat$\mathbf{i}_1$
11 $\mathbf{t}_1$\$thecatintheh$\mathbf{a}_1$
12 $\mathbf{t}_2$hecatinthehat$\mathbf{\$}_2$
13 $\mathbf{t}_3$hehat\$thecati$\mathbf{n}_1$
14 $\mathbf{t}_4$inthehat\$thec$\mathbf{a}_2$

If we know the letter count in the text
a : 2
c : 1
e : 2
h : 3
i : 1
n : 1
t : 4

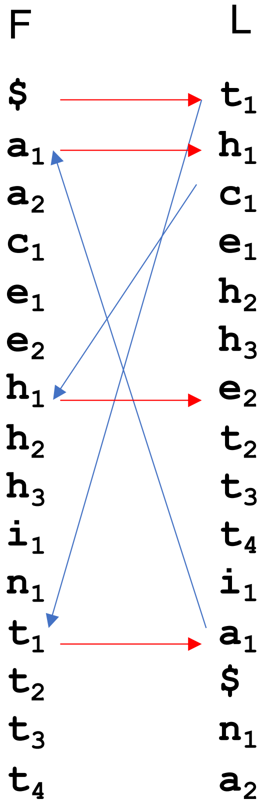
The index of the row where e_2 should be is 2(a) + 1(c) + 2(e) = 5

Can we reconstruct the text if we only have the F and L columns?

We'll do it from the end of the text towards the beginning, knowing that

- the character in L precedes the one in F and
- the rank-order of characters in first and last column is preserved.

F		L
$\$$	` thecatinthehat`	**t_1**
a_1	`t$thecatinthe`	**h_1**
a_2	`tinthehat$the`	**c_1**
c_1	`atinthehat$the`	**e_1**
e_1	`catinthehat$t`	**h_2**
e_2	`hat$thecatint`	**h_3**
h_1	`at$thecatinthe`	**e_2**
h_2	`ecatinthehat$`	**t_2**
h_3	`ehat$thecatin`	**t_3**
i_1	`nthehat$theca`	**t_4**
n_1	`thehat$thecat`	**i_1**
t_1	`$thecatintheha`	**a_1**
t_2	`hecatinthehat`	**$\$_2$**
t_3	`hehat$thecati`	**n_1**
t_4	`inthehat$thec`	**a_2**



F	L
$\$$	**t_1**
a_1	**h_1**
a_2	**c_1**
c_1	**e_1**
e_1	**h_2**
e_2	**h_3**
h_1	**e_2**
h_2	**t_2**
h_3	**t_3**
i_1	**t_4**
n_1	**i_1**
t_1	**a_1**
t_2	**$\$$**
t_3	**n_1**
t_4	**a_2**

$\$$	
$t_1\$$	preceding?
$t_1\$$	row of first t?
$a_1t_1\$$	preceding?
$a_1t_1\$$	row of first a?
$h_1a_1t_1\$$	preceding?
$h_1a_1t_1\$$	row of first h?
$e_2h_1a_1t_1\$$	preceding?
$e_2h_1a_1t_1\$$	row of second e?

⋮

$t_2h_2e_1c_1a_2t_4i_1n_1t_3h_3e_2h_1a_1t_1\$$

Does the substring the occur in the text?

We’ll look for it, again starting with its last letter.

Which row starts with e, the last letter of our search pattern?

F		L	F	L
$	thecatinthehat	$\mathbf{t}_1$	**$**	$\mathbf{t}_1$
$\mathbf{a}_1$	t$thecatinthe	$\mathbf{h}_1$	$\mathbf{a}_1$	$\mathbf{h}_1$
$\mathbf{a}_2$	tinthehat$the	$\mathbf{c}_1$	$\mathbf{a}_2$	$\mathbf{c}_1$
$\mathbf{c}_1$	atinthehat$th	$\mathbf{e}_1$	$\mathbf{c}_1$	$\mathbf{e}_1$
$\mathbf{e}_1$	catinthehat$t	$\mathbf{h}_2$	$\mathbf{e}_1$	$\mathbf{h}_2$
$\mathbf{e}_2$	hat$thecatint	$\mathbf{h}_3$	$\mathbf{e}_2$	$\mathbf{h}_3$
$\mathbf{h}_1$	at$thecatinth	$\mathbf{e}_2$	$\mathbf{h}_1$	$\mathbf{e}_2$
$\mathbf{h}_2$	ecatinthehat$	$\mathbf{t}_2$	$\mathbf{h}_2$	$\mathbf{t}_2$
$\mathbf{h}_3$	ehat$thecatin	$\mathbf{t}_3$	$\mathbf{h}_3$	$\mathbf{t}_3$
$\mathbf{i}_1$	nthehat$theca	$\mathbf{t}_4$	$\mathbf{i}_1$	$\mathbf{t}_4$
$\mathbf{n}_1$	thehat$thecat	$\mathbf{i}_1$	$\mathbf{n}_1$	$\mathbf{i}_1$
$\mathbf{t}_1$	$thecatintheh	$\mathbf{a}_1$	$\mathbf{t}_1$	$\mathbf{a}_1$
$\mathbf{t}_2$	hecatinthehat	$\mathbf{\$}_2$	$\mathbf{t}_2$	**$**
$\mathbf{t}_3$	hehat$thecati	$\mathbf{n}_1$	$\mathbf{t}_3$	$\mathbf{n}_1$
$\mathbf{t}_4$	inthehat$thec	$\mathbf{a}_2$	$\mathbf{t}_4$	$\mathbf{a}_2$

Does the substring the occur in the text?

We’ll look for it, again starting with its last letter.

F		L
$\$$	thecatintheha	$\mathbf{t}_1$
$\mathbf{a}_1$	t$thecatinthe	$\mathbf{h}_1$
$\mathbf{a}_2$	tinthehat$the	$\mathbf{c}_1$
$\mathbf{c}_1$	atinthehat$th	$\mathbf{e}_1$
$\mathbf{e}_1$	catinthehat$t	$\mathbf{h}_2$
$\mathbf{e}_2$	hat$thecatint	$\mathbf{h}_3$
$\mathbf{h}_1$	at$thecatinth	$\mathbf{e}_2$
$\mathbf{h}_2$	ecatinthehat$	$\mathbf{t}_2$
$\mathbf{h}_3$	ehat$thecatin	$\mathbf{t}_3$
$\mathbf{i}_1$	nthehat$theca	$\mathbf{t}_4$
$\mathbf{n}_1$	thehat$thecat	$\mathbf{i}_1$
$\mathbf{t}_1$	$thecatintheh	$\mathbf{a}_1$
$\mathbf{t}_2$	hecatinthehat	$\$_2$
$\mathbf{t}_3$	hehat$thecati	$\mathbf{n}_1$
$\mathbf{t}_4$	inthehat$thec	$\mathbf{a}_2$

F	L
$\$$	$\mathbf{t}_1$
$\mathbf{a}_1$	$\mathbf{h}_1$
$\mathbf{a}_2$	$\mathbf{c}_1$
$\mathbf{c}_1$	$\mathbf{e}_1$
$\mathbf{e}_1$	$\mathbf{h}_2$
$\mathbf{e}_2$	$\mathbf{h}_3$
$\mathbf{h}_1$	$\mathbf{e}_2$
$\mathbf{h}_2$	$\mathbf{t}_2$
$\mathbf{h}_3$	$\mathbf{t}_3$
$\mathbf{i}_1$	$\mathbf{t}_4$
$\mathbf{n}_1$	$\mathbf{i}_1$
$\mathbf{t}_1$	$\mathbf{a}_1$
$\mathbf{t}_2$	$\$$
$\mathbf{t}_3$	$\mathbf{n}_1$
$\mathbf{t}_4$	$\mathbf{a}_2$

Which row starts with e, the last letter of our search pattern?

Which of these e’s is preceded by an h?

Does the substring the occur in the text?

We’ll look for it, again starting with its last letter.

F		L	F		L
$\$$	thecatintheha	t_1	$\$$		t_1
a_1	t$thecatinthe	h_1	a_1		h_1
a_2	tinthehat$the	c_1	a_2		c_1
c_1	atinthehat$th	e_1	c_1		e_1
e_1	catinthehat$t	h_2	e_1	→	h_2
e_2	hat$thecatint	h_3	e_2	→	h_3
h_1	at$thecatinth	e_2	h_1		e_2
h_2	ecatinthehat$	t_2	h_2		t_2
h_3	ehat$thecatin	t_3	h_3		t_3
i_1	nthehat$theca	t_4	i_1		t_4
n_1	thehat$thecat	i_1	n_1		i_1
t_1	$thecatintheh	a_1	t_1		a_1
t_2	hecatinthehat	$\$_2$	t_2		$\$$
t_3	hehat$thecati	n_1	t_3		n_1
t_4	inthehat$thec	a_2	t_4		a_2

Which row starts with e, the last letter of our search pattern?

Which of these e’s is preceded by an h?

Locate h_2 and h_3 in F

Does the substring the occur in the text?

We'll look for it, again starting with its last letter.

F		L
`$`	`thecatintheha`	t_1
a_1	`t$thecatinthe`	h_1
a_2	`tinthehat$the`	c_1
c_1	`atinthehat$th`	e_1
e_1	`catinthehat$t`	h_2
e_2	`hat$thecatint`	h_3
h_1	`at$thecatinth`	e_2
h_2	`ecatinthehat$`	t_2
h_3	`ehat$thecatin`	t_3
i_1	`nthehat$theca`	t_4
n_1	`thehat$thecat`	i_1
t_1	`$thecatintheh`	a_1
t_2	`hecatinthehat`	$\$_2$
t_3	`hehat$thecati`	n_1
t_4	`inthehat$thec`	a_2

F		L
`$`		t_1
a_1		h_1
a_2		c_1
c_1		e_1
e_1	→	h_2
e_2	→	h_3
h_1		e_2
h_2		t_2
h_3		t_3
i_1		t_4
n_1		i_1
t_1		a_1
t_2		`$`
t_3		n_1
t_4		a_2

Which row starts with e, the last letter of our search pattern?

Which of these e's is preceded by an h?

Locate h_2 and h_3 in F

Does the substring the occur in the text?

We’ll look for it, again starting with its last letter.

F		L
`$`	`thecatintheha`	t_1
a_1	`t$thecatinthe`	h_1
a_2	`tinthehat$the`	c_1
c_1	`atinthehat$th`	e_1
e_1	`catinthehat$t`	h_2
e_2	`hat$thecatint`	h_3
h_1	`at$thecatinth`	e_2
h_2	`ecatinthehat$`	t_2
h_3	`ehat$thecatin`	t_3
i_1	`nthehat$theca`	t_4
n_1	`thehat$thecat`	i_1
t_1	`$thecatintheh`	a_1
t_2	`hecatinthehat`	$\$_2$
t_3	`hehat$thecati`	n_1
t_4	`inthehat$thec`	a_2

F		L
`$`		t_1
a_1		h_1
a_2		c_1
c_1		e_1
e_1	→	h_2
e_2	→	h_3
h_1		e_2
h_2		t_2
h_3		t_3
i_1		t_4
n_1		i_1
t_1		a_1
t_2		`$`
t_3		n_1
t_4		a_2

Which row starts with e, the last letter of our search pattern?

Which of these e’s is preceded by an h?

Locate h_2 and h_3 in F

Which of these h’s is preceded by a t?

Does the substring the occur in the text?

We'll look for it, again starting with its last letter.

F		L
$	thecatintheha	t_1
a_1	t$thecatinthe	h_1
a_2	tinthehat$the	c_1
c_1	atinthehat$th	e_1
e_1	catinthehat$t	h_2
e_2	hat$thecatint	h_3
h_1	at$thecatinth	e_2
h_2	ecatinthehat$	t_2
h_3	ehat$thecatin	t_3
i_1	nthehat$theca	t_4
n_1	thehat$thecat	i_1
t_1	$thecatintheh	a_1
t_2	hecatinthehat	$\$_2$
t_3	hehat$thecati	n_1
t_4	inthehat$thec	a_2

F		L
$		t_1
a_1		h_1
a_2		c_1
c_1		e_1
e_1	→	h_2
e_2	→	h_3
h_1		e_2
h_2	→	t_2
h_3	→	t_3
i_1		t_4
n_1		i_1
t_1		a_1
t_2		$
t_3		n_1
t_4		a_2

Which row starts with e, the last letter of our search pattern?

Which of these e's is preceded by an h?

Locate h_2 and h_3 in F

Which of these h's is preceded by a t?

Locate t_2 and t_3 in F

Does the substring the occur in the text?

We'll look for it, again starting with its last letter.

F		L	F		L
\$	thecatinthehat	t_1	\$		t_1
a_1	t\$thecatinthe	h_1	a_1		h_1
a_2	tinthehat\$the	c_1	a_2		c_1
c_1	atinthehat\$th	e_1	c_1		e_1
e_1	catinthehat\$t	h_2	e_1	→	h_2
e_2	hat\$thecatint	h_3	e_2	→	h_3
h_1	at\$thecatinth	e_2	h_1		e_2
h_2	ecatinthehat\$	t_2	h_2	→	t_2
h_3	ehat\$thecatin	t_3	h_3	→	t_3
i_1	nthehat\$theca	t_4	i_1		t_4
n_1	thehat\$thecat	i_1	n_1		i_1
t_1	\$theccatintheh	a_1	t_1		a_1
t_2	hecatinthehat	$\$_2$	t_2		\$
t_3	hehat\$thecati	n_1	t_3		n_1
t_4	inthehat\$thec	a_2	t_4		a_2

Which row starts with e, the last letter of our search pattern?

Which of these e's is preceded by an h?

Locate h_2 and h_3 in F

Which of these h's is preceded by a t?

Locate t_2 and t_3 in F

The substring the occurs in two locations in the text