COMP90042 Lecture 19
Index compression and efficient query processing

Matthias Petri

Wed 10/5/17

What we'll learn today

- Efficient storage and access of **Posting lists**
- Efficient inverted index query processing

Index compression

Benefits of index compression:

- Reduce storage requirements
- Keep larger parts of the index in memory
- Faster query processing

Example: A state-of-the-art inverted index of 25 million websites (426GB raw data) requires only 5GB (1.2%) and can answer queries in ≈ 10 milliseconds.

Practice: Companies such as Facebook keep their index in main memory. Efficient, compact representations can save lots of money!

Compression Principles

- Compressibility is bounded by the information content of a data set
- Information content of a text T is a characterized by its *Entropy* H:

$$H(T) = -\sum_{s \in \Sigma} \frac{f_s}{n} \log_2 \frac{f_s}{n}$$

  where f_s is the frequency of symbol s in T and n is the length of T
- For example, $H(abracadabra) = 2.040373$ bits with $n = 11, f_a = 5, f_b = 2, f_c = 1, f_d = 1, f_r = 2$
- Intuition: Spend less bits on items that occur often

Posting list Compression - Terminology

Inverted index terminology : Terms, Documents, Postings Lists, Postings

Example

`the`	25	26	29	...	12345	12347	12349
`house`	5123	5234	5454	5591	...		
`aeronaut`	251235	251239	251273				

Posting list Compression - Requirements

- Minimize storage costs
- Fast access

Sample collection

Documents N	$24{,}622{,}347$
Terms	$35{,}636{,}425$
Postings	$5{,}742{,}630{,}292$
Uncompressed Storage Cost[1]	16.71 GB

[1] Only document identifier, no frequencies or positions

Posting list Compression - Concepts

- Postings list corresponds to an increasing sequence of integers
- Each integer can be in $[1, N]$ requiring $\lceil \log_2(N) \rceil$ bits.
  Example: $N = 24{,}622{,}347$. $\log_2(N) = 24.55346 = 25$ bits
- Idea: Gaps between two adjacent integers can be much smaller

`the`	ids:	25	26	29	...	12345	12347	12349
	gaps:	25	1	3	...	1	2	2
`house`	ids:	5123	5234	5454	5591	...		
	gaps:	5123	1	220	137	...		
`aeronaut`	ids:	251235	251239	251273				
	gaps:	251235	4	34				

Variable Byte Compression

Idea

Use variable number of bytes to represent integers. Each byte contains 7 bits "payload" and one continuation bit.

Examples

Number	Bits	Encoding	
824	1100111000	00111000	10000110
5	101	10000101	

Variable Byte Compression - Storage Cost

Storage Cost

Range Start	Range Stop	Power of 2 Range	Number of Bytes
0	127	$0 - 2^7 - 1$	1
128	$16,383$	$2^7 - 2^{14} - 1$	2
$16,384$	$2,097,151$	$2^{14} - 2^{21} - 1$	3
$2,097,152$	$268,435,455$	$2^{21} - 2^{28} - 1$	4
$268,435,456$	$34,359,738,368$	$2^{28} - 2^{35} - 1$	5
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Variable Byte Compression - Example

Compress number 512312 or 1111101000100111000 in binary.

- How many bytes?

Variable Byte Compression - Example

Compress number 512312 or 1111101000100111000 in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!

Variable Byte Compression - Example

Compress number 512312 or 1111101000100111000 in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?

Variable Byte Compression - Example

Compress number 512312 or 1111101000100111000 in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?
  1. Extract the lowest 7 bits.

Variable Byte Compression - Example

Compress number 512312 or 11111010001001110000 in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?
  1. Extract the lowest 7 bits.
     $512312 \mod 128 = 56 = 0111000$

Variable Byte Compression - Example

Compress number 512312 or $11111010001001110 00$ in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?
  1. Extract the lowest 7 bits.
     $512312 \mod 128 = 56 = 0111000$
  2. Discard lowest 7 bits.

Variable Byte Compression - Example

Compress number 512312 or $11111010001001110 00$ in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?

1. Extract the lowest 7 bits.
   $512312 \mod 128 = 56 = 0111000$
2. Discard lowest 7 bits.
   $512312 \div 128 = 4002$ (or $512312 >> 7$)

Variable Byte Compression - Example

Compress number 512312 or 1111101000100111000 in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?

1. Extract the lowest 7 bits.
   $512312 \mod 128 = 56 = 0111000$
2. Discard lowest 7 bits.
   $512312 \div 128 = 4002$ (or $512312 >> 7$)
3. Extract the lowest 7 bits.

Variable Byte Compression - Example

Compress number 512312 or 11111010000100111000 in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?

1. Extract the lowest 7 bits.
   $512312 \mod 128 = 56 = 0111000$
2. Discard lowest 7 bits.
   $512312 \div 128 = 4002$ (or $512312 >> 7$)
3. Extract the lowest 7 bits.
   $4002 \mod 128 = 34 = 0100010$

Variable Byte Compression - Example

Compress number 512312 or 11111010000100111000 in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?

1. Extract the lowest 7 bits.
   $512312 \mod 128 = 56 = 0111000$
2. Discard lowest 7 bits.
   $512312 \div 128 = 4002$ (or $512312 >> 7$)
3. Extract the lowest 7 bits.
   $4002 \mod 128 = 34 = 0100010$
4. Discard lowest 7 bits.
   $4002 \div 128 = 31$ (or $4002 >> 7$)

Variable Byte Compression - Example

Compress number 512312 or 11111010000100111000 in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?

1. Extract the lowest 7 bits.
   $512312 \mod 128 = 56 = 0111000$
2. Discard lowest 7 bits.
   $512312 \div 128 = 4002$ (or $512312 >> 7$)
3. Extract the lowest 7 bits.
   $4002 \mod 128 = 34 = 0100010$
4. Discard lowest 7 bits.
   $4002 \div 128 = 31$ (or $4002 >> 7$)
5. Number smaller than 128. Write in lowest 7 bits and set top bit to 1. $31 = 11111$ So we write 10011111 which is $31 + 128 = 159$

Variable Byte Compression - Example

Compress number 512312 or 1111101000100111000 in binary.

- How many bytes? $11111|0100010|0111000$. 3 bytes!
- How do we compress the number?

1. Extract the lowest 7 bits.
   $512312 \mod 128 = 56 = 0111000$
2. Discard lowest 7 bits.
   $512312 \div 128 = 4002$ (or $512312 >> 7$)
3. Extract the lowest 7 bits.
   $4002 \mod 128 = 34 = 0100010$
4. Discard lowest 7 bits.
   $4002 \div 128 = 31$ (or $4002 >> 7$)
5. Number smaller than 128. Write in lowest 7 bits and set top bit to 1. $31 = 11111$ So we write 10011111 which is $31 + 128 = 159$

Variable Byte Compression - Algorithm

Encoding

```
1: function ENCODE(x)
2:     while x >= 128 do
3:         WRITE(x mod 128)
4:         x = x ÷ 128
5:     end while
6:     WRITE(x + 128)
7: end function
```

Decoding

```
1: function DECODE(bytes)
2:     x = 0
3:     y =READBYTE(bytes)
4:     while y < 128 do
5:         x = 128 × x + y
6:         y =READBYTE(bytes)
7:     end while
8:     x = 128 × x + (y − 128)
9:     return x
10: end function
```

OptPForDelta Compression

Idea

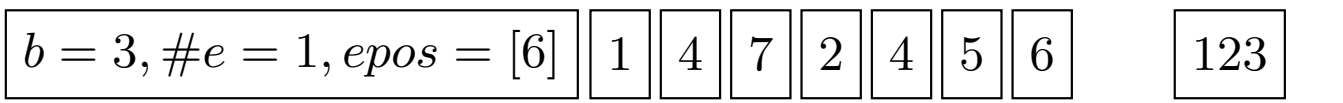
Group k gaps and encode using fixed number of bits. Encode numbers $> 2^b$ separately as an *exception. Pick* b *"optimally" for each block so there are* $\approx 10\%$ *exceptions.*

Example $k = 8$

`[1 4 7 2 4 5 123 6] [1 4 5 232 523 7 2 3] [3 4 755 15 12 1 8 4]`

b=3 b=8 b=4

Encode `[1 4 7 2 4 5 123 6]` as:



header content exceptions

16 bit 21 bit 8 bit

Postings List Decompression Speeds/Space Usage

Algorithm	Space [bits/posting]	Speed [Million Integers/sec]
Uncompressed	32	≈ 5400
Variable Byte	8.7	≈ 680
OptPForDelta	4.7	≈ 710
Simple-8b	4.8	≈ 780
SIMD-BP128	11	≈ 2300
⋮	⋮	

Postings List Compression - Optimizations

- Commonly lists are split into blocks of 128 integers
- Choose optimal compression for each block
- Often, long lists ("the") are represented more efficiently using bitvectors of size N
- State-of-the-art implementations use SIMD Instructions and bit-parallelism to increase decoding speed

Postings List Compression - Fast Searching

- Often decompressing the whole list is unnecessary
- Allow of efficient searching in the list is an important operation

Operation `GEQ(x)` - Find num in list that is Greater or Equal to x

Example:
List: [1,2,5,9,12,15]; $GEQ(6) = 9$; $GEQ(12) = 12$.

Postings List Compression - Fast Searching (GEQ)

- Compress postings list in blocks of 128 integers at a time
- For each block store an uncompressed sample value representing the largest (or smallest) value in the block
- Use sample values to efficiently seek to any position in the postings list without decompressing everything

Operation `GEQ(x)` algorithm:

- Binary search over uncompressed sample values to find destination block
- Decompress destination block to determine final offset in postings list

Postings List Compression - Summary

- Compress increasing integer sequences
- Support iterating and searching the compressed sequence
- Store gaps between adjacent numbers
- Different compression schemes provide many time-space tradeoffs

Efficient Query Processing (Top-k Retrieval)

IDEA:

Retrieve the top k items for a given query without having to evaluate all documents.

- Web search engines return only the top-10 results to the user
- For most queries most users generally do not retrieve more documents
- No need to score all possible documents to produce a "complete" ranking

Top-k Retrieval - Concepts

- Avoid scoring documents we know will not appear in the top-k result list
- For a given similarity metric (for example BM25), prestore some information for each term to avoid scoring
- Incorporate with block based compression schemes for efficient query processing
- Utilise the `GEQ`(x) operation to avoid decompression of large parts of postings lists

Top-k Retrieval - Review BM25

BM25 computation for one document:

$$S_{Q,d}^{\text{BM25}} = \sum_{t \in Q} \frac{(k_1 + 1) f_{d,t}}{k_1 \left(1 - b + b \frac{L_d}{L_{\text{avg}}}\right) + f_{d,t}} \cdot \ln \left(\frac{N - f_t + 0.5}{f_t + 0.5} \right)$$

Symbol	Description
Q	Multi set of query terms
t	Query term in Q
d	Evaluated document
$f_{d,t}$	Frequency of t in d (Term frequency)
N	Number of documents in the collection
f_t	Number of documents containing t (Document frequency)
L_d	Length of document d
L_{avg}	Average document length in the collection
k_1, b	Constants

Top-k Retrieval - Inefficient Evaluation of BM25

BM25 computation for one document:

$$S_{Q,d}^{\text{BM25}} = \sum_{t \in Q} \underbrace{\frac{(k_1+1) f_{d,t}}{k_1 \left(1 - b + b \frac{L_d}{L_{\text{avg}}}\right) + f_{d,t}}}_{=TF} \cdot \underbrace{\ln\left(\frac{N - f_t + 0.5}{f_t + 0.5}\right)}_{=IDF}$$

Inefficient Evaluation:

- For each t in Q compute IDF in $O(|Q|)$ time
- For each d in the document collection containing any t in Q evaluate TF (Potentially $O(N)$ time!)
- Return the top-k highest scoring documents

Top-k Retrieval - Evaluation with WAND

Basic Idea:

- Keep track of the top-k highest scoring documents
- For each unique term in the collection store the **maximum contribution** it can have to *any* document score in the collection
- Skip over documents that can not enter the top-k ranked result list

Citation: Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, Jason Y. Zien: Efficient query evaluation using a two-level retrieval process. CIKM: 426-434 (2003)

WAND - Maximum contribution of a term t

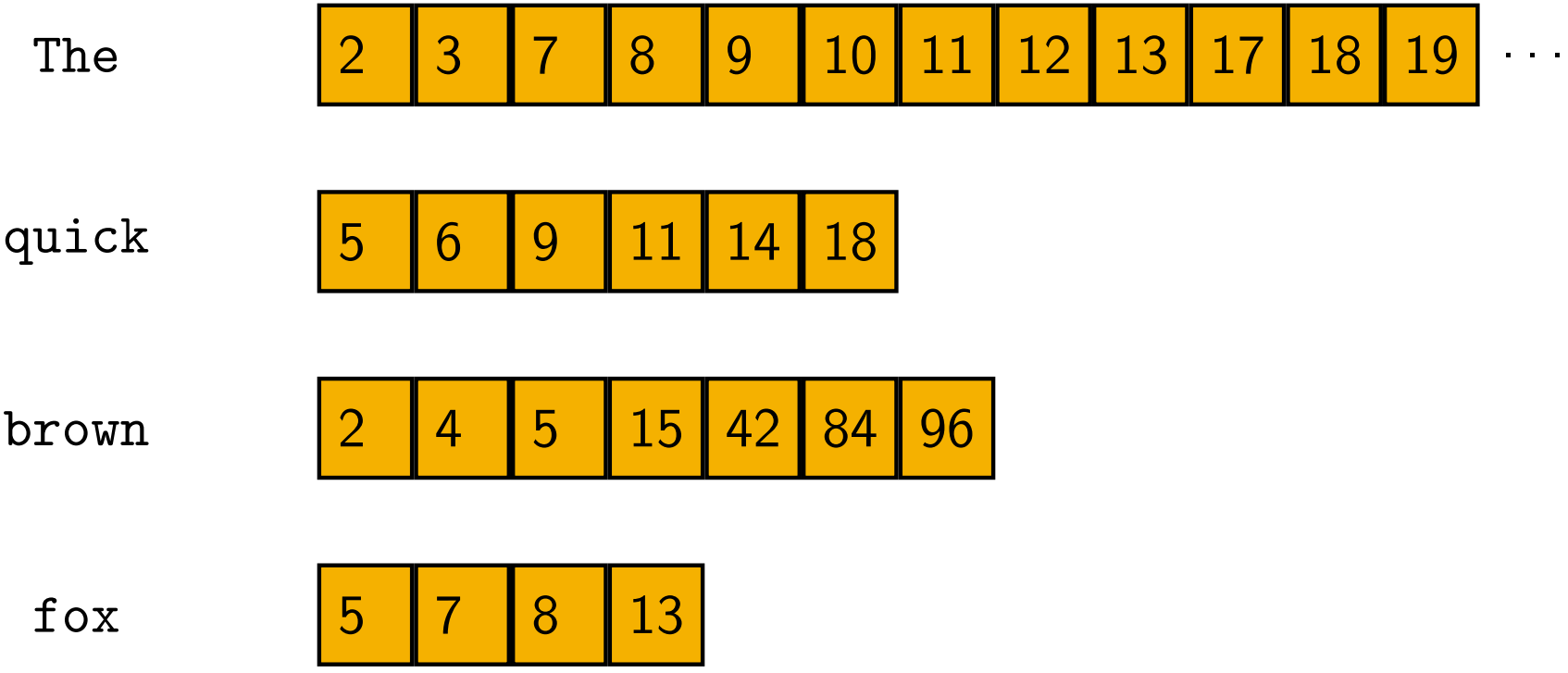
Maximum contribution

The **Maximum contribution** of a term t is the largest score **any** document in the collection can have for the query Q_t only consisting of t.

- Depends on the similarity measure
- Can be computed at construction time of the index
- Only requires storing a single floating point number for each list
- Can be used to **overestimate** the score of a document in a multi term query

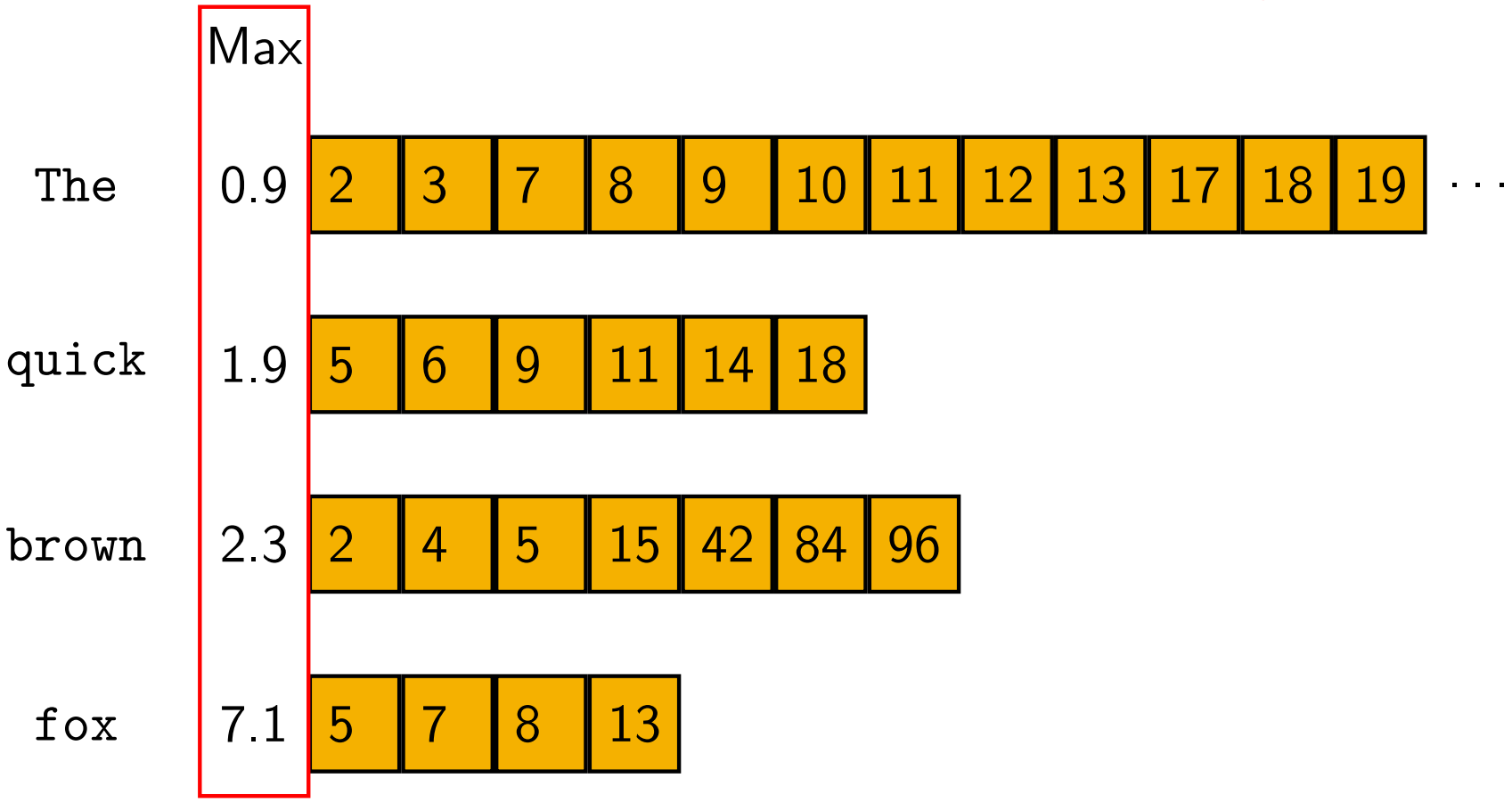
WAND - Example

Query Q : `The quick brown fox` with $k = 2$



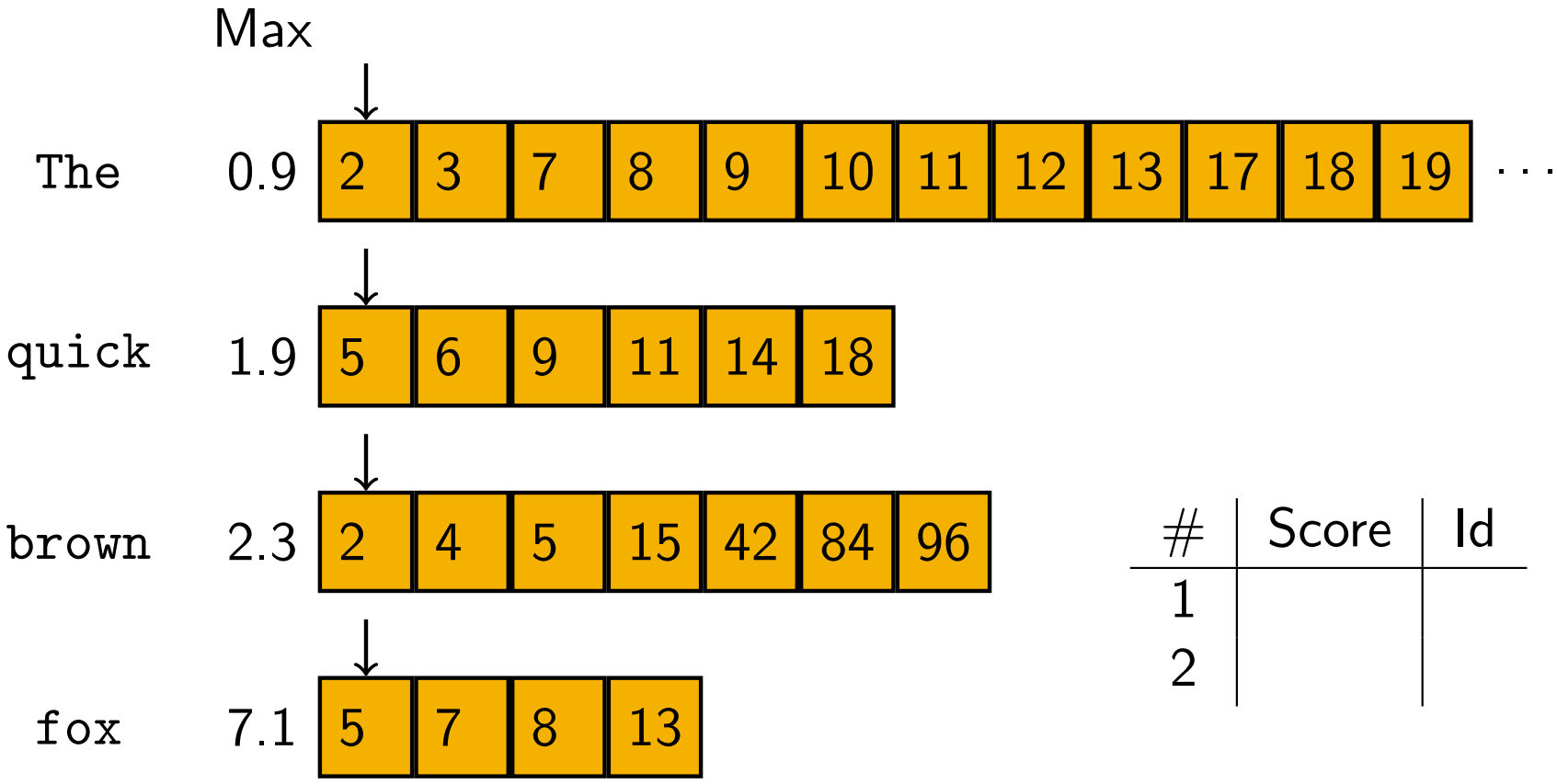
WAND - Example

Query Q : `The quick brown fox` with $k = 2$



WAND - Example

Query Q : `The quick brown fox` with $k = 2$

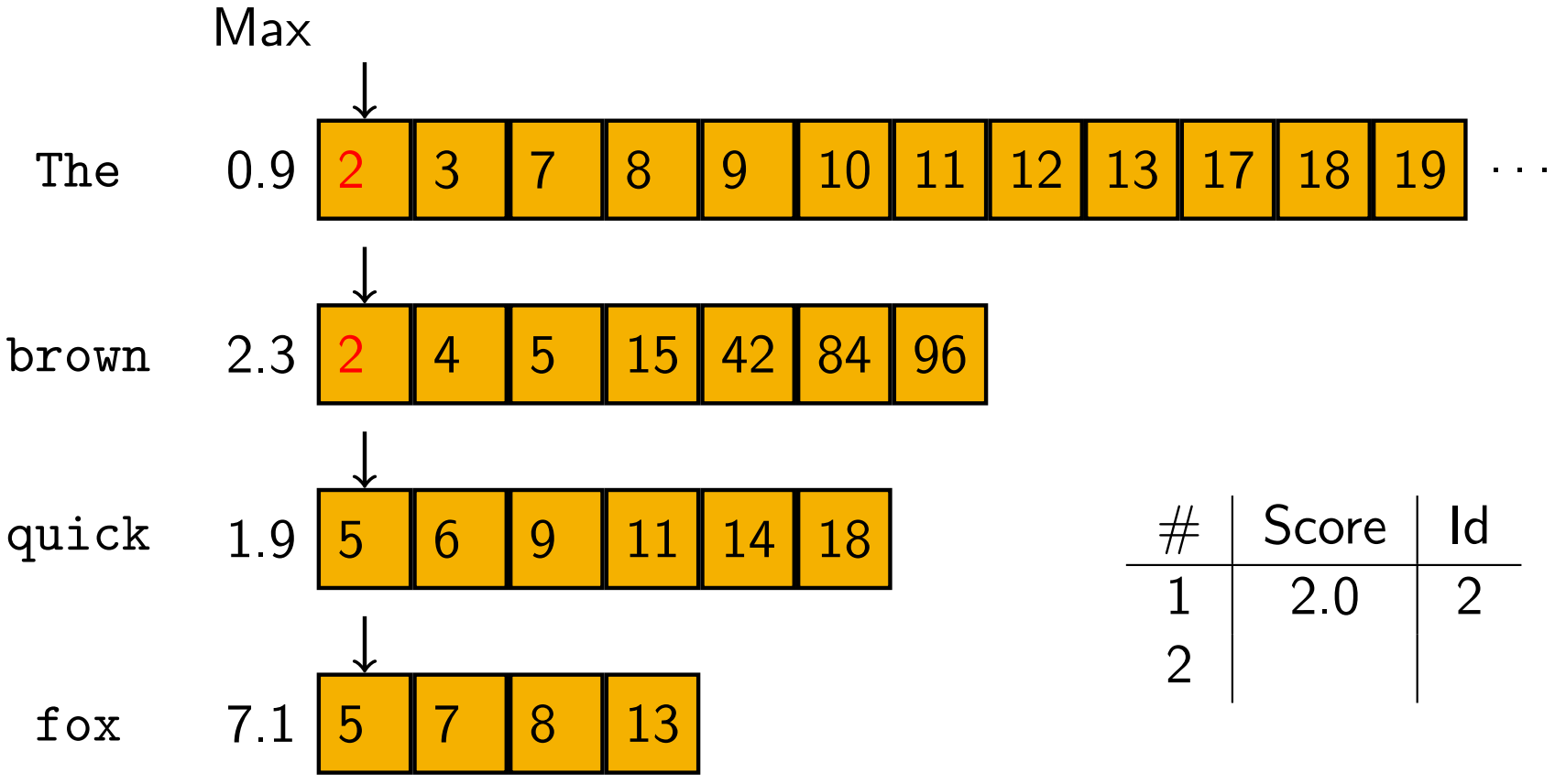


#	Score	Id
1		
2		

WAND - Example

Query Q : `The quick brown fox` with $k = 2$

(1) Reorder based on current id and score smallest.

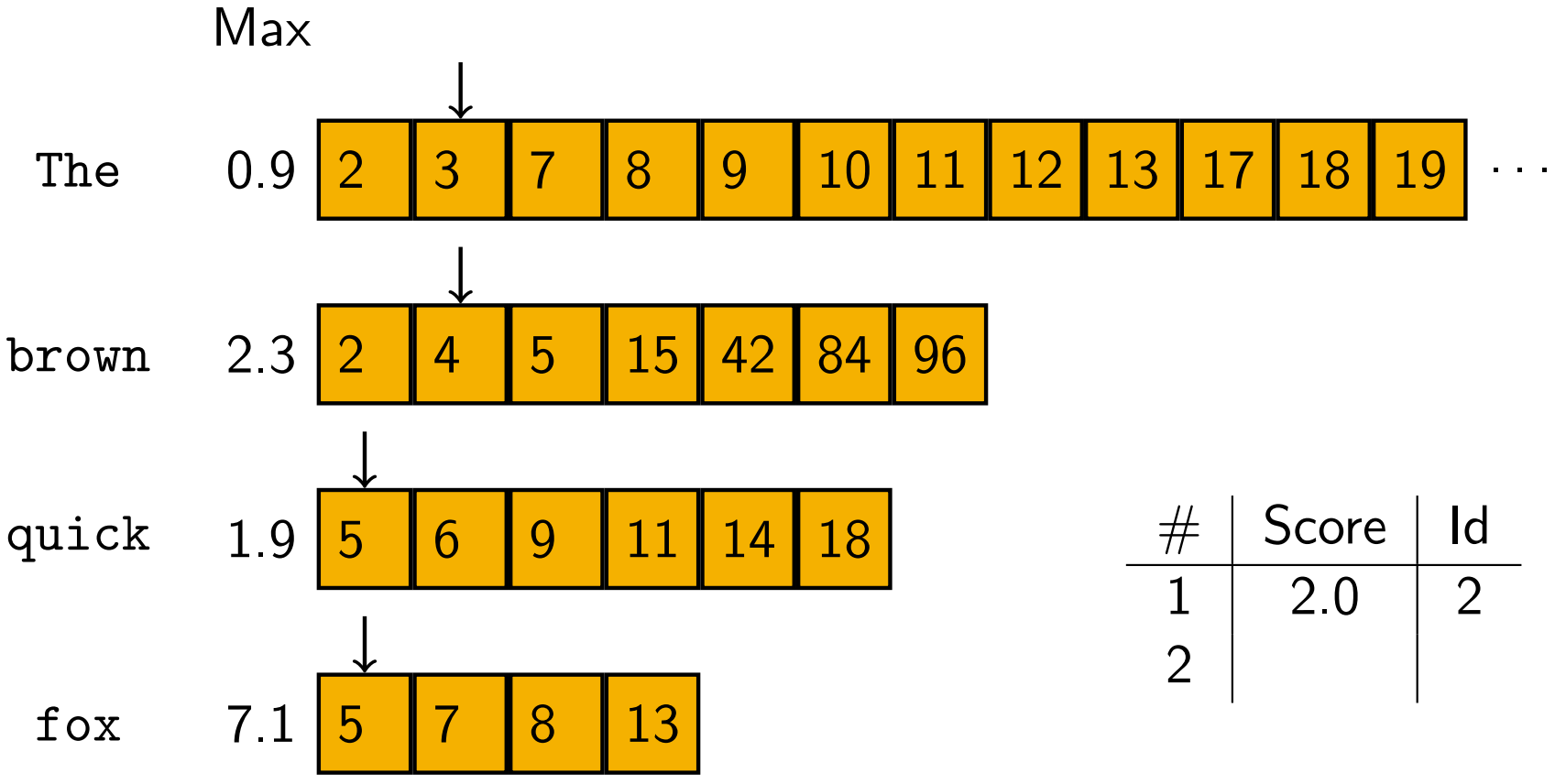


#	Score	Id
1	2.0	2
2		

WAND - Example

Query Q : `The quick brown fox` with $k = 2$

(2) Move pointer of scored elements.

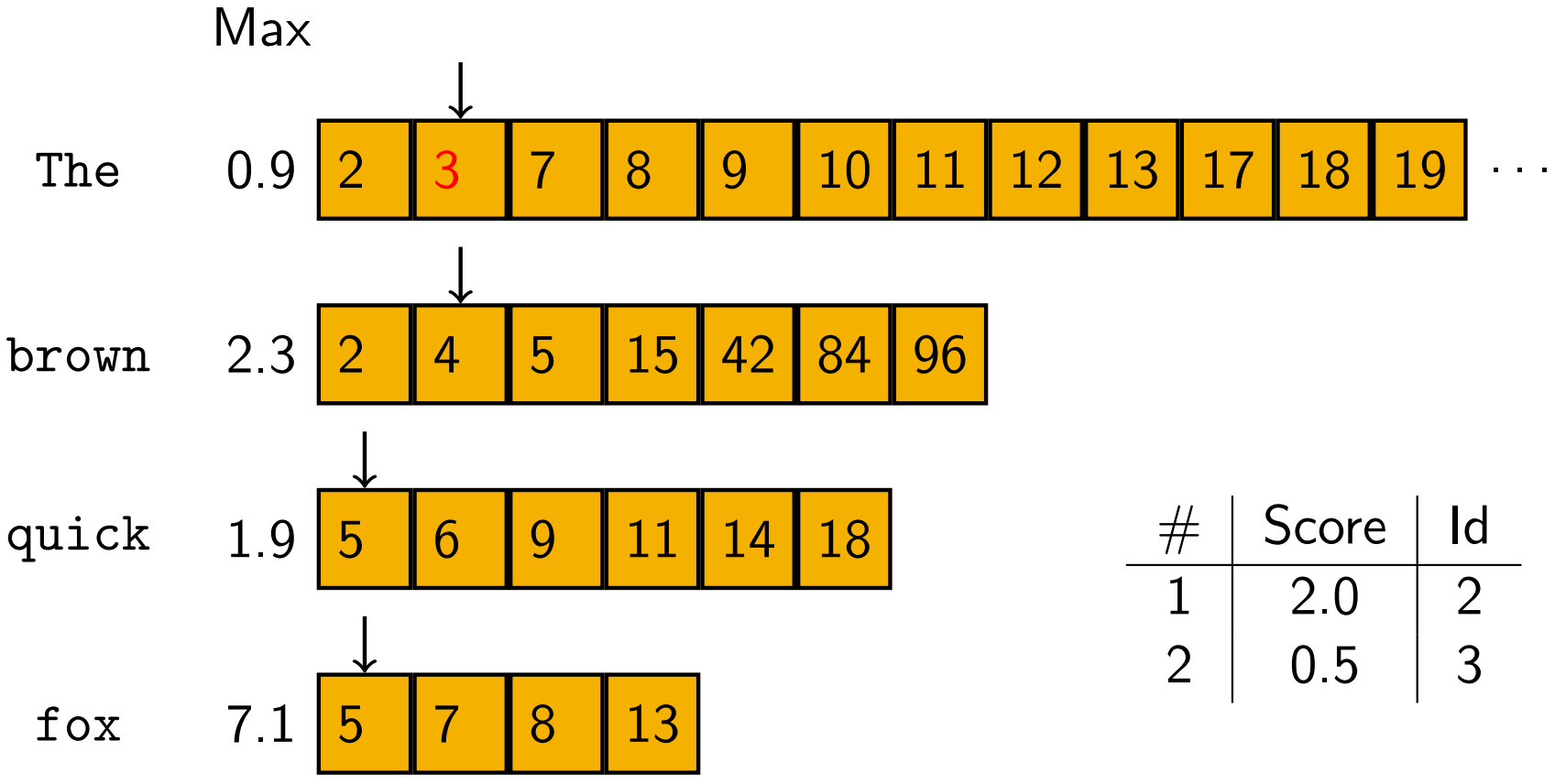


#	Score	Id
1	2.0	2
2		

WAND - Example

Query Q : `The quick brown fox` with $k = 2$

(3) Reorder based on current id and score smallest.

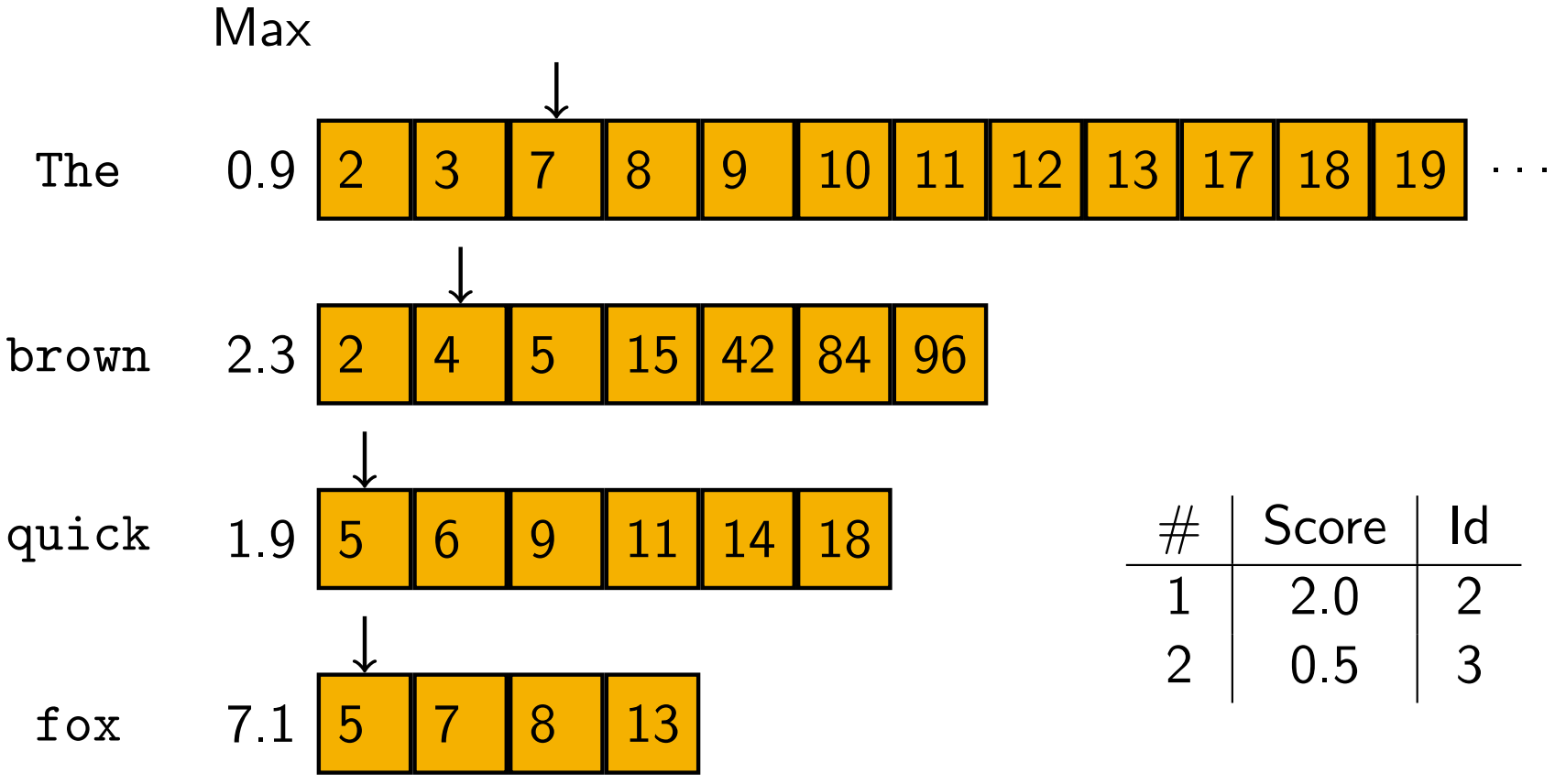


#	Score	Id
1	2.0	2
2	0.5	3

WAND - Example

Query Q : `The quick brown fox` with $k = 2$

(4) Move pointer of scored elements

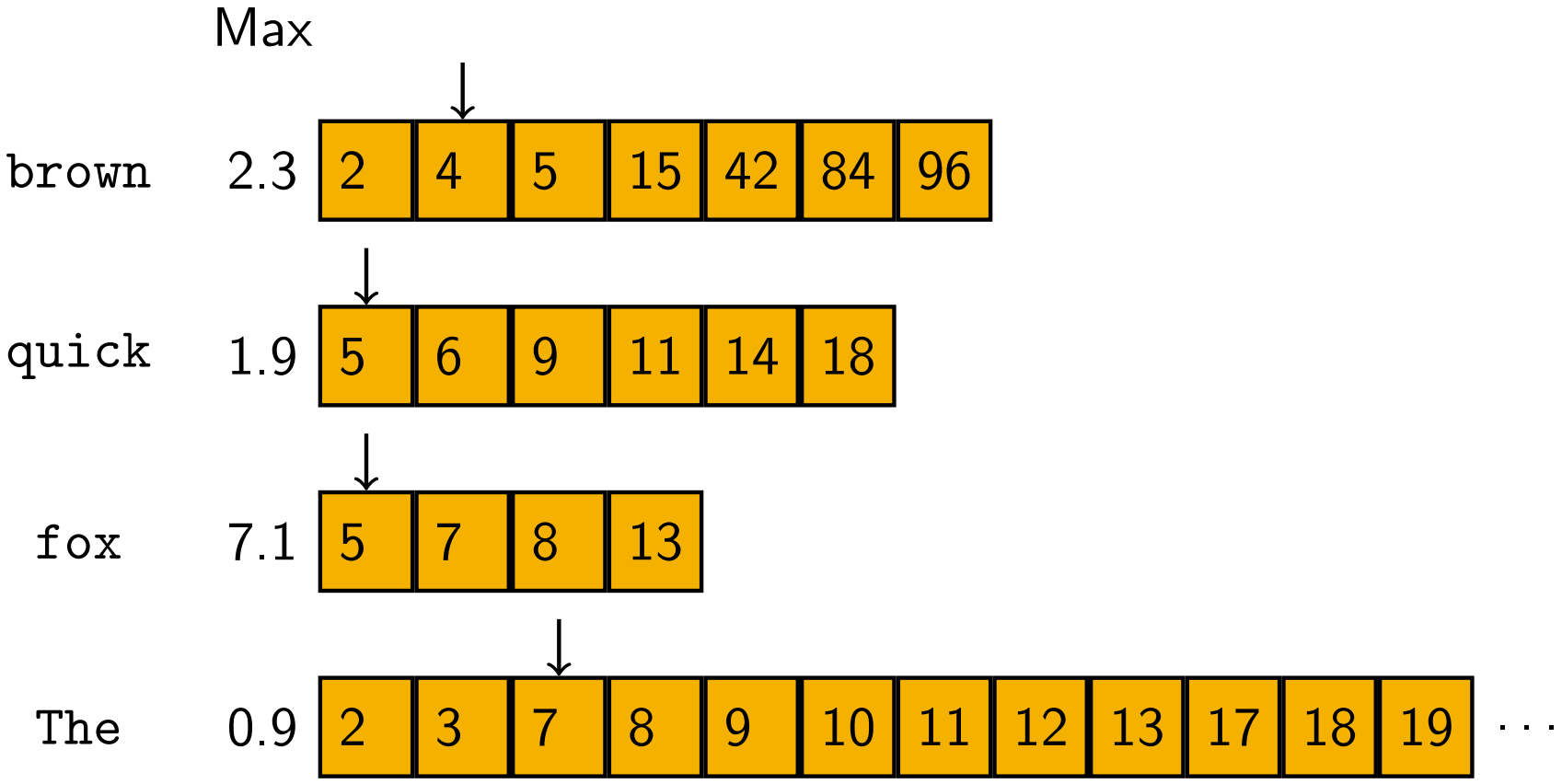


#	Score	Id
1	2.0	2
2	0.5	3

WAND - Example

Query Q : `The quick brown fox` with $k = 2$

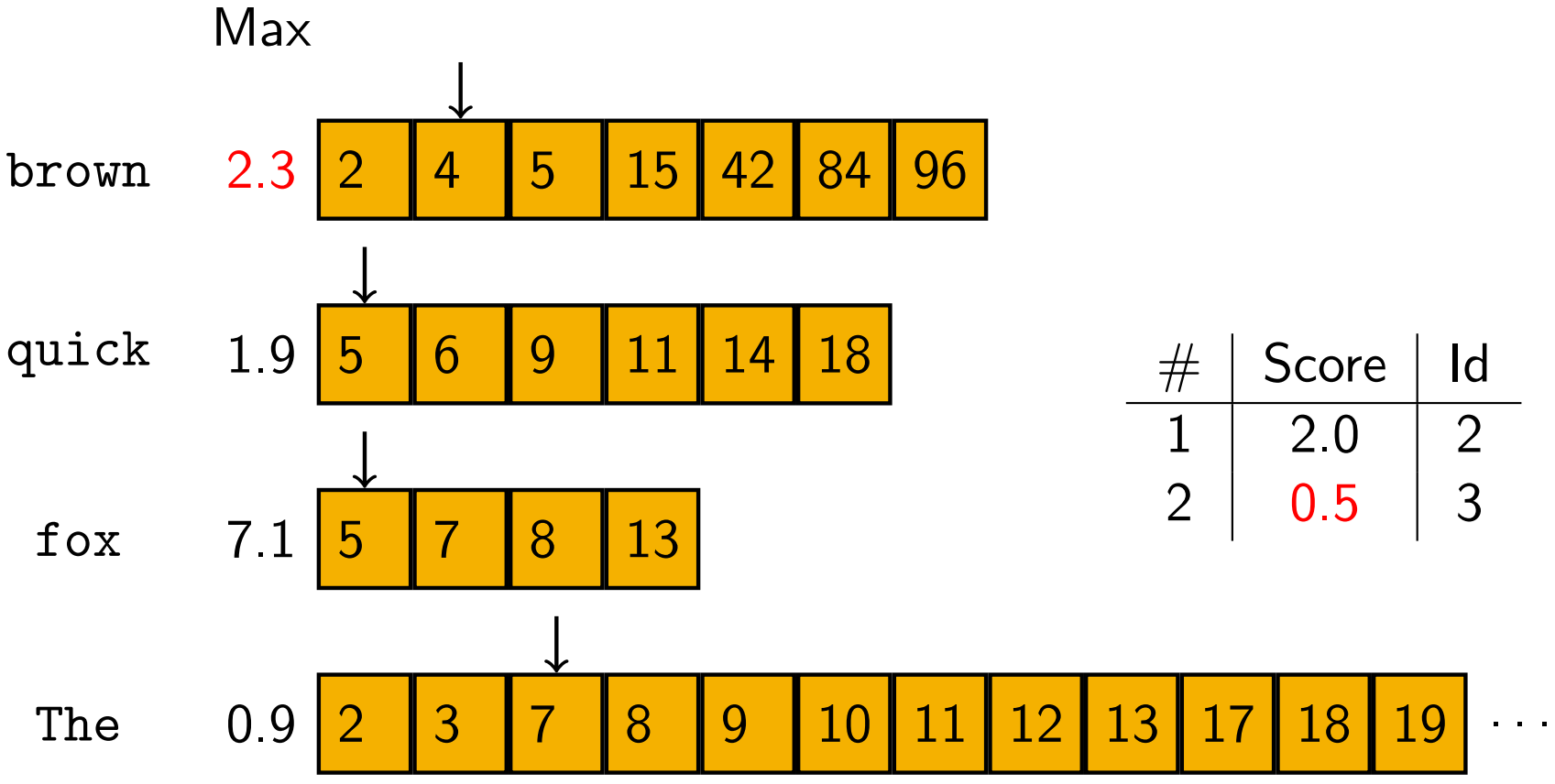
(5) Reorder based on current id and score smallest.



WAND - Example

Query Q : `The quick brown fox` with $k = 2$

(6) Decide if we need to score smallest id.

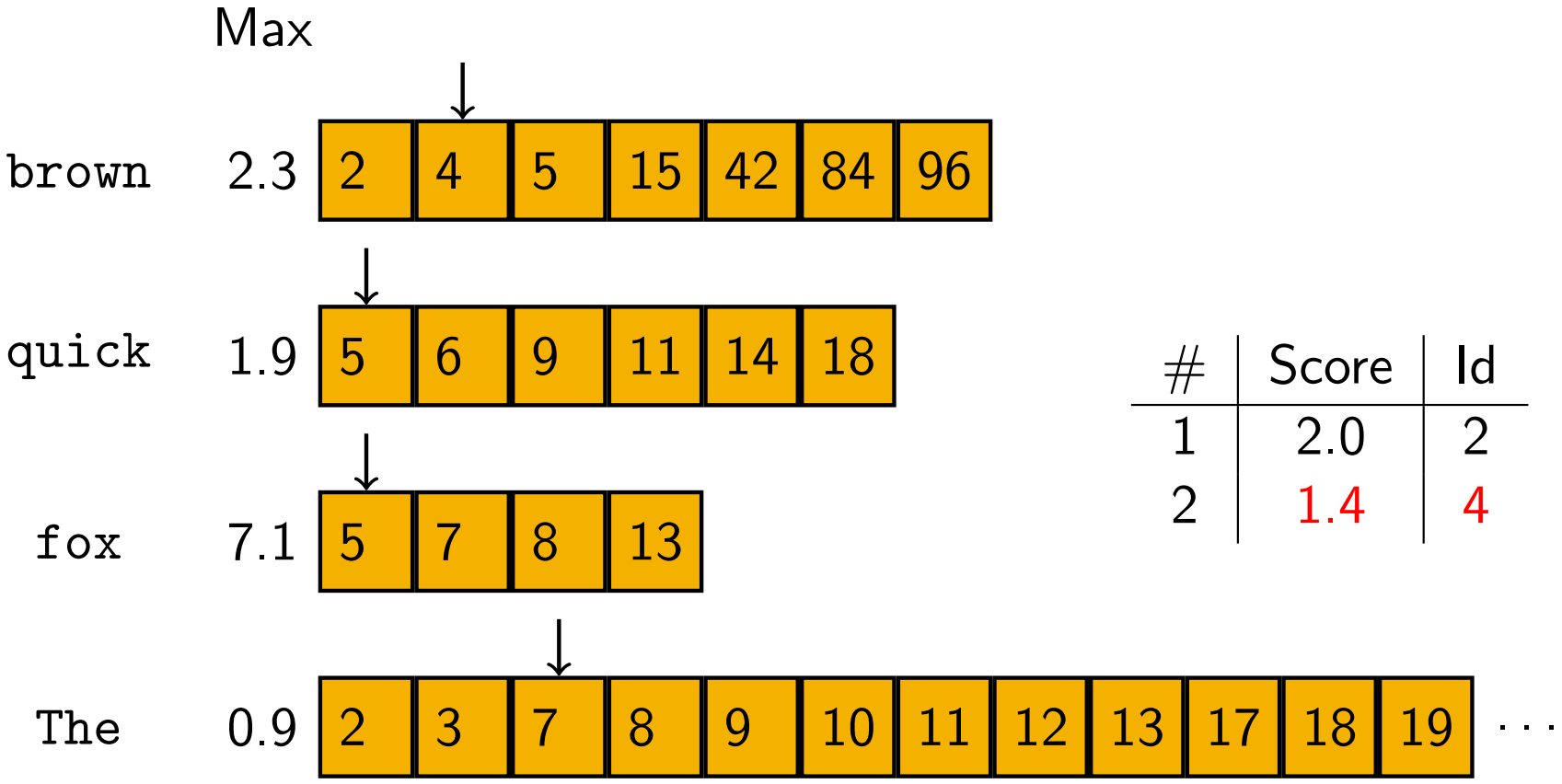


#	Score	Id
1	2.0	2
2	0.5	3

WAND - Example

Query Q : `The quick brown fox` with $k = 2$

(7) Replace 3 with 4 on the heap.

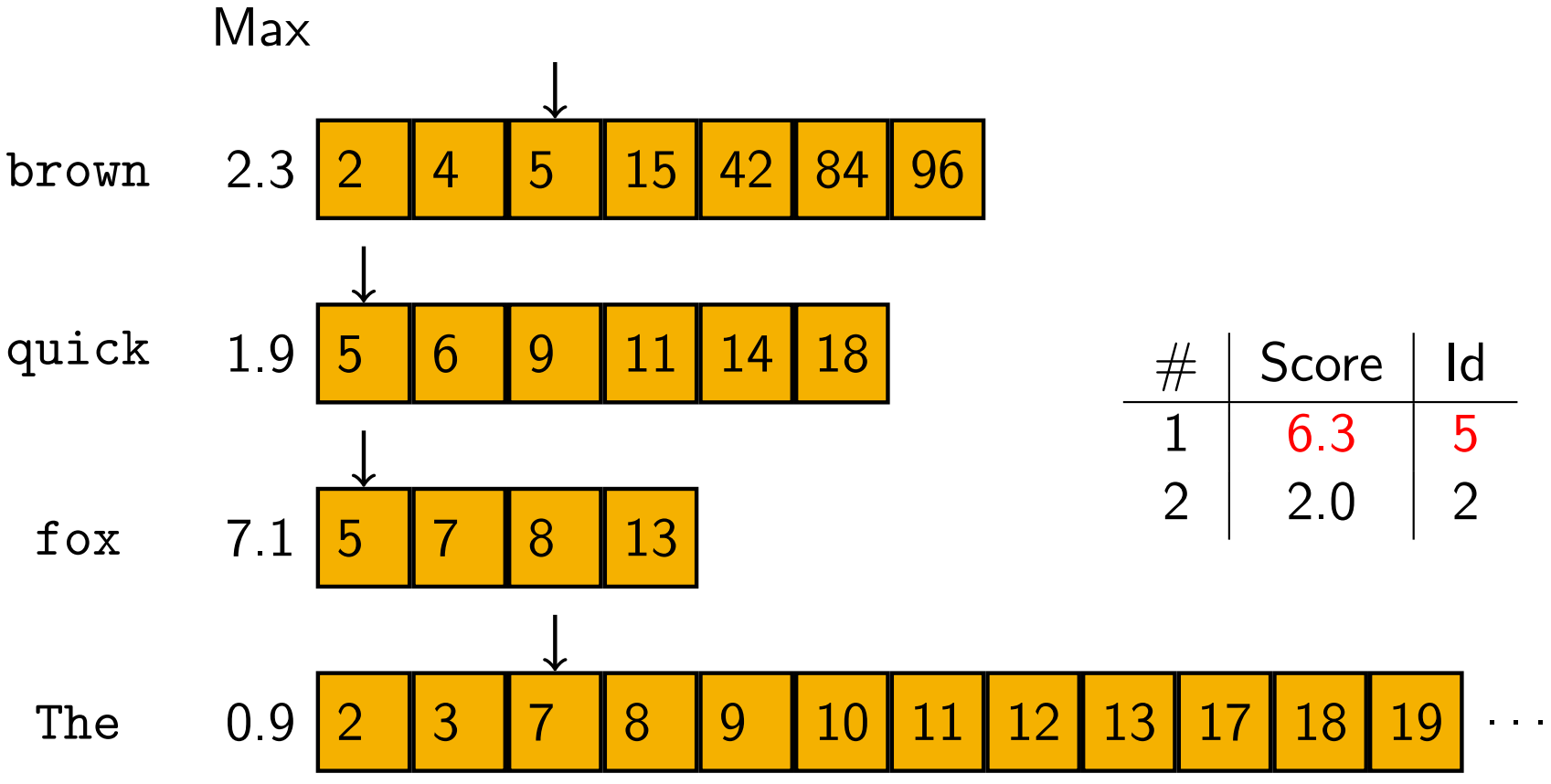


#	Score	Id
1	2.0	2
2	1.4	4

WAND - Example

Query Q : `The quick brown fox` with $k = 2$

(8) Sort by current id. Evaluate 5. Add to Heap.

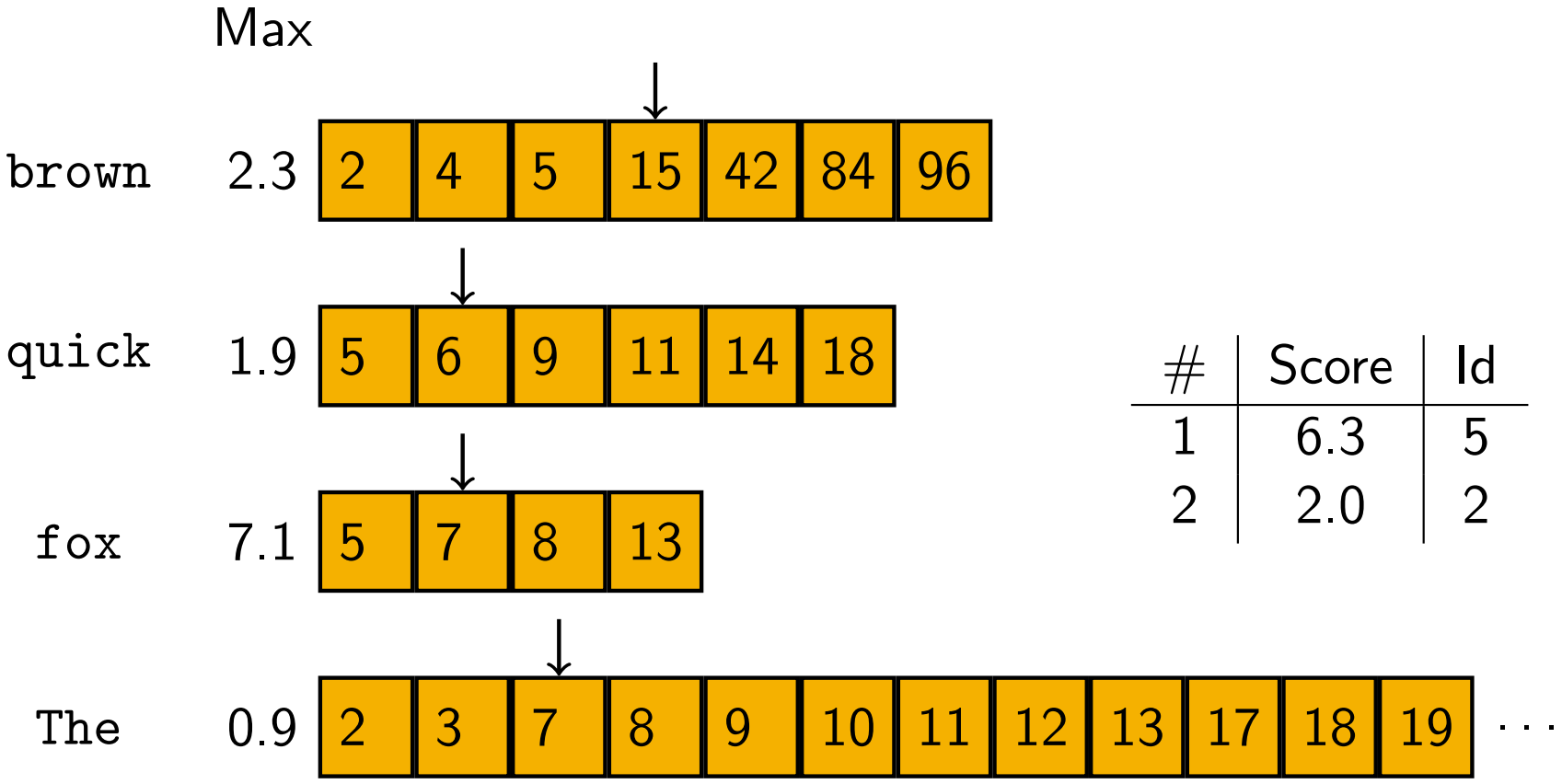


#	Score	Id
1	6.3	5
2	2.0	2

WAND - Example

Query Q : `The quick brown fox` with $k = 2$

(9) Move pointers and sort.

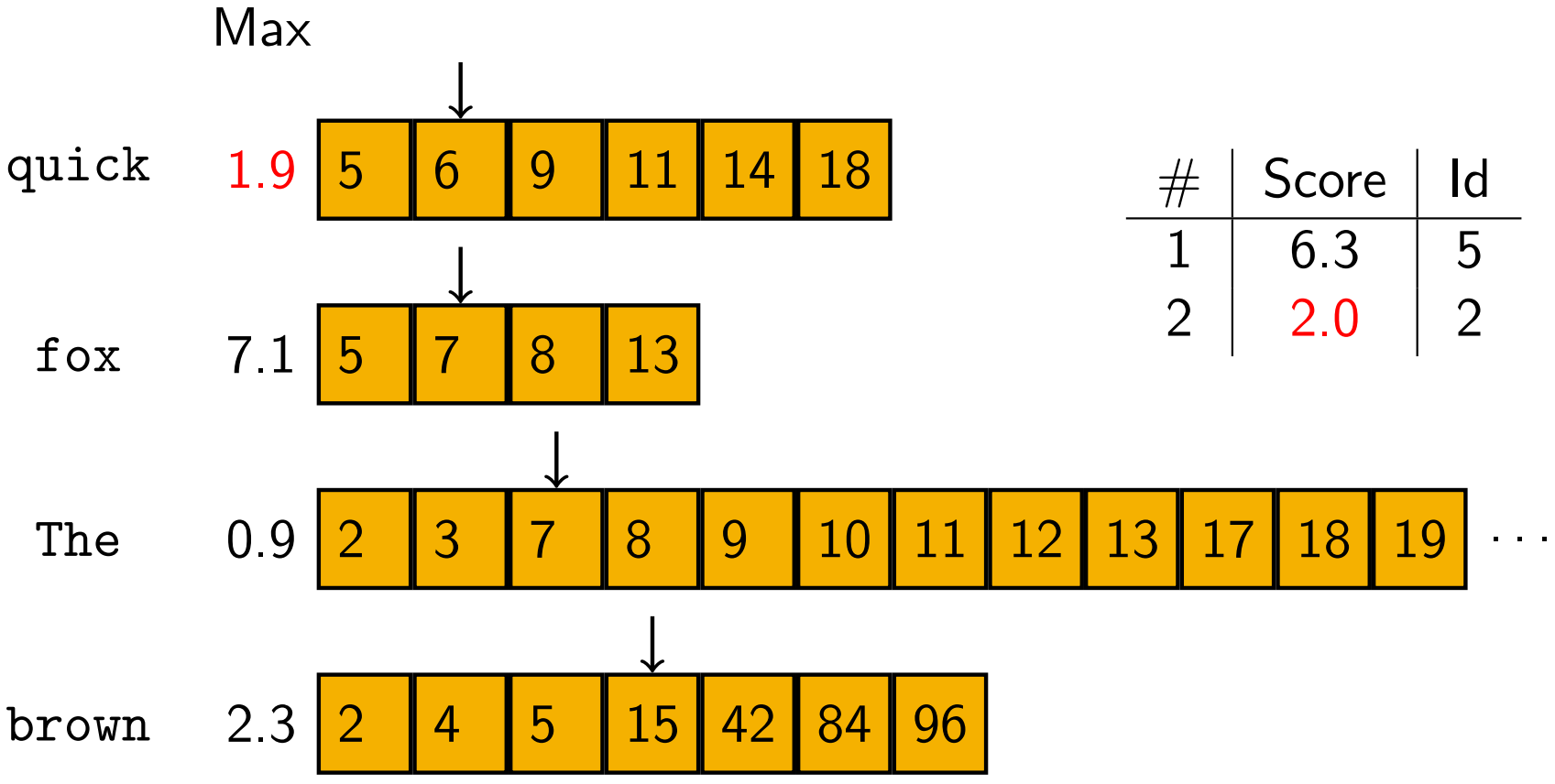


#	Score	Id
1	6.3	5
2	2.0	2

WAND - Example

Query Q : `The quick brown fox` with $k = 2$

(10) Use max to skip scoring smallest.

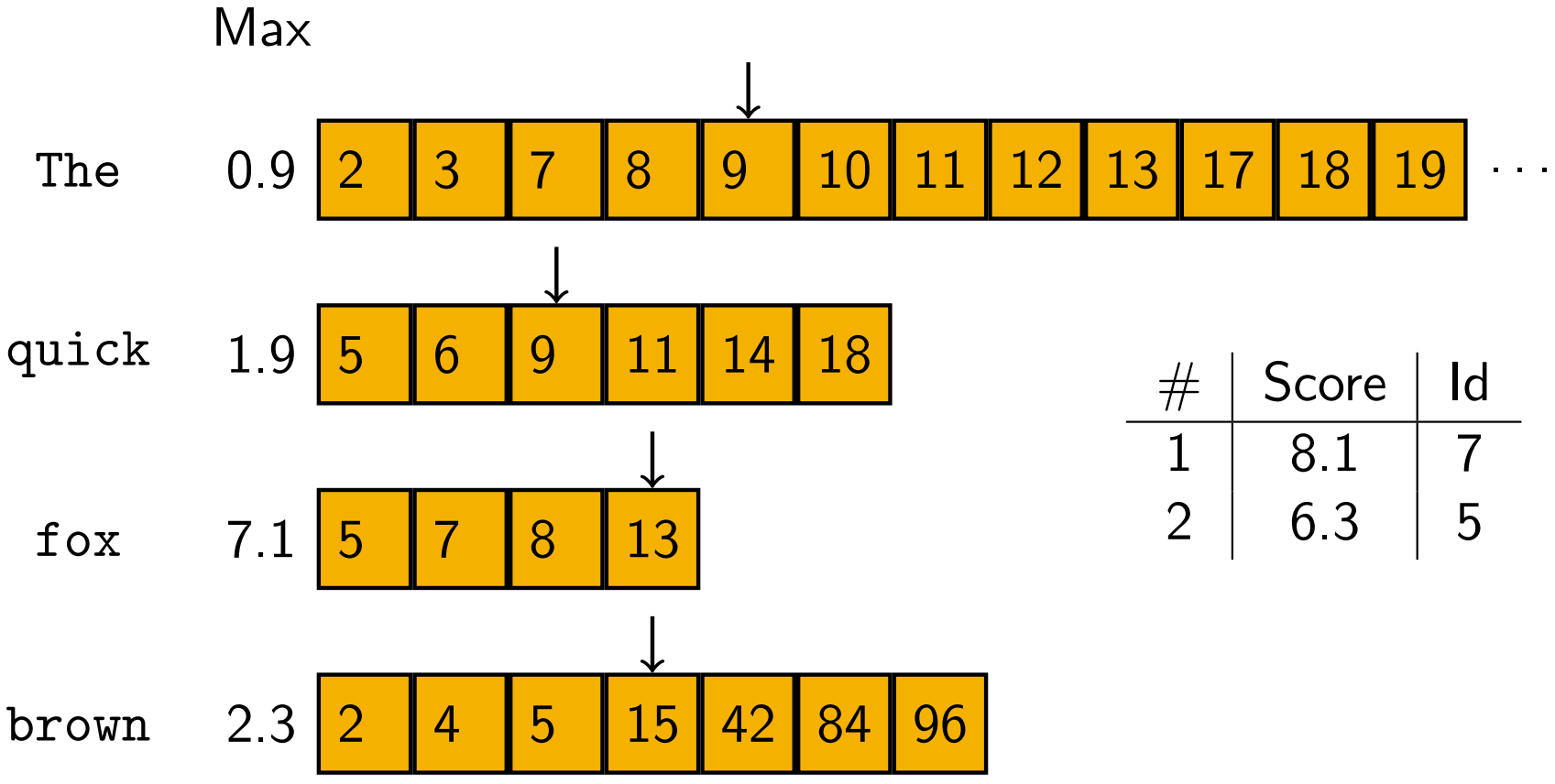


#	Score	Id
1	6.3	5
2	2.0	2

WAND - Example - Fast Forward

Query Q : `The quick brown fox` with $k = 2$

Do we have to evaluate document **9**?

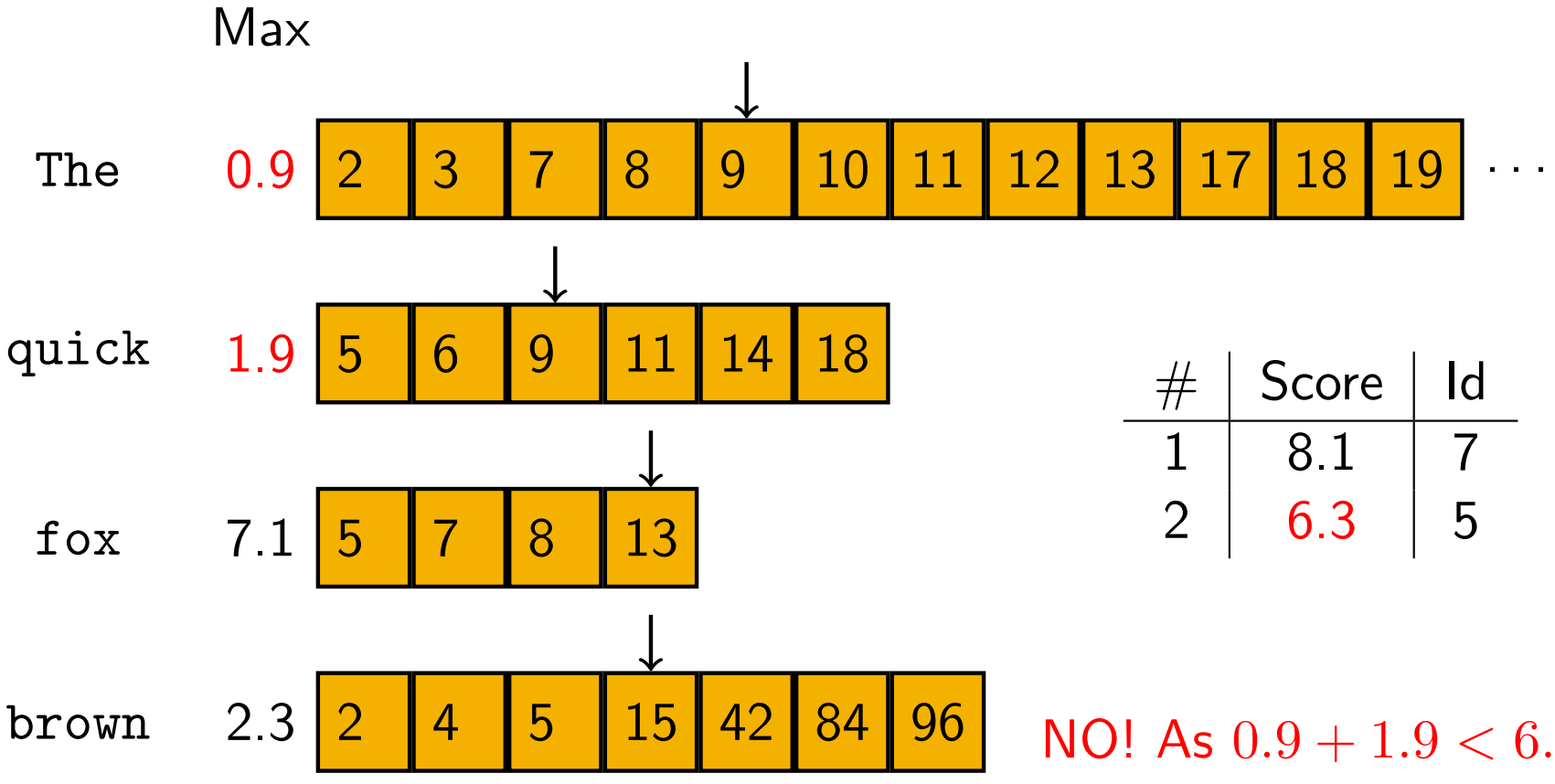


#	Score	Id
1	8.1	7
2	6.3	5

WAND - Example - Fast Forward

Query Q : `The quick brown fox` with $k = 2$

Do we have to evaluate document **9**?



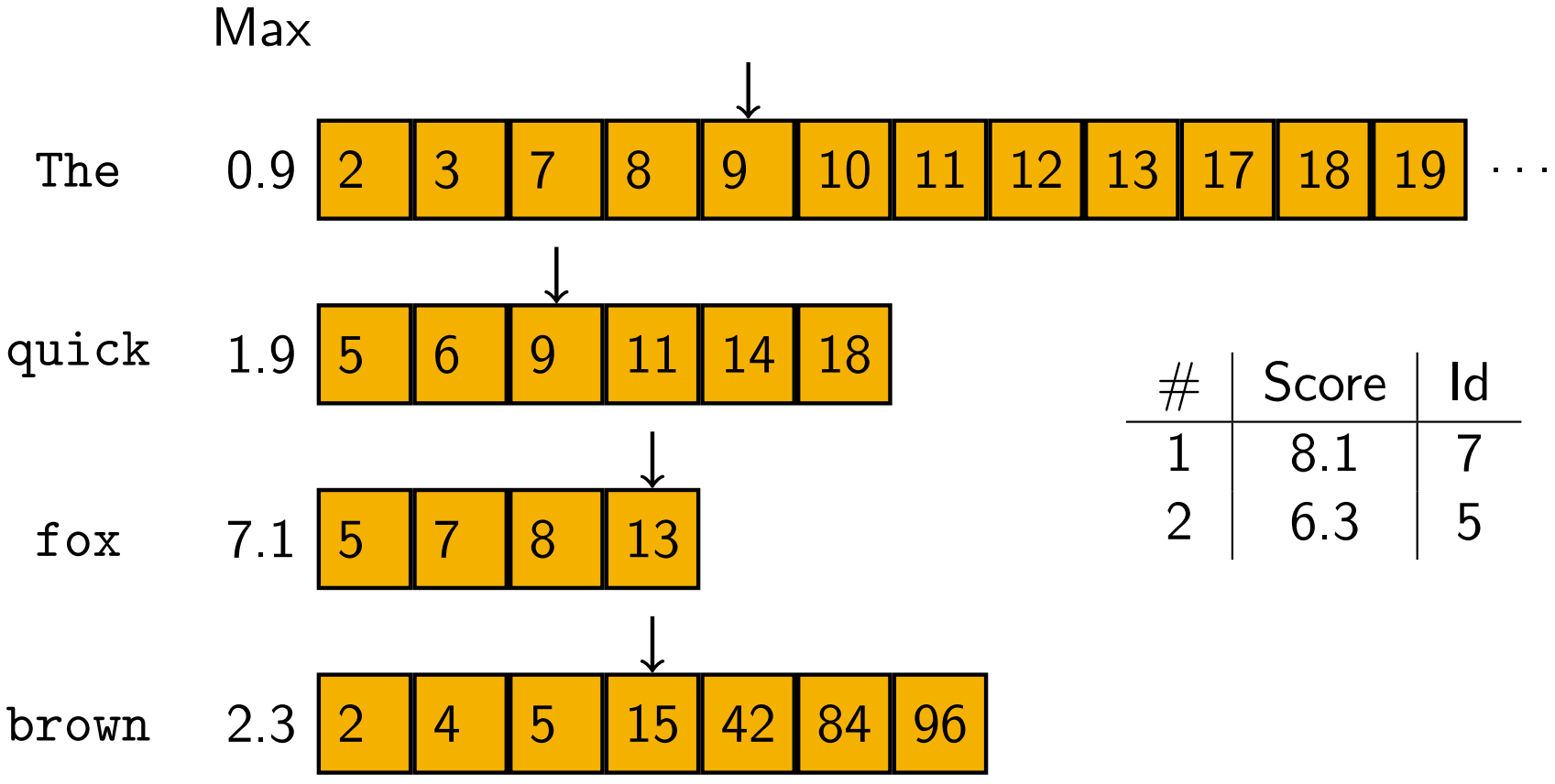
#	Score	Id
1	8.1	7
2	6.3	5

NO! As $0.9 + 1.9 < 6.3$!

WAND - Example - Fast Forward

Query Q : `The quick brown fox` with $k = 2$

What is the next document that has to be evaluated?

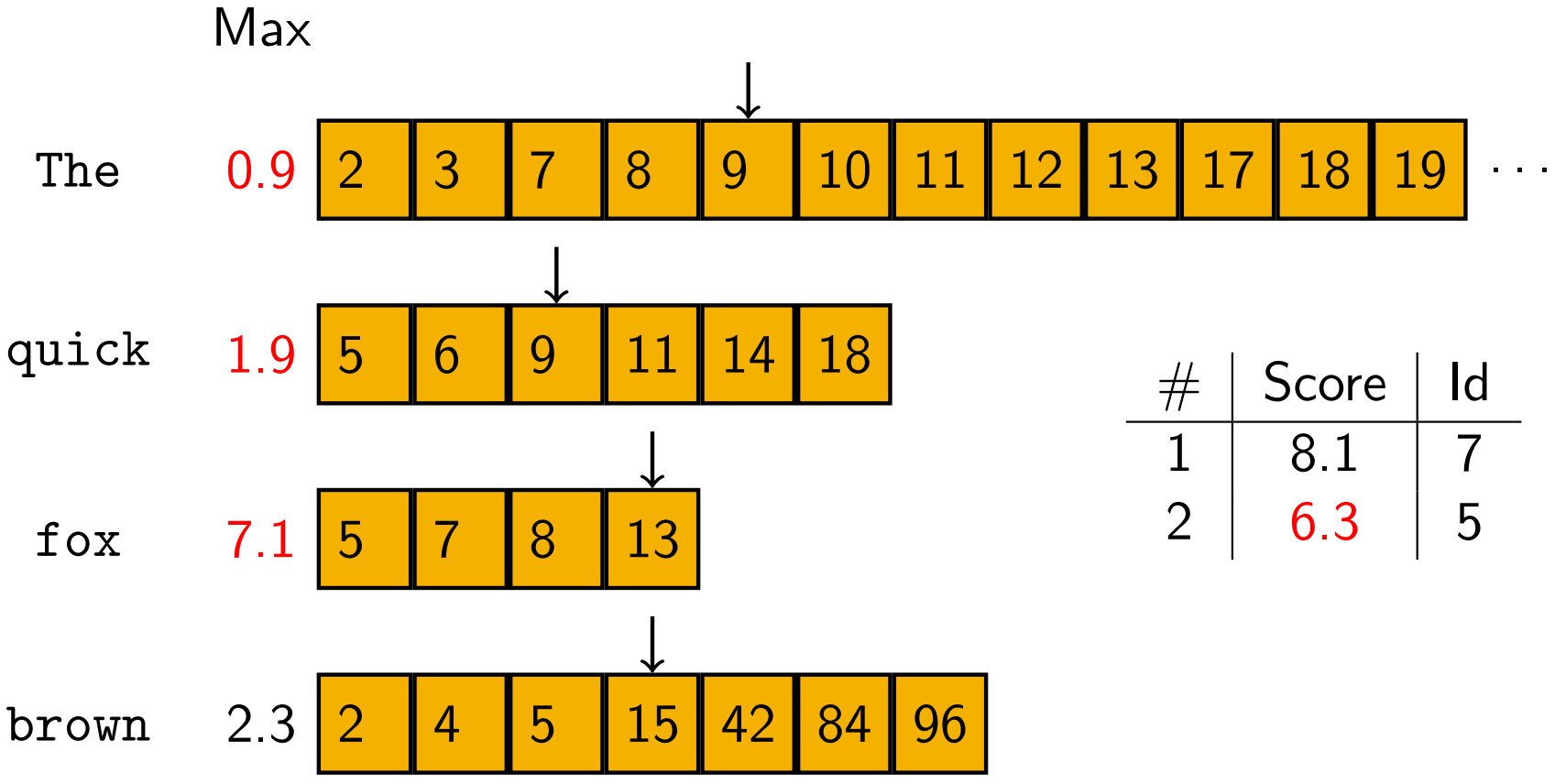


#	Score	Id
1	8.1	7
2	6.3	5

WAND - Example - Fast Forward

Query Q : `The quick brown fox` with $k = 2$

What is the next document that has to be evaluated?



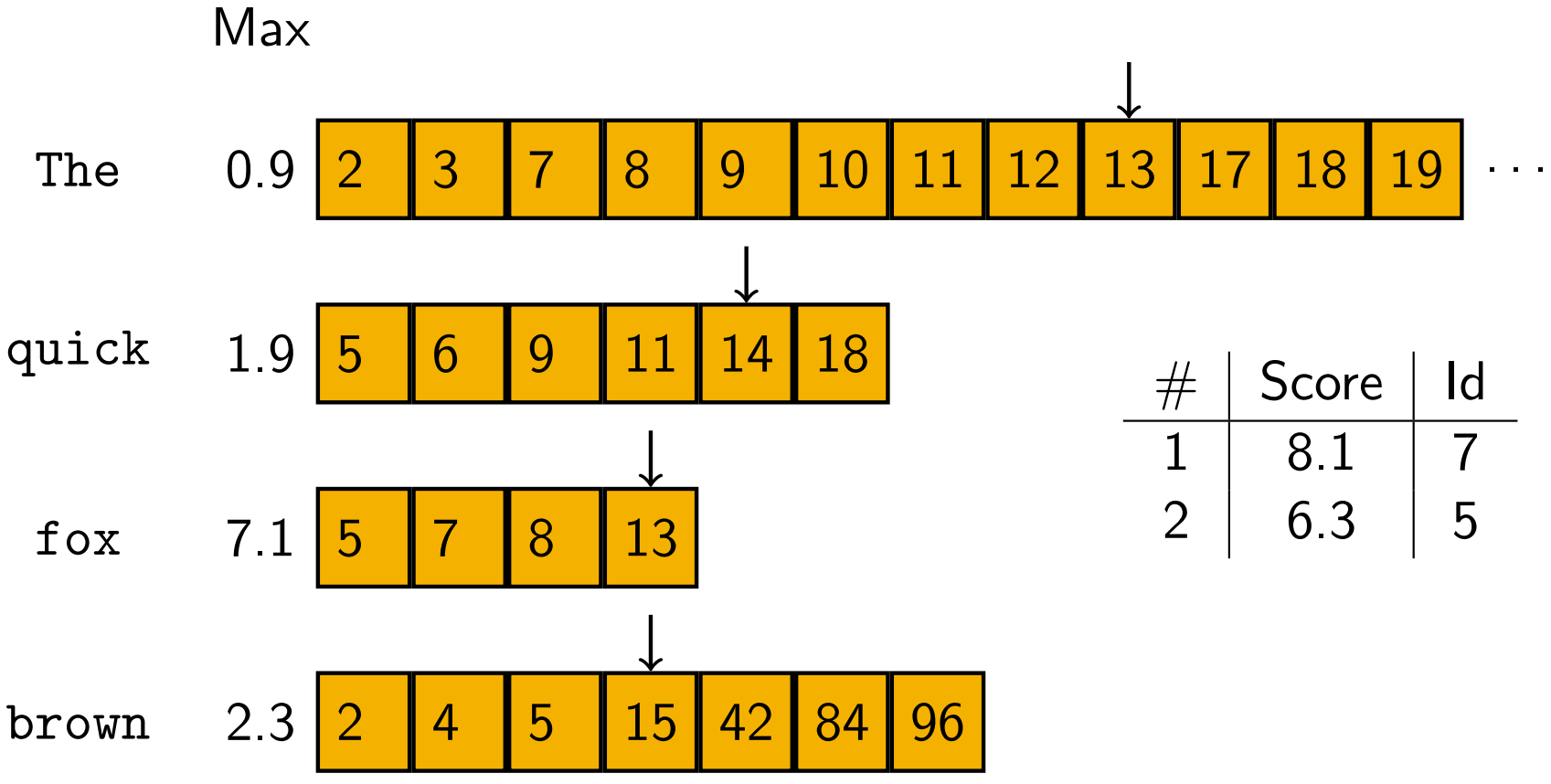
#	Score	Id
1	8.1	7
2	6.3	5

13, as $0.9 + 1.9 + 7.1 > 6.3$!

WAND - Example - Fast Forward

Query Q : `The quick brown fox` with $k = 2$

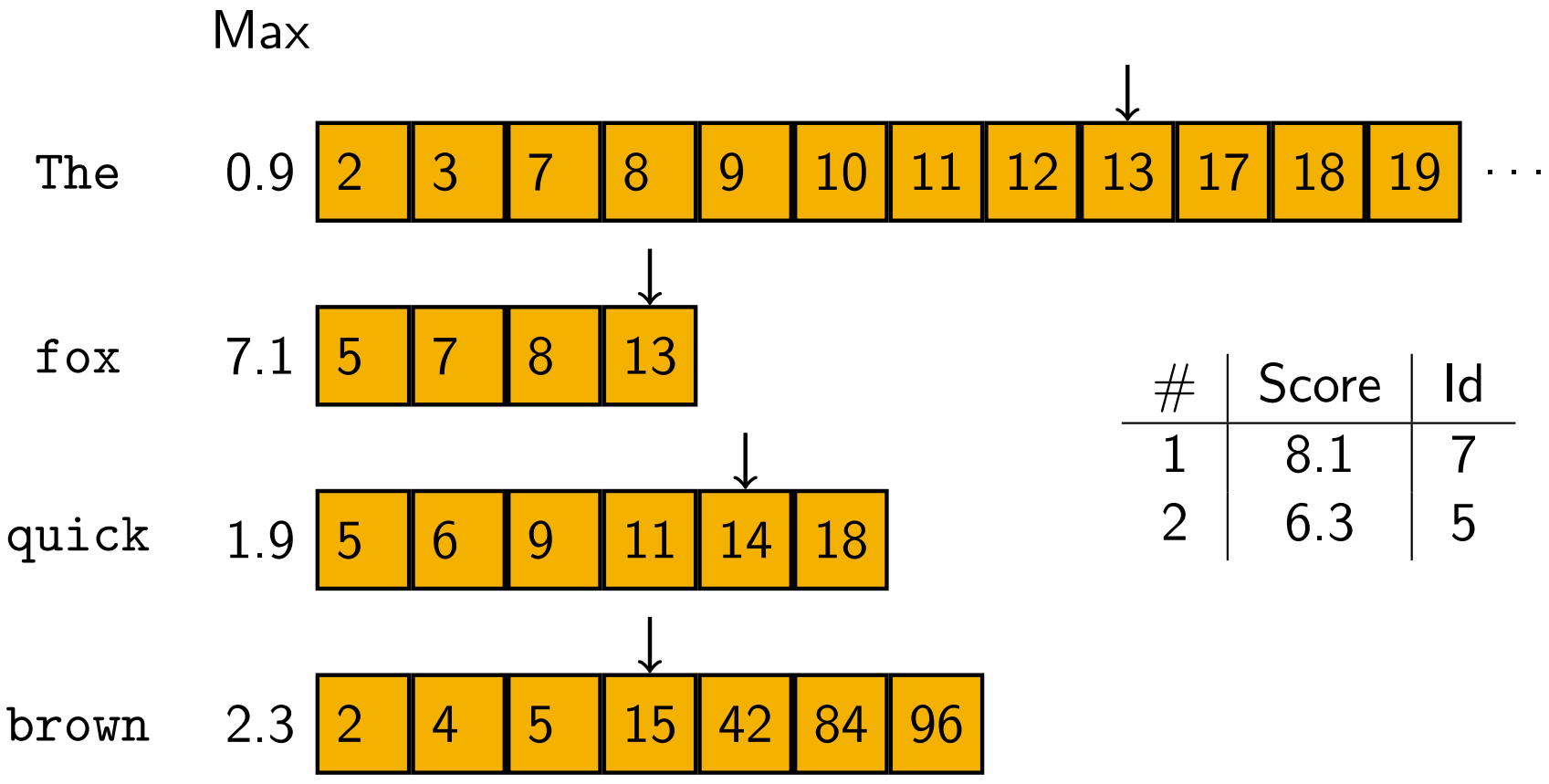
Fast forward smaller ids to 13 (GEQ) and sort.



#	Score	Id
1	8.1	7
2	6.3	5

WAND - Example - Evaluate 13?

Query Q : `The quick brown fox` with $k = 2$



#	Score	Id
1	8.1	7
2	6.3	5

WAND - Algorithm

Given Q, k and the postings lists $L[0 \dots |Q|-1]$ with:

$L[i].max \mathrel{\widehat{=}}$ the **maximum contribution** of the list
$L[i].cur \mathrel{\widehat{=}}$ the current element of the list

1: **function** WAND(Q,k,$L[0 \dots |Q|-1]$)
2: *TopDocs* $= \varnothing$ ▷ Min Heap of size k
3: Threshold $\Theta = 0$ ▷ Smallest score in *TopDocs*
4: **while** Not all lists are processed **do**
5: Sort L based on $L[i].cur$
6: Select pivot list p such that $\sum_0^{p-1} L[i].max >= \Theta$
7: Forward all lists $L[0 \dots |p|-1]$ to $d_p = L[p].cur$
8: Compute S^{BM25}_{Q,d_p} and insert into *TopDocs* if score $> \Theta$
9: $\Theta = \min($*TopDocs*$)$ or 0 if $|$*TopDocs*$| < k$.
10: **end while**
11: Return $TopDocs$
12: **end function**

WAND - Discussion

- Use max contribution of query term to overestimate score of a document.
- Do not score document if it can not enter the top-k heap.
- Utilize GEQ function of compressed representation to skip over large parts of the postings lists.
- Similarity metric fixed at index construction time.
- Works very well in practice.

WAND - Performance

Method	$k = 10$		$k = 1{,}000$	
	Avg.	Med.	Avg.	Med.
Exhaustive	100.0	100.0	100.0	100.0
WAND	3.5	1.0	28.0	11.7

Figure: Fraction of pointers processed as a percentage of the total number of pointers associated with each query, GOV2, using TREC topics 701–850. Across the set of queries, the average number of postings per query for exhaustive processing is 1,460,562, and the median number of postings is 1,080,008. The percentages shown in the table are relative to these two numbers.

WAND - Performance II

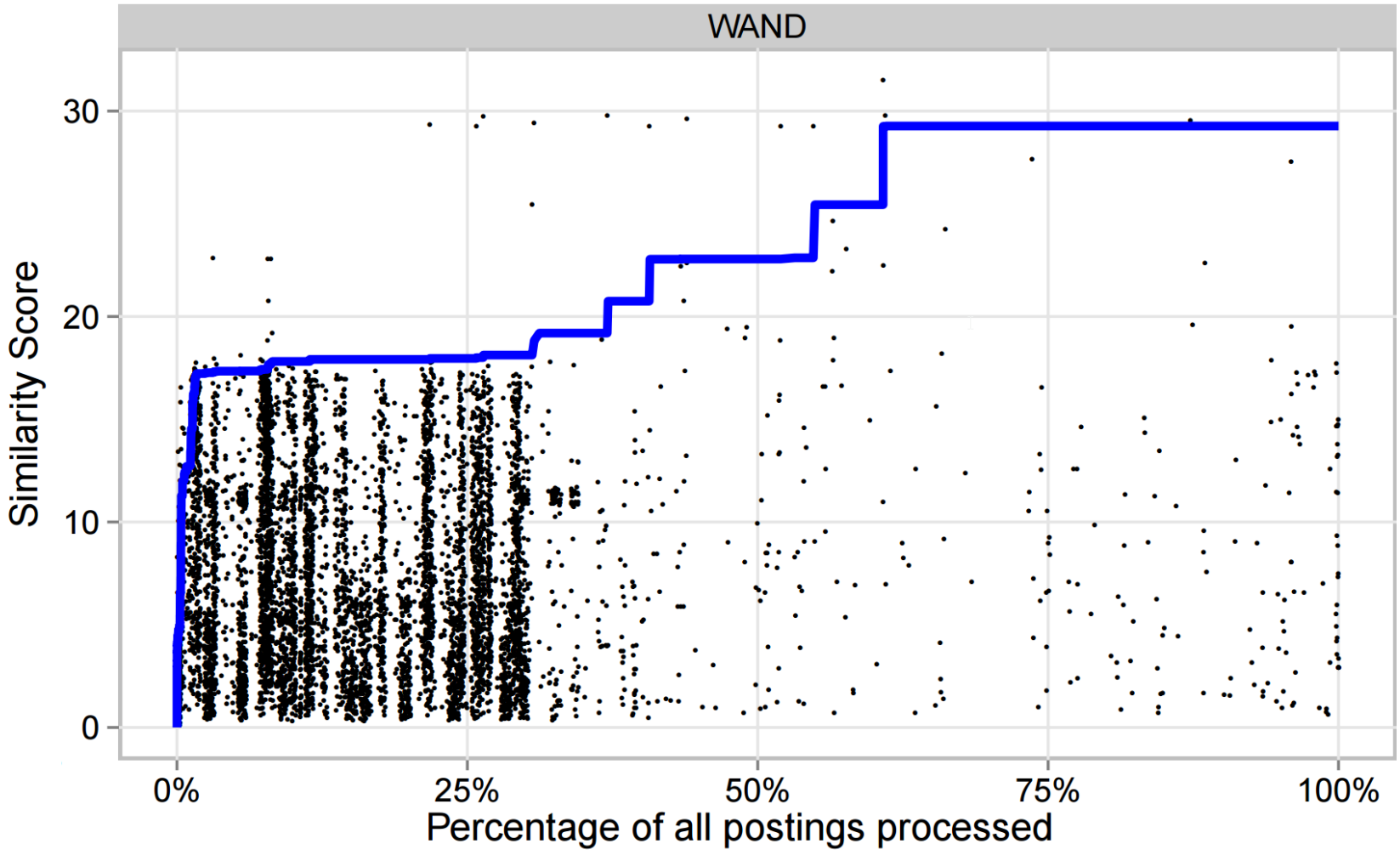


Figure: Evaluation of one query "north korean counterfeiting" for $k = 10$.

Further Reading

Reading:

- Manning, Christopher D; Raghavan, Prabhakar; Schtze, Hinrich; Introduction to information retrieval, Cambridge University Press 2008. (Chapter 5)

Additional References:

- Daniel Lemire, Leonid Boytsov: Decoding billions of integers per second through vectorization. Softw., Pract. Exper. 45(1): 1-29 (2015)

- Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, Jason Y. Zien: Efficient query evaluation using a two-level retrieval process. CIKM: 426-434 (2003)