CSCE 313-200
Introduction to Computer Systems
Spring 2025

Practice III

Dmitri Loguinov
Texas A&M University

April 8, 2025

String Search

- How fast is homework #3 with 216K keywords?
  - Roughly 9.1 KB/s, 38 days to parse the big file
- Using all 8M unique words in large Wikipedia?
  - Speed 240 bytes/s, roughly 4 years to finish (using 12 cores)
- Focus of computer science has always been efficiency
  - Quicksort vs bubble sort, hashing vs sorting, binary vs linear search, min-heap vs linear min()
  - Substring search is another example
- Start with single-string search
  - Assume some text and a given keyword
  - Need to find all occurrences of keyword in text
  - Matches do not have to be complete words

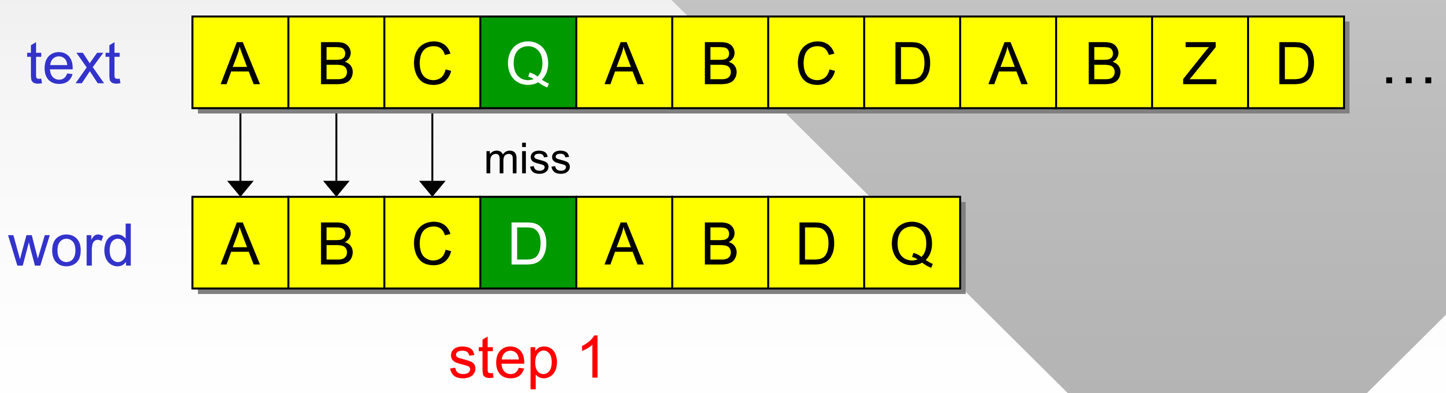
Single String

```
while (off < bufSize - wordLen) {
    if (memcmp (buf + off, word, wordLen) == 0)
          found ++;
    off ++;
}
```

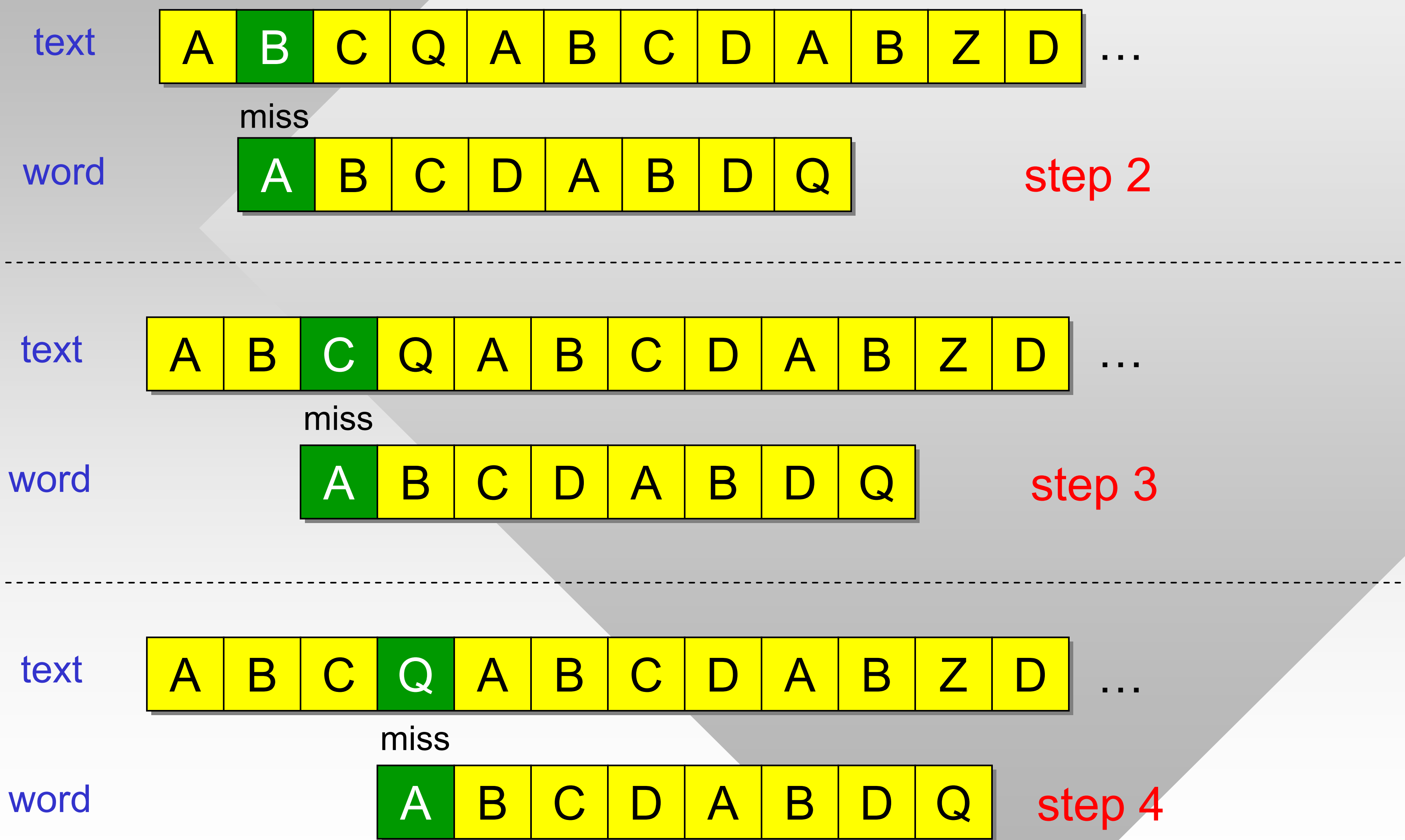
- Naïve method #1: use strcmp or memcmp
- Naïve method #2: use strstr
  - Runs somewhat faster, but still far from optimal

```
char *match = buf;
buf [bufSize] = 0;
while (true) {
    match = strstr (match, word);
    if (match == NULL)
        break;
    found ++;
    match ++;
}
```

- Example of method #1:
  - Worst-case complexity?
  - N = length of text, M = word size, then (N-M)*M



Single String



Single String

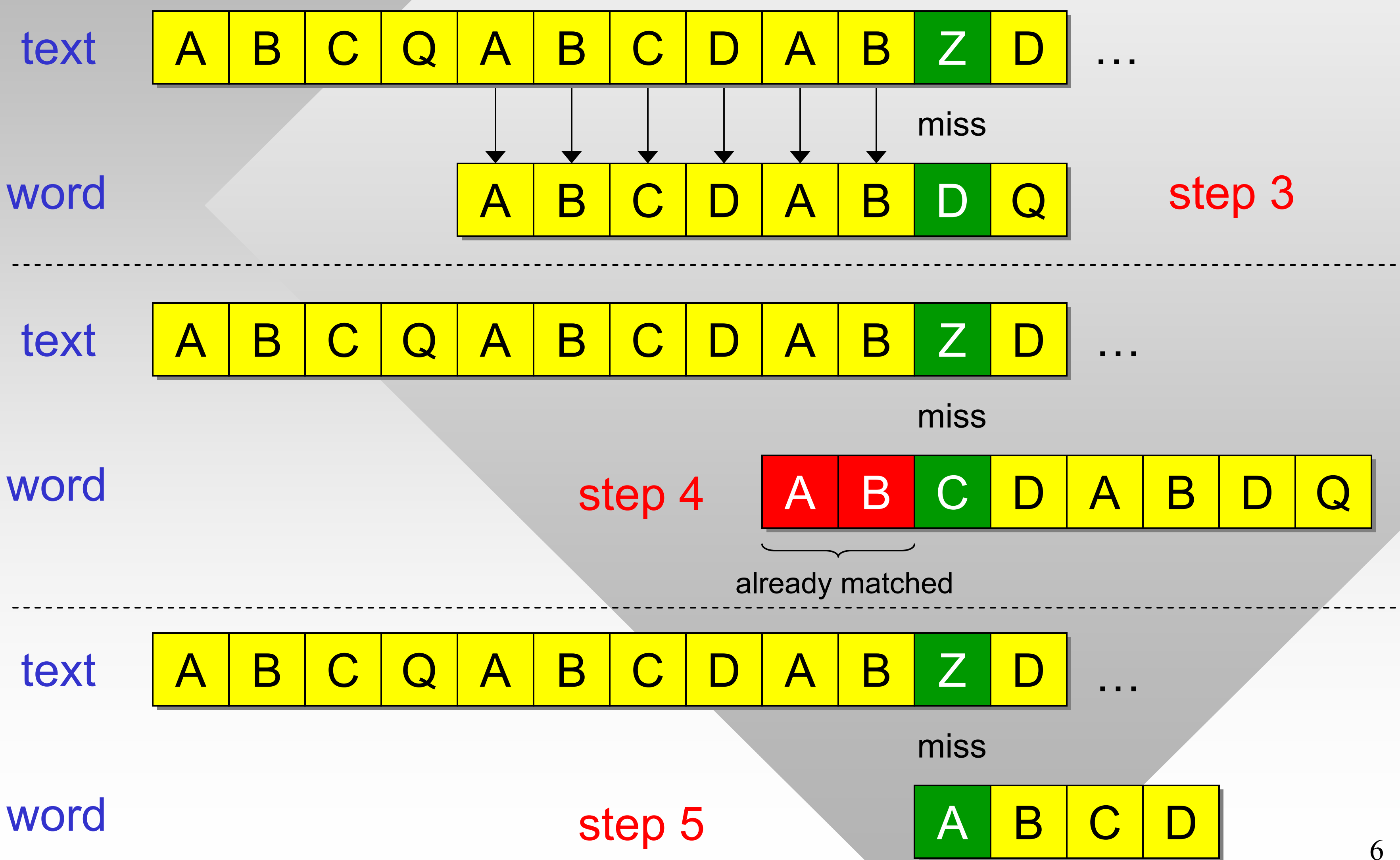
- Naïve takes 7 comparisons to move 4 bytes
  - Total complexity of getting past 12 bytes is 23 comparisons
- Knuth-Morris-Pratt (KMP), 1977:

text	A	B	C	Q	A	B	C	D	A	B	Z	D	…
				miss									
word	A	B	C	D	A	B	D	Q					step 1

---

text	A	B	C	Q	A	B	C	D	A	B	Z	D	…
				miss									
word				A	B	C	D	A	B	D	Q		step 2

Single String



Single String

- Total 6 steps, 15 comparisons to pass 12 bytes
- How does it work?
  - Each character needs two lookup tables (LUTs) – by how many bytes to move after a non-match in this position and where in the word to re-start on the next attempt

word	A	B	C	D	A	B	D	Q
move	1	1	2	3	5	5	4	7
re-start	0	0	0	0	0	0	2	0

tables built offline,
fit in L1 cache

Single String

- Boyer-Moore (BM), 1977:
  - Uses not just distance, but also the mismatched character
- Matching goes right to left, until a mismatch
  - Off is examined position in text

								off							
text	A	B	C	Q	A	B	C	C	Q	B	Z	D	Q	1	2
								miss							
word	A	B	C	D	A	B	D	Q							

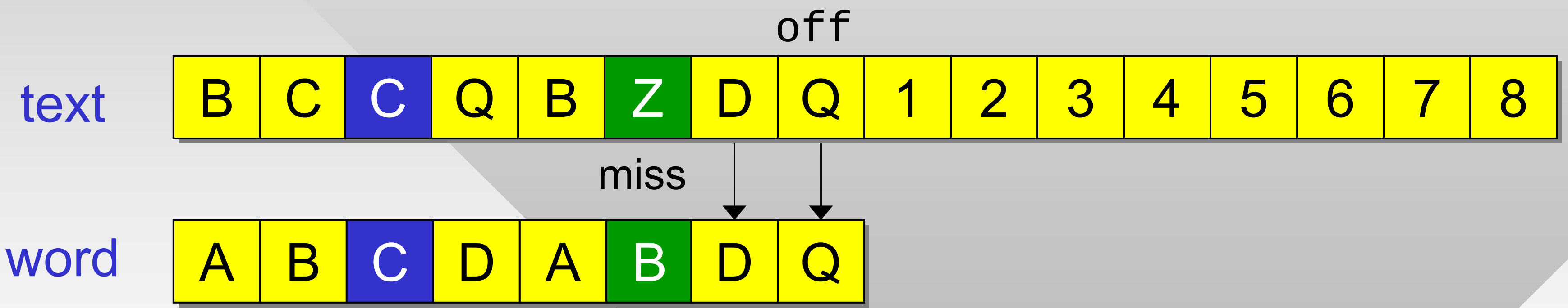
- After a miss, two hash tables move the word forward:
  - Slide[dist]: based on the # of matched characters
  - Shift[char]: based on mismatched character text[off]

Single String

- In the example above
  - Mismatch distance is 0, so slide by 1 char
  - Mismatch char = C, so shift by 5

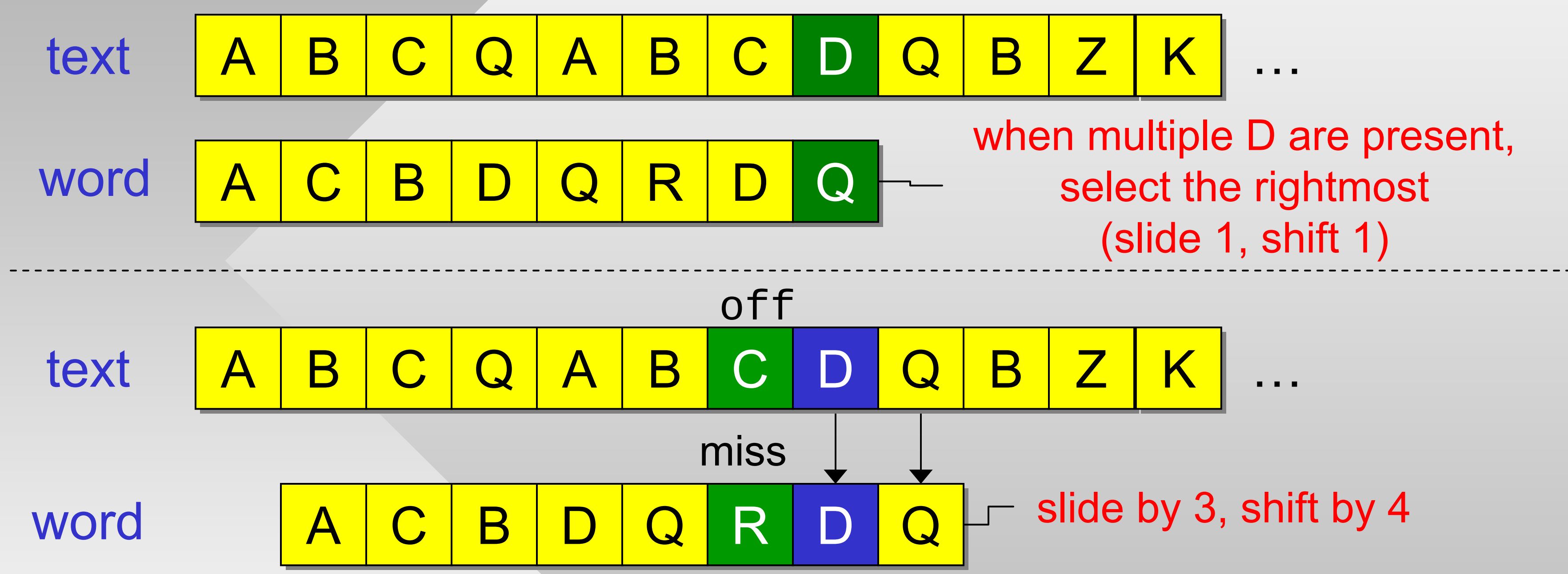
when moving forward, take the larger of the two

- After moving off by 5:



- In this case, mismatch occurs at `text[off] = Z`:
  - Mismatch distance = 2, slide word by 8
  - Mismatch char = Z, shift word by 6

Single String



- For words that have rare letter combinations, we can be skipping by M each time
  - Best case complexity is sub-linear, i.e., N/M comparisons
- Typically faster than KMP for larger M

Single String

- Can we do better?
- Notice that BM gets stuck on popular characters, while ideally it should skip most examined locations
  - E.g., “zebra” incurs detailed inspection any time it hits an ‘e’
- Idea: set up a hash table with 2-byte combinations
  - E.g., “ze”, “eb”, “br”, “ra” which are much more rare
  - Then scan the text using an `unsigned short` (2-byte) pointer
- Caveat: don’t know alignment of the word, may hit something like “_z” and miss the word
  - Need to set up wildcard entries ?z and a? for all possible leading and trailing characters
  - If only full words are needed, ? will be a white space

Multiple Strings

- Why was homework #3 so inefficient?

text

Z	B	C	Q	A	B

hash H

keyword list
apple
banana
mango
zebra
…

looking at ‘z’, no need to attempt a match to apple, banana, mango

- Idea: do not compare current byte to all strings, only to those that can potentially be a match
- Rabin-Karp (RK), 1987
  - Assume M is the smallest keyword length
  - Compute a hash H of the next M chars from current location
  - Hit a hash table, compare with words that tie for that hash
  - Speed is only based on the length of collision chains

Multiple Strings

example with M = 3, B = 10

- After hash table lookup, slide by one byte forward, recompute the hash of the next M chars

text

Z	B	C	Q	A	B

hash H_0

hash H_1

3	5	7	8	2	4

$H_0 = 357$

$H_1 = 578$

- Notice that M-1 chars are the same in both hashes
  - Main twist of the algorithm is to use a rolling hash, which obtains H_{i+1} from H_i in O(1) time
- Treating hashes as base-B integers, we have
  - $H_0 = str[0] * B^{M-1} + str[1] * B^{M-2} + \ldots + str[M-1]$
  - $H_{i+1} = (H_i * B + str[i+M]) \% B^M$

Wrap-up

- Larger M means fewer collisions and faster operation
- With M = 3 and 216K strings, RK runs at 20MB/s
  - 2000 times faster than the naïve method
- Indexing a file with unknown keywords is slightly different, but the idea is similar to RK
  - Homework #4 explores this in more detail
- Main goal is to design code that processes all 4.5B words in large Wikipedia in ~35 sec (135M wps)
  - 3.7M times faster than the method in homework #3
- Homework #4 has 3 checkpoints
  - The first two should be done early
  - Checkpoint #3 is more complex, uses virtual memory