Text Factorization of gzip

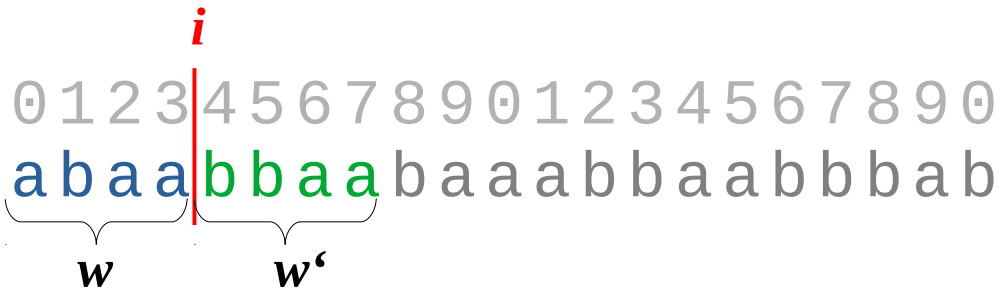
LZ77 with Sliding Windows

LZ77 with Sliding Windows

```
0123456789012345678 90
abaabbaabaaaabbaabbbab
```

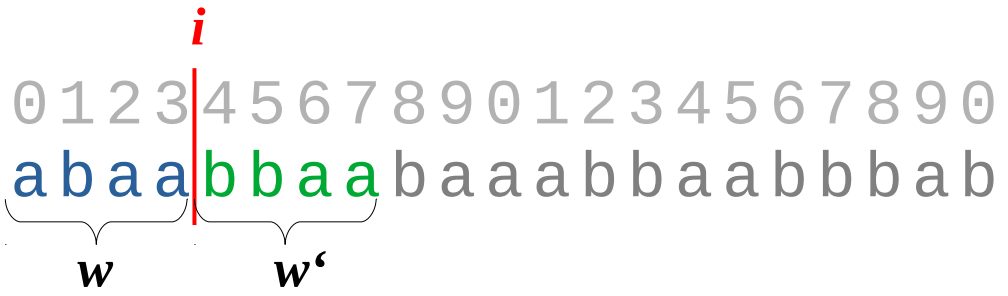
- **Goal**: Keep memory use small (order of window size)

LZ77 with Sliding Windows



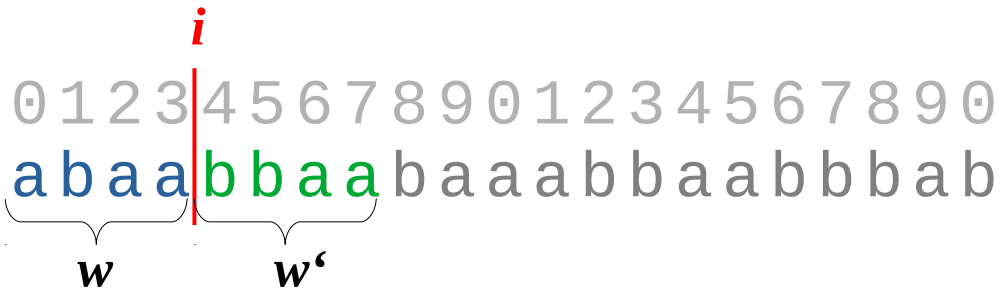
- **Goal**: Keep memory use small (order of window size)

LZ77 with Sliding Windows



- **Goal**: Keep memory use small (order of window size)
- Window of size w
  - Already factorized

LZ77 with Sliding Windows



- **Goal**: Keep memory use small (order of window size)
- Window of size w
  - Already factorized
- Lookahead window of size $w‘$
  - Factorize using **pattern matching** in entire window of size $w+w‘$

gzip Parameters

gzip Parameters

- Minimum match m := 3
  - Shorter matches will not be replaced by references

gzip Parameters

- Minimum match m := 3
  - Shorter matches will not be replaced by references
- Maximum match M := 258
  - References cannot be longer

gzip Parameters

- Minimum match m := 3
  - Shorter matches will not be replaced by references
- Maximum match M := 258
  - References cannot be longer
  - Since M-m = 255, reference lengths can be encoded in a byte alphabet

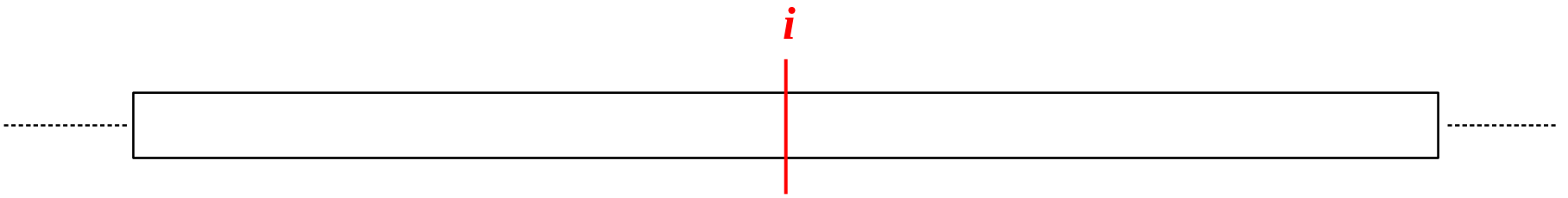
gzip Parameters

- Minimum match m := 3
  - Shorter matches will not be replaced by references
- Maximum match M := 258
  - References cannot be longer
  - Since M-m = 255, reference lengths can be encoded in a byte alphabet
- Window size w‘ = w := 2^{15}
  - Ensures compatibility to 16-bit architectures

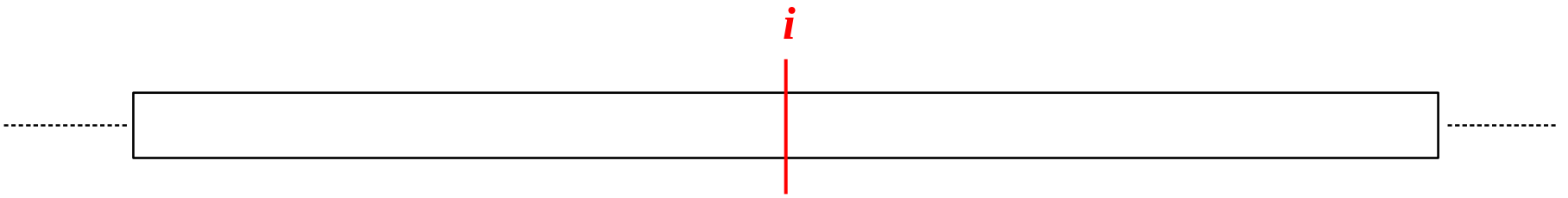
gzip Parameters

- Minimum match m := 3
  - Shorter matches will not be replaced by references
- Maximum match M := 258
  - References cannot be longer
  - Since M-m = 255, reference lengths can be encoded in a byte alphabet
- Window size w‘ = w := 2^{15}
  - Ensures compatibility to 16-bit architectures
- Compression level (0 to 9)
  - Determine ***heuristic*** parameters, details follow

Pattern Matching Scheme

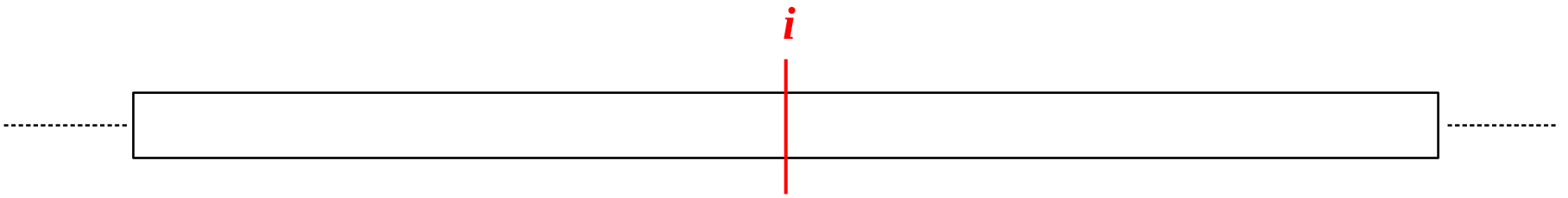


Pattern Matching Scheme



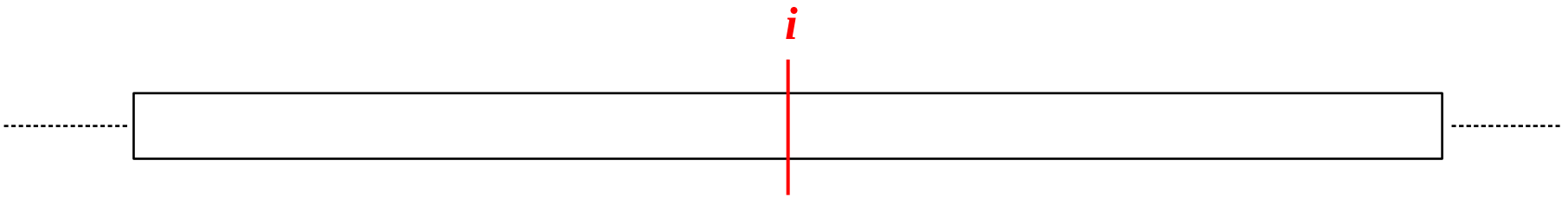
- At current position ***i***, find set of candidate positions $i \geq j \geq i - w$

Pattern Matching Scheme



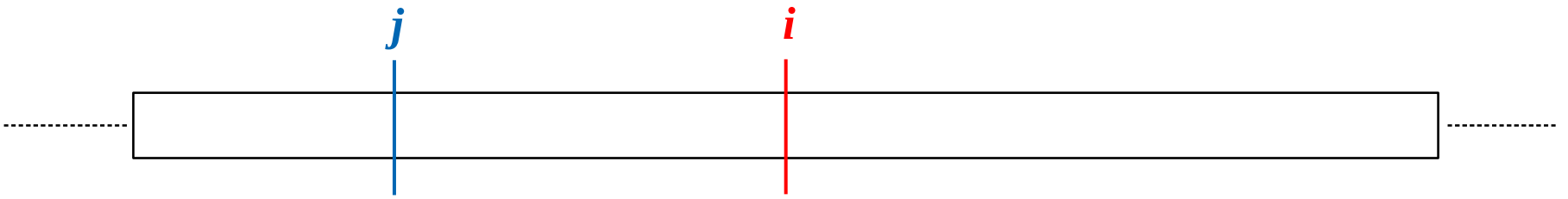
- At current position ***i***, find set of candidate positions $i \geq j \geq i - w$
  - We only ever match against these candidates
  - Details follow in later slides

Pattern Matching Scheme

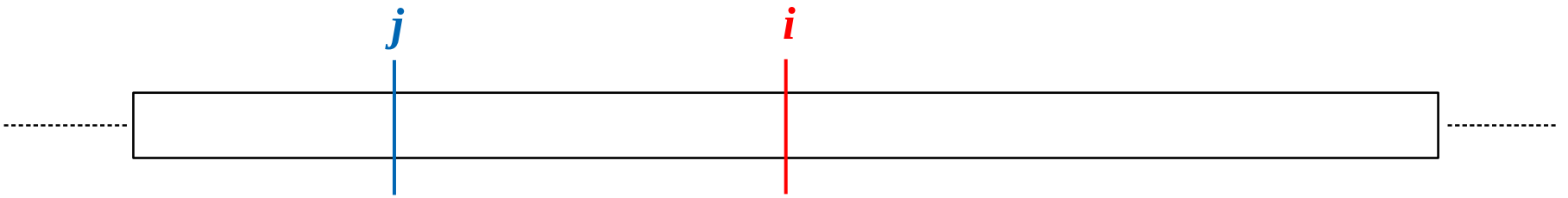


- At current position i, find set of candidate positions $i \geq j \geq i - w$
  - We only ever match against these candidates
  - Details follow in later slides
- Keep track of longest match of length L among the candidates

Pattern Matching Instance

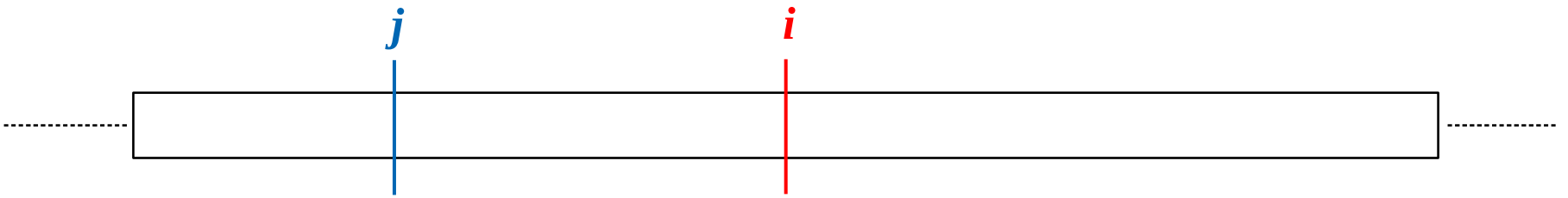


Pattern Matching Instance



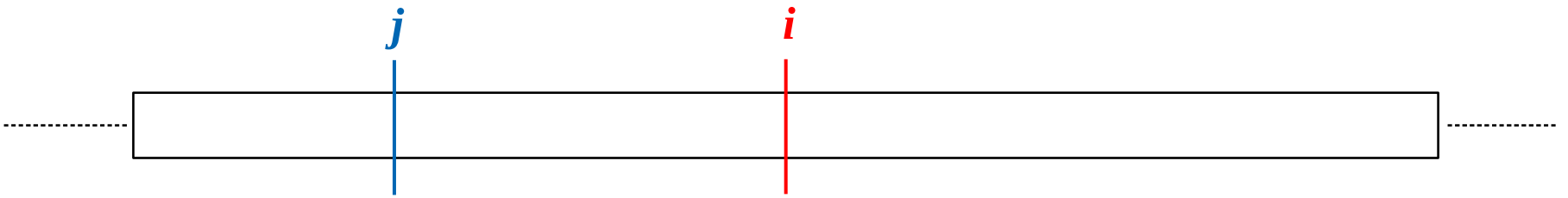
- Given current position i and candidate position $i \geq j \geq i - w$

Pattern Matching Instance



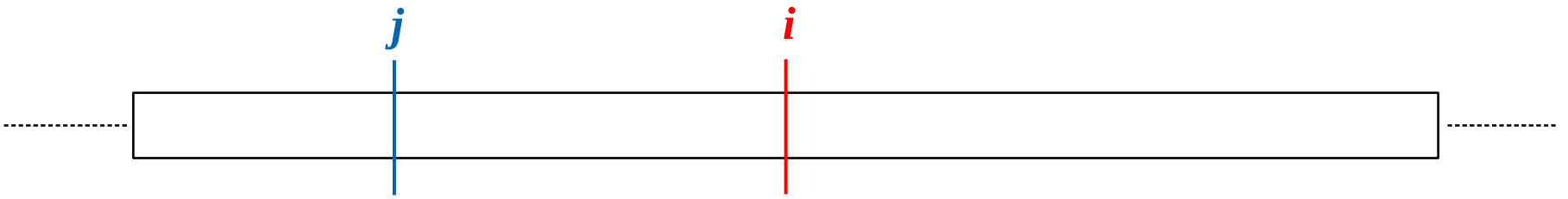
- Given current position i and candidate position $i \geq j \geq i - w$
  - Naively compare up to M bytes

Pattern Matching Instance



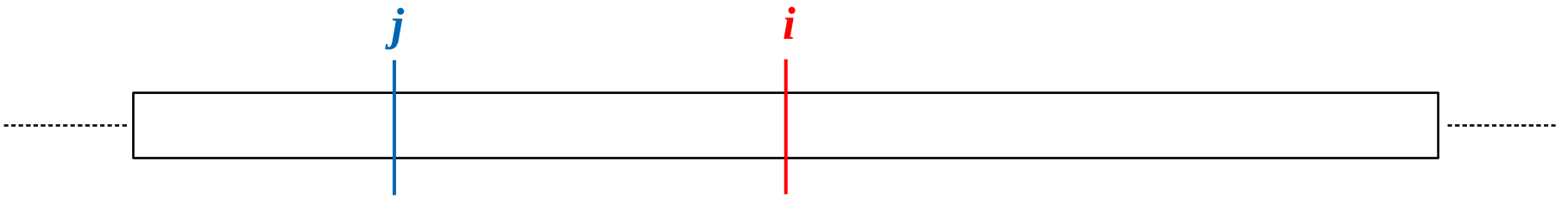
- Given current position i and candidate position $i \geq j \geq i - w$
  - Naively compare up to M bytes
  - Pack bytes into 16-bit words until mismatch, then do one final comparison
    - Ensures compatibility to 16-bit architectures

Pattern Matching Instance



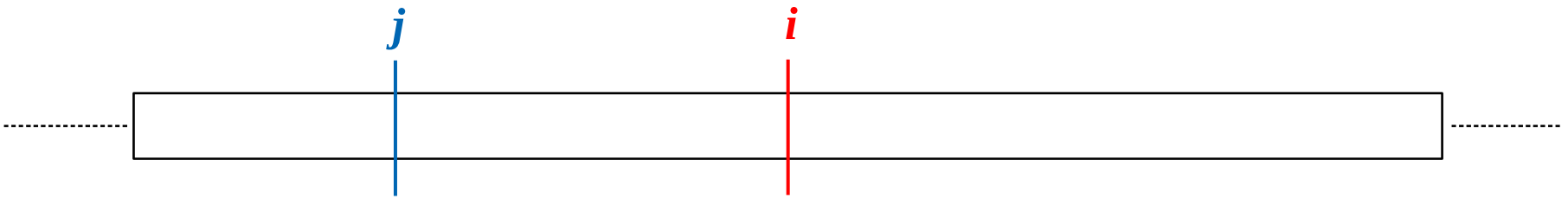
- Given current position i and candidate position $i \geq j \geq i - w$
  - Naively compare up to M bytes
  - Pack bytes into 16-bit words until mismatch, then do one final comparison
    - Ensures compatibility to 16-bit architectures
- Let $\ell := \min\{\text{lce}(i, j), M\}$ be the length of the match

Pattern Matching Instance



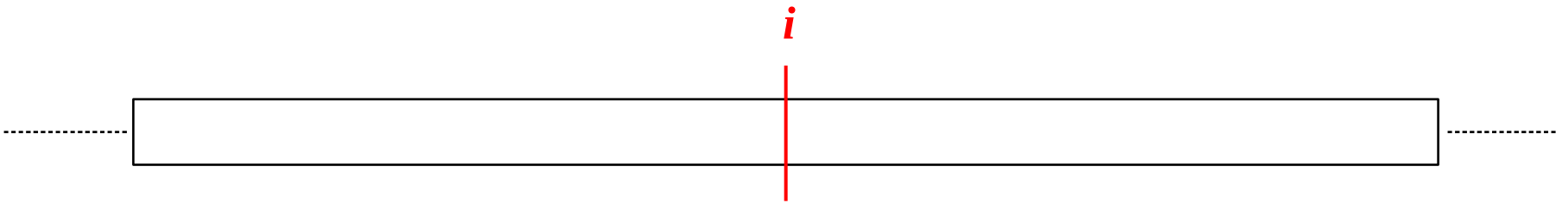
- Given current position i and candidate position $i \geq j \geq i - w$
  - Naively compare up to M bytes
  - Pack bytes into 16-bit words until mismatch, then do one final comparison
    - Ensures compatibility to 16-bit architectures
- Let $\ell := \min\{\text{lce}(i, j), M\}$ be the length of the match
- If ℓ is a ***nice*** match, immediately emit (j, ℓ), and skip any further candidates

Pattern Matching Instance

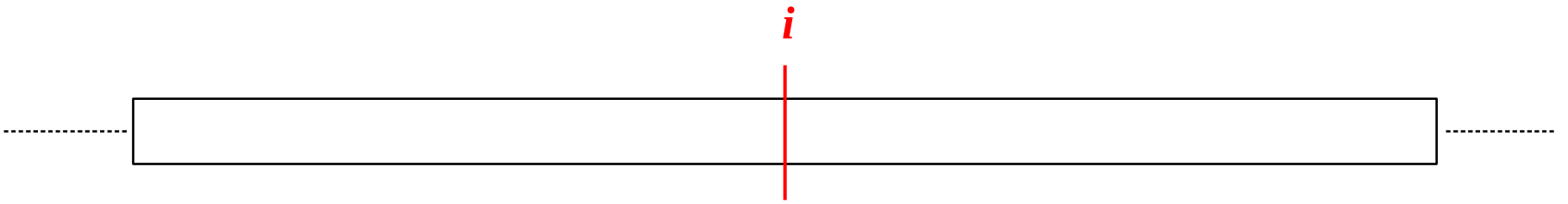


- Given current position i and candidate position $i \geq j \geq i - w$
  - Naively compare up to M bytes
  - Pack bytes into 16-bit words until mismatch, then do one final comparison
    - Ensures compatibility to 16-bit architectures
- Let $\ell := \min\{\text{lce}(i, j), M\}$ be the length of the match
- If ℓ is a ***nice*** match, immediately emit (j, ℓ), and skip any further candidates
- Otherwise, let $L := \max\{L, \ell\}$ and go on to next candidate

Candidate Management

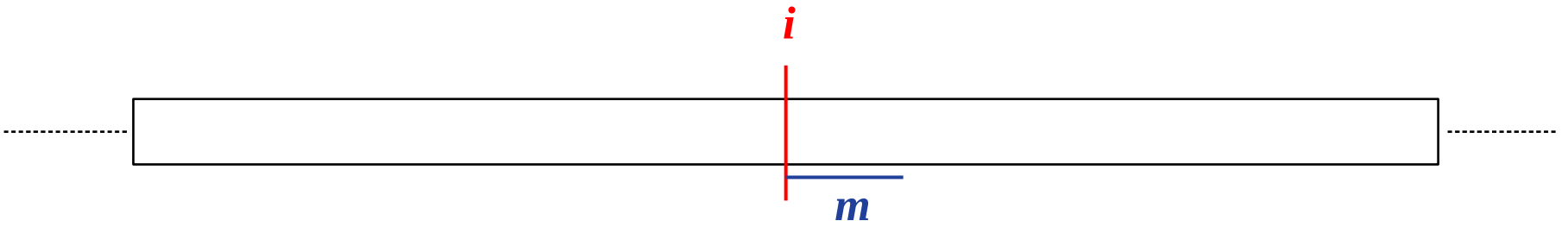


Candidate Management



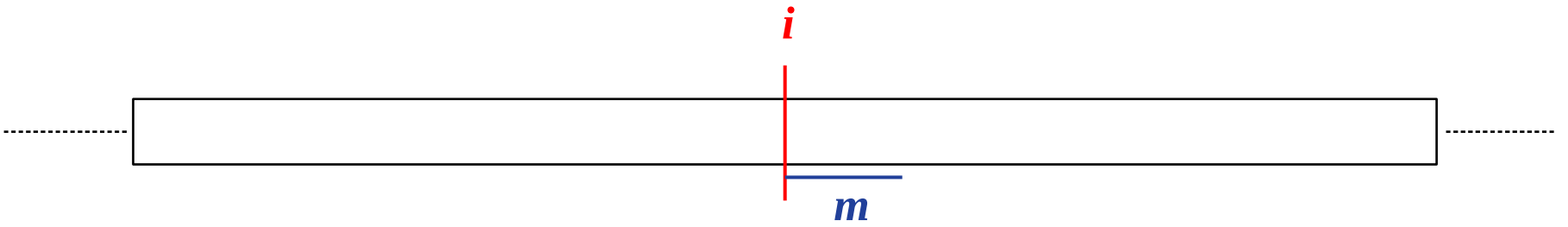
- Candidates are maintained in a *hash table of chains*

Candidate Management



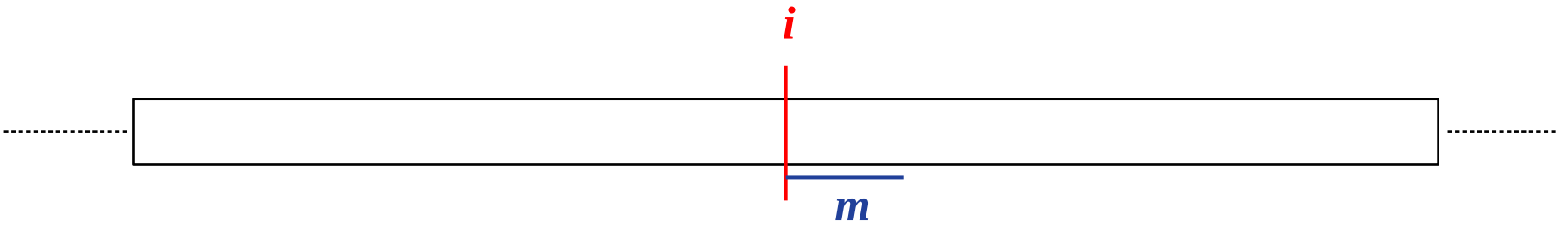
- Candidates are maintained in a *hash table of chains*
  - Let ***h*** be a hash function s.t. ***h***(***i***) computes a hash value for ***m*** bytes starting at ***i***

Candidate Management



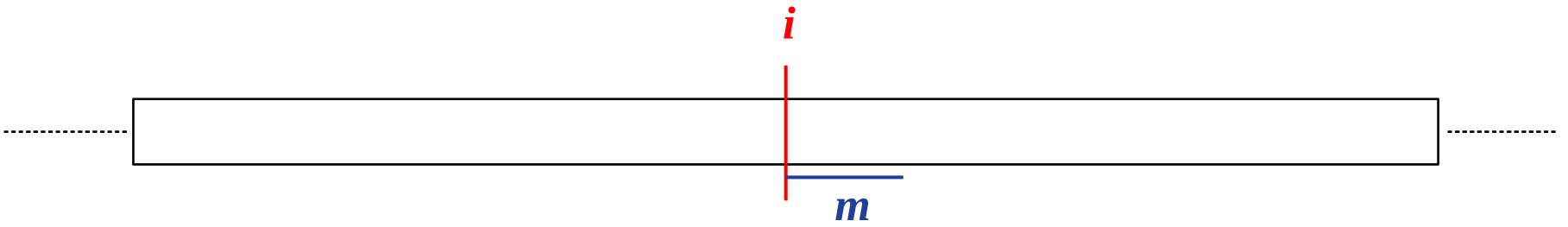
- Candidates are maintained in a *hash table of chains*
  - Let ***h*** be a hash function s.t. ***h***(***i***) computes a hash value for ***m*** bytes starting at ***i***
    - gzip implements ***h*** as a rolling hash function

Candidate Management



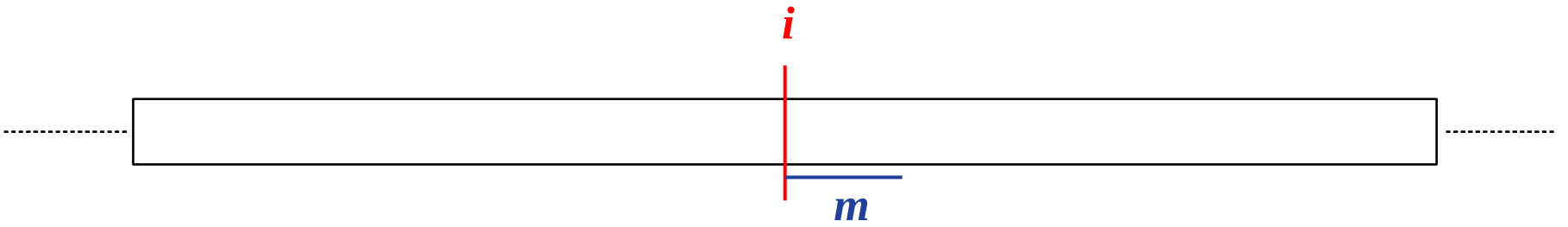
- Candidates are maintained in a *hash table of chains*
  - Let ***h*** be a hash function s.t. ***h***(***i***) computes a hash value for ***m*** bytes starting at ***i***
    - gzip implements ***h*** as a rolling hash function
    - ***j*** = ***Head***[***h***(***i***)] is the head of the chain for ***h***(***i***) (|***Head***|=2^{15})

Candidate Management



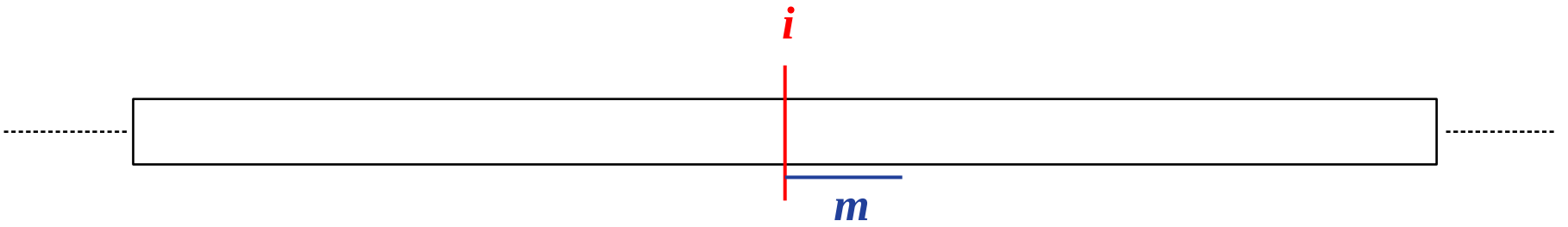
- Candidates are maintained in a *hash table of chains*
  - Let h be a hash function s.t. $h(i)$ computes a hash value for m bytes starting at i
    - gzip implements h as a rolling hash function
    - $j = Head[h(i)]$ is the head of the chain for $h(i)$ ($|Head|=2^{15}$)
  - $j' = Prev[j]$ is the position that was previously stored at $Head[h(i)]$ ($|Prev|=w$)

Candidate Management



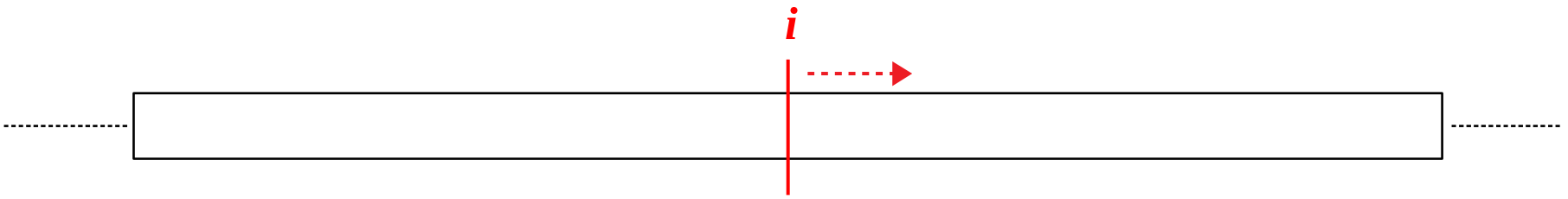
- Candidates are maintained in a *hash table of chains*
  - Let h be a hash function s.t. $h(i)$ computes a hash value for m bytes starting at i
    - gzip implements h as a rolling hash function
    - $j = Head[h(i)]$ is the head of the chain for $h(i)$ ($|Head|=2^{15}$)
  - $j' = Prev[j]$ is the position that was previously stored at $Head[h(i)]$ ($|Prev|=w$)
    - This forms a chain, where collisions are possible, but do not hurt

Candidate Management

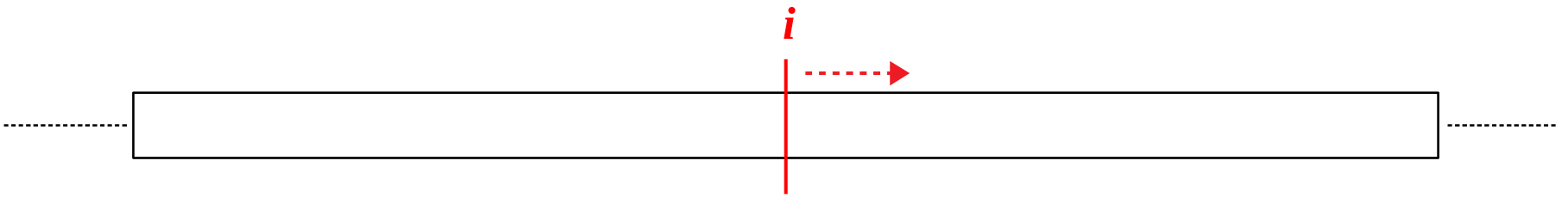


- Candidates are maintained in a *hash table of chains*
  - Let h be a hash function s.t. $h(i)$ computes a hash value for m bytes starting at i
    - gzip implements h as a rolling hash function
    - $j = Head[h(i)]$ is the head of the chain for $h(i)$ ($|Head|=2^{15}$)
  - $j' = Prev[j]$ is the position that was previously stored at $Head[h(i)]$ ($|Prev|=w$)
    - This forms a chain, where collisions are possible, but do not hurt
    - For any i, we follow the chain at most c times (for `gzip -9`, $c := 4096$)

Pattern Matching Aftermath

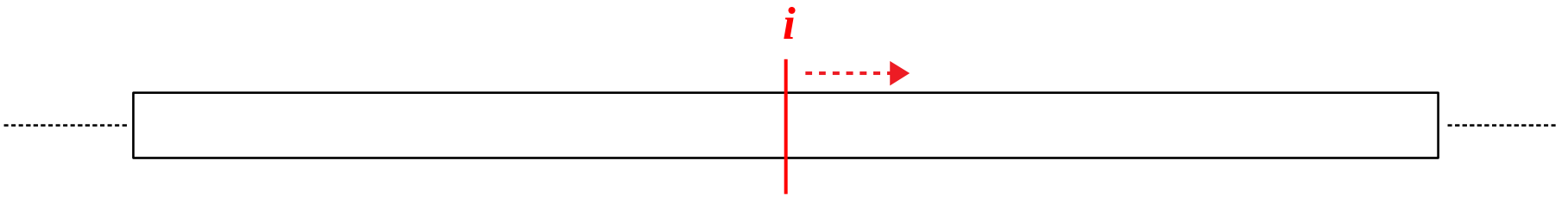


Pattern Matching Aftermath



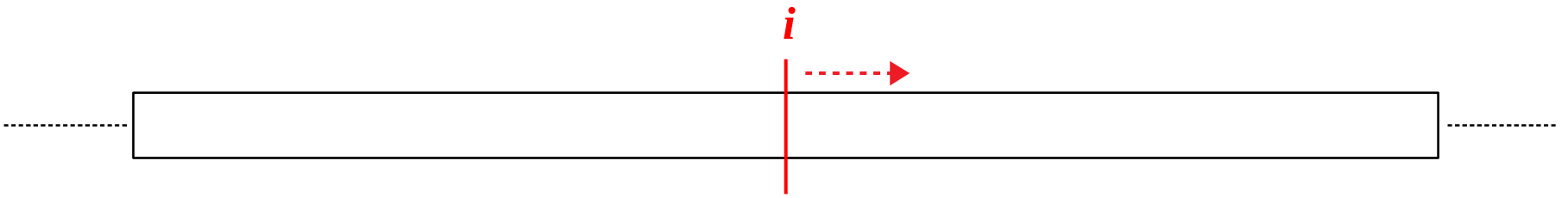
- Take care of some business before advancing to position ***i+1***

Pattern Matching Aftermath



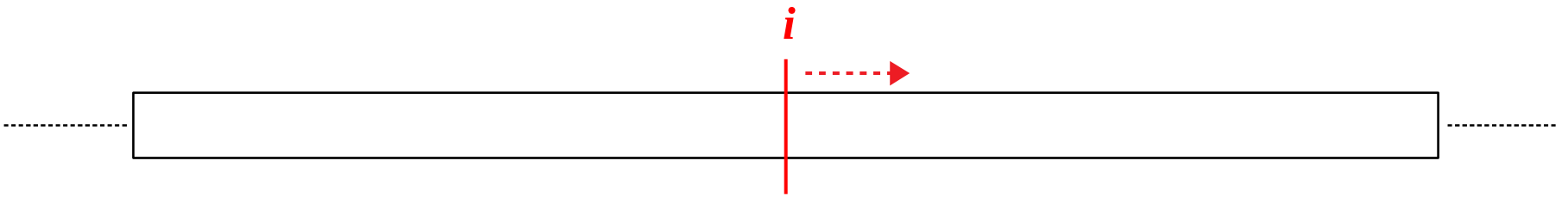
- Take care of some business before advancing to position ***i+1***
- Update chain by setting ***Prev***[***Head***[***h***(***i***)]] := ***Head***[***h***(***i***)] and ***Head***[***h***(***i***)] := ***i***

Pattern Matching Aftermath



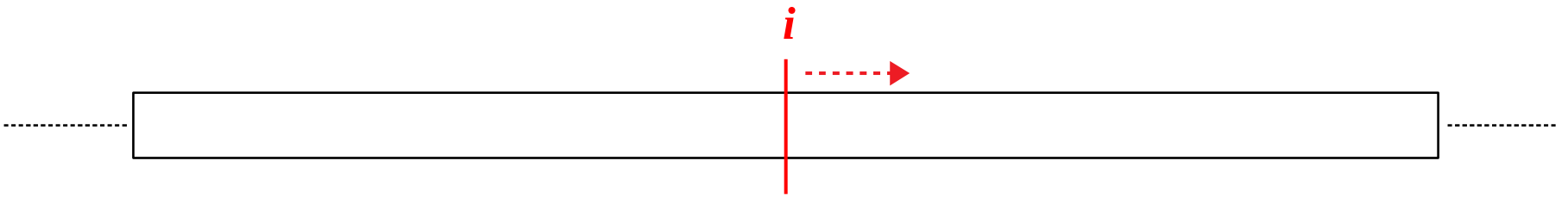
- Take care of some business before advancing to position ***i+1***
- Update chain by setting ***Prev***[***Head***[***h***(***i***)]] := ***Head***[***h***(***i***)] and ***Head***[***h***(***i***)] := ***i***
  - Thus, chains assume a descending order, helping find a *rightmost* parsing

Pattern Matching Aftermath



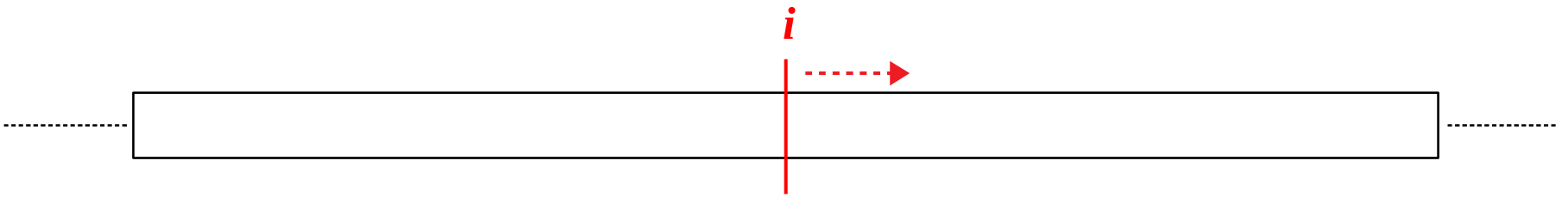
- Take care of some business before advancing to position ***i+1***
- Update chain by setting ***Prev***[***Head***[***h***(***i***)]] := ***Head***[***h***(***i***)] and ***Head***[***h***(***i***)] := ***i***
  - Thus, chains assume a descending order, helping find a *rightmost* parsing
- Consider previous longest match ***L'*** (found at position ***i-1***)

Pattern Matching Aftermath



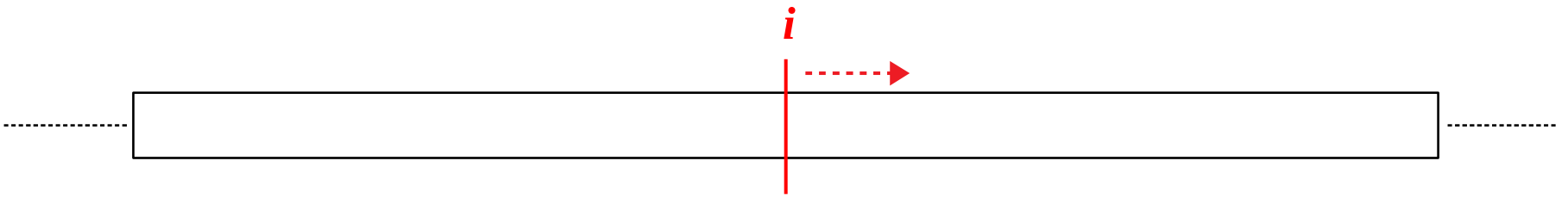
- Take care of some business before advancing to position ***i+1***
- Update chain by setting ***Prev***[***Head***[***h***(***i***)]] := ***Head***[***h***(***i***)] and ***Head***[***h***(***i***)] := ***i***
  - Thus, chains assume a descending order, helping find a *rightmost* parsing
- Consider previous longest match ***L'*** (found at position ***i-1***)
  - If ***L*** > ***L'***, act greedily by emitting ***T***[***i-1***] as a literal

Pattern Matching Aftermath



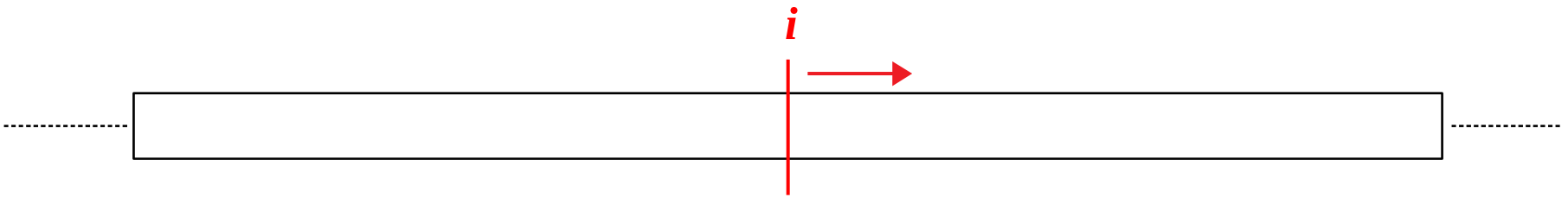
- Take care of some business before advancing to position ***i+1***
- Update chain by setting ***Prev***[***Head***[***h***(***i***)]] := ***Head***[***h***(***i***)] and ***Head***[***h***(***i***)] := ***i***
  - Thus, chains assume a descending order, helping find a *rightmost* parsing
- Consider previous longest match ***L'*** (found at position ***i-1***)
  - If ***L*** > ***L'***, act greedily by emitting ***T***[***i-1***] as a literal
  - Otherwise, emit reference corresponding to previous longest match

Pattern Matching Aftermath

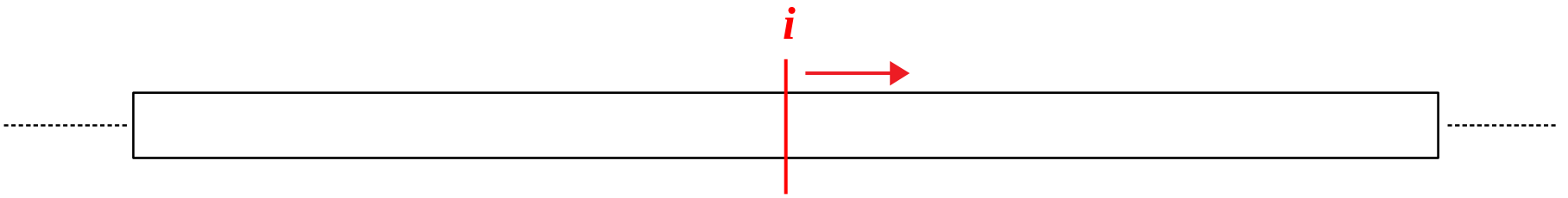


- Take care of some business before advancing to position ***i+1***
- Update chain by setting ***Prev***[***Head***[***h***(***i***)]] := ***Head***[***h***(***i***)] and ***Head***[***h***(***i***)] := ***i***
  - Thus, chains assume a descending order, helping find a *rightmost* parsing
- Consider previous longest match ***L'*** (found at position ***i-1***)
  - If ***L*** > ***L'***, act greedily by emitting ***T***[***i-1***] as a literal
  - Otherwise, emit reference corresponding to previous longest match
- Set ***L'*** := ***L*** for the next step

Heuristics for Next Step

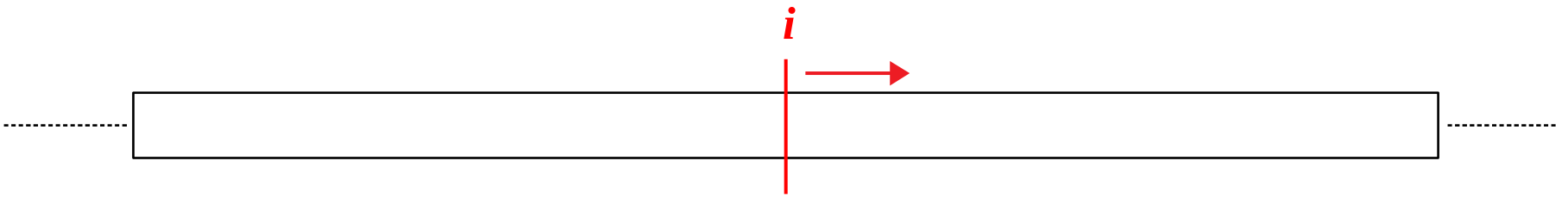


Heuristics for Next Step



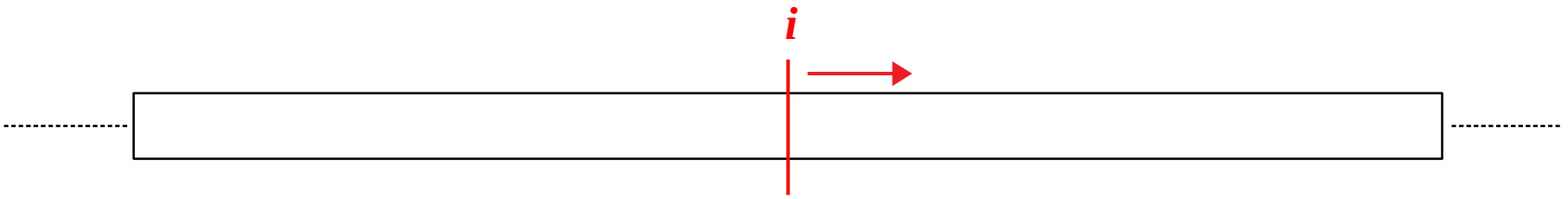
- We advance to position ***i+1***

Heuristics for Next Step



- We advance to position ***i+1***
- If longest previous match ***L'*** exceeds the ***lazy*** threshold, don't bother doing any pattern matching in this step
  - The corresponding reference is emitted right away

Heuristics for Next Step



- We advance to position ***i+1***
- If longest previous match ***L‘*** exceeds the ***lazy*** threshold, don‘t bother doing any pattern matching in this step
  - The corresponding reference is emitted right away
- Otherwise, if ***L‘*** is a ***good*** match, reduce chain follow limit to $c/4$ for this step

Compression Level

Compression Level

- Governs heuristic parameters related to pattern matching

Compression Level

- Governs heuristic parameters related to pattern matching
  - Length of ***nice*** matches that immediately eliminate further candidates

Compression Level

- Governs heuristic parameters related to pattern matching
  - Length of ***nice*** matches that immediately eliminate further candidates
  - Maximum number c of candidates to browse in any step

Compression Level

- Governs heuristic parameters related to pattern matching
  - Length of ***nice*** matches that immediately eliminate further candidates
  - Maximum number c of candidates to browse in any step
    - Reduced to $c/4$ if a ***good*** match is already available

Compression Level

- Governs heuristic parameters related to pattern matching
  - Length of ***nice*** matches that immediately eliminate further candidates
  - Maximum number c of candidates to browse in any step
    - Reduced to $c/4$ if a ***good*** match is already available
  - **Lazy** acceptance of references exceeding a certain length

Compression Level

- Governs heuristic parameters related to pattern matching
  - Length of ***nice*** matches that immediately eliminate further candidates
  - Maximum number c of candidates to browse in any step
    - Reduced to $c/4$ if a ***good*** match is already available
  - **Lazy** acceptance of references exceeding a certain length
- Level 0 indicates no compression at all (simply store input files)

Compression Level

- Governs heuristic parameters related to pattern matching
  - Length of ***nice*** matches that immediately eliminate further candidates
  - Maximum number c of candidates to browse in any step
    - Reduced to $c/4$ if a ***good*** match is already available
  - **Lazy** acceptance of references exceeding a certain length
- Level 0 indicates no compression at all (simply store input files)
- Levels 1-3 switch to a faster variant that redefines laziness
  - Enter i into hash chain only if longest match L exceeds ***lazy*** threshold

Compression Level

```
246  local config configuration_table[10] = {
247  /*      good lazy nice chain */
248  /* 0 */ {0,    0,  0,    0},  /* store only */
249  /* 1 */ {4,    4,  8,    4},  /* maximum speed, no lazy matches */
250  /* 2 */ {4,    5, 16,    8},
251  /* 3 */ {4,    6, 32,   32},
252
253  /* 4 */ {4,    4, 16,   16},  /* lazy matches */
254  /* 5 */ {8,   16, 32,   32},
255  /* 6 */ {8,   16, 128, 128},
256  /* 7 */ {8,   32, 128, 256},
257  /* 8 */ {32, 128, 258, 1024},
258  /* 9 */ {32, 258, 258, 4096}}; /* maximum compression */
```

Conclusions

Conclusions

- gzip‘s factorization by itself is not „good“

Conclusions

- gzip‘s factorization by itself is not „good“
  - Instead of emitting as few factors as possible, focus is on speed

Conclusions

- gzip‘s factorization by itself is not „good“
  - Instead of emitting as few factors as possible, focus is on speed
  - At the core lie naive string comparisons using simple word packing

Conclusions

- gzip‘s factorization by itself is not „good“
  - Instead of emitting as few factors as possible, focus is on speed
  - At the core lie naive string comparisons using simple word packing
  - Various heuristics help process the input quickly

Conclusions

- gzip‘s factorization by itself is not „good“
  - Instead of emitting as few factors as possible, focus is on speed
  - At the core lie naive string comparisons using simple word packing
  - Various heuristics help process the input quickly
- gzip‘s encoding takes care of the rest

Conclusions

- gzip‘s factorization by itself is not „good“
  - Instead of emitting as few factors as possible, focus is on speed
  - At the core lie naive string comparisons using simple word packing
  - Various heuristics help process the input quickly
- gzip‘s encoding takes care of the rest
  - Format (DEFLATE) specified in RFC 1951

Conclusions

- gzip‘s factorization by itself is not „good“
  - Instead of emitting as few factors as possible, focus is on speed
  - At the core lie naive string comparisons using simple word packing
  - Various heuristics help process the input quickly
- gzip‘s encoding takes care of the rest
  - Format (DEFLATE) specified in RFC 1951
  - References are encoded as (distance, length)-tuples

Conclusions

- gzip‘s factorization by itself is not „good“
  - Instead of emitting as few factors as possible, focus is on speed
  - At the core lie naive string comparisons using simple word packing
  - Various heuristics help process the input quickly
- gzip‘s encoding takes care of the rest
  - Format (DEFLATE) specified in RFC 1951
  - References are encoded as (distance, length)-tuples
  - Literals and lengths share a Huffman tree that additionally contains utility codes („end of block“ and further coding flags)

Conclusions

- gzip‘s factorization by itself is not „good“
  - Instead of emitting as few factors as possible, focus is on speed
  - At the core lie naive string comparisons using simple word packing
  - Various heuristics help process the input quickly
- gzip‘s encoding takes care of the rest
  - Format (DEFLATE) specified in RFC 1951
  - References are encoded as (distance, length)-tuples
  - Literals and lengths share a Huffman tree that additionally contains utility codes („end of block“ and further coding flags)
  - Encoding is done block-wise, each block receiving a new Huffman tree