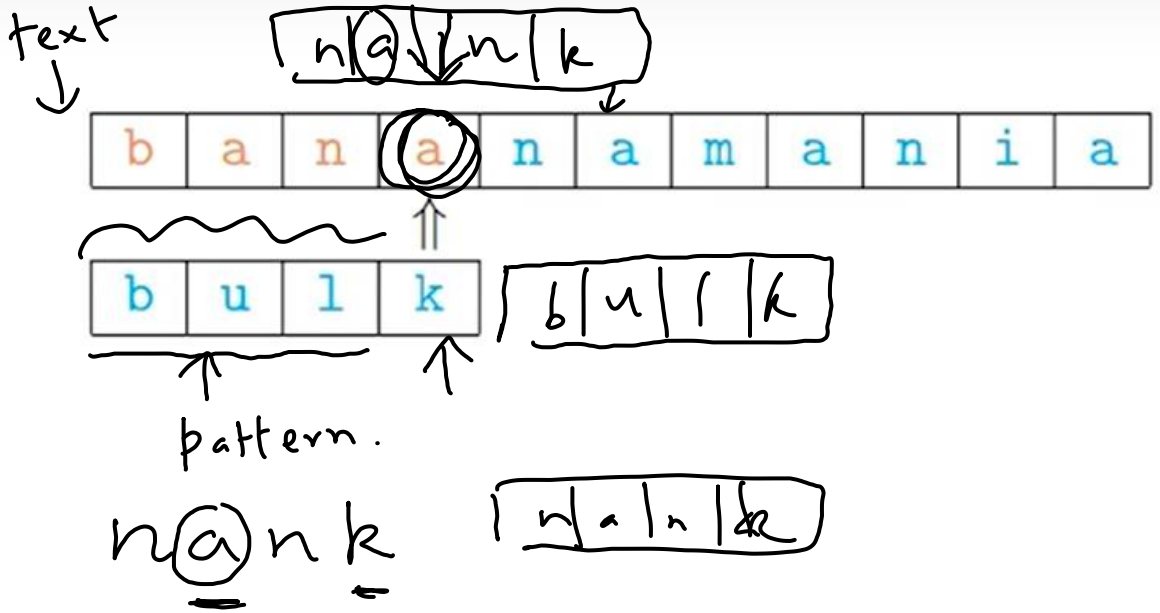
String Matching

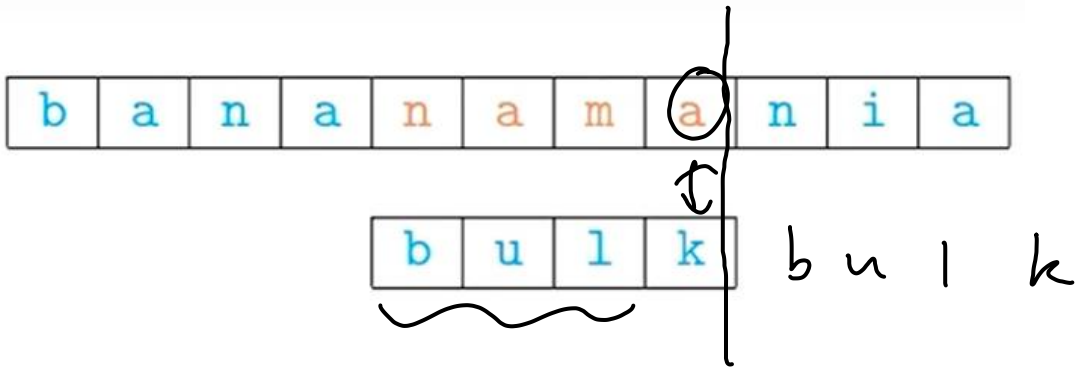
Boyer Moore - Intuition

- Text `t`, pattern `p` of of lengths n, m
- For each starting position `i` in `t`, compare `t[i:i+m]` with `p`
  - Scan `t[i:i+m]` right to left
- While matching, we find a letter in `t` that does not appear in `p`
  - `t = bananamania`, `p = bulk`

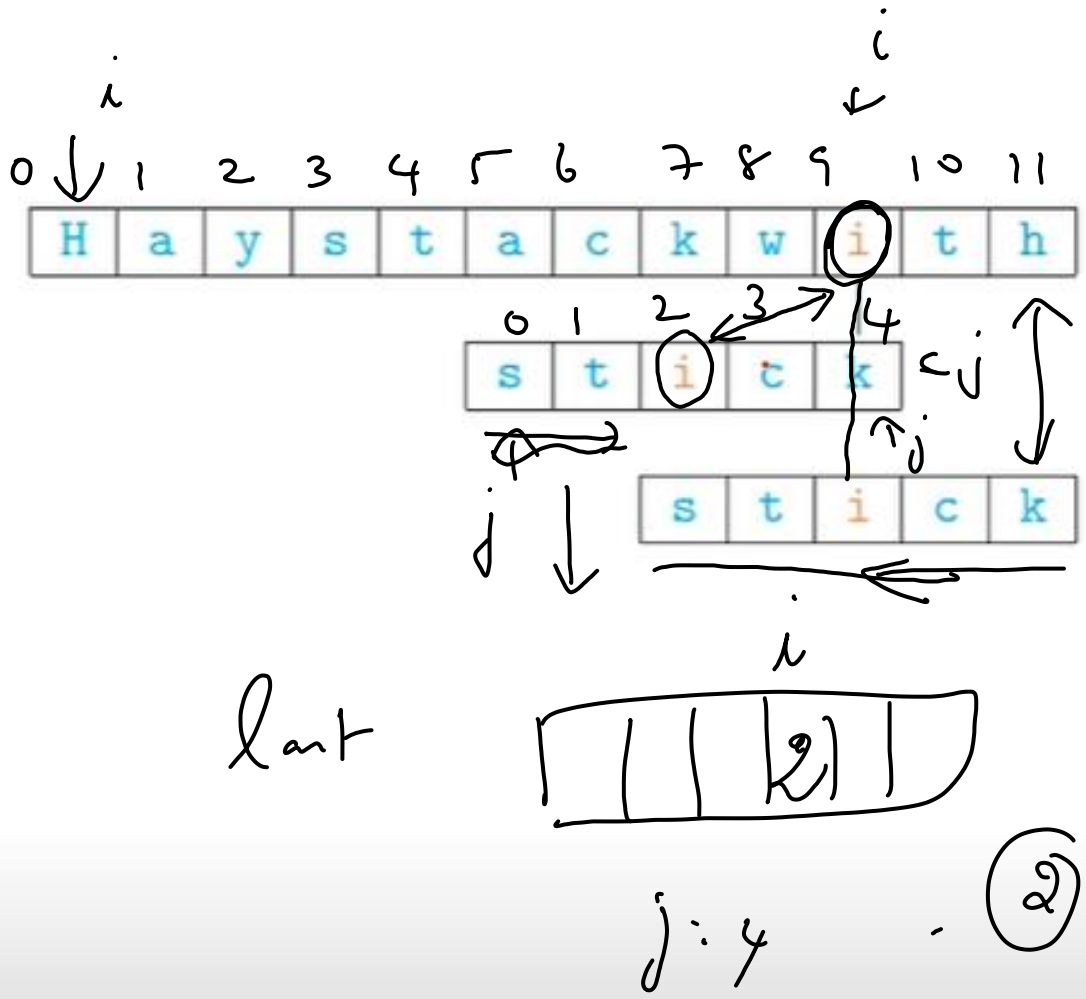


Boyer Moore - Intuition

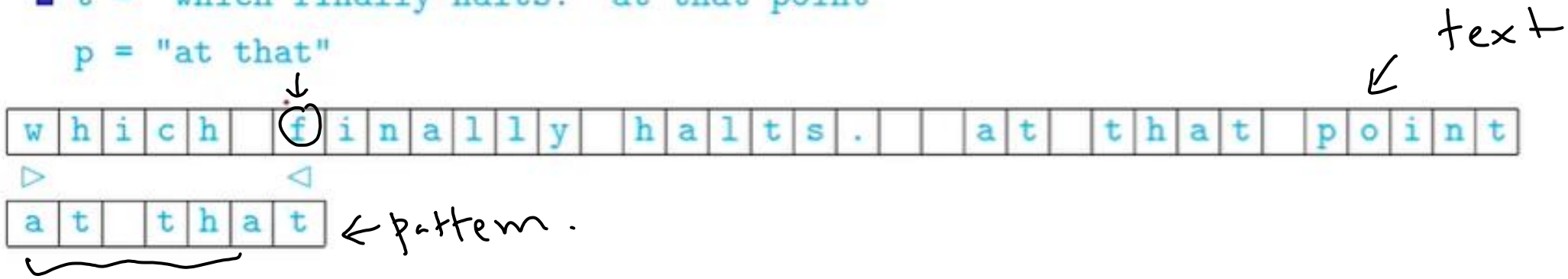
- Text `t`, pattern `p` of of lengths n, m
- For each starting position `i` in `t`, compare `t[i:i+m]` with `p`
  - Scan `t[i:i+m]` right to left
- While matching, we find a letter in `t` that does not appear in `p`
  - `t = bananamania`, `p = bulk`
- Shift the next scan to position after mismatched letter
- What if the mismatched letter does appear in `p`?



- Suppose `c = t[i+j] != p[j]`, but `c` does occur somewhere in `p[j]`
- Align rightmost occurrence of `c` in `p` with `t[i+j]`
- Scan this substring of `t` next
- Use a dictionary `last`
  - For each `c` in `p`, `last[c]` records right most position of `c` in `p`
  - Shift pattern by `j - last[c]`
- If `c` not in `p`, shift pattern by `j+1`

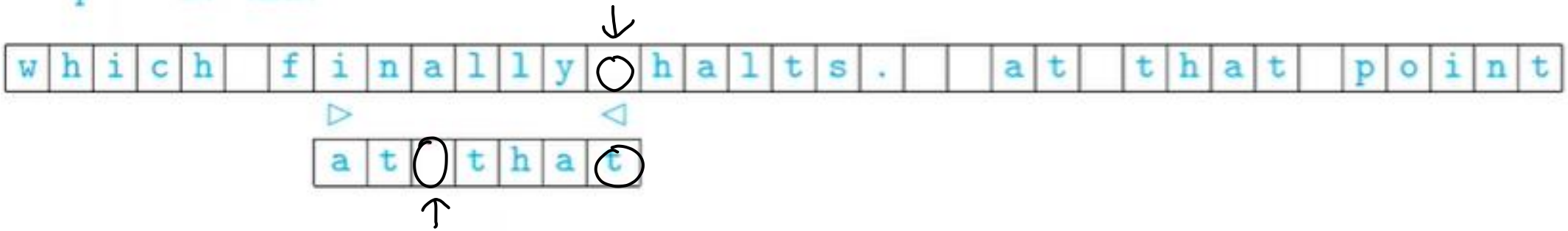


- t = "which finally halts. at that point"
  p = "at that"



- t[0:7] == "which f", "f" not in pattern, shift by 7, index 7

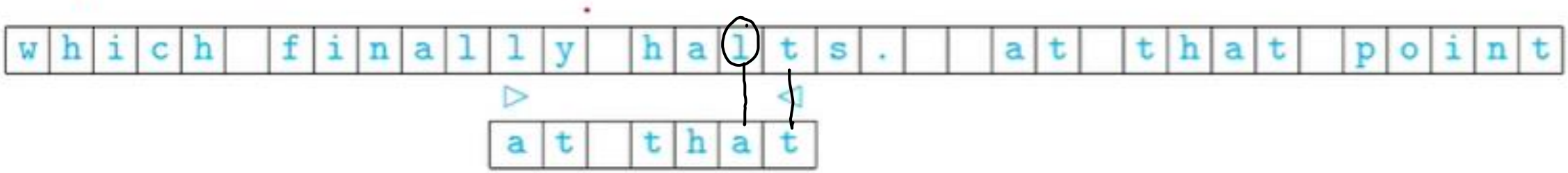
- t = "which finally halts. at that point"
  p = "at that"



- t[0:7] == "which f", "f" not in pattern, shift by 7, index 7
- t[7:14] == "inally ", " " in pattern, shift by 4 to align " ", index 11

- t = "which finally halts. at that point"

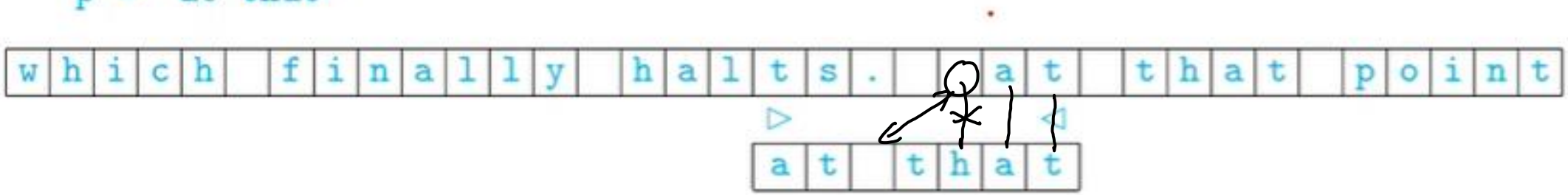
  p = "at that"



- t[0:7] == "which f", "f" not in pattern, shift by 7, index 7
- t[7:14] == "inally ", " " in pattern, shift by 4 to align " ", index 11

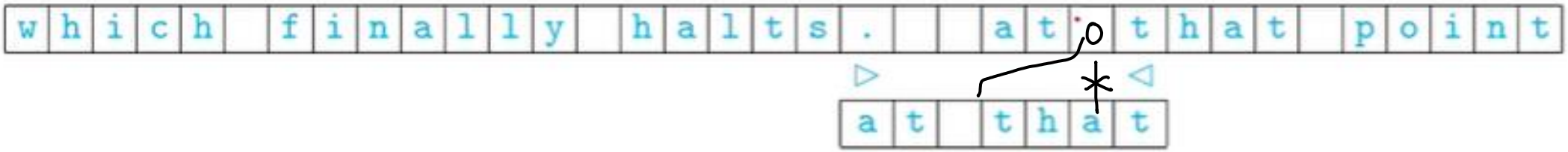
- t = "which finally halts. at that point"

  p = "at that"



- t[0:7] == "which f", "f" not in pattern, shift by 7, index 7
- t[7:14] == "inally ", " " in pattern, shift by 4 to align " ", index 11
- t[11:18] == "ly halt", "l" not in pattern, shift by 6, index 17

- t = "which finally halts. at that point"
  p = "at that"

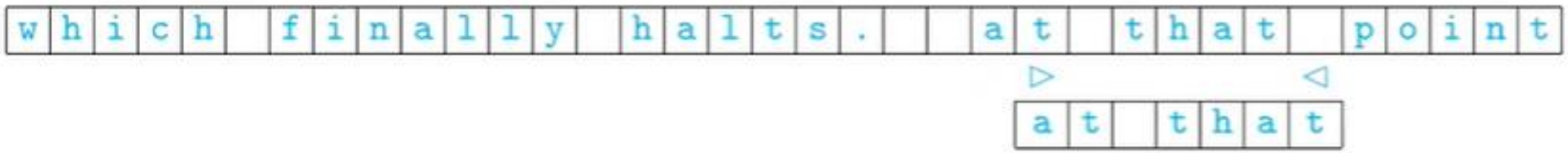


- t[0:7] == "which f", "f" not in pattern, shift by 7, index 7
- t[7:14] == "inally ", " " in pattern, shift by 4 to align " ", index 11
- t[11:18] == "ly halt", "l" not in pattern, shift by 6, index 17
- t[17:24] == "ts. at", " " in pattern, shift by 2, index 19
- t[19:26] == ". at t", " " in pattern, shift by 3, index 22

- t = "which finally halts. at that point"
  p = "at that"

- t[0:7] == "which f", "f" not in pattern, shift by 7, index 7
- t[7:14] == "inally ", " " in pattern, shift by 4 to align " ", index 11
- t[11:18] == "ly halt", "l" not in pattern, shift by 6, index 17
- t[17:24] == "ts. at", " " in pattern, shift by 2, index 19
- t[19:26] == ". at t", " " in pattern, shift by 3, index 22

- t = "which finally halts. at that point"
  p = "at that"



- t[0:7] == "which f", "f" not in pattern, shift by 7, index 7
- t[7:14] == "inally ", " " in pattern, shift by 4 to align " ", index 11
- t[11:18] == "ly halt", "l" not in pattern, shift by 6, index 17
- t[17:24] == "ts. at", " " in pattern, shift by 2, index 19
- t[19:26] == ". at t", " " in pattern, shift by 3, index 22
- t[22:29] == "at that", report match at index 22, shift by 1, index 23

- `t = "which finally halts.  at that point"`

  `p = "at that"`

- `t[0:7] == "which f"`, `"f"` not in pattern, shift by 7, index 7
- `t[7:14] == "inally "`, `" "` in pattern, shift by 4 to align `" "`, index 11
- `t[11:18] == "ly halt"`, `"l"` not in pattern, shift by 6, index 17
- `t[17:24] == "ts.  at"`, `" "` in pattern, shift by 2, index 19
- `t[19:26] == ".  at t"`, `" "` in pattern, shift by 3, index 22
- `t[22:29] == "at that"`, report match at index 22, shift by 1, index 23
- `t[23:30] == "t that "`, `" "` in pattern, shift by 4, index 27

- Initialize `last[c]` for each `c` in `p`
  - Single scan, rightmost value is recorded
- Nested loop, compare each segment `t[i:i+len(p)]` with `p`
- If `p` matches, record and shift by `1`
- We find a mismatch at `t[i+j]`
  - If `j > last[t[i+j]]`, shift by `j - last[t[i+j]]`
  - If `last[t[i+j]] > j`, shift by 1
    - Should not shift `p` to left!
  - If `t[i+j]` not in `p`, shift by `j+1`

```
def boyermoore(t,p):
  last = {}                          # Preprocess
  for i in range(len(p)):
    last[p[i]] = i

  poslist,i = [],0                   # Loop
  while i <= (len(t)-len(p)):
    matched,j = True,len(p)-1
    while j >=0 and matched:
      if t[i+j] != p[j]:
        matched = False
      j = j - 1
    if matched:
      poslist.append(i)
      i = i + 1
    else:
      j = j + 1
      if t[i+j] in last.keys():
        i = i + max(j-last[t[i+j]],1)
      else:
        i = i + j + 1
  return(poslist)
```

- Worst case remains $O(nm)$
  - `t = aaa...a, p = baaa`
- Without dictionary, computing `last` is a bottleneck, complexity is $O(|\Sigma|)$
- Boyer-Moore works well, in practice
  - "Sublinear"
  - Experimentally — English text, 5 character pattern, average number of comparisons is 0.24 per character
  - Performance improves as pattern length grows — more characters skipped
- Often used in practice — `grep` in Unix

Rabin-Karp

- Suppose $\Sigma = \{0, 1, \ldots, 9\}$
- Any string over Σ can be thought of as a number in base 10
- Pattern `p` is an m-digit number n_p
- Each substring of length m in the text `t` is again an m-digit number
- Scan `t` and compare the number n_b generated by each block of m letters with the pattern number n_p

Converting a string to a number

- Can convert a block `t[i:i+m]` to an integer n_i in one scan

```
num = 0
for j in range(m):
    num = 10*num + t[i+j]
```

- Computing n_i from `t[i:i+m]` for each block from scratch will take time $O(nm)$
- Instead
  - Subtract $10^{m-1} \cdot t[i-1]$ from n_{i-1} — drop leading digit
  - Multiply by 10 and add $t[i+m-1]$ to get n_i

$n_{i-1} \rightarrow n_i$

$t[i-1 \cdots i+m-1]$ $t[i : i+m]$

```
def rabinkarp(t,p):
  poslist = []

  numt,nump = 0,0
  for i in range(len(p)):
    numt = 10*numt + int(t[i])
    nump = 10*nump + int(p[i])

  if numt == nump:
    poslist.append(0)

  for i in range(1,len(t)-len(p)+1):
    numt = numt - int(t[i-1])*(10**(len(p)-1))
    numt = 10*numt + int(t[i+len(p)-1])
    if numt == nump:
      poslist.append(i)
  return(poslist)
```

- First convert `t[0:m]` to n_0 and `p` to n_p
- In the loop, incrementally convert n_{i-1} to n_i
- Whenever $n_i = n_p$ report a match

- For $\Sigma = \{a_1, a_2, \ldots, a_k\}$, treat each letter as a digit in base k
- Naively, repeat the previous algorithm in base k
  - Multiply/divide by k rather than 10
- In practice, for realistic k, the numbers are too large to work with directly
- Instead, do all computations and comparisons modulo a suitable prime q

- p = 31415, t = 2359023141526739921
- q = 13
- 31415 mod 13 = 7
- 23590 mod 13 = 8
  35902 mod 13 = 9
  ...
  31415 mod 13 = 7
  ...
  67399 mod 13 = 7
  ...
- False positives — must scan and validate each block that appears to match

- Preprocessing time is $O(m)$
  - To convert `t[0:m]`, `p` to numbers
- Worst case for general alphabets is $O(nm)$
  - Every block `t[i:i+m]` may have same remainder modulo q as the pattern `p`
  - Must validate each block explicitly, like brute force
- In practice number of spurious matches will be small
- If $|\Sigma|$ is small enough to not require modulo arithmetic, overall time is $O(n+m)$, or $O(n)$, since $m \ll n$
  - Also if we can choose q carefully to ensure $O(1)$ spurious matches