COMP26120: Algorithms and Imperative Programming

Lecture 8: Dictionary Look-up and String Matching

Ian Pratt-Hartmann

Room KB2.38: email: ipratt@cs.man.ac.uk

2015–16

Outline

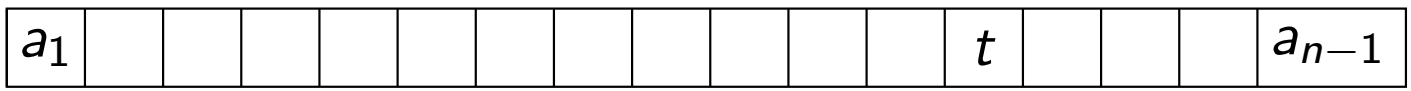
- Suppose `A` is an array or `int`s of length n.

- Consider the problem of determining whether a given `int`, say t, occurs in `A`.
- Well we could just search it ...

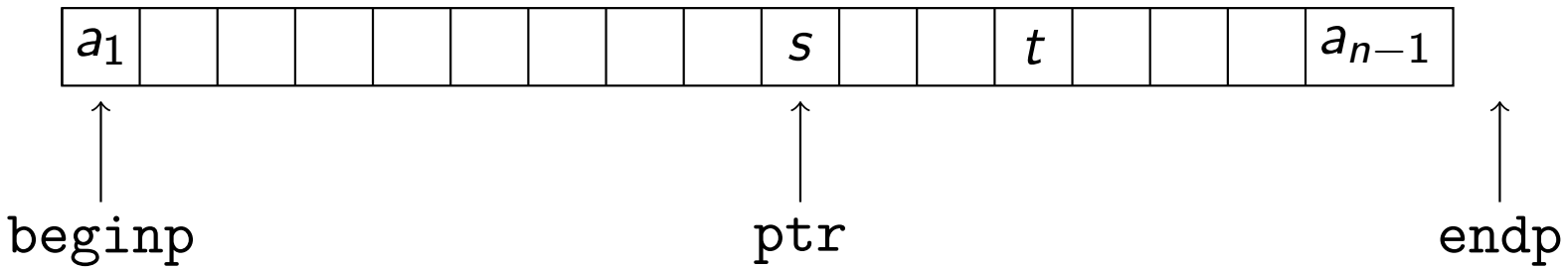
```
for(int i= 0; i < n; i++)
    if(a[i] == t)
      printf("Found it!);
printf("Does not compute!");
```

- But that wouldn't be a good thing to do too often.

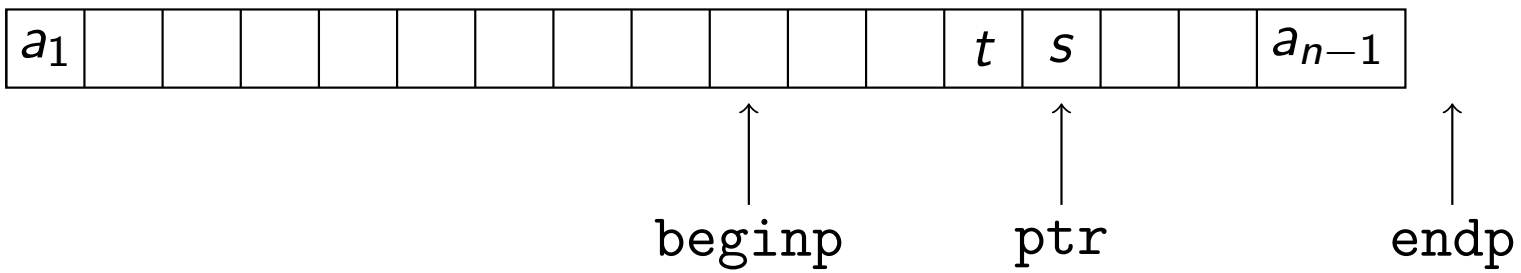
- Suppose the list is sorted as $a_0 \leq \cdots \leq a_{n-1}$



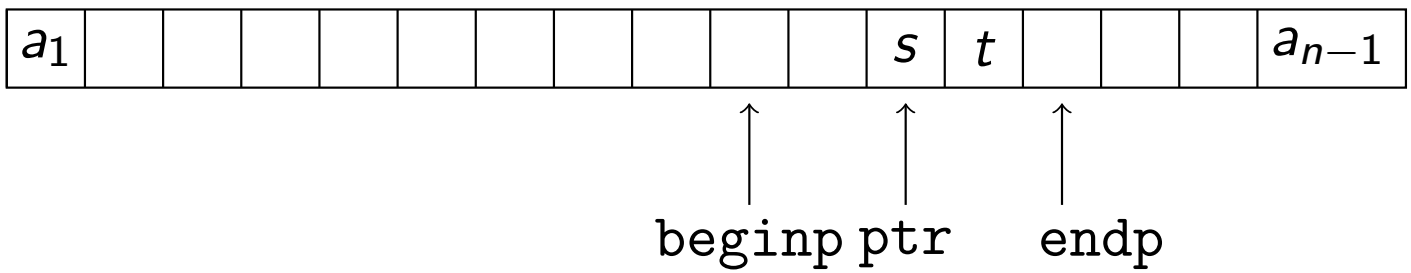
- Suppose the list is sorted as $a_0 \leq \cdots \leq a_{n-1}$



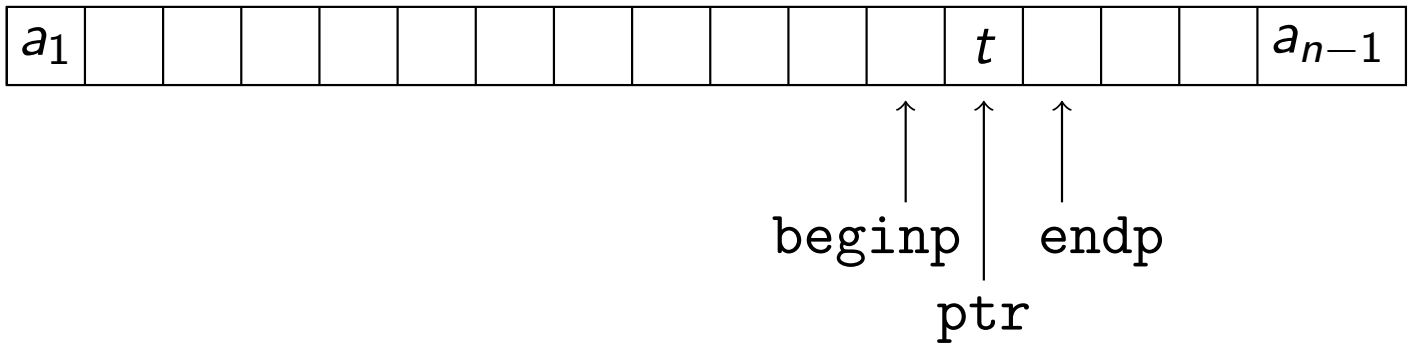
- Suppose the list is sorted as $a_0 \leq \cdots \leq a_{n-1}$



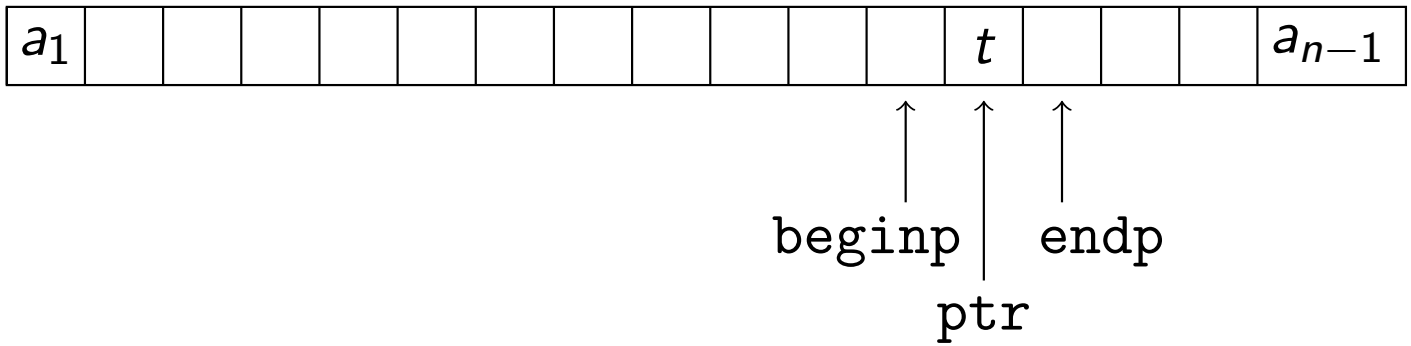
- Suppose the list is sorted as $a_0 \leq \cdots \leq a_{n-1}$



- Suppose the list is sorted as $a_0 \leq \cdots \leq a_{n-1}$



- Suppose the list is sorted as $a_0 \leq \cdots \leq a_{n-1}$



- Of course, this works not only for `ints`, but for any orderable type.

- Here is the algorithm where the data type is strings:

Algorithm DictLookup:
1: **Input:** a word t, a dictionary D of n words
2: $beginp \leftarrow 0$, $ptr \leftarrow n/2$, $endp \leftarrow n$
3: **while** $(beginp < endp)$ **do**
4: $val \leftarrow \text{strncmp}(t, D[ptr])$
5: **if** $val = 0$
6: print "word found !", exit
7: **if** $val > 0$
8: $beginp \leftarrow ptr + 1$
9: **else**
10: $endp \leftarrow ptr$
11: $ptr \leftarrow (beginp + endp)/2$

- Running time is $O(\log_2 n)$, where n is the number of elements in the array (assuming constant-time comparisons).

- Let's make a spell-checker:

Algorithm SpellChecker:
1: **input:** a text file of n words, a dictionary D of m words
2: read the text file and put words into an array, $word[]$
3: **for** $i \leftarrow 1$ to n **do**
4: $\quad found? \leftarrow \text{DictLookup}(word[i], D)$
5: $\quad$ **if** $\neg found?$
6: $\quad\quad$ print $word[i]$ "misspelled? [newline]"

- This algorithm uses the dictionary lookup from above.
- For an input of n words and a dictionary of size m, it will run in worst case $O(n \log_2 m)$ time.

Outline

- Suppose we are given some English text

  > *A blazing sun upon a fierce August day was no greater rarity in southern France then, than at any other time, before or since. Everything in Marseilles, and about Marseilles, had stared at the fervid sky, and been stared at in return, until a staring habit had become universal there.*

  and a search string, say "Marseilles".
- We would like to find all instances (or just the first instance) of the search string in the text.
- What's the best way?

- Suppose we are given some English text

  > *A blazing sun upon a fierce August day was no greater rarity in southern France then, than at any other time, before or since. Everything in Marseilles, and about Marseilles, had stared at the fervid sky, and been stared at in return, until a staring habit had become universal there.*

  and a search string, say "Marseilles".
- We would like to find all instances (or just the first instance) of the search string in the text.
- What's the best way?

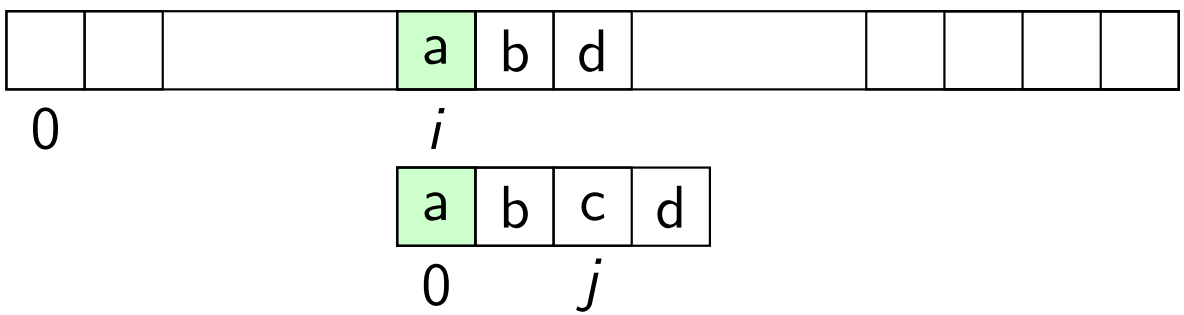
- As usual, we start by modelling the data:
  - let Σ be a finite non-empty set (the alphabet);
  - let $T = T[0], \ldots, T[n-1]$ be a string length n over a fixed alphabet Σ;
  - let $P = P[0], \ldots, P[m-1]$ be a string length m over Σ;
- We formalize the notion of an occurrence of one string in another:
  - string P occurs with shift i in string T if $P[j] = T[i+j]$ for all j $(0 \leq i < |P|)$.
- The we have the following problem

MATCHING
Given: strings T and P over some fixed alphabet Σ.
Return: smallest i such that P occurs in T with shift i;
-1 if P does not occur in T.

- Here is a really stupid algorithm

```
begin naiveMatch(T, P)
    for i = 0 to |T| − |P|
        j ← 0
        until j = |P| or T[i + j] ≠ P[j]
            j++
        if j = |P|
            return i
    return −1
end
```

- Graphically

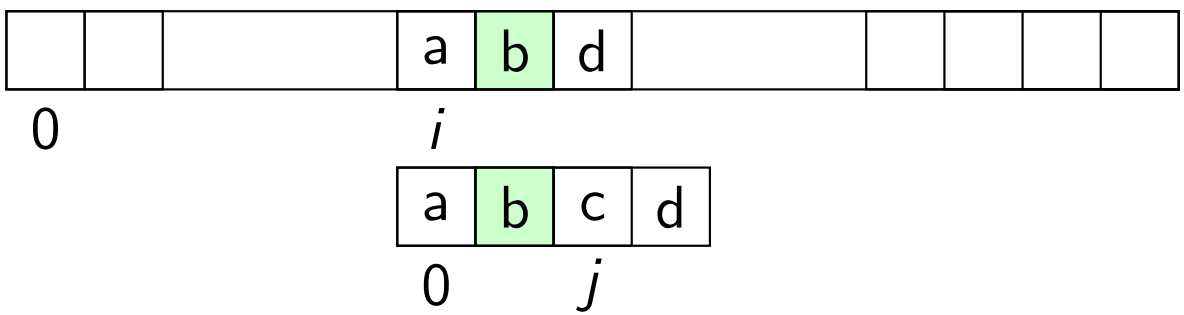


- Running time is $O(|T| \cdot |P|)$.

- Here is a really stupid algorithm

```
begin naiveMatch(T,P)
    for i = 0 to |T| − |P|
        j ← 0
        until j = |P| or T[i + j] ≠ P[j]
            j++
        if j = |P|
            return i
    return −1
end
```

- Graphically

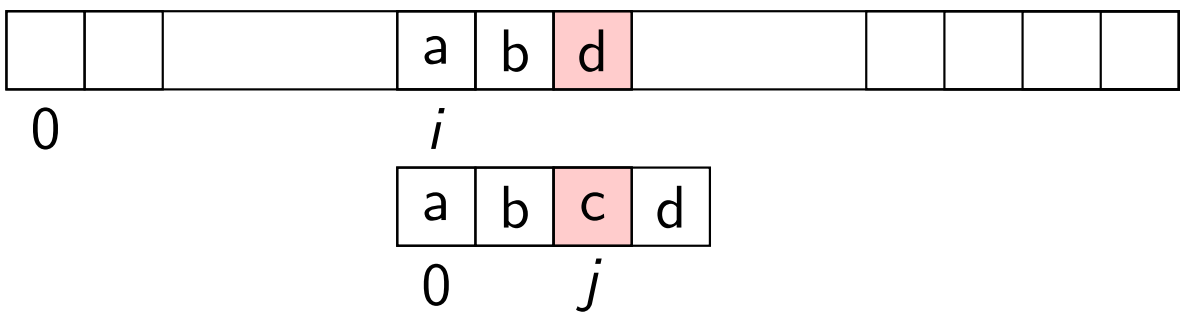


- Running time is $O(|T| \cdot |P|)$.

- Here is a really stupid algorithm

```
begin naiveMatch(T, P)
    for i = 0 to |T| − |P|
        j ← 0
        until j = |P| or T[i + j] ≠ P[j]
            j++
        if j = |P|
            return i
    return −1
end
```

- Graphically



- Running time is $O(|T| \cdot |P|)$.

Outline

- Let $n = |T|$ and $m = |P|$.
- Think of the elements of Σ as digits in a base-b numeral, where $b = |\Sigma|$.
- Then P is the number $P[0] \cdot b^{m-1} + \cdots + P[m-1] \cdot b^0$.
- Similarly, $T[i, \cdots, i+m-1]$ is $T[i] \cdot b^{m-1} + \cdots + T[i+m-1] \cdot b^0$.
- To calculate $T[i+1, \cdots, i+m]$ from $T[i, \cdots, i+m-1]$, write:

$$T[i+1, \cdots, i+m] = (T[i, \cdots, i+m-1] - T[i] \cdot b^{m-1}) \cdot b + T[i+m].$$

- These numbers can get a bit large.
- However, we can word modulo q, for some constant q (usually a prime) such that bq is about the size of a computer word.
- Of course, we have

$$T[i+1,\cdots,i+m] = (T[i,\cdots,i+m-1] - T[i]\cdot b^{m-1})\cdot b + T[i+m] \pmod{q}.$$

- If $T[i,\cdots,i+m-1] \neq P \bmod q$, then we know we do not have a match at shift i.
- If $T[i,\cdots,i+m-1] = P \bmod q$, then we simply check explicitly that $T[i,\cdots,i+m-1] = P$.

- The worst-case running time of this algorithm is also $O(|T| \cdot |P|)$.
- On average, however, it works must better:
  - A rough estimate of the probability of a spurious match is $1/q$, since this is the probability that a random number will take a given value modulo q. (Well, that's actually nonsense, but never mind.)
  - A reasonable estimate of the number of matches is $O(1)$, since patterns are basically rare.
- This leads to an expected performance of about $O(n + m + m(n/q))$
- Thus, expected running time will be about $O(n + m)$, since presumably $q > m$.

- Here is the algorithm

```
begin Rabin-Karp(T,S,q,b)
    I ← ∅
    m ← |P|
    t ← T[0]·b^{m-1} + ··· + T[m-1]·b^0  mod q
    p ← P[0]·b^{m-1} + ··· + P[m-1]·b^0  mod q
    i → 0
    while i ≤ |T| − m
        if p = t
            j ← 0
            while P[j] = T[i + j] and j < |P|
                j++
            if j = |P|
                I ← I ∪ {i}
        t ← (t − T[i]·b^{m-1})·b + T[i + m]  mod q
        i++
    return I
end
```