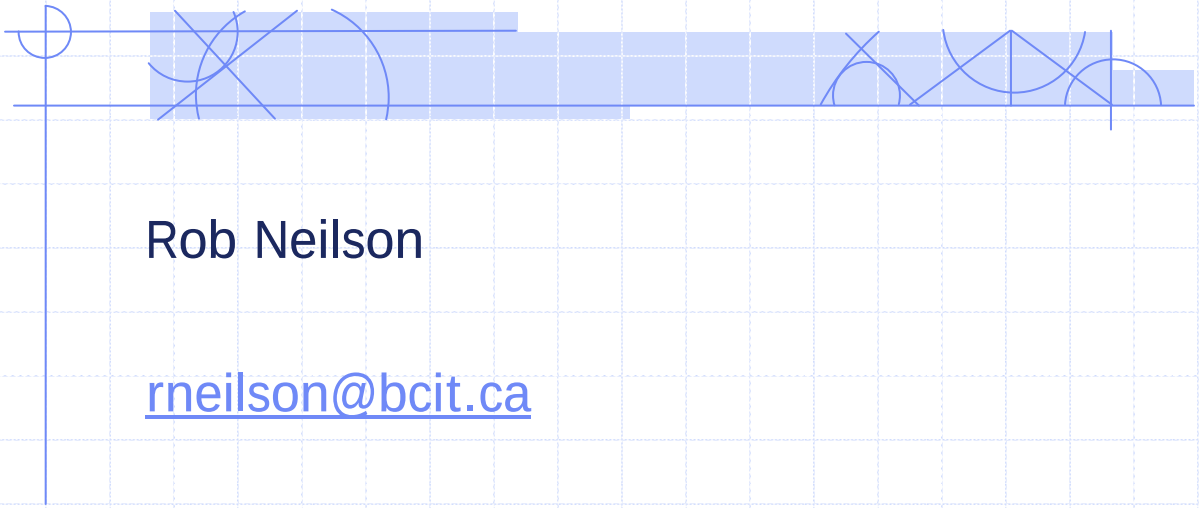
COMP 3760: Algorithm Analysis and Design

Lesson 12: Input Enhancement



Rob Neilson

rneilson@bcit.ca

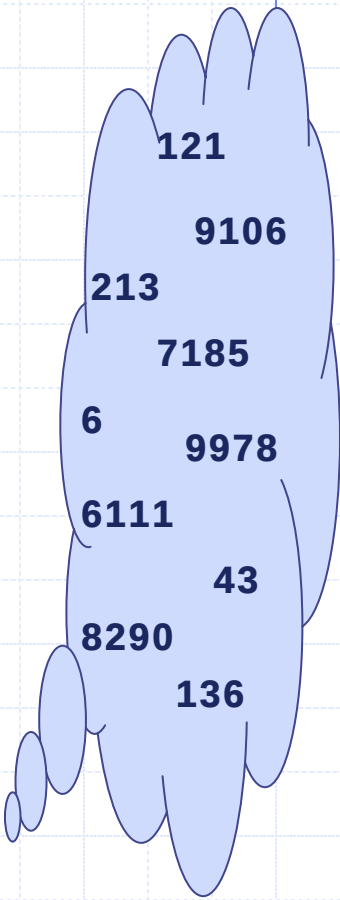
Administrivia

1. This weeks homework can be handed in to me (in person – paper only) during any lecture or lab this week; all sets need to do the homework
2. No labs next week; office hours instead; I will be in my office so you can get help as needed;
3. No new reading or assignments due next week; you should just study for the midterm
4. Today is a make up lecture; I missed it a couple weeks back; the corresponding textbook material is in chapters 7.1

Input Enhancement (Chps 6.1, 7.1)

- this is a technique in which we transform/organize the input data into a form that is easier to apply an algorithm to
- we do this all the time in our day-to-day lives, for example:
  - Assume I give you a small bag of coins and ask you to tell me how many nickels are in it. What would you do?
  - Chances are you would first sort out the nickels, and then count them
- this is simply an example of input enhancement
- Note: probably the most common enhancement done for algorithms is to pre-sort the input data. We do this quite frequently, for a couple of purposes:
  1. because it makes the data easier to process (ie: we have some knowledge of the order we will receive input in)
  2. it often allows us to select a more flexible data structure to store the data in within our algorithm

Input Enhancement: Example 1



- the textbook gives an algo, ComparisonCountingSort, that exploits the property of a set of input, namely:
  - it exploits the fact that:

> given a set N of n input items, then for any item t ∈ N, there is some subset S of N that contains only elements smaller than t

  - for example, if I give you 10 random numbers, and tell you:
    - one of the numbers is 136
    - there are exactly 3 of the 10 numbers that are smaller than 136
  - ***then you know for sure that when these 10 numbers are sorted, 136 must be in the 4th position***

posn	1	2	3	4	5	6	7	8	9	10
value	6	43	121	**136**	213	6111	7185	8290	9106	9978

3 numbers smaller than 136

6 numbers larger than 136

Example 1 continued …

- OK, so let's take it one step further … assume I give you a table that tells you exactly how many elements are smaller than each number in the input … then you can sort the numbers … right?
- for example (using the same input set as previous slide):

input	121	9106	213	7185	6	9978	6111	43	8290	136
Count	2	8	4	6	0	9	5	1	7	3

- so to create this array we could do...

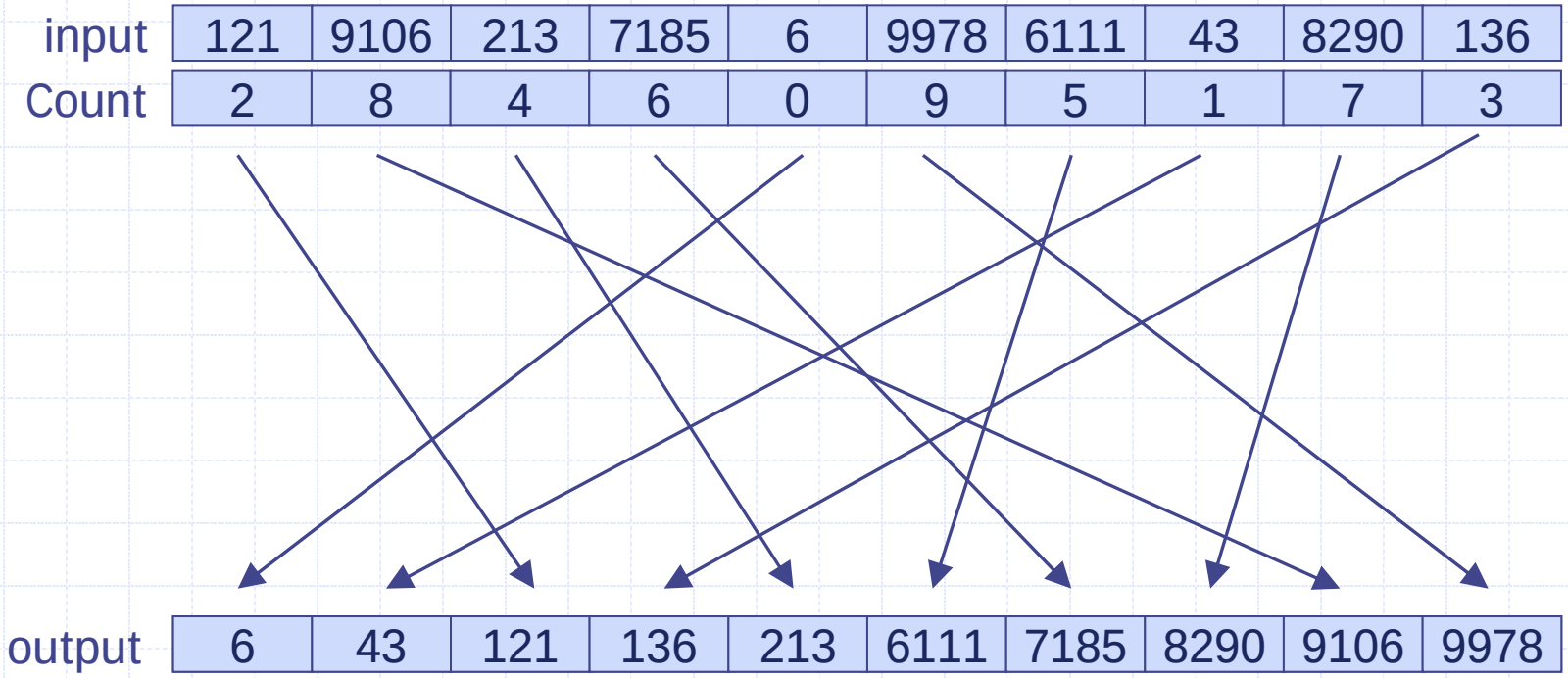
```
for i ← 0 to n-2
    for j ← i+1 to n-1
        if input[i] < input[j]
            Count[j]++
        else
            Count[i]++
```

note:
this "pre-structuring"
is $O(n^2)$

Example 1 continued …

- well now sorting is easy, as we just need to move each input element to it's corresponding position in an output array with

```
for i ← 0 to n-1
    output[Count[i]] ← input[i]
```



Analysis of Comparison Counting Sort

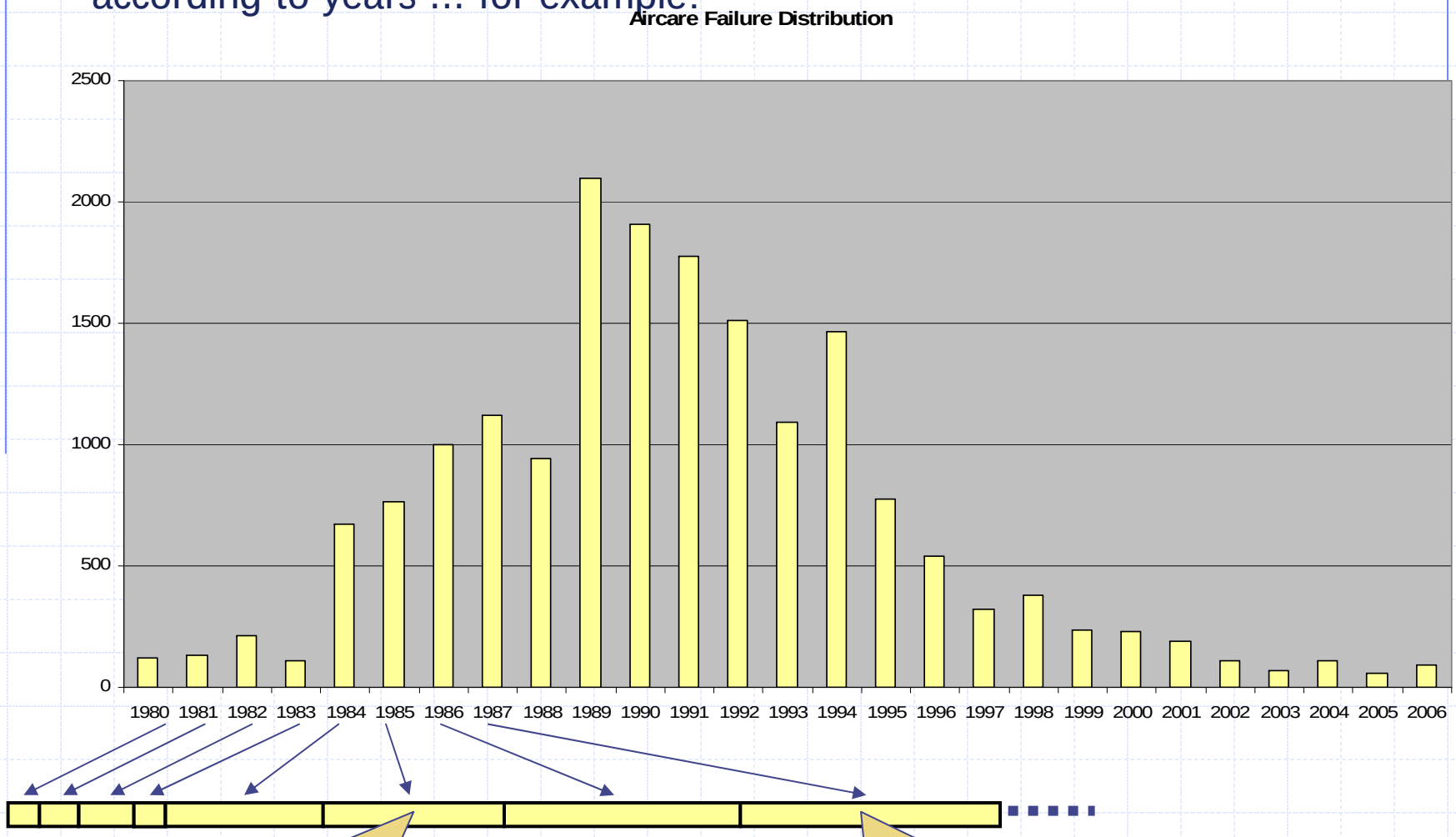
- Analysis:
  - the input enhancement part (counting) was $O(n^2)$
  - the actual sorting part was a single loop ... $O(n)$
  - so the overall efficiency is $O(n) + O(n^2) = O(n^2)$
  - not so good given that other sorts (mergesort, heapsort, quicksort etc) that are all O(nlogn)
- Does the sort have any other redeeming qualities ...
  - it isn't "in place" (ie: needs additional memory)
  - it only works for sets of distinct elements (ie: no duplicates)
  - is it 'stable'? ... well ... this doesn't really apply as there are no duplicates (recall that a stable sorting algorithm maintains the order of duplicate items)

Distribution Counting

- let's look at a more practical example: DistributionSort
  - this algo works for any set of input where the elements of the set are drawn from a ***limited range of inputs***
  - for example, we might have aircare records of all the cars that failed on their first attempt, and for each car we have the year it was produced
  - let us assume that we need to sort the records by vehicle year
  - since aircare only applies to vehicles between 1980 and 2006, we have a ***limited range of inputs*** (ie: we have only 26 possible values)
  - note the we can still have (potentially) 10's of thousands of cars to sort ... it is just that there are only 26 years (the sort key)

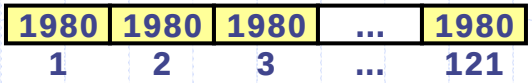
Distribution Sort cont …

- we can sort by exploiting the fact that the input values will be distributed according to years … for example:

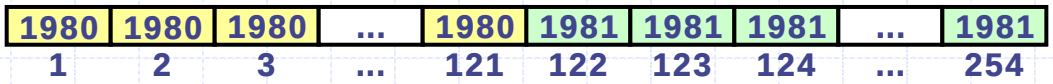


So how exactly did we get them sorted …

- the first number in the range to be sorted was “1980”, and there were 121 occurrences of this number (we don’t have the full source data, but I am taking a guess from the graph)
- these 121 occurrences must be the first entries in the sorted list …



- the next value was 1981, and there are approx 133 occurrences, so they come next in the sorted list …



- this process continues until all the values are sorted

The Algorithm ....

- to how to devise an algorithm to sort like this?
- First:
  - assume we have the input in an array A[0..n-1]
  - assume also that the input contains only values in the range from *lowVal* to *highVal*
    - *eg: in the prev example, 1980 is lowVal and 2006 is highVal*
  - assume that the output will be returned in a new array S[0..n-1]
  - finally, for our algorithm to work we are going to require some structure to contain the frequency and distribution count for each value
    - this will be an array that is big enough to contain the count for each legal value (ie: it could be smaller or larger than the size of A[]
    - we will use an array D[0..(highVal-lowVal)]

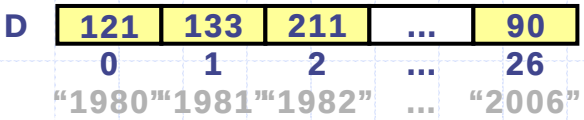
The Algorithm …. (cont)

To create the algorithm, we require ...

1. some way to count the frequencies (# times a value occurs in the input data)
   - we can do this simply by making one pass through the array, and counting the number of occurrences of each value

```
for i ← 0 to n-1
    D[A[i]-lowVal]← D[A[i]-lowVal] + 1  // incr freq count
```

   - the result of this is a count of the number of occurrences of each value



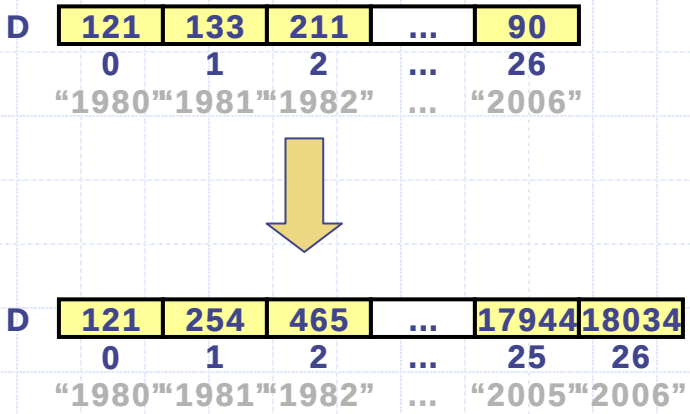
Notes:

- implementation of this would be a little bit easier in a language that supports associative arrays
- alternatively, in a language like Java, one might choose to use Map object (such as HashMap in the Java Collections Classes

The Algorithm …. (cont)

2. some way to convert the count into the starting or ending position of each value in the output array
   - we can do this by iterating over D[], and changing the frequency to the ending position (in the output) for each value

```
for j ← 1 to highVal-lowVal
    D[j] ← D[j-1] + D[j]
```



The Algorithm .... (cont)

3. some way to re-organize the input values according to their distribution
   - now we go through the input A[], from right to left
   - for each value V in the input we:
     - move it to the posn (in S[]) given by the current distribution value D[V]
       - ie: D[V] tells us the last place in the output that will get value V
     - then we decrement D[V], so that the next occurrence of V can be placed in the next place for a V in the output

```
for i ← n-1 downto 0
    j ← A[i] – lowVal      // find posn of ith value in D[]
    S[D[j]-lowVal] ← A[i] // put ith value in correct posn
    D[j] ← D[j] – 1        // decrement the dist count
```

Analysis

- the complete algorithm is given in your textbook, along with a complete working example

- the first part of the algorithm makes exactly one pass through the input
  - ie, it is O(n)
- the second part of the algorithm makes one pass through the D[] array
  - ie, it is O(highVal-lowVal)
- the last part of the algo makes exactly one pass through the input
  - ie, it is also O(n)
- this means the efficiency (worst case) is
  - O(n) + O(highVal-lowVal) + O(n)

- so, as long as the range of valid input values is roughly less than or equal to the number of input values (n), the algorithm is O(n)

this is very good efficiency

Input Enhancement in String Matching

- previously we considered a brute force approach to string matching
- it compared each character in the search pattern P with each posn of the input text T, giving us an O(nm) algorithm (where n is the size of the T and m is the size of P)

How can we improve this by using the concept of input enhancement?

- the key observation is that each time we have a "mismatch" (ie: a pattern char doesn't match the corresponding text char), we may be able to ***shift more than one character*** before starting to compare again

String Matching: Key Observation

- the example from the textbook uses the pattern BARBER

  Note: the examples in the book are comparing the chars from right to left (ie: we first compare R, then E, then B etc)

- assume we are comparing the last R with some character in the text, and it causes a mismatch, for example

```
T = [ ... IAMABARBIENOTABARBERNAMEDJOE ... ]
P:            BARBER
```

- in this case the brute force algo would shift by one posn, ie:

```
T = [ ... IAMABARBIENOTABARBERNAMEDJOE ... ]
P:            →BARBER
```

- but we know there is no "I" in BARBER, so really we should shift the pattern all the way past the "I", for example

```
T = [ ... IAMABARBIENOTABARBERNAMEDJOE ... ]
P:            →→→→→→→BARBER
```

String Matching: Input Enhancement Cases

- it turns out that there are only 4 possibly cases that can occur when we encounter a mismatch:

  1. the text char being compared (c) is not in the pattern at all
  2. the text char being compared does exist in the pattern but not in the last position
  3. the text char being compared is the last char in the pattern, and occurs only once in the pattern
  4. the text char being compared is the last char in the pattern, and occurs multiple times in the pattern

- in cases 1 and 3, we can shift the pattern by its entire length
- in cases 2 and 4, we can shift the pattern such that the rightmost occurrence of (c) is aligned with the (c) that is being compared in the text

Note: this is illustrated in detail on page 256 of your textbook

The Strategy

- so how to use this observation for input enhancement?
- Strategy:
  - we are going to create a “shift table”. It will have one entry for each possible value in the *input alphabet* (the *input alphabet* is the set of valid characters in the input text and/or pattern)
  - our shift table will indicate the number of positions to shift the pattern when there is a mismatch between a char c in the text and a different char x in the pattern
    - ie: we will do a lookup using c as a key in the shift table, and shift by the number of positions given by Table[c]

The Shift Table

- How to construct the shift table:
  - it will have a size equal to the number of elements in the input alphabet (so we have to know this in advance!)
  - we initialize each entry in the table to the size of the pattern (ie: the maximum shift)
  - then for each char c in the pattern we set the shift to the distance from the last occurrence of c to the end of the pattern
- Example:
  - assume our alphabet is {A B C D E F G H I J}
  - assume our pattern is IDIGDAB (m=7)
  - then our shift table will be ...

Using the shift table …

- we compare the chars in the pattern to the chars in the text, from right to left
- when there is a mismatch we do a lookup in the shift table, using the character in the *text* that is aligned with the rightmost char in the pattern
- we shift by the number of posn's indicated in the shift table

Example:

- assume the pattern is IDIGDAB as on prev slide
- assume the text is IBBAGHJCDBADABCCIDIJEFADGJHIACDDD

```
IBBAGHJCDBADABCCIDIJEFADGJHIACDDD
     IDIGDAB
```

- in this case we lookup **table["D"]**, which returns **2**, so we shift by 2 …

```
IBBAGHJCDBADABCCIDIJIDIGDABHIACDDD
       →→IDIGDAB
```

Continuing the example …

Table

1	7	7	2	7	7	3	7	4	7
0	1	2	3	4	5	6	7	8	9
"A"	"B"	"C"	"D"	"E"	"F"	"G"	"H"	"I"	"J"

```
IBBAGHJCDBADABCCIDIJIDIGDABHIACDDD
      IDIGDAB
```

- in this case the B, then A, then D all match, but the A/G mismatch
- so, we lookup **table["B"]**, which returns **7**, so we shift by 7 …
- note that the lookup is against the last aligned char in the text (ie: the char shown in **GREEN** in the above text
- this shift results in …

```
IBBAGHJCDBADABCCIDIJIDIGDABHIACDDD
      →→→→→→→IDIGDAB
```

- so, we lookup **table["I"]**, which returns **4**, so we shift by 4 (ie: line up the I's)

```
IBBAGHJCDBADABCCIDIJIDIGDABHIACDDD
            →→→→IDIGDAB
```

Continuing the example (2) …

Table	1	7	7	2	7	7	3	7	4	7
	0	1	2	3	4	5	6	7	8	9
	"A"	"B"	"C"	"D"	"E"	"F"	"G"	"H"	"I"	"J"

```
IBBAGHJCDBADABCCIDIJIDIGDABHIACDDD
                    IDIGDAB
```

- we lookup **table["D"]**, which returns **2**, so we shift by 2 …

```
IBBAGHJCDBADABCCIDIJIDIGDABHIACDDD
                      →→IDIGDAB
```

- and now all chars match, so the algorithm returns 20 (the starting posn of the pattern in the text)

Note: the algorithm is spelled out in detail in your textbook. It is Horspool's algorithm.

The End