COMP20007 Workshop Week 11

LAB	**Preparation:** - *have draft papers and pen ready* - ready to work with assignment 2 Counting & Radix sort: Questions 11.1-11.3 Horspool's Algorithm: Questions 11.4-11.6 Assignment 2: Q&A Assignment 2 Revision on demands: complexity, recurrences, master theorem

Counting Sort

Simple Distribution Sort = Counting Sort

Conditions:

- keys are integers in a small range (small in comparison with n), for example: array of positive integers, each ≤ 2:

```
input array: {0,1,2,0,0,1,2,1,1,0,0,0}
```

freq(0) =6 freq(1)= 4 freq(2)=2

Sorted array: { 0,0,0,0,0,0, 1,1,1,1, 2, 2 }

keys 0 start from index 0

keys 1 starts from index **6**

6 = freq(0)
= freq(**<**1)

keys 2 starts from index 10

10 = freq(0 & 1)
= freq(**<2**)

Counting Sort for sorting array A[0..n-1]

Input: `A[0..n-1]` where
`0 ≤ A[i] ≤ k`, and
`k` is small (`k ≪ n`)

Output: `B[0..n-1]` which is the sorted version of `A[]`

Step 1: build array `C[0..k]` such that `C[i]`= starting index of keys `i`, by:

- First, `C[i+1]`← freq(**`i`**)
- Then accumulate:

```
for i := 1 to k do
    C[i] := C[i-1] + C[i]
```

Step 2: scan `A[]` again and copy to `B`.
For `A[j]` :

```
x := A[j]
B[  C[x] ] := x
C[x]= C[x]+1
```

`A[0..11]= {2,0,1,0,3,1,2,1,1,0,0,0}`
`k= 3`

`C[]= table of frequencies`

idx	0	1	2	3	4=k+1
		5	4	2	1

`C →`

0	5	9	11	

`A[]= {2,0,1,0,3,1,2,1,1,0,0,0}`

`B:`

- complexity=? is stable? is in-place?
- What if: `min ≤ A[i] ≤ max & max-min is small`

Counting Sort

- Can be applied when A[i] in range min..max, where k= max-min+1 is small
- Time complexity: $\Theta(n+k)$, or $\Theta(n)$ if k could be considered as a small constant
- In-place: NO additional memory: $\Theta(n+k)$, or $\Theta(n)$ if k small
- Stable: YES

Bucket Sort:

- counting sort is a special case of bucket sort, where k is the number of buckets
  - gather keys into buckets, 1 bucket for each distinct value of keys
  - concatenate the buckets (in order of key values)
- general bucket sort
  - gather keys into K≤k buckets
  - sort each bucket using a stable auxiliary sort
  - concatenate the sorted buckets

Radix Sort

Applied when all keys can be represented as same-size strings over a small alphabet σ. Examples:

{1, 12, 7, 10, 6, 9, 8, 3}

→ {0001, 1100, 0111, 1010, 0110, 1001, 1000, 0011} σ= {0,1}

{1,22,17,167,26,19,28,173,…}

→ {001, 022, 017, 167, 026, 019, 028, 173,…} σ= {0,1,…,9}

→ {01, 16, 11, A7, 1A, 13, 1C, AD…} σ= {0,1,…,9,A,B,..F}

Radix Sort:

From **rightmost to leftmost position** of strings: sort the string by that position by:

- Put items into buckets defined by the symbol at that position
- Concatenate (join) buckets in increasing order of symbols

Complexity: n * string_size

Q11.1 - Counting Sort: Use counting sort to sort the following array of characters:

[a, b, a, a, c, d, a, a, f, c, b]

How much space is required if the array has n characters and our alphabet has k possible letters.

Q11.2 - Radix Sort: Use radix sort to sort the following strings:

`abc bab cba ccc bbb aac abb bac bcc cab aba`

As a reminder radix sort works on strings of length k by doing k passes of some other (stable) sorting algorithm, each pass sorting by the next most significant element in the string. For example in this case you would first sort by the 3rd character, then the 2nd character and then the 1st character.

Q11.3: Which property is required to use counting sort to sort an array of tuples by only the first element, leaving the original order for tuples with the same first element. For example the input may be:

(8, campbell), (6, tal), (3, keir), . . . (6, gus), (0, nick), (8, tom)

Discuss how you would ensure that counting sort satisfies this property. Can you achieve this using only arrays? How about using auxiliarry linked data structures?

Q11.1 - Counting Sort: Use counting sort to sort the following array of characters:

[a, b, a, a, c, d, a, a, f, c, b]

How much space is required if the array has n characters and our alphabet has k possible letters.

Your Solution:

Q11.2 - Radix Sort: Use radix sort to sort the following strings:

```
abc bab cba ccc bbb aac abb bac bcc cab aba
```

As a reminder radix sort works on strings of length k by doing k passes of some other (stable) sorting algorithm, each pass sorting by the next most significant element in the string. For example in this case you would first sort by the 3rd character, then the 2nd character and then the 1st character.

Your Solution:

String Searching

Input:

- A (long) text `T[0..n-1]`. Example: `T`= "SHE SELLS SEA SHELLS", with `n`=20
- A (short) pattern `P[0..m-1]`. Example: `P`="HELL", `m`=4.

Output:

- index `i` such that `T[i..i+m-1]=P[0..m-1],` or NOTFOUND

Algorithms:

- Naïve: brute force, complexity O(nm) (max= (n-m+1)*m character comparison):
  - shift pattern left to right on the text, 1 position each time
  - compare pattern with text from left to right
- Horspool's: also O(mn) but practically fast:
  - shift pattern left to right on the text, *at least* 1 position each time
  - compare pattern with text **from right to left**

How to run Horspool's *manually*

a	b	a	b	y	a	y	b		a	a	b	a	b	c	a

a	b	a	b	c

SHIFT m because
y not in P

mismatch found at the first comparison
no matter where mismatch happens, the shift is totally decided by the rightmost examined char of T, **y**

Shift until having the first match of character on P
with that rightmost **y** (here, no match found)

a	b	a	b	y	a	y	b		a	a	b	a	b	c	a

a	b	a	b	c

The number of positions to shift totally depend on the pattern `P` and can be easily pre-computed!

Horspool's Algorithm Review

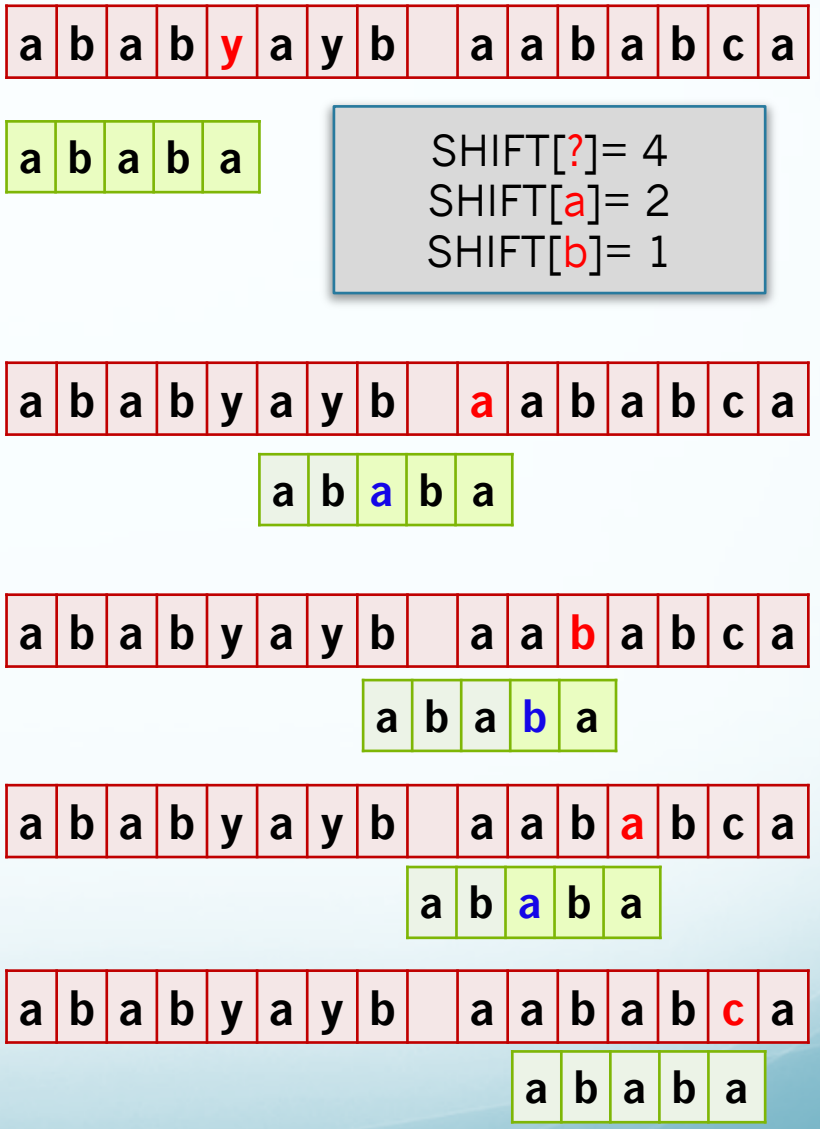
The task: Searching for a pattern `P` (such as "ababa" that has length m=5) in a text `T` (such as "ababyayb aababca", having length n=16).

***Stage 1**: build `SHIFT[x]` for all `x`, by:*

1. set `SHIFT[x]= m` for all `x`, then
2. for each `x` in `P`, *except for the last one*: `SHIFT(x)`= distance from the last appearance of `x` to the end of `P`

***Stage 2:** searching, by first set `i=m-1`, then*

1. set `c= T[i]`, align `P` with `T` so that `P[m-1]` aligned with `T[i];`
2. compare characters *backwardly* from the last character of `P` until the start or until finding a mismatch:
3. if no mismatch found: return solution which is `i-m+1`
4. otherwise, set `i= i+ SHIFT[c]`, back to step 1



Horspool's Algorithm

Q11.4: Use Horspool's algorithm to search for the pattern `GORE` in the string `ALGORITHM`

Q11.5: How many character comparisons will be made by Hor-spool's algorithm in searching for each of the following patterns it the binary text of one million zeros?

(a) 01001 (b) 00010 (c) 01111

Q11.6 - Horspool's Worst-Case Time Complextity: Using Horspool's method to search in a text of length n for a pattern of length m, what does a worst-case example look like?

Horspool's Algorithm – Your Solutions

Q11.4: Use Horspool's algorithm to search for the pattern `GORE` in the string `ALGORITHM`

Q11.5: How many character comparisons will be made by Hor-spool's algorithm in searching for each of the following patterns it the binary text of one million zeros?

(a) 01001 (b) 00010 (c) 01111

Q11.7: Review on recurrences

- done last week
- ask questions later (if any)

Assignment 2: Q&A (Part 1, Part 2, Part 3)

Lab: Assignment 2

- Make sure that you understand the tasks of A2, know what to do, ask questions if in doubt.
- Do assignment 2, further questions, and/or
- Review complexity, recurrences, and other parts.

A2.P3 and A2.P2: Questions?

Pseudocode in A2.P2:

For simplicity you can suppose that a tree node include

- array `child[0..3]` of children
- array `val[1..3]` of keys

(but if you do so, you should clearly state the supposition).

Pseudocode should be clear, but also be concise (ie. with no/minimal redundancy).

Other questions?

A2.P1: more on C

C bitwise operators: AND `&`, OR `|`, XOR `^`, … operate on integers or unsigned integers at binary level, ie. by processing each bit of the binary representations.

With unsigned integers there are less confusion. Example of unsigned int datatypes in C:

- `uint64_t` : 8-byte non-negative int, convenient for very big numbers (up to $2^{64}-1$)
- `uint8_t` : 1-byte non-negative int. A text of `N` characters can be declared as

```
uint8_t text[N];
```

and represented as a pair `(uint8_t *text, unit64_t N)` [no \0 at the end].

Exclusive OR `^` and some important properties:

```
a ^ a  =  0
a ^ b  =  b ^ a
(a ^ b) ^ c  =  a ^ (b ^ c)
```

So:

```
c  =  s ^ t   →   t  =  s ^ c
```

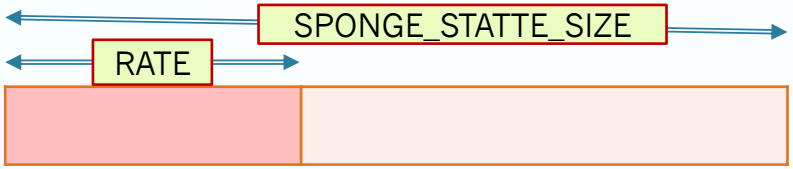
a	b	a^b	example
0	0	0	10001101 151
0	1	1	10001010 138
1	0	1	---------- -----
1	1	0	00000111 7

Other C facilities: not crucial, but it would be better if you `google` or use `man` to know about `memset` and `memcpy`.

A2.P1: Sponge

A sponge is defined by its state, which is array of SPONGE_STATTE_SIZE bytes:

uint8_t state[SPONGE_STATE_SIZE];



init: zeroing the whole `state`

read: copying the first `num` bytes of state into a buffer `uint8_t *dest`

write: using the first `num` bytes of the text `uint8_t *src` and

- copying it into `state[0..num-1]` if the flag `bw_xor` is `false`
- `XOR` it into `state[0..num-1]` if the flag `bw_xor` is `true`

demarcate: XOR a byte `uint8_t delimiter` into `state[i]`

permute: using function *permutation_384* to replace state with a sequence of `SPONGE_STATTE_SIZE` bytes in a deterministic and reversible way

A2.P1: Hashing

// Hashes an input text `T` of `len` bytes to produce the hash value `H` of `H_len` bytes.

```
void hash(uint8_t *H, uint64_t H_len, uint8_t const *T, uint64_t len);
```

Step 0: start with a zeroed sponge `s`

Step 1 – Absorb `T` into `s`:

divide `T` into blocks T_0, T_1, T_2, … T_k of `RATE` bytes (T_k has $0 \le r \le RATE-1$ bytes)

for each block T_i in that order:

- absorb T_i into `s` using `sponge_write` with `bw_xor`= `true`
- permute `s` (note: special treatment for the last block)

Step 2 – Demarcation: do 2 specific demarcations

Step 3 – Squeezing: getting value for H block-by-block

while (`H` is not full)

- read `RATE` bytes from `s` and append it to `H`
- permute `s`

A2.P1: MAC

// Creates authentication tag of size tag_len bytes from key of size CRYPTO_KEY_SIZE bytes and an input text T msg len bytes.

```
void mac(uint8_t *tag, uint64_t tag_len, uint8_t const *key,
            uint8_t const *T, uint64_t len);
void hash(uint8_t *H, uint64_t H_len, uint8_t const *T, uint64_t len);
```

Step 0:

start with a zeroed sponge `s`

absorb key (of `CRYPTO_KEY_SIZE` bytes) into `s`

Step 1,2,3:

same as hashing,

but in Step 3, squeeze into `tag` (of `tag_len` bytes) instead of `H`

A2.P1: encrypt is similar to MAC

For simplicity you can suppose

```
void mac(                    tag, tag_len, key, msg,       msg_len);
void auth_encr(ciphertext, tag, tag_len, key, plaintext, text_len);
```

the only difference is in Step 1 (absorbing), when we build the encrypted `ciphertext` which also has length of `text_len` bytes

A2.P1: decrypt should mimic the encrypt

and you should make sure that the sponge is the same in both encryption and decryption before each permutation. Read the XOR properties again and figure out how to get back the original plain_text.

```
void auth_encr(ciphertext, tag, tag_len, key, plaintext, text_len);
int auth_decr (plaintext, key, ciphertext, text_len, tag, tag_len);
```