


# COMP10002 Workshop Week 8

## It's about Assignment 1

	ASS1: <b>Progress?</b> Marking rubric – some specific topics? Q&A?
	<div>?</div> <ul style="list-style-type: none"> <li>• string searching, KMP &amp; BMH</li> <li>• do 7.16</li> <li>• Q&amp;A on ASS1 &amp; other exercises</li> </ul> <div>?</div> <ul style="list-style-type: none"> <li>• ASS1</li> <li>• Q&amp;A</li> </ul>
LAB	<p>Finish ass1 during this workshop or today! Submissions close at <b>6pm this Friday</b>.</p>  <p>Done ass1? Implement 7.16, then exercises from lec06</p>
LMS	<p><b>Workshops:</b></p> <ul style="list-style-type: none"> <li>• <i><b>Your most important goal this week is to complete Assignment 1.</b></i></li> <li>• Submission is <b>via the LMS Assignment 1</b> page, and <b>not</b> by grok</li> </ul>

# Your assignment1:

A- All done

B- only stages 1+2 fully done

C- only stage 1 fully done

D- none of the above

## Input:

A (normally long) text  $T[0..n-1]$ . Example:  $T = \text{"SHE SELLS SEA SHELLS"}$ , with  $n=20$

A (normally short) text pattern  $P[0..m-1]$ . Example:  $P = \text{"HELL"}$ ,  $m=4$ .

## Output:

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

## Algorithms:

- Naïve: brute force, complexity  $O(nm)$   
max number of character comparisons =  $(n-m+1)*m$   
Note: `strcmp` also compares strings character-by-character
- BMH: also  $O(mn)$  but practically fast  $m \ll n$
- KMP:  $2n+m$  iteration max  $\rightarrow O(n+m)$

# String matching: the task

Input:

- a text  $T[]$  such as “abab yxy aababcb”
- a pattern  $P[]$  such as “ababc”

Output

- first position where  $P$  appears in  $T$ , or NOTFOUND

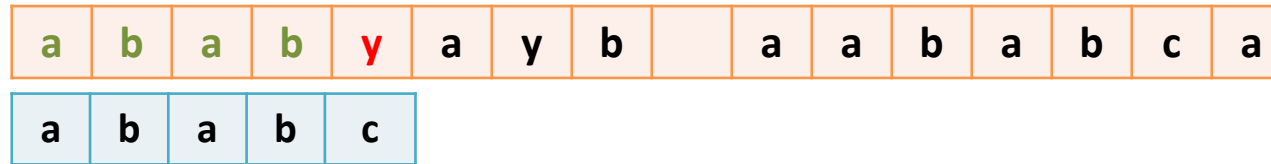
Output for the example:

- 10
- how many (pattern) shifts?
- how many comparisons?)

# String matching: KMP & BMH

Both algorithms:

- start with aligning **P** with the start of **T**
- repeatedly shift **P** to the right as far as possible by comparing **P** with **T** character-by-character



- *The number of positions to shift totally depends on P and hence is pre-computed at the start (with complexity  $O(m)$ )*

But

- **KMP** compare pattern (with text) from *left to right*, why?
- **BMH** compare pattern (with text) from *right to left*, why?

# String matching: KMP & BMH

- KMP compare pattern (with text) from left to right, why?  
*because the letters K, M, P are in alphabetic order 😊*
- BMH compare pattern (with text) from right to left, why?  
*but the letters B, M, H are not 😊*

# How to run KMP *manually*

Compare **T** with **P** from left to right

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

a	b	a	b	c
---	---	---	---	---

5 comp until mismatch;

When mismatch, examine **only** the *matched part* of **P** to decide the shift:

a	b	a	b	
---	---	---	---	--

--	--

--	--

prefix "a", suffix "b": not matched  
prefix "ab", suffix "ab": matched, length= 2

and shift **P** to the right  $s$  positions where so that:

$s$  = length of the largest matching between *prefix* and *suffix*  
( can be pre-computed just using the pattern)

# How to run KMP *manually*

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

a	b	a	b	c
---	---	---	---	---

max: prefix "ab" matches suffix "ab"

shift P for that prefix-suffix match (shift 2)

Then repeat the process from the *mismatched position in T* ie. from the first y,  
ie. from comparing y with a here

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

a	b	a	b	c
---	---	---	---	---

(here, shift 1 because no prefix-suffix match in "ab")

Note: if the matched part of P  $\leq 1$  char, or has no prefix-suffix match, just shift P by 1 position



# How to run KMP *manually*

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

a	b	a	b	c
---	---	---	---	---

5 comp until mismatch; shift 2

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

a	b	a	b	c
---	---	---	---	---

a	b	a	b	c
---	---	---	---	---

a	b	a	b	c
---	---	---	---	---

a	b	a	b	c
---	---	---	---	---

1 comp (with y); shift 1

1 comp (with y); shift 1

1 comp (with y); shift 1

2 comp(a-a, y-b); shift 1

then 4 times: 1 comp; shift 1 (comparing a with y b a

a	b	a	b	c
---	---	---	---	---

5 comp; done

# Understand the BMH algorithm

First align the pattern P with the text T.

Loop:

Start from the *right end* of the pattern, and work to the left, comparing  $P[i]$  with the corresponding character in T

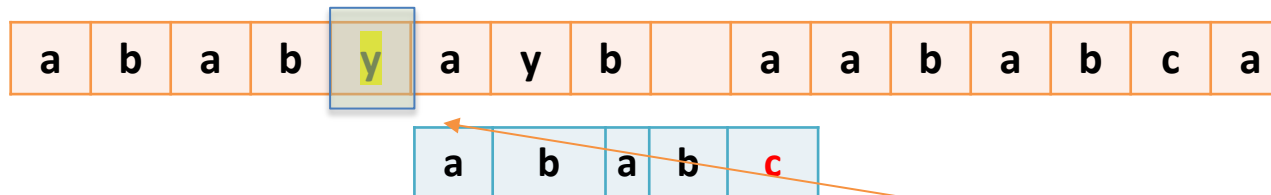
if mismatched:

shift pattern P to the right [ how many positions?]

else

FOUND

return NOTFOUND



SHIFT  $m=5$  because  
**y** not in P

- The shifts depends on the rightmost-examined T, here **y**
- Shift pattern P to the right at least 1 position and until that **y** matches with a character in P (here: shift 5 because no match is possible)

# How to run BMH *manually*

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

a	b	a	b	c
---	---	---	---	---

SHIFT  $m=5$   
because **y** not in P

1 comp until mismatch

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
---	---	---	---	---

1 comp, SHIFT 2 = distance from last **a** to

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

a	b	a	b	c
---	---	---	---	---

a	b	a	b	c
---	---	---	---	---

a	b	a	b	c
---	---	---	---	---

**The task:** Searching for a pattern  $P$  (such as “HELL” that has length  $m=5$ ) in a text  $T$  (such as “SHE SELLS SEA SHELLS”, having length  $n=20$ ).

**The Algorithm:**

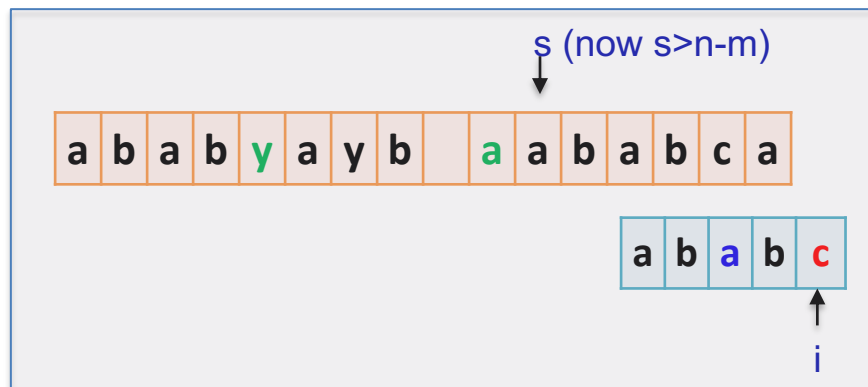
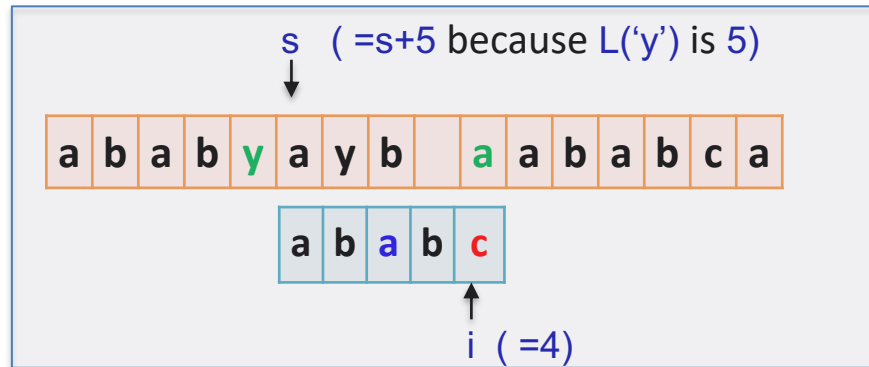
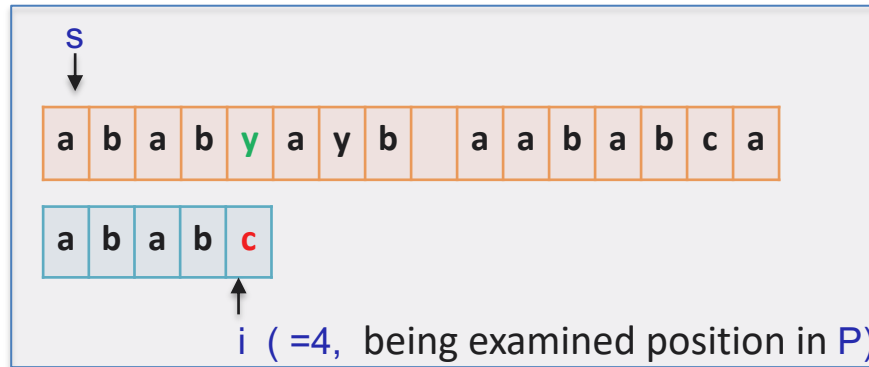
need to do a pre-processing of the pattern before performing the search  
normally,  $|P| \ll |T|$ , this step doesn't affect the overall complexity

**Pre-processing:** build  $L[x]$  for every possible character  $x$ , ie. for all  $x$  from the alphabet (in the lecture, the alphabet has  $\sigma$  symbols), by:

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

```
for  $v \leftarrow 0$  to  $\sigma - 1$   
     $L[v] \leftarrow m$   
for  $i \leftarrow 0$  to  $m - 2$   
     $L[P[i]] = m - i - 1$ 
```

# BMH - searching



```
s=0; // current start of P in T
i= m-1; // current position in P
c= T[s+m-1]; // the pilot character
```

```
while (s+i not passing the end of T) {
    if (mismatched at P[i]) {
        s = s + L[c];
        i = m-1;
        c= T[s+m-1];
    } else {
        if (i==0)
            return s;
        else
            i--;
    }
}
return NOTFOUND;
```

$s, i \leftarrow 0, m-1$

```
while s ≤ n-m
    if T[s+i] ≠ P[i]
        s, i ← s + L[T[s+m-1]], m-1
    else if i = 0
        return s
    else
        i ← i-1
return not_found
```

# Assignment 1 or 7.16 + exercises from lec06.pdf

7.12 (palindrome),  
7.14 (atoi),  
7.15 (anagram, similar to Exercise 3 )  
7.16 (word frequencies)

7.03: sorts the array then removes duplicates  
7.08: k'th smallest in array  
7.x2: Count distinct values  
7.x3: Longest ascending run

**Exercise 1** Write a function `is_subsequence(char *s1, char *s2)` that returns 1 if the characters in `s1` appear within `s2` in the same order as they appear in `s1`. For example, `is_subsequence("bee", "abbreviate")` should be 1, whereas `is_subsequence("bee", "acerbate")` should be 0.

**Exercise 2** Ditto arguments, but determining whether every occurrence of a character in `s1` also appears in `s2`, and 0 otherwise. For example, `is_subset("bee", "rebel")` should be 1, whereas `is_subset("bee", "brake")` should be 0.

**Exercise 3** Write a function `is_anagram(char *s1, char *s2)` that returns 1 if the two strings contain the same letters, possibly in a different order, and 0 otherwise, ignoring whitespace characters, and ignoring case. For example, `is_anagram("Algorithms", "Glamor Hits")` should return 1.

**Exercise 4** Write a function `next_perm(char *s)` that rearranges the characters in a string argument and generates the lexicographically next permutation of the same letters. For example, if the string `s` is initially `"51432"`, then when the function returns `s` should be `"52134"`.

**Exercise 5** If the two strings are of length `n` (and, if there are two, `m`), what is the asymptotic performance of your answers to Exercises 1–4?

# Additional Slides



# Ass1: Q&A make sure that you understand the tasks

*Carefully read the spec and the marking rubric*

*Read the Discussion Forum and:*

- *answer if you are confident,*
- *post **new** questions if needed*

*Check your code against the marking rubric*

*Examine the 2020 sample solution: you still can learn something from here even if you don't understand the task and the code.*

*Questions on marking rubric?*

## **Section E: Academic Honesty**

*missing Authorship Declaration at top of program, -5.0;*

*significant overlap detected with another student's submission, -10.0;*

*use of external code without attribution (minor), -2.0;*

*use of external code without attribution (major), -10.0;*

# Parts of execution marks

## *Deductions*

- failure to compile, -6.0;
- unnecessary warning messages in compilation, -2.0;

# Marking Rubric: The importance of Style & Structure

Stage	Presentation	Structure	Execution	max accumulated mark
1	+6 +1 +1	+4	+2	12
2		+1	+2	16
3		+1	+2	20
all	8	6	6	

Failed code  
could even  
get 14

Absolutely  
correct program  
could get only 6

Correct and  
good Stage 1+2  
alone could get  
16

# ASS1 Marking Rubric: a few keys in Presentation

- use of magic numbers, -0.5;
- #defines not in upper case, -0.5;
- bad choices for variable names, -0.5;
- bad choice for function names, -0.5;

use #define for meaningful constants  
#define-ed names should be in upper case  
variable names in lower case (except single-letter array name)  
variable names should be expressive

- absence of function prototypes, -0.5;

add function prototypes before main()  
no function implementation before main()!

- inconsistent bracket placement, -0.5;
- inconsistent indentation, -0.5;

you can use sample programs in lectures as the model

- excessive commenting, -0.5;
- insufficient commenting, -0.5;

each function header should have a comment  
add comments for non-trivial code segment

- lack of whitespace (visual appeal), -0.5;
- lines >80 chars, -0.5;

• other issue: minor -0.5, major -1.0

- comment at end that says "algorithms are fun", +0.5;
- overall care and presentation, +0.5;

DO IT !  
Easy way to get back 0.5 or 1 mark

# ASS1 Marking Rubric: a few keys in Structure

- global variables, -0.5;

Global variables are NOT allowed!

- main program too long or too complex, -0.5;
- functions too long or too complex, -0.5;
- overly complex function argument lists, -0.5;

- function should not be long
- and should not have too many arguments

- insufficient use of functions, -0.5;
- duplicate code segments, -0.5;

- when having a few line, or a complicated line similarly-duplicated, think about creating a new function!

- **overly complex algorithmic approach, -1.0;**

- avoid too many levels of nesting `if` and loops
- don't make the marker wonder too much to understand your code!

- unnecessary duplication/copying of data, -0.5;

- For example, think carefully before you copy an array!

- **other structural issue: minor -0.5, major -1.0;**

# Parts of execution marks

## *Deductions*

- failure to compile, -6.0;
- unnecessary warning messages in compilation, -2.0;

Examples of overly complicated:

sort the data when not actually required

too many levels of nested loops/if



## duplicate code segments: turn similar segments into a function

having 2 or more lines similar? Think if you should form a new function.

# ASS1 Marking Rubric: a few keys in Execution

**failure to compile, -6.0;**

**unnecessary warning messages in compilation, -2.0;**

Your program might be compiled OK in your computer, without any warning.  
But it might have compiler errors or warnings on the testing machine.  
→ carefully check with `grok`, and check again when submitting

incorrect Stage X layout or values in any test, -0.5 ;

different Stage X layout or values in any test, -0.5 ;

X can be: 1, 2, 3a, 3b

Again, check the verification report!

When testing compare your outputs with expected outputs using command `diff`:

```
./ass1 alice feet < data1.txt > out1.txt  
diff out1.txt data1-S1-out.txt
```

Desirable outcome: EMPTY output from command `diff`.

If you have non-empty output:

- + The lines starting with `<` is from the first file of the `diff`
- + The lines starting with `>` is from the second file
- + You can just do the testing submit to see the diff

# Assignment 1: testing

## TESTING IN YOUR COMPUTER

*Using redirection when running/testing your program:*

```
./myass1 < test0.txt > test0-myout.txt
```

*Your program's output must be the same as the expected, ie. the command*

```
diff test0-myout.txt test0-out.txt
```

*must give empty output (that is, no difference).*

*Remember that your code might work well on the 4 supplied data sets, but fail on some other...*

*A better check in your computer, that not only test for correctness, but also for compilation warnings:*

- *copy `Makefile` from grok to your directory*
- *run “`make`” for compiling*
- *run “`make test0`”, “`make test1`”, “`make test2`”, “`make test3`” to run the tests*

*or you can copy your code to grok and do the testing there.*

## Compiler warnings: -2.0

*If you don't use make (why not?), you should test for compilation warning with (command copied from Makefile):*

```
clang -Wall -Wextra -Werror -Wno-newline-eof -Wno-unused-parameter -pedantic  
-std=c99 -ferror-limit=1 -o myass1 myass1.c
```

and the above command must give no warning messages at all!

## Assignment 1 testing: seems OK even before the submission

```
bash$ ./myass1 < test0.txt > test0-myout.txt
```

```
bash$ diff test0-myout.txt test0-out.txt
```

```
bash$
```

*must give empty output (that is, no difference).*

line starting with **<** is from the left file (ie. our output)  
**here:** our line is not identical to the expected

*Understand the output from the above `diff`,*

```
bash$ diff test2-myout.txt test2-out.txt
```

```
5d4
```

```
< operation not available yet
```

```
7c6
```

```
< register b: 123,456,789
```

```
---
```

```
> register b: 2,592,592,569
```

```
9d7
```

line starting with **>** is from the right file (ie. expected output)  
**here:** we missed 1 blank line in our output

## Ex 7.16 and others

Combine Alistair's `getword.c` and `words.c` into one `.c` file, then change it to meet the requirement of Ex 7.16.

# Case Study & Ex 7.16 – The Task

Use the program of Figures 7.13 and 7.14 of the textbook (`words.c` and `getword.c` on Page 4 of `lec06.pdf`).

*Design* and implement a program that reads text from `stdin`, and writes a list of the distinct words that appear, together with their frequencies.

First step:

Make sure you understand the task, that you can imagine what's the input and output.

# Case Study & Ex 7.16 – Understanding The Task

Design and implement a program that reads text from stdin, and writes a list of the distinct words that appear, together with their frequencies.

Sample texts:

A cat in a hat!

+ - abc 10e12 e 1abc #e#abc.abcdefghijklm=xyz

Input = ?

- How to get the input text?

Output = ?

- How to store output, which data structure?
- And how to produce output?

Assumptions/limits:

- What's a *word*?
- Other assumptions?



# Case Study & Ex 7.16 – Alistair's getword

```
int getword(char W[], int limit) {
    int c, len=0;
    /* first, skip over any non alphanumerics */
    while ((c=getchar()) != EOF && !isalpha(c)) {
        /* 12+34 aWord ?-? is the first word */
    }
    if (c==EOF) return EOF;

    /* ok, first character of next word has been found */
    W[len++] = c;
    while (len<limit && (c=getchar())!=EOF && isalpha(c)) {
        /* 12+34 aWord ?-? is the first word */
        W[len++] = c;
    }

    /* now close off the string */
    W[len] = '\0'; // W is the string aWord
    return 0;
}
```

# Alistair's words.c

```
#define MAXCHARS 10
/* Max chars per word */
#define MAXWORDS 1000
/* Max distinct words */

typedef char word_t
    [MAXCHARS+1];
/* word_t word; now is
   equivalent to
   char word [MAXCHARS+1];
   */

int getword(word_t W,
            int limit);

#include "getword.c"

int
main(int argc,
     char *argv[]) {
```

```
    word_t one_word, all_words[MAXWORDS];
    int numdistinct=0, totwords=0, i, found;

    while (getword(one_word, MAXCHARS) != EOF) {
        totwords = totwords+1;
        /* linear search in array of previous words...*/
        found = 0;
        for (i=0; i<numdistinct && !found; i++) {
            found = (strcmp(one_word, all_words[i]) == 0);
        }
        if (!found && numdistinct<MAXWORDS) {
            strcpy(all_words[numdistinct], one_word);
            numdistinct ++;
        }
        /* NB - program silently discards words after
           MAXWORDS distinct ones have been found */
    }

    printf("%d words read\n", totwords);
    for (i=0; i<numdistinct; i++) {
        printf("word #%d is \"%s\"\n", i, all_words[i]);
    }
    return 0;
}
```

# Some small, easy, but important topics

`typedef`

`argc, argv`

# Program arguments

Write a program `sum` that accept two numbers and print out their sum. Example of execution:

```
$ ./sum 12 5
```

```
12.00 + 5.00 = 17.00
```

```
?: int main(int argc, char *argv[])
```