DALHOUSIE UNIVERSITY
Inspiring Minds

Fall 2018 (Sep4-Dec4)
Faculty of Computer Science

CSCI 2132 — Software Development

Assignment 7— Sample Solutions

---

1) (10 marks) Briefly describe the `malloc` function, how it works, and its time efficiency. Describe briefly the `free` function, how it works, and its time efficiency.

Record and submit your answer via SVN in the file: *CSID*`/a7/a7q1.txt` where *CSID* is your CSID and it is your main SVN directory for the course.

Solution:

The `malloc` function reserves a block of memory on the heap, of the required size and returns a pointer to it. (Optional: The function also records the size of the block in a memory location just before the block.) The function searches for a block of the appropriate size by traversing a linked list of free blocks. If it does not find a block of the appropriate size, it uses a system call to request more memory from the operating system kernel, and adds this memory to the heap. If it does not succeed to get appropriate additional memory, it returns NULL. The malloc function is typically fast, but it may take some time if the list of the free blocks is very long, and if it needs to execute the system call.

The `free` function takes a parameter, which is a pointer to a block of previously reserved memory, and releases it. The function knows the size of the block because it is recorded in the heap, and the block is released by returning it to the list of free blocks. This is generally an efficient operation.

Marking scheme:
1pt — malloc reserves a block of memory
2pt — size given as parameter
1pt — returns a pointer
1pt — returns NULL if not successful
1pt — traverses a linked list of blocks, so it may take some time

— may need to call system call for more memory (if this is mentioned it is a bonus 1pt, but total cannot go over 10pt)

2pt — free takes a parameter, which is a pointer to a block to be released
1pt — releases block to the list of free blocks
1pt — relatively efficient function

---

2) (15 marks) Submit the answer to this question as a C file named `a7q2.c`, which includes definitions of the required functions below. The file must be submitted via SVN in the pathname: *CSID*`/a7/a7q2.c` where *CSID* is your CSID and it is your main SVN directory for the course.

The file must be valid C code that can be compiled and tested with other code.

Let us assume that the following structure is used to create a linked list:

```
struct node {
```

```
  int val;
  struct node *next;
};
```

a) (5 marks) Write a recursive function `list_sum_r` with the following prototype:

```
int list_sum_r(struct node *head);
```

which returns 0 if the parameter `head` is NULL, and otherwise it returns the sum of all elements of the list that starts with the `head` pointer.

b) (5 marks) Write a non-recursive version of the function in a), called `list_sum_i` with the same prototype:

```
int list_sum_i(struct node *head);
```

and the same result.

Solution:

```
int list_sum_r(struct node *head) {
  if (head == NULL)
    return 0;
  else
    return head->val + list_sum_r(head->next);
}

int list_sum_i(struct node *head) {
  int sum=0;
  struct node *p = head;
  while (p != NULL) {
    sum += p->val;
    p = p->next;
  }
  return sum;
}
```

---

c) (5 marks) Write a function `list_reverse` with the following prototype:

```
struct node * list_reverse(struct node *head);
```

that reverses the list and returns the pointer to the new head of the list.

Solution:

```
struct node *list_reverse(struct node *head) {
  struct node *p, *new_head;
  if (head == NULL || head->next == NULL)
    return head;
  p = head->next;
```

```
  head->next = NULL;
  new_head = list_reverse(p);
  p->next = head;
  return new_head;
}
```

---

3) (10 marks) Make sure that you complete the Lab 9 as required. There is a set of files that need to be submitted in this lab in your GitLab project directory "`git/csci2132/lab9`".

Solution:

You need to follow the instructions given in Lab 9. As a result of the lab your csci2132 project in GitLab should have a directory named `lab9` that contains the following files, with content as specified in the lab:

- bit.h
- dec2bin.c
- hello.c
- Makefile — with added FLAGS variable
- sortlist.c
- stack.c
- stack.h
- sortlist.h — created file with specified contents
- main.c — created file with specified contents
- list.c — created file with specified contents
- aux.c — created file with specified contents
- testlist1.in — created file with specified contents
- testlist1.out — created file with specified contents

---

4) (15 marks) Your task in this question is to write a C program named `I.c` according to the specifications below. You will submit the program using SVN in the pathname *CSID*`/a7/I.c` where *CSID* is your CSID and the name of your main SVN directory for the course. You will also need to submit the program as a part of the Practicum 5. The Practicum 5 link is available at the course web site.

Problem description: You should be familiar with the program sort that can sort lines of a textual file in various ways using the keys. A simple sort will just sort lines alphabetically, or to be more precise in the ASCII order using the characters in the line and using the lexicographic order.

In some situations, we may want to keep several lines together and we do not want to break them in sorting. For example we can indicate that a line continues into the next line by using the backslash character at the very end of the line. Your task is to write a sorting program that sorts lines in such way that the continued lines are not broken. The continued lines should be sorted and compared to each other as long strings using the function `strcmp`.

Since there is not specified limit on the number of lines or the length of one line, it is recommended that you use `malloc` and `realloc` functions to allocate for each continued line exactly as much memory as required. You can read all continued lines into a linked list, then sort the list, and print them from the list.

Input: The standard input includes an arbitrary text. There are no specified limitations to the number of of lines or the length of lines.

Output: Your program must keep continued lines in a group, where any line continues to the next line if there is the backslash character at the very end of the line, and the program must printed the continued lines in a sorted order. Of course, the continued line can be mixed with single lines that are not part of any group.

You can find more details and sample input and output of the problem at the practicum site.

a) [10 marks] Functionality: To satisfy this requirement, your program must satisfy the program requirements as specified and pass the practicum testing.

b) [5 marks] Implementation: To receive these marks, the program must be implemented in the following way, and it will be marked by a marker. You are required to write a head comment that includes your name, date of the program, the course and assignment number, and a short description of the program. The program should follow the organization presented in the class.

The program should use an appropriate indentation. Each block should have a consistent indentation, which can be any number of spaces you choose, between 2 and 8. You can choose a reasonable brace placement style (for example, see
`https://en.wikipedia.org/wiki/Indentation_style`). K&R style would be recommended, but any other known style is acceptable as well. Your program lines should not be longer than 80 characters.

The program should use allocation functions (such as `malloc` and `realloc`,) to be flexible about line length and allocate the appropriate amount of memory. It is preferable that you use linked list and merge sort, but it is acceptable if you use array-based sorting, as long as you do not use more memory than minimally required by the program.

When allocating memory, you should start with allocation of certain memory size, and whenever you need more memory, you should use realloc to double the memory size. Once you know the exact memory size that you need, you should again use realloc to shrink the block to the exactly needed memory size.

Solution:

```
/* CSCI 2132, Fall 2018, Practicum 5, 2018-11-22 (Assignment 7)
   Solution to Problem I: Sort of Continued Lines
   Author: Vlado Keselj
   Description:  The program reads the standard input and prints it to
     the output after sorting the continued lines.  A line is
     continued if the last character before the newline is backslash
     (\).  A continued line is connected to the next line and so on
     until a line does not end with backslash.  A continued line is
     is not broken in the sort.
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int EOF_indicator=0; /* Set to 1 once EOF is reached. */
```

```
/* Structure used for the linked list of continued lines.
   Each node contains one continued line. */
struct node {
  char *line;
  struct node *next;
};

char *read_continued_line(); /* Reads a continued line from the
                                standard input, allocating space on
                                the heap for it, and returns a pointer
                                to it. */

/* Functions for MergeSort of the linked list. */
struct node* merge_sort(struct node* head);
struct node* merge(struct node *p, struct node* q);

int main() {
  struct node *head=NULL, *p;
  head = (struct node*) malloc(sizeof(struct node));
  head->line = read_continued_line();
  head->next = NULL;
  p = head;
  while (!EOF_indicator) {
    char *line = read_continued_line();
    if (strlen(line) == 0) {
      free(line); break;
    }
    p->next = (struct node*) malloc(sizeof(struct node));
    p = p->next;
    p->next = NULL;
    p->line = line;
  }

  head = merge_sort(head);
  for (p=head; p!=NULL; p = p->next)
    printf("%s", p->line);
  return 0;
}

char *read_continued_line() {
  int ch, buff_cap, buff_i;
  char *buff;
  buff = (char*) malloc( buff_cap = 1024 );
  buff_i=0;

  while (EOF != (ch = getchar())) {
    if (buff_i >= buff_cap-2) {
      buff = (char*) realloc( buff, buff_cap * 2 );
      buff_cap *= 2;
    }
    buff[buff_i++] = ch;
```

```
    if (ch=='\n' && (buff_i<2 || buff[buff_i-2]!='\\'))
      break;
  }
  if (ch == EOF) EOF_indicator = 1;
  buff[buff_i] = '\0';
  buff = (char*) realloc( buff, buff_i+1 );
  return buff;
}

struct node* merge_sort(struct node* head) {
  struct node *p, *q;
  if (head==NULL || head->next == NULL)
    return head;
  for(p=head, q=head->next; q != NULL && q->next != NULL; ) {
    q = q->next->next;
    p = p->next;
  }
  q = p->next;
  p->next = NULL;
  p = merge_sort(head);
  q = merge_sort(q);
  head = merge(p, q);
  return head;
}

struct node* merge(struct node *p, struct node* q) {
  if (p == NULL) return q;
  if (q == NULL) return p;
  if (strcmp(p->line, q->line) <= 0) {
    p->next = merge(p->next, q);
    return p;
  }
  q->next = merge(p, q->next);
  return q;
}
```

---

5) (15 marks) Your task in this question is to write a C program named `J.c` according to the specifications below. You will submit the program using SVN in the pathname *CSID*`/a7/J.c` where *CSID* is your CSID and the name of your main SVN directory for the course. You will also need to submit the program as a part of the Practicum 5. The Practicum 5 link is available at the course web site.

Problem description: We covered in class generation of all permutations of numbers 1, 2, ... n. However, as we noticed, the generated permutations were not lexicographically ordered. For example, for $n = 4$ the generated permutations were:

```
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 3 2
...
```

while the lexicographically ordered permutations would be:

```
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 2 3
...
```

If you are not familiar with lexicographic order, we can define it in the following way: We compare two sequences of numbers (or letters, or other comparable elements), from the start and keep doing it while the sequence elements are equal. When we come across the first elements that are different, we decide that sequences compare in the same way as those two elements. In the above example, when we compare sequences (1,4,3,2) and (1,4,2,3), we see that the sequences differ first time at the third position, and the sequence with 2 at the third position should come before the sequence with 3 at third position; i.e., the order should be (1,4,2,3) and then (1,4,3,2).

You should write a program that generates permutations in the lexicographically ordered way. The program must have two modes of operation, in one the program must produce all permutations, and in other it must produce only the permutation that comes at certain position in the permutations list.

Method: You should base your method on the algorithm covered in class. If you remember the algorithm (one version) from Lecture 19 was:

```
1: IF k == n-1 THEN print A
2: ELSE
3:   Permute(A, k+1, n)
4:   FOR i = k+1 TO n-1 DO
5:     swap A[k] with A[i]
6:     Permute(A, k+1, n)
7:     swap A[k] with A[i]    /* swap back */
```

In order to get lexicographic order, in the step 5, you should not just swap $A[k]$ and $A[i]$, but you need to rotate all elements from $A[k]$ to $A[i]$, by moving $A[k+1]$ into $A[k]$, $A[k+2]$ into $A[k+1]$, and so on, until you move $A[i]$ into $A[i-1]$, and finally move the old value of $A[k]$ into $A[i]$. Similarily in the step 7 you need to reverse this rotation.

Input:

Each line of input contains two integers m and n. The input ends with integers 0 and 0. Other than the final two numbers, n is always the positive integer. If the first number m is 0, you program must print all permutations of numbers 1, 2, ..., n in the lexicographic order. If the number m is positive, then your program must print only the m'th permutation of numbers 1, 2, ..., n in the lexicographic order. If the number m is so large that you run out of permutation, your program should not print anything.

Output: For each par of numbers m and n in the input, your program should first produce the line that looks as follows:
If n is 0, the line should be:
`End of output`
otherwise, if m is 0, the line should be:
`Permutations of` n`:`
otherwise, the line should be:
`Permutation of` n `number` m`:`

After that, if $m = 0$ and $n > 0$ the program must produce all permutations as specified, one permutation

per line; or if $m > 0$ only the specified permutation, or no permutations at all if m is larger than the number of permutations, and if both m and n are zero, then the program should end.

You can find more details and sample input and output of the problem at the practicum site.

a) [10 marks] Functionality: To satisfy this requirement, your program must satisfy the program requirements as specified and pass the practicum testing.

b) [5 marks] Implementation: To receive these marks, the program must be implemented in the following way, and it will be marked by a marker. You are required to write a head comment that includes your name, date of the program, the course and assignment number, and a short description of the program. The program should follow the organization presented in the class.

The program should use an appropriate indentation. Each block should have a consistent indentation, which can be any number of spaces you choose, between 2 and 8. You can choose a reasonable brace placement style (for example, see
`https://en.wikipedia.org/wiki/Indentation_style`). K&R style would be recommended, but any other known style is acceptable as well. Your program lines should not be longer than 80 characters.

The program should use the above method, although alternative methods are also acceptable if they are correct and equally or more efficient. If the input requires that you print only one permutation, the program should not continue generating unnecessary permutations internally, but move to the next case.

Solution:

```
/* CSCI 2132, Fall 2018, Practicum 5, 2018-11-22 (Assignment 7)
   Solution to Problem J: Ordered Permutations
   Author: Vlado Keselj
   Description: The program reads two integers at each input line,
     m and n, and repeats the following for each line.
     It is assumed that both m and n are non-negative integers.
     It first prints values of those integers in format "m=m n=n".
     If n==0, the program ends.  If m==0 and n>0, the program then
     prints all permutations of numbers 1..n in the lexicographic
     order.  If m>0 and n>0, the program prints m'th permutation of
     numbers 1..n, according to the lexicographic order.  If m is
     larger than the number of permutations, then nothing is printed
     after the "m=m n=n" line.
*/
#include <stdio.h>

/* Global variables */
int m;                 /* The current input value of "m" as described */
int permutation_count; /* Permutation count, starting from 1 */

/* The function permute below is a variation of the permute function
   covered in class.  This function lists permutations in
   lexicographic order.  It is a recursive function with the following
   parameters:
   n - the length of array a[] and the final number n for permutations
       of 1..n
   k - the first k elements of a[] are fixed
*/
void permute(int n, int k, int a[]);
```

```
int main() {
  int n, i;
  while (2 == scanf("%d%d", &m, &n)) {
    if (n == 0) {
      printf("End of output\n");
      break;
    } else {
      int a[n];
      if (m == 0)
        printf("Permutations of %d:\n", n);
      else
        printf("Permutation of %d number %d:\n", n, m);
      for (i=0; i<n; i++)
        a[i] = i+1;
      permutation_count = 0;
      permute(n, 0, a);
    }
  }
  return 0;
}

void permute(int n, int k, int a[]) {
  if (m != 0 && m < permutation_count)
    return;
  if (k == n-1) {
    permutation_count++;
    /* Decide whether we print or not */
    if (m == 0 || m == permutation_count) {
      int i;
      for (i=0; i<n; i++)
        printf(" %d", a[i]);
      printf("\n");
    }
  } else {
    int i, j;
    permute(n, k+1, a);

    /* The for-loop below is the key difference from the algorithm
       used in class.  We do not simply swap elements, but move the
       i'th element to the position k, and shift all intermediate elements
       to the right to keep the lexicographic order. */
    for (i = k+1; i < n; i++) {
      int hold = a[i];
      for( j = i; j > k; j--)
        a[j] = a[j-1];
      a[k] = hold;

      permute(n, k+1, a);

      /* move elements back */
```

```
      hold = a[k];
      for( j=k; j < i; j++)
        a[j] = a[j+1];
      a[i] = hold;
    }
  }
}
```
