ZJU-UIUC Institute
Second Midterm Exam, ECE 220

Friday 4 December 2020

Name (pinyin and Hanzi):
PARTIAL SOLUTION IN RED
(SOME PARTS NOT CHANGED FROM ORIGINAL)
Student ID: Lab TA Name:

- **Be sure that your exam booklet has 12 pages.**
- **Write your name, student ID, and lab section TA name on the first page.**
- **Do not tear the exam apart other than to remove the reference sheet.**
- **This is a closed book exam. You may not use a calculator.**
- **You are allowed TWO handwritten A4 sheets of notes (both sides).**
- **YOU MAY NOT USE EXTRA PAPER! WRITE ON THE EXAM!**
- **Absolutely no interaction between students is allowed.**
- **Show all work, and clearly indicate any assumptions that you make.**
- **Challenge problems are marked with ***.**
- **Don't panic, and good luck!**

Problem 1	19 points	_____
Problem 2	25 points	_____
Problem 3	25 points	_____
Problem 4	30 points	_____
Correct Room	1 point	_____
Total	100 points	_____

Problem 1 (19 points): Short Answer Questions

Tingkai is working on a filesystem for his operating system in ECE391. His plan is to organize the filesystem as a set of data blocks, each containing 4 kB (4,096 bytes), using the following structure:

```
struct data_block_t {
    uint8_t bytes[0x1000];
};
```

1. **(3 points)** After struggling for a while, Tingkai found a function that returns a pointer to the first block of the file system:

   ```
   struct data_block_t* fblock = get_fs_pointer ();
   ```

   Tingkai then wrote following variable declaration below to copy a block from the filesystem:

   ```
   struct data_block_t blk = *(fblock - 7 + N * 3);
   ```

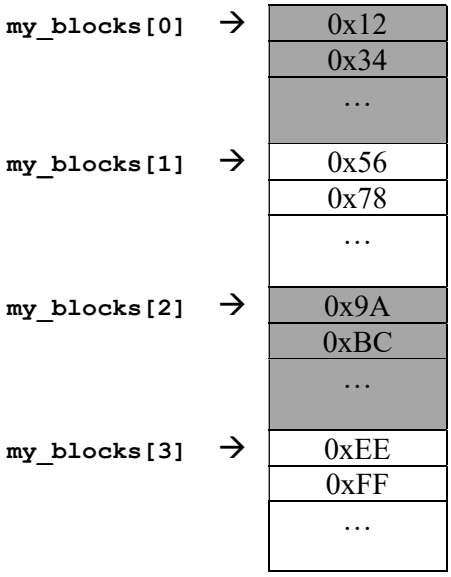
   His code doesn't look elegant. Please help him by rewriting the declaration using array notation (`N` is a variable).

   `struct data_block_t blk =` **fblock[N * 3 – 7]** `;`

Now consider an array of data blocks:

```
struct data_block_t my_blocks[4];
```

Assuming a machine with byte-addressable memory (one memory address in memory holds one byte), the array appears as shown to the right.



2. **(3 points)** What is the value of

   `my_blocks[4].bytes[-0xFFF]`?

   Write "bits" if the value cannot be determined.

   0xFF

3. **(3 points)** Given the declaration:

   ```
   struct data_block_t* foo = my_blocks + 18;
   ```

   and assuming that `my_blocks` has value `0xECE220`, what is the value stored in `foo` (in hexadecimal)? Show your work for partial credit.

0xECE220 + 18 * 0x1000 = 0xEE0220

Problem 1, continued:

4. **(4 points)** In ECE220, Bob learned that a one-dimensional array can be passed as a pointer to a function. He wanted to extend this idea to two-dimensional arrays and wrote the following code:

```
void foo (char** arr) {
    // Some code
}

int main () {
    char a[10][20];
    foo (a);
    return 0;
}
```

Is this code correct? **USING 30 WORDS OR FEWER**, explain.

_____

_____

5. **(6 points)** Bob also struggled with the idea of dynamic allocation. On a recent midterm, he was asked to write a function to free a linked list of `thing_t`s, given a pointer to a variable holding a pointer to the head of the list. The function should also set the original list head variable to NULL.

Bob wrote the code shown below. Unfortunately, it contains TWO BUGS.
USING 20 WORDS OR FEWER (per bug), explain each bug and how to fix the problem.

Bug 1: _____

_____

Bug 2: _____

_____

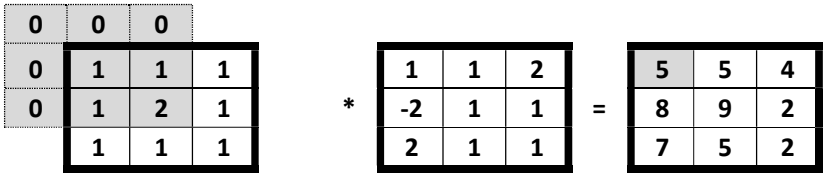
```
typedef struct thing_t thing_t;
struct thing_t {
    // Other fields don't matter.
    thing_t* next;
};

void free_list (thing_t** head_ptr)
{
    for (thing_t* thing = *head_ptr; NULL != thing; thing = thing->next) {
        free (thing);
    }
    head_ptr = NULL;
}
```

Problem 2 (25 points): Arrays and Debugging with Deep Neural Networks

1. **(15 points)** In this problem, you must implement a convolution, an important tool for image processing and deep neural networks (DNNs). Given a **`size`** × **`size`** input matrix **`in`** and a 3×3 mask matrix **`mask`**, your function must calculate a **`size`** × **`size`** matrix **`out`**, as described below.

   In a convolution, each output element is calculated by first aligning the center of the mask matrix over the corresponding input element (the element with the same row and column indices as the output element being calculated), as illustrated by the shaded region of the input in the example below for (row,col)=(0,0). Notice that part of the mask may fall outside of the input for some output elements (including the example shown), and that elements outside the actual input matrix are treated as 0. After alignment, each element of the mask is multiplied by the corresponding input element (or 0), and all nine products are summed to produce the single output element.

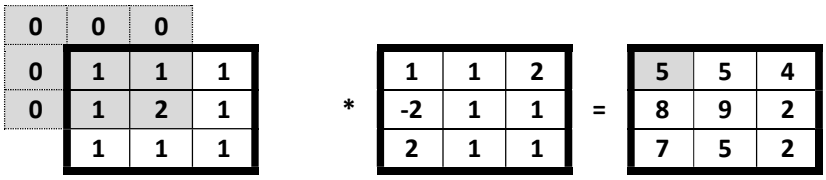


in mask out

   Specifically, in the figure above, all three matrices—**`in`**, **`mask`**, and **`out`**—are 3×3. To calculate the (0,0) element of the output matrix **`out`** (the shaded element), we align the center of the mask over the (0,0) element of the input matrix **`in`**—the shaded region shows the position of the **`mask`** matrix. Elements of the shaded region that fall outside of the input matrix use the value 0 instead of input matrix values, as shown in the figure. To compute the output value, we multiply each of the values from the mask by the corresponding element of the input (or 0), then sum up the nine products. In this case, starting from the upper left, we obtain 0×1 + 0×1 + 0×2 = 0 from the first row, 0×(-2) + 1×1 + 1×1 = 2 from the second row, and 0×2 + 1×1 + 2×1 = 3 from the third row, for a total of 5, which we write into **`out`** at position (0,0).

   The problem is on the next page.

Problem 2, continued:



in mask out

(Image replicated for your convenience.)

USING AT MOST SIX LINES (not counting braces and variable declarations), which must fit within the blanks provided, complete the inner loop body below to handle one product term of each output element. Include comments for more partial credit.

Each of the three matrices—**`in`**, **`mask`**, and **`out`**—is given as a one-dimensional array. Calculate array indices in the same manner as discussed in class and used in your MPs.

You may ignore the possibility of overflow in your calculations.

```
void conv_layer (int32_t* in, int32_t* mask, int32_t* out, int32_t size)
{
  int32_t x, y, p, q;

  // For each value in the input matrix
  for (y = 0; y < size; y++) {
    for (x = 0; x < size; x++) {

      // Initialize out to 0
      out[y * size + x] = 0;

      // For each value in the mask matrix
      for (p = 0; p < 3; p++) {
        for (q = 0; q < 3; q++) {

          _____________________________________________

          _____________________________________________

          _____________________________________________

          _____________________________________________

          _____________________________________________

          _____________________________________________

        }
      }
    }
  }
}
```

Problem 2, continued:

2. (**10 points**) Pooling is a second important operation in DNNs. In max pooling, a square submatrix of values is replaced with a single value—the **maximum** among the values in the square submatrix. The picture below, for example, shows 2×2 max pooling applied to a 5×5 input matrix.

29	**35**	**26**	**167**	**55**
0	**100**	**45**	**22**	**33**
14	**14**	**7**	**34**	**21**
14	**14**	**22**	**56**	**22**
16	**-6**	**2**	**155**	**23**

in

-->

100	**167**
14	**56**

out

Your friend has implemented 2×2 max pooling in the subroutine below. Given a `size` × `size` input matrix `in`, the subroutine produces an appropriately sized output matrix `out` (if `size` is odd, the code should ignore the last row and column).

Unfortunately, **your friend's subroutine HAS TWO BUGS**. For each bug, give one example of an input matrix `in` (for example: `in = {1, 2, 3, 4}`) that exposes the bug. Then, **USING TWENTY WORDS OR FEWER**, explain the bug.

Bug 1: `in = {` anything with 4 or more elements and more than one distinct value `}`

Reason 1: expression for "value to be compared" computes min, not max

---

Bug 2: `in = {` anything with size 4+ and outputs that get smaller in loop order (when correct) `}`

Reason 2: val_max initialized outside of loops—should be reset for each output value

---

```
void maxpooling_layer(int32_t *in, int32_t *out, int32_t size) {
  int32_t out_size = (size >> 1);
  int32_t x, y, p, q;
  int32_t val_max = 0x80000000, val_cmp;

  // For each value in the output matrix
  for (x = 0; x < out_size; x++) {
    for (y = 0; y < out_size; y++) {
      for (p = 0; p < 2; p++) {
        for (q = 0; q < 2; q++) {
          // value to be compared
          val_cmp = in[(2 * y + q) * size + (2 * x + p)];
          val_max = (val_max <= val_cmp ? val_max : val_cmp);
        }
      }
      out[y * out_size + x] = val_max;
    }
  }
}
```

Problem 3 (25 points): Functions and Dynamic Resizing

Recall the AI quantum magic button that you implemented during Midterm 1 for D-331. To further improve security, you decide to allocate a unique ID (a `uint32_t`) for each student, and to store these IDs in a dynamically resized array. The ID list is stored in a dynamically resized array using the following file-scope variables.

```
uint32_t* id_list;     // pointer to the dynamically allocated ID array
uint32_t  num_ids;     // current number of elements stored in ID array
uint32_t  max_ids;     // current ID array size
```

Complete each function below by filling in the blanks as necessary. Not all blanks may be needed. You may **USE ONLY THE BLANKS PROVIDED**. Additional code will earn no credit.

1. **(3 points)** In class, Prof. Lumetta claimed that maintaining a dynamically resized array in sorted order makes insertion and deletion slow. But, as you know from your experience with the MPs, one can perform insertion and deletion into a sorted array of N elements in log(N) time. **USING 10 OR FEWER WORDS**, explain how.

   ____Any reference to binary tree-like structure is fine here.____

2. **(7 points)** Complete the function below to check whether the ID given by parameter `id` is already present in the array of IDs, ***assuming that the array is sorted from small to large***. Return 1 if it is present, or 0 if it is not present. A solution that ignores the sorted order will earn little or no credit.

```
int32_t is_duplicate (uint32_t id) {

    int low = 0, high = num_ids - 1, mid;

    while (low <= high) {

        mid = low + (high - low) / 2;

        if (id < id_list[mid]) {

             high = mid - 1;

        else if (id > id_list[mid]) {

             low = mid + 1;

        } else {

            return 1;
        }

    }

    return 0;

}
```

Problem 3, continued:

```
uint32_t* id_list;     // pointer to the dynamically allocated ID array
uint32_t  num_ids;     // current number of elements stored in ID array
uint32_t  max_ids;     // current ID array size
```

(List of file-scope variables replicated for your convenience.)

3. **(15 points)** Finally, complete the function below to insert the ID given by the parameter id to the end of the array of IDs. The function should return 1 for success, and 0 for failure.
   - Duplicate IDs should be rejected (by returning failure).
   - You may assume that the array ID pointer is valid when your function is called (it will not be NULL).
   - If the array does not have enough space for a new ID, grow the array by a factor of 3, updating file-scope variables as necessary. Return failure if no memory is available.
   - See the reference sheet for dynamic allocation functions available in the C standard library.

   Your code **MUST USE the helper function** that you developed for **Part 2** as well as the following function to insert a non-duplicate id into the sorted array of IDs:

```
void insert_id (uint32_t id);
```

```
int32_t register_id (uint32_t id) {

    _____________________________________________________________________

    _____________________________________________________________________

    // Check for duplicate IDs.

    _____________________________________________________________________

    _____________________________________________________________________

    _____________________________________________________________________

    // Resize the array if necessary.

    if ( _____________________________________________ ) {

        _____________________________________________________________________

        _____________________________________________________________________

        _____________________________________________________________________

        _____________________________________________________________________

        _____________________________________________________________________

        _____________________________________________________________________

        _____________________________________________________________________

    }
    // Insert the ID.

    _____________________________________________________________________

    return 1;
}
```

Problem 4 (30 points): Merge Sort on Linked Lists

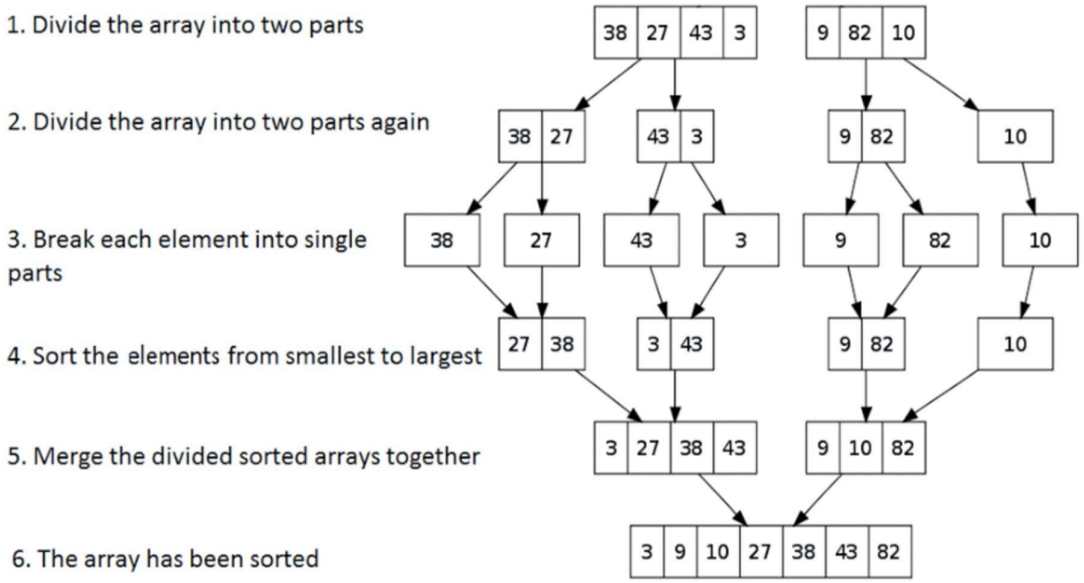
In this problem, you must implement a recursive merge sort on a linked list of values. Merge sort works as follows:

- Divide the original data into two unsorted parts (of roughly equal size).
- Sort both parts (recursively).
- Merge the two sorted parts into a sorted whole.

Note that any list consisting of a single element is already sorted, and forms a stopping condition for the recursion.

For each part of this problem, complete each function below by filling in the blanks as necessary. Not all blanks may be needed. You may **USE ONLY THE BLANKS PROVIDED**. Additional code will earn no credit.

As a further aid to understanding, the diagram below illustrates the merge sort process for an array.



Your recursive merge sort will operate on a linked list of the structure shown below, each of which has an integral value and a next pointer.

```
typedef struct element_t element_t;
struct element_t {
    int32_t    value;
    element_t* next;
};
```

Problem 4, continued:

```
typedef struct element_t element_t;
struct element_t {
    int32_t    value;
    element_t* next;
};
```

(Structure definition replicated for your convenience.)

Before implementing the main recursive merge sort function, you must implement two helper functions.

1. ***(12 points)** Complete the function below to divide the list starting with **`head`** into two sublists. The sublists should have equal length if the original list has even length. If the original list has odd length, the extra element should be put into the first sublist. A pointer to the head of the first sublist should be written to the address given by **`firstp`**, and a pointer to the head of the second sublist should be written to the address given by **`secondp`**.

   Not all blanks may be needed. You may **USE ONLY THE BLANKS PROVIDED**. Additional code will earn no credit.

   For this function, the list given by **`head`** must not be empty.

   For full credit, do not write any additional loops (other than the one given).

```
void divide_list (element_t* head, element_t** firstp, element_t** secondp)
{
    element_t* middle = head;
    element_t* end    = head->next;
    element_t* sublist;

    while (end != NULL) {

        _____________________________________________________________________

        _____________________________________________________________________

        _____________________________________________________________________

        _____________________________________________________________________

        _____________________________________________________________________

        _____________________________________________________________________
    }

    sublist      = middle->next;
    middle->next = NULL;           // middle node ends one sublist

    _____________________________________________________________________

    _____________________________________________________________________
}
```

Problem 4, continued:

```
typedef struct element_t element_t;
struct element_t {
    int32_t    value;
    element_t* next;
};
```

(Structure definition replicated for your convenience.)

Not all blanks may be needed. You may **USE ONLY THE BLANKS PROVIDED**. Additional code will earn no credit.

2. **(8 points)** Complete the recursive function below to merge two sorted lists, **`fst`** and **`sec`**, into a single sorted list and return a pointer to the merged list. **All lists are sorted in decreasing order of their value fields.**

```
element_t* merge_list (element_t* fst, element_t* sec)
{
    element_t* result;

    if (fst == NULL)

        return _sec_____________________________________________________;
    if (sec == NULL)

        return _fst_____________________________________________________;

    if (_fst->value > second->value_____________________________________) {

        result = _fst___________________________________________________;

        result->next = _merge list (fst->next, sec)_____________________;
    } else {

        result = __sec__________________________________________________;

        result->next = _merge list (fst, sec->next)_____________________;
    }
    return result;
}
```

3. **(10 points)** Now write merge sort. Complete the recursive function below to sort the list pointed to by the parameter **`head_ptr`** using merge sort and replace it with a pointer to the sorted list. The original list may be empty. Your code **MUST USE** the helper functions that you wrote in **Part 1** and **Part 2**.

```
void merge_sort (element_t** head_ptr)
{
    element_t* fst;
    element_t* sec;

    if (_NULL == *head ptr || NULL == (*head ptr)->next_____________) {
        return;
    }
    _divide list (*head ptr, &fst, &sec);_____________ _____________

    _merge sort (&fst);_____________________________________________

    _merge sort (&sec);_____________________________________________

    _*head_ptr = merge_list (fst, sec);_____________________________
}
```

dynamic allocation routines from C's standard I/O library

```
// returns pointer to new memory, or NULL on failure
void* malloc (size_t size);

// returns pointer to 0-filled new memory, or NULL on failure
void* calloc (size_t nmemb, size_t size);

// returns pointer to resized block, or NULL on failure
void* realloc (void* ptr, size_t size);

// frees previously allocated block
void free (void* ptr);
```