Pre-requisite: knowledge about how to implement doubly linked list data structure

size_t : It's a type which is used to represent the size of objects in bytes and is therefore used as the return type by the sizeof operator. (Good to know: It is guaranteed to be big enough to contain the size of the biggest object the host system can handle).

void initmem(strategies strategy, size_t sz) :

It is important to notice here that initmem receives the strategy and size of memory (total size of the memory pool)

The fundamental task of this function is to initialize chosen strategy and the size of memory. If memory has already been initialized before, it has to be erased the data structures, before the new size can be initialized.

Not only is it necessary to erase the memory size, but each block initialized before must also be released. After, and only after, releasing the other memory, used for keeping track of the doubly link list, the memory can be re-initialized. The memory size must be allocated as one block of a certain size for example 500 bytes(as done on line no 505 of mymem.c).

In the example bellow, the shaded part is allocated memory which has been allocated previously (75,50,90). Note: the figure doesn't represent the head and last pointers, which doubly link list normally have:

The line

```
if (myMemory != NULL) free(myMemory);
```

Does free the `myMemory` that was dynalically allocated with `malloc(sz)` but rest of the data structures that were being used to manage `myMemory` like `head`, `last etc`. and these corresponding data members, which are in place, needs to be freed.

So, here you have 2 things to do

After the statement below, on line 56 in mymem.c

```
if (myMemory != NULL) free(myMemory);
```

You need to `release any other memory you were using for bookkeeping when doing a re-initialization! */`

Check if head (already declared globally) is not null meaning there is memory allocation of atleast one block.

So something like

```
if (head != NULL)
```

Create a way (pointer) to traverse through the doubly link list.(hint: you should do something similar to line 31 in mymem.c `- struct memoryList *trav;`)

For traversal you need a for loop which would check the member called next of the doubly link list is not null; free() the member called last and then free() the created pointer to traverse.

To traverse through the list –

```
for(trav=head;trav->next != NULL; trav=trav->next)
```

and then you would ofcourse release the memory every time by

```
free(trav->last);
```

and then you free trav

```
free(trav);
```

Once you have learnt this, you can use malloc to allocate memory of the size (sz) and point it into `myMemory`

Good to know: *malloc*() shall return a pointer to the allocated space

```
/* TODO: initialize your bookkeeping structures... */
```

As you use to create a new node in doubly link list, you would do the same here.

Refer to the guide by Armandas.

```
void *mymalloc(size_t requested):
```

last= NULL	500	next= NULL

As the function *initmem(strategy, total size)* have initialized the structure of the memory, we are now ready to allocate memory blocks into this memory. This needs to be implemented by first making sure that the requested block is not larger in size than the available memory. Is the requested block size less than the available memory size.

OBS. A block must be marked, when being allocated, to keep track of the blocks.

OBS. The return type of the function, consider what you want to return.

Block memory must be allocated as the given strategy requires --> Strategies

I have not specified which variables you would need to initialized, I assume from the explanation you would be able to deduce that.

So firstly, start by checking if the first node has been created, or in other words, head is not null.

Implementing the strategy assigned to you.

First–fit:

Loop condition: if the size of the block requested is greater than the first node (hint: you should have done something similar to `static struct memoryList *head;` so that you can compare to the memory block you created as a first node for instance, initialize it to `head`. Let's call it trav).

In the same conditional, you should also check that the trav->alloc is not 1

Then traverse to the next block

Create a new node with malloc ……*very important similar to how you did in first node reference from Armandas's guide.

Connect that new node (let's call is temp)such that you have this new node as the starting block and unallocated part at the end (this means that the `last` of the new node points to the pointer you created as mentioned as hint in the loop condition).

For connecting the new node you will have to

1. `trav = head;`
2. Put your thinking hats on
    a. You need to skip two types of blocks

        i. `((trav->size < req)`
        ii. `(trav->alloc == 1))`
3. Traverse to find the one which you can allocate
4. Create a new new, call it temp
5. If the `((trav->size > req)`
    a. `temp->last= trav //` if you use this line your code you have to explain what you are doing here in a comment to this line (Mandatory)
    b. similarly `temp->next= trav->next`
6. Now consider the scenario when `trav->next` is not null, it means there is a node ahead of trav
7. Do the following
    a. `trav->next->last = temp;` if you use this line your code you have to explain what you are doing here in a comment to this line (Mandatory)
8. `trav->next = temp;`
9. set all the rest of the members of the temp to trav
    a. remember setting `temp->size`
        i. `temp->size = trav->size - req;`
        ii. `temp->ptr = trav->ptr+req;`
    b. set alloc to 0 and 1 for temp and trav, which one should be 0 and which one should be one from the above code
    c. set `trav->size = req;`
10. return `trav->ptr`

Similarly for Best, worst and next fit you would do something similar(remember in next you have to start from the current node.)

void myfree(void* block)

As mentioned in *initmem()* a block must be freed, where *free()* must not be implemented but just called in *myfree()* to free the selected block in the memory. This function must be able to locate a memory block and then delete it. Remember to remove the mark from when the block was allocated.
In order to free the memory,

1. you should start by doing something similar to `static struct memoryList *head;` so that you can compare to the memory block you created as a first node for instance, initialize it to `head`. Let's call it trav
2. Find the block in the list
    a. Loop until `trav->ptr` is same as block
3. Change the alloc for trav to 0
4. Do the following
    a. `if ((trav->last != NULL) && (trav->last->alloc == 0))` ; if you use this line in code; comment what this line is doing
    b. `trav->last->size += trav->size;` if you use this line in code; comment what this line is doing
5. `temp = trav;`
6. you need to check if this was a last node inside the if statement

a. if it is not null that you need to change the `next` pointer of current node & `last` pointer of the next node to skip the node to be freed. (I hope by now you know how to do that)
b. `free(temp);`

7. similary
   a. `if ((trav->next != NULL) && (trav->next->alloc == 0))`
   b. `trav->last->size += trav->size;` if you use this line in code; comment what this line is doing
   c. `temp=trav->next`
   d. if next node's `next` pointer is not null that you need to change the `next` pointer of current node & `last` pointer of the next node to skip the node to be freed. (I hope by now you know how to do that)
   e. `free(temp);`

mem_holes():

How many free blocks are in memory?
This function returns the number of contiguous areas of free space in memory.
Traverse through the list and count the number of allocs are 0

mem_allocated():

How much memory is currently allocated?
Use this function to get the number of bytes allocated in memory as integer.
Traverse through the list and count the number of allocs are 1

mem_free():

How much memory is NOT allocated?
Use this function to get the number of bytes available in the memory as integer.
Traverse through the list and add the size of all allocs which are 0

Mem_largest_free() :

How large is the largest free block?
In this function you should think about going through the memoryList and check for longest sequence of unallocated bytes(alloc = 0). Keep track of the largest sequence and update it accordingly.

mem_small_free(int size):

How many small unallocated blocks are currently in memory compared to the size provided in function
Same logic as above just instead of max compare with size

Testrunner.c

With the testrunner file provided you can test your implementation to see if it passes the tests. This can be done by running "./mem -test <test> <strategy>" as example you could try and run all of them with worst-fit. This would look like this "./mem -test all worst". This will run all the tests for the worst fit algorithm. If your strategy works, the test will pass.
It is good to run the test, because it makes sure to catch mistake you might have overlooked. It can sometimes be hard to decode the "./mem -try <strategy> output and see if it actually works properly.

Memorytest.c

With this class, you can test the memory in different ways. There are multiple methods that performs various tests: one method can do a randomized test, another does a stress tests. Either you can invoke all tests by writing 'all' as an argument when running or you can specify the test with its suite that you would like to run.