Problem 1 (2 parts, 22 points)  

Part A (14 points) Consider the following C function. Using the MIPS register linkage convention, indicate how many load/store operation pairs $N$ (to save and restore the register contents) would be required if the given variable were assigned to a T register or an S register.

```c
int bar(int, int); /* prototype of bar */
int foo(int sum) {
    int i, u, v, r = 1;

    u = sum * SCALE; /* SCALE is a constant */
    for (i = 0; i < 100; i++)
        for (j = 0; j < 100; j++)
            v = r * i;
            v = bar(v, i);
            r = r / (u * v);
}
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>$N$ if assigned to S register</th>
<th>$N$ if assigned to T register</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>1</td>
<td>10/00</td>
</tr>
<tr>
<td>v</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>10/00</td>
</tr>
<tr>
<td>r</td>
<td>1</td>
<td>10/00</td>
</tr>
</tbody>
</table>

Part B (8 points) Consider the following C code. Indicate the number of instructions saved in bar and in foo if the MIPS register convention for call ($s3-s0$) and return ($v1, v0$) is used compared to passing arguments and return values on the activation frame. (Assume all argument are referenced in foo)

```c
int * foo(int i, int *p, float x, int A[], float z);  
void bar() {
    int l, m, *n, B[10];
    float u, v;

    code block 1

    l = foo(m, n, u, B, v);

    code block 2
}
```

<table>
<thead>
<tr>
<th>Instructions saved in bar</th>
<th>Instructions saved in foo</th>
</tr>
</thead>
<tbody>
<tr>
<td>call 4</td>
<td>4</td>
</tr>
<tr>
<td>return 1</td>
<td>1</td>
</tr>
</tbody>
</table>
Problem 2 (7 parts, 28 points)

Dynamic Memory Allocation on Heap

Consider a memory allocator (malloc and free), such as described in class. Inside the C-code for the allocator, unsigned *heapPtr is the address of the next word that could be allocated to the heap, and unsigned **freePtr is the address of the first block on the free list (and the word at the address of each block on the free list is a pointer to the next block on the free list). The allocator uses a best fit strategy with an unsorted free list, and never splits blocks.

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
<th>addr</th>
<th>value</th>
<th>addr</th>
<th>value</th>
<th>addr</th>
<th>value</th>
<th>addr</th>
<th>value</th>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>8000</td>
<td>8</td>
<td>8032</td>
<td>20</td>
<td>8064</td>
<td>4</td>
<td>8096</td>
<td>8</td>
<td>8048</td>
<td>8128</td>
<td>8</td>
<td>8160</td>
</tr>
<tr>
<td>8004</td>
<td>1234</td>
<td>8036</td>
<td>0</td>
<td>8068</td>
<td>12</td>
<td>8100</td>
<td>8104</td>
<td>8132</td>
<td>8004</td>
<td>8164</td>
<td>0</td>
</tr>
<tr>
<td>8008</td>
<td>4</td>
<td>8040</td>
<td>43</td>
<td>8072</td>
<td>8036</td>
<td>8104</td>
<td>4</td>
<td>8136</td>
<td>4</td>
<td>8168</td>
<td>22</td>
</tr>
<tr>
<td>8012</td>
<td>16</td>
<td>8044</td>
<td>12</td>
<td>8076</td>
<td>8144</td>
<td>8108</td>
<td>2</td>
<td>8140</td>
<td>42</td>
<td>8172</td>
<td>7000</td>
</tr>
<tr>
<td>8016</td>
<td>8072</td>
<td>8048</td>
<td>8096</td>
<td>8080</td>
<td>8</td>
<td>8112</td>
<td>12</td>
<td>8144</td>
<td>43</td>
<td>8176</td>
<td>12</td>
</tr>
<tr>
<td>8020</td>
<td>8052</td>
<td>8052</td>
<td>12</td>
<td>8084</td>
<td>4</td>
<td>8116</td>
<td>0</td>
<td>8148</td>
<td>427</td>
<td>8180</td>
<td>41</td>
</tr>
<tr>
<td>8024</td>
<td>8132</td>
<td>8056</td>
<td>8</td>
<td>8088</td>
<td>0</td>
<td>8120</td>
<td>4</td>
<td>8152</td>
<td>8</td>
<td>8184</td>
<td>40</td>
</tr>
<tr>
<td>8028</td>
<td>8116</td>
<td>8060</td>
<td>8116</td>
<td>8092</td>
<td>16</td>
<td>8124</td>
<td>30</td>
<td>8156</td>
<td>0</td>
<td>8188</td>
<td>0</td>
</tr>
</tbody>
</table>

Suppose heapPtr = 8128 and freePtr = 8016. Consider each part below independently.

Part A: (4 pts)

How many blocks and usable bytes are on the free list? blocks = 3 bytes = 48

Part B: (12 pts)

B.1) What value would be returned by the call malloc(10); 80+2

B.2) Which (if any) values in the above map would be changed by the call in B.1?

addr 8016 value 8036 addr value addr value no change

(fill in the address/value pairs above. There may be more pairs than needed)

B.3) Fill in the values after the call in B.1: heapPtr = 8128 freePtr = 8016

Part C: (8 pts)

C.1) What value would be returned by the call malloc(22); 8132

C.2) Which (if any) values in the above map would be changed by the call in C.1?

addr 8128 value 24 addr value addr value no change

Part D: (4 pts)

Which (if any) values in the above map would be changed by the call free(8116)?

addr 8116 value 8016 addr value addr value no change
Problem 3 (4 parts, 20 points)  

Consider the following three heap management strategies:

1. First fit with free list sorted by increasing size (smallest to largest).
2. First fit with free list sorted by decreasing size (largest to smallest).
3. Best fit with unsorted free list.

Part A (5 points) Which strategy (1, 2, or 3) has fastest average speed of `malloc`? 

Why? Either first block on free list fits or grow the heap.

Part B (5 points) Which strategy (1, 2, or 3) has slowest average speed of `malloc`? 

Why? All blocks on free list must be searched (unless there is an exact fit).

Part C (5 points) Which strategy (1, 2, or 3) has fastest average speed of `free`? 

Why? Freed block are placed on the head of the list.

Part D (5 points) Which strategy (1, 2, or 3) has the worst internal fragmentation? 

Why? This is worst fit strategy. There is a high probability of significant unused space within blocks.
Problem 4 (6 parts, 30 points)

Suppose we have the following definition which is used to create singly-linked lists.

```c
typedef struct Element {
    int    Num;
    struct Element *Next;
} Link;
```

Part A (4 points) Fill in the blank below to allocate space for a Link structure using malloc and make the variable L1 point to the object allocated.

```c
Link *L1 = malloc(sizeof(Link));
```

Part B Complete the following function which takes a pointer to a pointer to the first Link of a linked list (called Headptr) and a pointer (called NewEnd) to a Link. The function places NewEnd on the linked list as its last element. Follow the steps specified below.

```c
void Append(Link **Headptr, Link *NewEnd) {
    Link *ThisLink = *Headptr;
    NewEnd->Next = NULL;  /* part B.1 */

    if (ThisLink == NULL) *Headptr = NewEnd;  /* List was empty */
    else {
        while (ThisLink->Next != NULL) ThisLink = ThisLink->Next;  /* part B.2 */
        ThisLink->Next = NewEnd;  /* part B.3 */
    }
}
```

Part B.1 (4 points) Set the link to be appended (NewEnd) to terminate the modified list

Part B.2 (8 points) Walk the list and set ThisLink to point to the last link in the list

Part B.3 (4 points) Append the link pointed to by NewEnd to the list

Part C The following subroutine should free up all elements in the linked list whose first Link is pointed to by the input parameter Head. What error does it make? Write the correct code below.

```c
void FreeElements(Link *Head) {
    Link *h;
    for (h = Head; h != NULL; h = h->Next)
        free(h);
}
```

Part C.1 (4 points) What is the error?

```c
Dangling Pointer. Block is freed and then pointer to block is dereferenced.
```

Part C.2 (6 points) Write the correct version of FreeElements?

```c
void FreeElements(Link *Head) {
    Link *h;
    while (Head) {
        h = Head->Next;
        free(Head);
        Head = h;
    }
```