1. (Sorting, 30%)

The following pseudo code implements some sorting technique:

```
SOME_SORT(A, p, r)
  If p < r then
    q = PARTITION (A, p, r)
    SOME_SORT(A, p, q)
    SOME_SORT (A, q+1, r)

PARTITION(A, p, r)
  x = A[p]
  i = p-1
  j = r+1
  while TRUE
      repeat j = j -1
        until A[j] <= x
      repeat i = i+1
        until A[i] >= x
      if i < j then
        exchange A[i] and A[j]
      else
        return j
```

(1) (5%) What is the name of the sorting technique described by the procedure *SOME_SORT* ?

(2) (5%) Show that the computation complexity of the procedure *PARTITION* is $O(n)$ where $n = r-p+1$.

(3) (5%) The *worst-case* behavior for the above sorting method occurs when the partitioning routine *PARTITION* produces one region with $n-1$ elements and the other region with only 1 element. What is the worse-case computation complexity of the sorting method?

(4) (5%) The *best-case* behavior of the sorting method is when the partition procedure produces two regions of equal size of $n/2$. What is the best-case computation complexity of the sorting method?

(5) (5%) Is the *average-case* computation complexity of the sorting method closer to the worst-case situation or the best-case situation? Why?

(6) (5%) Please give the names of five other different sorting methods you can think of.

2. (Tree, 20%)

A B-tree T is a rooted tree (with root denoted by *root[T]*) having the following properties:

(i) Every node x has the following fields:

a. $n[x]$, the number of keys currently stored in node x,

b. $key_1[x]$, $key_2[x]$, ..., $key_{n[x]}[x]$, the $n[x]$ keys stored in node x in non-decreasing order: $key_1[x] \le key_2[x] \le \Lambda \le key_{n[x]}[x]$, and

c. *leaf[x]*, a Boolean value which is TRUE if x is a leaf and FALSE if x is an internal node.

d. If x is an internal node, it also contains $n[x]+1$ pointers $c_1[x]$, $c_2[x]$, ..., $c_{n[x]+1}[x]$ to x's children. Leaf nodes have no children, so their c_i fields are undefined.

(ii) The keys $key_i[x]$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $c_i[x]$, then $k_1 \le key_1[x] \le k_2 \le key_2[x] \le \Lambda \le key_{n[x]}[x] \le k_{n[x]+1}$.

(iii) Every leaf has the saem depth, which is the tree's height h.

(iv) There are lower and upper bounds on the number of keys a node can contain in terms of a fixed integer $t \ge 2$ called the **minimum degree** of the B-trees.

a. Every node other than the root must have at least $t-1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

b. Every node can contain at most $2t-1$ keys. Therefore an internal node can have at most $2t$ children. We say that a node is **full** if it contains exactly $2t-1$ keys.

(1) (5%) What is the value of t when the simplest case of B-tree occurs? In this case, how many children does every internal node have?

(2) (10%) Please prove the following worst-case height of a B-tree: If $n \ge 1$, then for any n-key B-tree T of height h and minimum degree $t \ge 2$, $h \le \log_t \frac{n+1}{2}$.

(3) (5%) Write a pseudo code to implement the operation of *B_TREE_SEARCH(x, k)*.

3. ALU Design (10%)

An ALU has two four-bit inputs, A and B, and one four-bit output C. A, B are **signed** integers, represented as two's complement binary numbers. The most significant bits of A, B and C are A[3], B[3] and C[3], respectively; while the least significant bits are A[0], B[0] and C[0], respectively. In addition, the ALU also generates two flag signals: OV (overflow) and LESS (less than). The ALU can be used to compare the two inputs by performing a subtraction C=A – B. The signal LESS is set to one if A is less than B (zero, if not). Derive the logic equation for the signal LESS based on the signal value of OV and/or the bit values of C.

4. (20%) Instruction Set

It is possible to imagine even simpler instruction sets. In this assignment, you are to consider a hypothetical machine called SIC, for Single Instruction Computer. As its name implies, SIC has only one instruction: subtract and branch if negative, or sbn for short. The sbn instruction has three operands, each consisting of the address of a word in memory:

```
sbn a,b,c # Mem[a] = Mem[a] - Mem[b];if (Mem[a]<0) go to c
```

The instruction will subtract the number in memory location b from the number in location a and place the result back in a, overwriting the previous value. If the result is greater than or equal to 0, the computer will take its next instruction from the memory location just after the current instruction. If the result is less than 0, the next instruction is taken from memory location c. SIC has no registers and no instructions other than sbn.

Although it has only one instruction, SIC can imitate many of the operations of more complex instruction sets by using clever sequences of sbn instructions. For example, here is a program to copy a number from location a to location b:

```
start: sbn temp,temp,.+1    # Sets temp to zero
       sbn temp,a,.+1       # Sets temp to -a
       sbn b,b,.+1          # Sets b to zero
       sbn b,temp,.+1       # Sets b to -temp, which is a
```

In the program above, the notation .+1 means "the address after this one," so that each instruction in this program goes on to the next in sequence whether or not the result is negative. We assume temp to be the address of a spare memory word that can be used for temporary results.

4.1 (10%) Write a SIC program to add a and b, leaving the result in a and leaving b unmodified.

4.2 (10%) Write a SIC program to multiply a by b, putting the result in c. Assume that memory location one contains the number 1. Assume that a and b are greater than 0 and that it's OK to modify a or b. (Hint: What does this program compute?)

```
c = 0; while (b > 0) {b = b - 1; c = c + a;}
```

6. Computer Performance (20%)

Consider two different implementations, M1 and M2, of the same instruction set. There are three classes of instructions (A, B, and C) in the instruction set. M1 has a clock rate of 400 MHz, and M2 has a clock rate of 200 MHz. The average number of cycles for each instruction class on M1 and M2 is given in the following table:

Class	CPI on M1	CPI on M2	C1 usage	C2 usage	Third-party usage
A	4	2	30%	30%	50%
B	6	4	50%	20%	30%
C	8	8	20%	50%	20%

The table also contains a summary of how three different compilers use the instruction set. C1 is a compiler produced by the makers of M1, C2 is a compiler produced by the makers of M2, and the other compiler is a third-party product. Assume that each compiler uses the same number of instructions for a given program but that the instruction mix is as described in the table. Using C1 on both M1 and M2, how much faster can the makers of M1 claim that M1 is compared with M2? Using C2 on both M2 and M1, how much faster can the makers of M2 claim that M2 is compared with M1? If you purchase M1, which compiler would you use? If you purchase M2, which compiler would you use? Which machine would you purchase if we assume that all other criteria are identical, including costs?

Note: CPI = cycles per instruction