Lab 4

Due No Due Date **Points** 1

You are suggested to use the teaching servers burrow.soic.indiana.edu or hulk.soic.indiana.edu or tank.soic.indiana.edu for practicing C programs.

Lab 4: Heap Sort algorithm

Recursion

Recursion is the magical tool supplied to the developer by all high level programming languages. Whenever we can divide our problem into sub-problem we can use recursion. By using recursion, we can code our program with ease and produce clean and readable codes.
Every recursive function must have a base case and a recursive case.
For example, about Fibonacci number we can say that: F[1] = 1, F[2] = 1, F[n>=3] = F[n-2]+F[n-1]
We can write a recursive function to calculate that:

```
int FIBO(int n)
{
        // Base case
        if (n==1 || n==2)
                return 1;
        else
                // Recursive case
                return FIBO(n-1)+FIBO(n-2);
}
```

Without defining the base case, a recursive function will go forerver.
All recursive code can be replaced by alternative loop version, for example we can write a looped version to calculate the fibonacci rather than the recursive case. But most of the cases, writing a loop version would be much harder to write than the recursive version.
For example, we can write a recursive function that print out all possible permutations of a string say "abcd" as

```
abcd
abdc
acbd
acdb
adbc
adcb
bacd
badc
bcad
bcda
bdac
bdca
cabd
cadb
cbad
cbda
cdab
cdba
```

```
dabc
dacb
dbac
dbca
dcab
dcba
```

Here is the code

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char *flag, *output;
char s[] = "abcde";

printperm(int index, int len)
{
        int i;
        if (index == 0 && len == strlen(s))
                printf("%s\n", output);

        if (flag[index]) return;
        flag[index] = 1;
        output[len] = s[index];

        for(i=0; s[i]; ++i)
                printperm(i, len+1);

        flag[index] = 0;
}

int main()
{
        int i, len;
        len = strlen(s);
        flag = (char *) malloc(len);
        output = (char *) malloc(len+1);

        memset(flag, 0, len);
        output[len] = 0;
        for(i=0; s[i]; ++i)
                printperm(i, 0);
}
```

If we try to do the same without recursion, there is definitely possible ways to do so, but it won't be as easy as it is with recursion.

Heap Sort

Heap sort uses a data structure called heap for sorting. Like merge sort, but unlike insertion sort, heapsort's running time is **O(nlgn)**. Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time. Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

Here we use an array implementation of **max** binary heap like in the book(page 152). Note that the array indices start with **0** rather than 1 as in the book.

An array A that represents a heap is an object with two attributes: A:length, which (as usual) gives the number of elements in the array, and A:heap-size, which represents how many elements in the heap are stored within array A. That is, although A[0...length-1] may contain numbers, only the elements in A[0...heap_size-1] are valid elements of the heap.

Pseudo Code:

```
heap A = {
     array[0 ... lenngth-1];
     length;
     heap_size;
}

left(i):{         //i is the node index in the array
    return 2*i+1  // returns the index of the left child of node i
}

right(i):{
    return 2*i+2  // returns the index of the right child of node i
}

parent(i):{
 return (i-1)/2 // returns only the integer part of the value
}

float_down(A,i): // push node i downwards the binary heap if it violates the heap property.
{
 // Note the precondition of this function is that the left and right subtrees of node i already satisfy th
e heap property
 // That is node i is the only node that possibly violates the heap property currently
 // So we need to push node i downward of the heap to restore the heap property
 l = left(i)
 r = right(i)
 max = the index of the node that has the maximum value among node l,r,i

 if(max != i): // if the parent node is not bigger than the child nodes
               // which violates the max-heap property
   swap(heap[i],heap[max]) // correct the violation by swapping the bigger node to the parent position
   float_down(heap,max) // check recursively if the heap property is valid in the subtree
}

Complexity = O(log(heap_size))

build_max_heap(A): // adjust array A so that it satisfies the heap propery
{
 A.heapsize = A.length
 for i from parent(A.heapsize-1) to 0: //starting from the second last level of the heap tree up to the roo
t
                            //the bottom-up order is crucial here
   float_down(A,i) // make sure every node satisfy the heap property
                   // in the end, the array is adjusted to be a heap
}

Note that, the Complexity = O(A.length)

Heap_sort(A): // input array A
{
 build_max_heap(A); // make A into a max heap, so A[0] is the maximum of the HEAP elements
 for i from A.length-1 to 1:
   swap(A[0],A[i]) // put the maximum at the end of the heap
```

```
   A.heap_size = A.heap_size - 1 // remove the last element from the heap(still on the array)
   float_down(A,0) // adjust the heap to restore heap property
}
Let n = A.length
Conmplexity = O(n) + n*O(logn) = O(nlogn)
```