Problem Set 4
Pointers. Arrays. Strings. Searching and sorting algorithms.

Problem 4.1

Consider the insertion sort algorithm described in Lecture 5 In this problem, you will re-implement the algorithm using pointers and pointer arithmetic.(a) The function shift element() takes as input the index of an array element that has been determined to be out of order. The function shifts the element towards the front of the array, repeatedly swapping the preceding element until the out-of-order element is in the proper location. The implementation using array indexing is provided below for your reference:

```
/∗ move pr evi ous e l ement s down u n ti l
i n s e r ti o n point reached ∗/
void s h i ft e l e m e n t (unsigned int i ) {
int i v a l u e ;
/∗ guard a g a ins t going out s ide ar ray ∗/
for ( i v a l u e = a r r [ i ] ; i && a r r [ i −1] > i v a l u e ; i −−)
a r r [ i ] = a r r [ i −1]; /∗ move element down ∗/
a r r [ i ] = i v a l u e ; /∗ i n s e r t element ∗/
}
Re-implement this function using pointers and pointer arithmetic instead of array indexing.
/∗ i n t ∗pElement − po int e r to the element
in a r r ( type i n t [ ] ) that i s out−of−pl a c e ∗/
void s h i ft e l e m e n t ( int ∗pElement) {
/∗ i n s e r t code he r e ∗/
}
```

<u>ANSWER:</u>

```
void shift_element ( int ∗pE) {
int ivalue = ∗pE ;
for ( ivalue = ∗pE ; pE > a r r && ∗( pE −1) > ivalue ; pE−−)
∗pE = ∗( pE −1);
∗(pE-1) = ivalue ;
}
```

(b) The function insertion sort() contains the main loop of the algorithm. It iterates through elements of the array, from the beginning, until it reaches an element that is out-of-order. It calls shift element() to shift the offending element to its proper location earlier in the array and resumes iterating until the end is reached. The code from lecture is provided below:

```
/∗ it e r a t e u n ti l out−of−orde r element found ;
```

```
s h i ft the element , and continue it e r a ti n g ∗/
void insertionsort ( void) {
unsigned int i , l en = a r r a y l e n g t h ( a r r );
for ( i = 1 ; i < l en ; i++)
i f ( a r r [ i ] < a r r [ i −1])
s h i ft e l e m e n t ( i );
}
```

Re-implement this function using pointers and pointer arithmetic instead of array indexing.Use the shift element() function you implemented in part (a).

<u>ANSWER:</u>

```
void insertion_sort(void)
{
      int *i, len = arraylength(arr);
        for (i= arr +1;i <len; i++)
          if (∗i < ∗( i −1))
            shift_element(i);

}
```

Problem 4.2

In this problem, we will use our knowledge of strings to duplicate the functionality of the C standard library's strtok() function, which extracts "tokens" from a string. The string is split using a set of delimiters, such as whitespace and punctuation. Each piece of the string, without its surrounding delimiters, is a token. The process of extracting a token can be split into two parts: finding the beginning of the token (the first character that is not a delimiter), and finding the end of the token (the next character that is a delimiter). The first call of strtok() looks like this:

```
char ∗ strtok(char ∗ str, const char ∗ delims);
```

The string str is the string to be tokenized, delims is a string containing all the single characters to use as delimiters (e.g. " \t\r\n"), and the return value is the first token in str. Additional tokens can be obtained by calling strtok() with NULL passed for the str argument:

```
char ∗ strtok(NULL, const char ∗ delims);
```

**Because strtok() uses a static variable to store the pointer to the beginning of the next token, calls to strtok() for different strings cannot be interleaved. The code for strtok() is provided
below:**

```
char ∗ strtok( char ∗ text , const char ∗ de lims ) {
/∗ i n i t i a l i z e ∗/
i f (! t ext )
t ext = pnexttoken ;
/∗ fi n d s t a r t o f token in t ext ∗/
t ext += s t r spn ( text , de lims );
i f (∗ t ext == '\0' )
return NULL;
/∗ fi n d end o f token in t ext ∗/
pnexttoken = t ext + s t r c spn ( text , de lims );
/∗ i n s e r t null−t e rminator at end ∗/
i f (∗ pnexttoken != '\0' )
∗ pnexttoken++ = '\0' ;
return t ext ;
}
```

(a) In the context of our string tokenizer, the function strspn() computes the index of the first non-delimiter character in our string. Using pointers or array indexing (your choice), implement the strspn() function. In order to locate a character in another string, you may use the function strpos(), which is declared below:

```
int strpos ( const char ∗ s t r , const char ch );
```

This function returns the index of the first occurrence of the character ch in the string str
or -1 if ch is not found. The declaration of strspn() is provided below:

```
unsigned int s t r spn ( const char ∗ s t r , const char ∗ de lims ) {
/∗ i n s e r t code he r e ∗/
}
```

Here, delims is a string containing the set of delimiters, and the return value is the index of the first non-delimiter character in the string str. For instance, strspn(" . This", " .") == 3. If the string contains only delimiters, strspn() should return the index of the null-terminator
('\0'). Assume '\0' is not a delimiter.

ANSWER:

```
unsigned int str(spn( const char* str, const char * delims)
   {
            unsigned int i;
            for(i = 0; str[i] != '\0' && strpos(delims, str[i]) > -1; i++);
              return i;
    }
```

(b) The function strcspn() computes the index of the first delimiter character in our string.
Here's the declaration of strcspn():

```
unsigned int s t r c spn ( const char ∗ s t r , const char ∗ de lims ) {
/∗ i n s e r t code he r e ∗/
}
```

If the string contains no delimiters, return the index of the null-terminator ('\0'). Implement this function using either pointers or array indexing.

<u>ANSWER:</u>

```
  unsigned int str(spn( const char* str, const char * delims)
{
        unsigned int i;
        for(i = 0; str[i] != '\0' && strchr(delims, str[i]) == -1; i++);
          return i;
 }
```

Problem 4.3
In this problem, you will be implementing the shell sort. This sort is built upon the insertion sort, but attains a speed increase by comparing far-away elements against each other before comparing closer-together elements. The distance between elements is called the "gap". In the shell sort, the array is sorted by sorting gap sub-arrays, and then repeating with a smaller gap size. As written here, the algorithm sorts in O(n2) time. However, by adjusting the sequence of gap sizes used, it is possible to improve the performance to O(n3/2), O(n4/3), or even O(n(log n)2) time. You can read about a method for performing the shell sort in O(n(log n)2) time on Robert Sedgewick's page at Princeton: http://www.cs.princeton.edu/~rs/shell/paperF.pdf Note that you can find complete C code for the shell sort at the beginning of the paper, so please wait until you have finished this exercise to read it. (a) First, we will modify the shift element() function from the insertion sort code for the shell
sort. The new function, started for you below, should start at index i and shift by intervals
of size gap. Write the new function, using array indexing or pointers. Assume that i ≥ gap.

```
void s h ift e l e m e n t b y g a p (unsigned int i , unsigned int gap ) {
/∗ i n s e r t code he r e ∗/
}
```

(b) Now, we need to write the main shell sort routine. Using the template below, fill in the missing code so that the insertion sort in the inner loop compares every element to the element gap spaces before it.

```
void s h e l l s o r t ( void) {
unsigned int gap , i ,
l en = a r r a y l e n g t h ( a r r );
/∗ sor t , comparing a g a ins t f a r t h e r away
e l ement s f i r s t , then c l o s e r e l ement s ∗/
for ( gap = l en / 2 ; gap > 0 ; gap /= 2) {
/∗ do i n s e r ti o n –l i k e sor t , but comparing
and s h i fti n g e l ement s in m u lti p l e s o f gap ∗/
for ( /∗ i n s e r t code he r e ∗/ ) {
i f ( /∗ i n s e r t code he r e ∗/ ) {
/∗ out o f order , do s h i ft ∗/
s h ift e l e m e n t b y g a p ( i , gap );
}
}
}
}
```