

Problem Set 4 – Solutions

Pointers. Arrays. Strings. Searching and sorting algorithms.

Out: Friday, January 15, 2010.

Due: Tuesday, January 19, 2010.

Problem 4.1

Consider the insertion sort algorithm described in Lecture 5 (slides 21-23). 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 previous elements down until
   insertion point reached */
void shift_element(unsigned int i) {
    int ivalue;
    /* guard against going outside array */
    for (ivalue = arr[i]; i && arr[i-1] > ivalue; i--)
        arr[i] = arr[i-1]; /* move element down */
    arr[i] = ivalue; /* insert element */
}
```

Re-implement this function using pointers and pointer arithmetic instead of array indexing.

```
/* int *pElement - pointer to the element
   in arr (type int[]) that is out-of-place */
void shift_element(int *pElement) {
    /* insert code here */
}
```

Answer: Here's one possible implementation:

```
/* int *pElement - pointer to the element
   in arr (type int[]) that is out-of-place */
void shift_element(int *pElement) {
    int ivalue = *pElement;
    for (ivalue = *pElement; pElement > arr && *(pElement-1) > ivalue; pElement--)
        *pElement = *(pElement-1);
    *pElement = 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:

```

/* iterate until out-of-order element found;
   shift the element, and continue iterating */
void insertion_sort(void) {
    unsigned int i, len = array_length(arr);
    for (i = 1; i < len; i++)
        if (arr[i] < arr[i-1])
            shift_element(i);
}

```

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

Answer: Here's one possible implementation:

```

/* iterate until out-of-order element found;
   shift the element, and continue iterating */
void insertion_sort(void) {
    int * pElement, *pEnd = arr + array_length(arr);
    for (pElement = arr+1; pElement < pEnd; pElement++)
        if (*pElement < *(pElement-1))
            shift_element(pElement);
}

```

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 * delims) {
    /* initialize */
    if (!text)
        text = pnexttoken;
    /* find start of token in text */
    text += strspn(text, delims);
    if (*text == '\0')
        return NULL;
    /* find end of token in text */
    pnexttoken = text + strcspn(text, delims);
    /* insert null-terminator at end */
    if (*pnexttoken != '\0')
        *pnexttoken++ = '\0';
    return text;
}

```

- (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 * str, 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 strspn(const char * str, const char * delims) {
    /* insert code here */
}
```

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: Here's a simple implementation of `strspn()`:

```
unsigned int strspn(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 strcspn(const char * str, const char * delims) {
    /* insert code here */
}
```

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

Answer: Here's a simple implementation of `strcspn()`:

```
unsigned int strcspn(const char * str, const char * delims) {
    unsigned int i;
    for (i = 0; str[i] != '\0' && strpos(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 $O(n^2)$ time. However, by adjusting the sequence of gap sizes used, it is possible to improve the performance to $O(n^{3/2})$, $O(n^{4/3})$, 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:

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 $i \geq \text{gap}$.

```
void shift_element_by_gap(unsigned int i, unsigned int gap) {
    /* insert code here */
}
```

Answer: One possible version of this function is:

```
void shift_element_by_gap(unsigned int i,
                          unsigned int gap) {
    int ival;
    unsigned int j;
    for (j = i, ival = arr[i]; j >= gap
         && arr[j-gap] > ival; j -= gap)
        arr[j] = arr[j-gap]; /* move */
    arr[j] = ival; /* insert element */
}
```

- (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 shell_sort(void) {
    unsigned int gap, i,
        len = array_length(arr);
    /* sort, comparing against farther away
       elements first, then closer elements */
    for (gap = len/2; gap > 0; gap /= 2) {
        /* do insertion-like sort, but comparing
           and shifting elements in multiples of gap */
        for (/* insert code here */) {
            if (/* insert code here */) {
                /* out of order, do shift */
                shift_element_by_gap(i, gap);
            }
        }
    }
}
```

Answer: One possible solution is shown below:

```
void shell_sort(void) {
    unsigned int gap, i, len = array_length(arr);
    /* compare against farther away elements first, then closer elements */
    for (gap = len/2; gap > 0; gap /= 2) {
        /* comparing and shifting elements in multiples of gap */
        for (i = gap; i < len; i++)
            if (arr[i-gap] > arr[i])
                shift_element_by_gap(i, gap); /* out of order, do shift */
    }
}
```

MIT OpenCourseWare
<http://ocw.mit.edu>

6.087 Practical Programming in C
January (IAP) 2010

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.