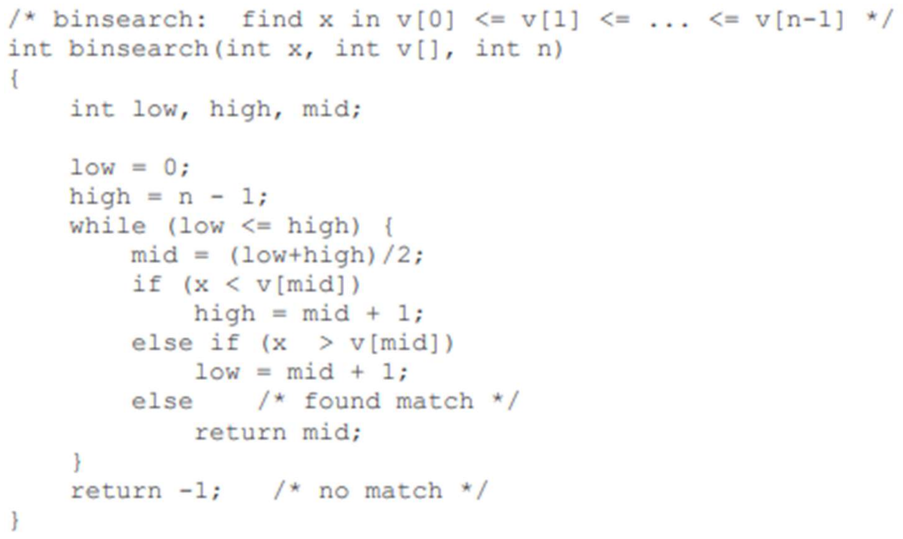
Documentation of Day 4

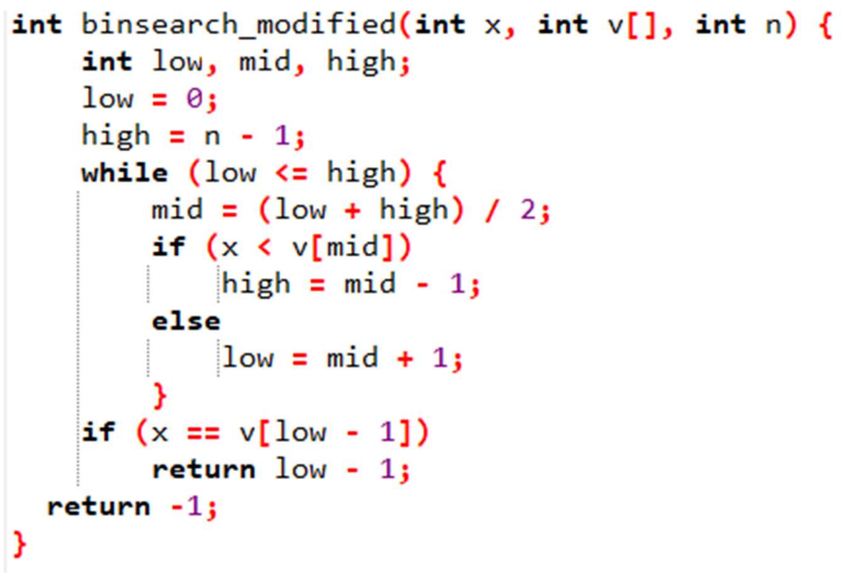
Exercise 3-1: A general binary search makes two tests inside the loop. I have modified that binary search program which only test one inside the loop and the other is tested outside the loop and measured the difference in run-time.

Here is the source code of general/basic binary search:



```
/* binsearch:  find x in v[0] <= v[1] <= ... <= v[n-1] */
int binsearch(int x, int v[], int n)
{
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high)/2;
        if (x < v[mid])
            high = mid + 1;
        else if (x  > v[mid])
            low = mid + 1;
        else    /* found match */
            return mid;
    }
    return -1;   /* no match */
}
```

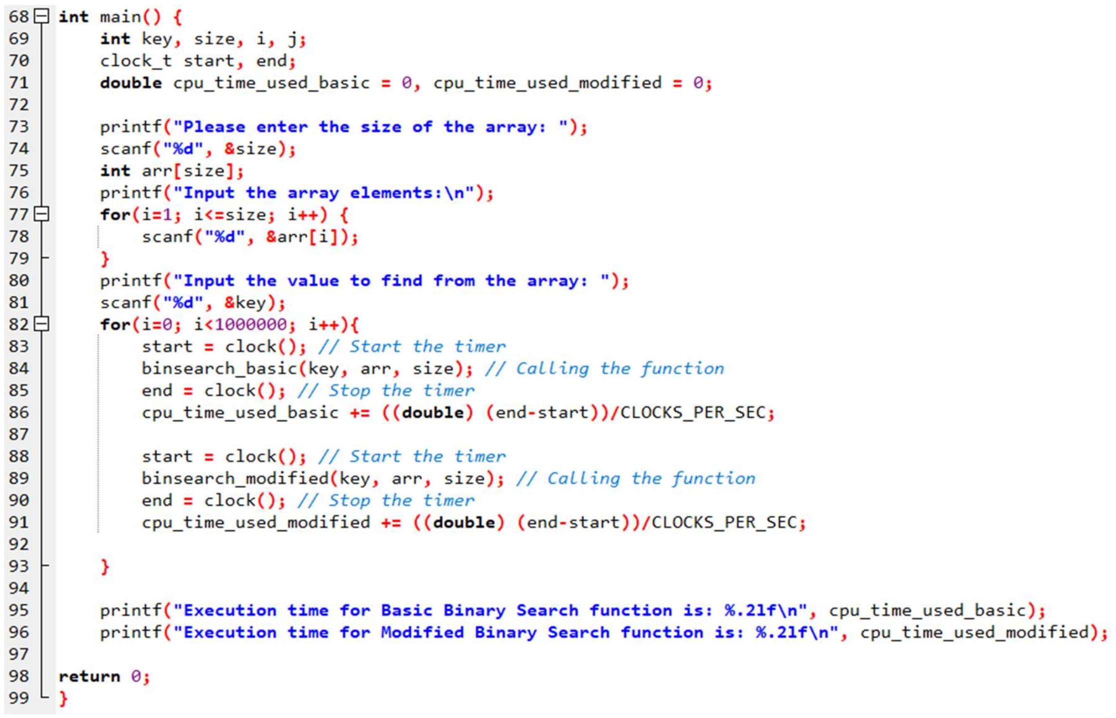
The Modified Binary Search Program is as follows:



```
int binsearch_modified(int x, int v[], int n) {
    int low, mid, high;
    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        if (x < v[mid])
            high = mid - 1;
        else
            low = mid + 1;
        }
    if (x == v[low - 1])
        return low - 1;
  return -1;
}
```

Here inside the while loop there is only one condition and outside the loop there is another condition is checked. Inside a while loop the statements inside the loop is executed as much time as the loop runs. Logically the program that have two conditions the while loop should take more time compared to the program that have one condition inside the loop and another outside the loop. Because outside the loop the executing statements over the condition have a constant time complexity but inside the loop the condition execution time is dependent on the loop. If the loop running time is a very high number then the condition execution statements inside the loop will affect the run-time of the program.

Here the program that calculates the run-time of a function is given:



```c
int main() {
    int key, size, i, j;
    clock_t start, end;
    double cpu_time_used_basic = 0, cpu_time_used_modified = 0;

    printf("Please enter the size of the array: ");
    scanf("%d", &size);
    int arr[size];
    printf("Input the array elements:\n");
    for(i=1; i<=size; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Input the value to find from the array: ");
    scanf("%d", &key);
    for(i=0; i<1000000; i++){
        start = clock(); // Start the timer
        binsearch_basic(key, arr, size); // Calling the function
        end = clock(); // Stop the timer
        cpu_time_used_basic += ((double) (end-start))/CLOCKS_PER_SEC;

        start = clock(); // Start the timer
        binsearch_modified(key, arr, size); // Calling the function
        end = clock(); // Stop the timer
        cpu_time_used_modified += ((double) (end-start))/CLOCKS_PER_SEC;

    }

    printf("Execution time for Basic Binary Search function is: %.2lf\n", cpu_time_used_basic);
    printf("Execution time for Modified Binary Search function is: %.2lf\n", cpu_time_used_modified);

return 0;
}
```

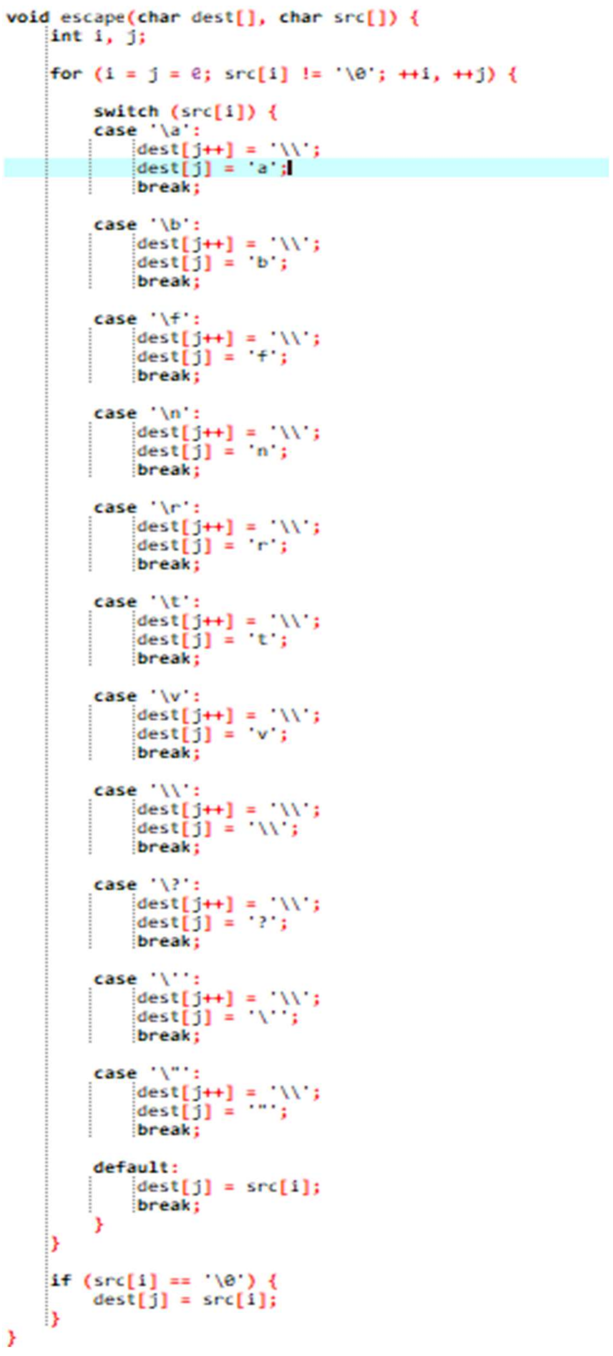
clock_t is a data type defined in the C standard library time.h. It is used to represent clock time or processor time. The clock_t type is capable of representing time values returned by the clock() function.

The clock() function returns the number of clock ticks elapsed since the program started, so we store the start and end times using clock(). We then calculate the execution time by subtracting the start time from the end time and dividing by CLOCKS_PER_SEC to get the time in seconds. We have taken two variables cpu_time_used_basic and cpu_time_used_modified to store the calculated value of general and modified binary search function respectively. Finally, we print the execution time.

Exercise 3-2:

Qunestion: Write a function escape(s,t) that converts characters like newline and tab into visible escape sequences like \n and \t as it copies the string t to s. Use a switch. Write a function for the other direction as well, converting escape sequences into the real characters.

Here's an implementation of the escape() function that converts newline and tab characters and other escape sequences into visible escape sequences:

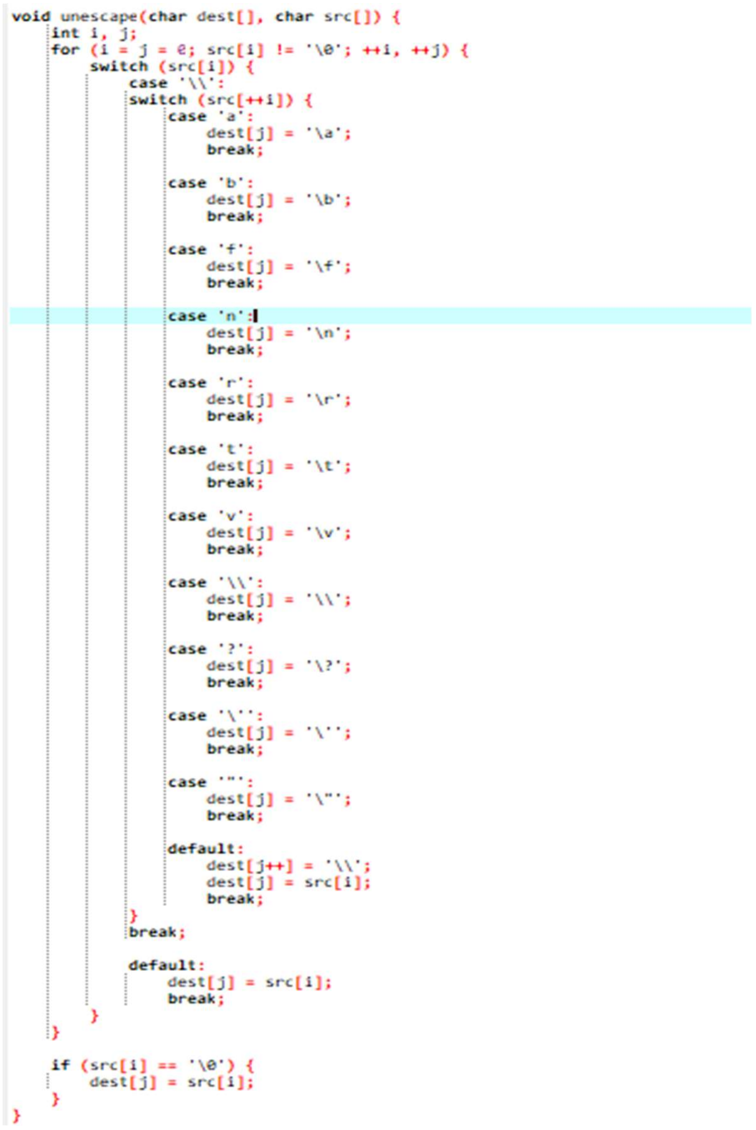


```c
void escape(char dest[], char src[]) {
    int i, j;

    for (i = j = 0; src[i] != '\0'; ++i, ++j) {

        switch (src[i]) {
        case '\a':
            dest[j++] = '\\';
            dest[j] = 'a';
            break;

        case '\b':
            dest[j++] = '\\';
            dest[j] = 'b';
            break;

        case '\f':
            dest[j++] = '\\';
            dest[j] = 'f';
            break;

        case '\n':
            dest[j++] = '\\';
            dest[j] = 'n';
            break;

        case '\r':
            dest[j++] = '\\';
            dest[j] = 'r';
            break;

        case '\t':
            dest[j++] = '\\';
            dest[j] = 't';
            break;

        case '\v':
            dest[j++] = '\\';
            dest[j] = 'v';
            break;

        case '\\':
            dest[j++] = '\\';
            dest[j] = '\\';
            break;

        case '\?':
            dest[j++] = '\\';
            dest[j] = '?';
            break;

        case '\'':
            dest[j++] = '\\';
            dest[j] = '\'';
            break;

        case '\"':
            dest[j++] = '\\';
            dest[j] = '"';
            break;

        default:
            dest[j] = src[i];
            break;
        }
    }

    if (src[i] == '\0') {
        dest[j] = src[i];
    }
}
```

The escape() function takes two character arrays as parameters: s (destination) and t (source). It iterates through the characters of the source string t and checks for newline (\n) and tab (\t) and other escape sequence characters using a switch statement. When it encounters these characters, it appends the corresponding escape sequences to the destination string s. For other characters, it simply copies them to s. Finally, it null-terminates the destination string.

The unescape() function performs the reverse operation. It scans the source string t and when it encounters any escape characters, it checks the next character to identify the escape sequence (n for newline, t for tab, or other characters). It then replaces the escape sequence with the corresponding character in the destination string s. If a backslash is followed by a character other than n or t, it treats it as a literal backslash and copies both the backslash and the following character to s. Finally, it null-terminates the destination string.

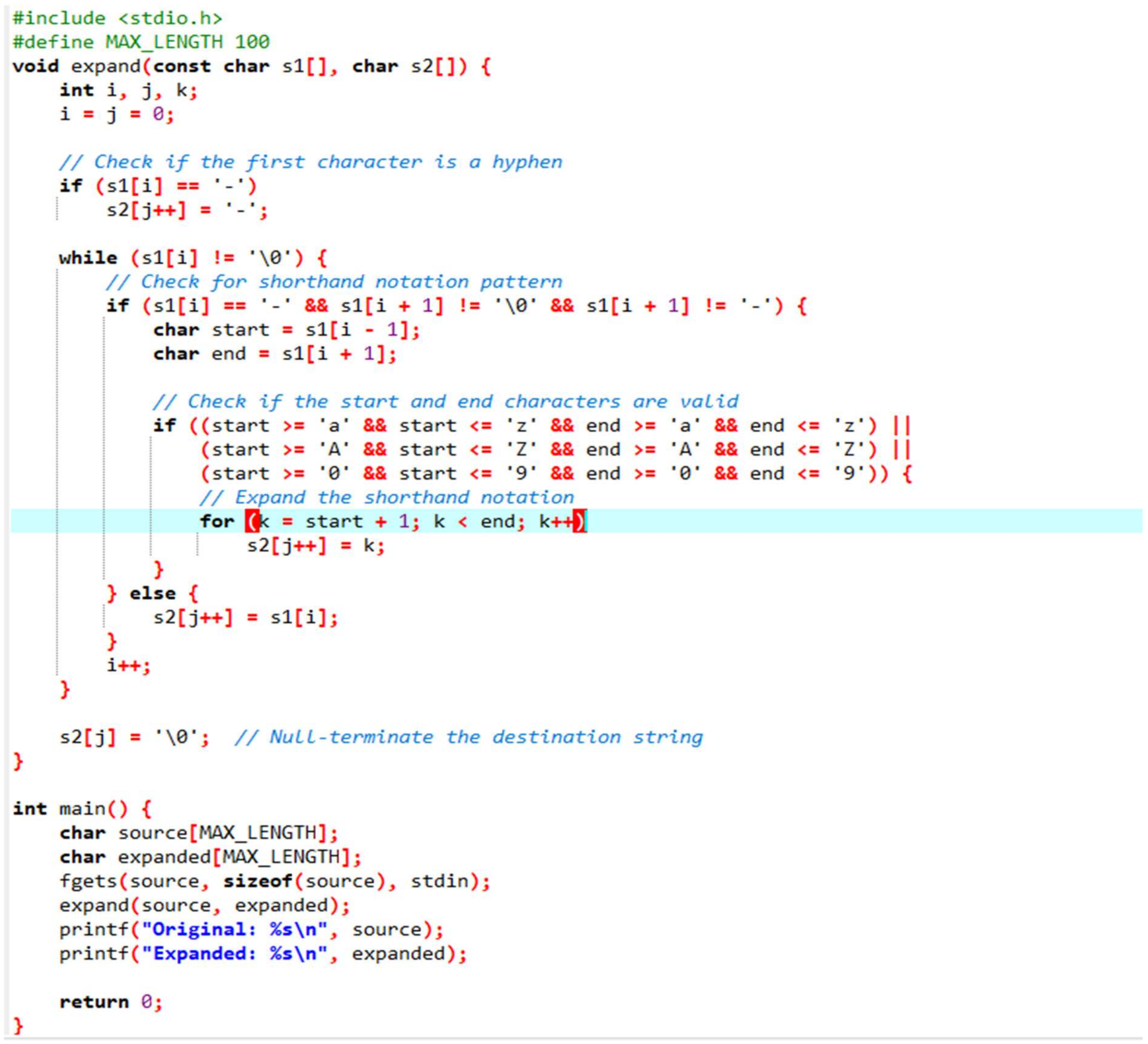
Here is the driver code of unescape() function:



```c
void unescape(char dest[], char src[]) {
    int i, j;
    for (i = j = 0; src[i] != '\0'; ++i, ++j) {
        switch (src[i]) {
            case '\\':
            switch (src[++i]) {
                case 'a':
                    dest[j] = '\a';
                    break;

                case 'b':
                    dest[j] = '\b';
                    break;

                case 'f':
                    dest[j] = '\f';
                    break;

                case 'n':
                    dest[j] = '\n';
                    break;

                case 'r':
                    dest[j] = '\r';
                    break;

                case 't':
                    dest[j] = '\t';
                    break;

                case 'v':
                    dest[j] = '\v';
                    break;

                case '\\':
                    dest[j] = '\\';
                    break;

                case '?':
                    dest[j] = '\?';
                    break;

                case '\'':
                    dest[j] = '\'';
                    break;

                case '"':
                    dest[j] = '\"';
                    break;

                default:
                    dest[j++] = '\\';
                    dest[j] = src[i];
                    break;
            }
            break;

            default:
                dest[j] = src[i];
                break;
        }
    }

    if (src[i] == '\0') {
        dest[j] = src[i];
    }
}
```

<u>Exercise 3-3:</u>

Here is an implementation of the expand() function that expands shorthand notations in a string:



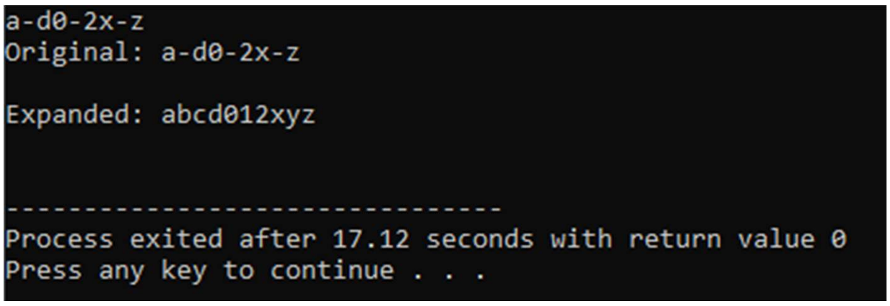
```c
#include <stdio.h>
#define MAX_LENGTH 100
void expand(const char s1[], char s2[]) {
    int i, j, k;
    i = j = 0;

    // Check if the first character is a hyphen
    if (s1[i] == '-')
        s2[j++] = '-';

    while (s1[i] != '\0') {
        // Check for shorthand notation pattern
        if (s1[i] == '-' && s1[i + 1] != '\0' && s1[i + 1] != '-') {
            char start = s1[i - 1];
            char end = s1[i + 1];

            // Check if the start and end characters are valid
            if ((start >= 'a' && start <= 'z' && end >= 'a' && end <= 'z') ||
                (start >= 'A' && start <= 'Z' && end >= 'A' && end <= 'Z') ||
                (start >= '0' && start <= '9' && end >= '0' && end <= '9')) {
                // Expand the shorthand notation
                for (k = start + 1; k < end; k++)
                    s2[j++] = k;
            }
        } else {
            s2[j++] = s1[i];
        }
        i++;
    }

    s2[j] = '\0';  // Null-terminate the destination string
}

int main() {
    char source[MAX_LENGTH];
    char expanded[MAX_LENGTH];
    fgets(source, sizeof(source), stdin);
    expand(source, expanded);
    printf("Original: %s\n", source);
    printf("Expanded: %s\n", expanded);

    return 0;
}
```

The expand() function takes two character arrays as parameters: s1 (source) and s2 (destination). It iterates through the characters of the source string s1 and checks for shorthand notation patterns using an if condition. When it encounters a hyphen (-) that is not at the beginning or end of the string, it extracts the start and end characters for the shorthand notation.

The function then checks if the start and end characters are valid. They can be either lowercase letters (a-z), uppercase letters (A-Z), or digits (0-9). If the characters are valid, it expands the shorthand notation by adding the characters from start to end (inclusive) to the destination string s2.

For example, if s1 contains "a-d0-2", the function will expand it to "abcd012".

The output is as follows:



```
a-d0-2x-z
Original: a-d0-2x-z

Expanded: abcd012xyz



--------------------------------
Process exited after 17.12 seconds with return value 0
Press any key to continue . . .
```