EE3980 Algorithm

HW5. Better Sortings

106061225 楊宇羲

1. Problem Description:

In our first assignment we had already compared four kinds of sortings, the results all point out that they have the time complexity of $O(n^2)$. However, there are some smart people that found a totally different accept to develop some astonishing fast solution toward sortings. Today we are going to use three among these powerful sorting methods, which are `HeapSort()`, `MergeSort()`, `QuickSort()`, respectively, to sort the 9 same datasets as homework 1, which are all made up by $10{\cdot}2^{n-1}$ words, where n represents the number of dataset, and the core concept of these methods will be discussed in the analysis part.

??? Unlike comparing numbers, the comparison between words starts with the first letter of the word: 'a' has the highest priority while 'z' has the lowest. If there are two words with the same priority then we will have to compare the next letter. Note that there are generally 2 types of sorting method, which is comparison-based and non-comparison-based, and the three methods we are going to discuss about are all belong to comparison based methods, so we can apply the `strcpy(str1, str2)` from `string.h` to compare the order of two words, then we can eventually list the array in non-decreasing order. Since the dominating element ??? is the `strcpy()` function, so the key to reduction of process time is to avoid unnecessary or redundant call of `strcpy()`.

Not clear!

2. Approach:

For comparing the average time, we are going to use the same methods as in homework 1, which will define a repetition time R = 500 to repeat, then we will store the original word array 'a' to the array that are going to be sorted 'list'. After copying the array, we are able to carry out each algorithm for R times. The overall procedure will be as follow:

a) start counting time
b) copy & execute algorithm for R times
c) Stop counting time, change to another algorithm then go back to a)
d) when all of the algorithm is done, print the CPU time and the sorted array.

The main function pseudo code will be like this:

```
Algorithm main(void)
{
        ReadData();

        t := GetTime();                         // Execute heap sort
        for i := 1 to Repeat do {
                CopyArray();                    // copy to the original array
                HeapSort(list, n);
        }
        t := GetTime() - t;                     // stop counting time
        Write (t/Repeat);

        t := GetTime();                         // Execute merge Sort
        for i := 1 to Repeat do {
                CopyArray();                    // copy to the original array
                MergeSort(list, low, high);
        }
        t := GetTime() - t;                     // stop counting time
        Write (t/Repeat);

        t := GetTime();                         // Execute quick sort
        for i := 1 to Repeat do {
                CopyArray();                    // copy to the original array
                QuickSort(list, low, high);
        }
        t := GetTime() - t;                     // stop counting time
        Write (t/Repeat);

        PrintArray();
}
```

As for variable & function declarations, there will be more functions to be called than in homework 1 since each sorting functions will derive a ??? function. Take heap sort for instance, except in addition to calling `HeapSort()`, we are also required to define a `Heapify()` to make a max heap array every time. This time I declared 3 global arrays & 2 global variables which are data size & repetition times. The function of the 3 arrays is for storing original array, sorting, and for heap function, respectively. In addition, the size of array is N + 1 and not N, this is because when we are going to implement quick sort, there is a partition function that has a chance to compare list[N], which is defined as ∞ in theorem, so I define it to be "zzzzzz", which is technically going to be bigger than any words.

After the results are found, we are then able to make a chart with x-axis the dataset size and the y-axis process time to compare each algorithms' difference of average process time. Furthermore, we can also add the previous quadratic sorts run time into the comparison to observe the difference.

3. Analysis:

The three algorithms we are going to apply on this assignment are heap sort, merge sort and quick sort, and the pseudo code of each of them and their sub-function are as the provided:

a) Heap sort:

The concept of heap sort is actually quite directive and it achieve its aim by a method slightly alike with bubble sort, but the bubble pop up with a way far faster than bubble sort. As bubble sort finds the max bubble by calling `Heapify()` and swapping it with its adjacent index, heap sort finds the max bubble by swapping it with the parent in the heap tree. You should define these terms. In the worst case, while bubble sort has to swap n times to pop the bubble, heap sort only requires lg$n$ times to pop the bubble. Hence, with the average heapify time of lg$n$ times and the total n elements to heapify, the total time complexity is able to be calculated , which is $O(n\lg n)$. The overall process will be calling the Heapify function first to initialize the whole array, then we start to call the heapify function of the range 1:i - 1 (the max are stored at the back) until the whole array is sorted.

Incomplete.

The space complexity of heap sort is actually simple as O(1) since there are no extra number to be stored anywhere else except swapping and the range and root position.

```
// Sort array in nondecreasing order with max heap method
// input: size n array list
// output: sorted array list with size n in nondecreasing order
Algorithm HeapSort (list, n)
{
        for i := ⌊n/2⌋ to 1 step -1 do {          // initialize the whole array to be max heap
                Heapify (list, 1, n);
        }
        for i := n to 2 step -1 do {               // make list[1:i-1] a max heap
                temp = list[i];                    // store max to the last element
                list[i] = list[1];
                list[1] = temp;
                Heapify (list, 1, i - 1);
        }
}
```

```
// To make array in max heap property with root i
// input: size n array list , root i
// output: updated list array with the property of max heap
Algorithm Heapify (list, i, n)
{
        j := 2 * i;                              // left child's position
        item := list[i];                         // target element
        done := 0;                               // check if it is in the right position
        while j <= n and !done do {
                if j < n and list[j] < list[j + 1] then j := j + 1;     // right child > left child
                if item > list[j] then done := 1;       // in the right position
                else {
                        list[⌊j/2⌋] := list[j];        // make the bigger element be the parent
                        j := 2 * j;
                }
        }
        list[⌊j/2⌋] := item;
}
```

b) Merge sort:

This method involves divide & conquer, which separates the problem into couple sub problems and solve it, then compose the final outcome with every sub problem's results. divides the array into 2 halves in each layer until the sub array size become 1. When the `Merge()` are called, it will divide the sub array in the argument into right half and left half array then merge it to a sorted sub array by comparing the divided arrays, until the whole array are sorted. To analyze its time complexity, we will assume the array size n be a power of 2, represent it as $n = 2^k$, note that in merge function we will take n comparisons, so we can now set $T(n) = a$ the running time when n = 1 and $T(n) = 2T(n/2) + c \cdot n$ the running time if n is bigger than 1. We can then write the formula below:

$$
\begin{aligned}
T(n) &= 2(2T(n/4) + c \cdot n/2) + c \cdot n \\
&= 4T(n/4) + 2c \cdot n \\
&= 4(2T(n/8) + c \cdot n/4) + 2c \cdot n \\
&= 2^k T(1) + k \cdot c \cdot n \\
&= a \cdot n + c \cdot n \cdot lg(n)
\end{aligned}
$$

Finally, we can obtain the result of the time complexity, which is *O(*nlgn*)*.

The space complexity is always O(n) since there is a temp array to store the merged array, which is the dominant cause of the complexity, the rest integers brings the space complexity of O(lg*n*).

```
// Sort array in nondecreasing order with merge sort method
// input: array list, int low, high
// output: sorted array list
 Algorithm MergeSort (list, low, high)
{
        if (low < high) then {                      //terminal condition
                mid := ⌊(low + high) / 2⌋;          // determine mid position
                MergeSort(list, low, mid);          // solve left part
                MergeSort(list, mid + 1, high);     // solve right part
                Merge(list, low, mid, high);        // merge the whole sub array
        }
}

// Separate the array to two halves then merge it to a sorted array
// input: array list, int low, high
// output: sorted array list
 Algorithm Merge(list, low, mid, high)
{
        h := low; i = low; j := mid + 1;            // initializations
        while (( h ≤ mid) and (j ≤ high)) do {      // compare the two half of array
                if (list[h] ≤ list[j])then {        // left one is smaller, store it first
                        temp[i] := list[h];
                        h := h + 1 ;                // check next one
                }else {                             // right one is smaller, store it first
                        temp[i] := list[j];
                        j := j + 1;                 // check next one
                }
                i := i + 1;
        }
        if (h > mid) then {                         // store the remaining element
                for k := j to high do {
                        temp[i] = list[k];
                        i := i + 1;
                }
        }else {                                     // store the remaining element
                for k := h to mid do {
                        temp[i] := list[k] ;
                        i := i + 1;
                }
        }
        for k := low to high do list[k] := temp[k];    // store it back to the list
}
```

c) Quick sort:

The last method quick sort has a similar structure as merge sort, but the sub function `partition()` and `merge()` approaches the result by different ways. The quick sort method will also call its recursive function and send in the divided sub array until the size of array is 1. However, while merge function merge the two arrays into one sorted array, partition merely classify the array by checking if the element is bigger than first element or not. Not clear. Moreover, what quick sort is different as merge sort is that the value of mid index is decided by `partition()` but not just take the mean of *low* and *high,* so the sub array size is varied.

So what has been done in the `partition()` function is that the function will always set the first element as a pivot, then we find an element from the front that is bigger than the pivot and find an element from the back that is smaller than the pivot, then swap it. After every elements are checked, we will insert the pivot that is originally placed at the first to the updated positioning return the position.

To analyze its time complexity we have to discuss its average and worst case since the order of the list will make the number of comparisons different. For average case, we can have a general equation: $C_A(n) = n + 1 + \frac{1}{n}\sum_{k=1}^{n}(C_A(k-1) + C_A(n-k))$, where $C_A(n)$ is the comparisons at top layer, and $C_A(k-1)$, $C_A(n-k)$ are comparisons from the left half and right half, respectively, since there are one element being pivot when calling the partition function every time. As we expand the sigma and multiply the equation by 2 we will get $nC_A(n) = n(n+1) + 2\sum_{i=0}^{n-1}C_A(i)$, knowing that $C_A(0)$ and $C_A(1)$ is 0 since it requires no comparison, if we replace n with n - 1 and subtract the two equation we can get :

$$\frac{C_A(n)}{n+1} = \frac{C_A(n-1)}{n} + \frac{2}{n+1}$$

$$= \frac{C_A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$= \frac{C_A(1)}{2} + 2\sum_{k=3}^{n+1}\frac{1}{k}$$, which is bounded below log*n.*

So we can get the final answer $C_A(n) \leq (n+1)lg(n) = O(n\lg n)$.

As for its worst case, it is easier than average case to analyze since we know that worst case occurs when the partition is set at the $n-1^{th}$ element and thus requires $(n+1)+(n)+(n-1)+\cdots$

$$= \sum_{i=2}^{n} i+1 = O(n^2).$$

The space complexity of quick sort:

Worst case: recursion is called by n - 1 times, so $S_W(n) = 2 + S_W(n-1) = O(n)$

Best case: $S_B(n) = S_W(\lfloor \frac{n-1}{2} \rfloor) = \mathrm{O}(\lg n)$

Average case: Also $\mathrm{O}(\lg n)$

Incomplete.

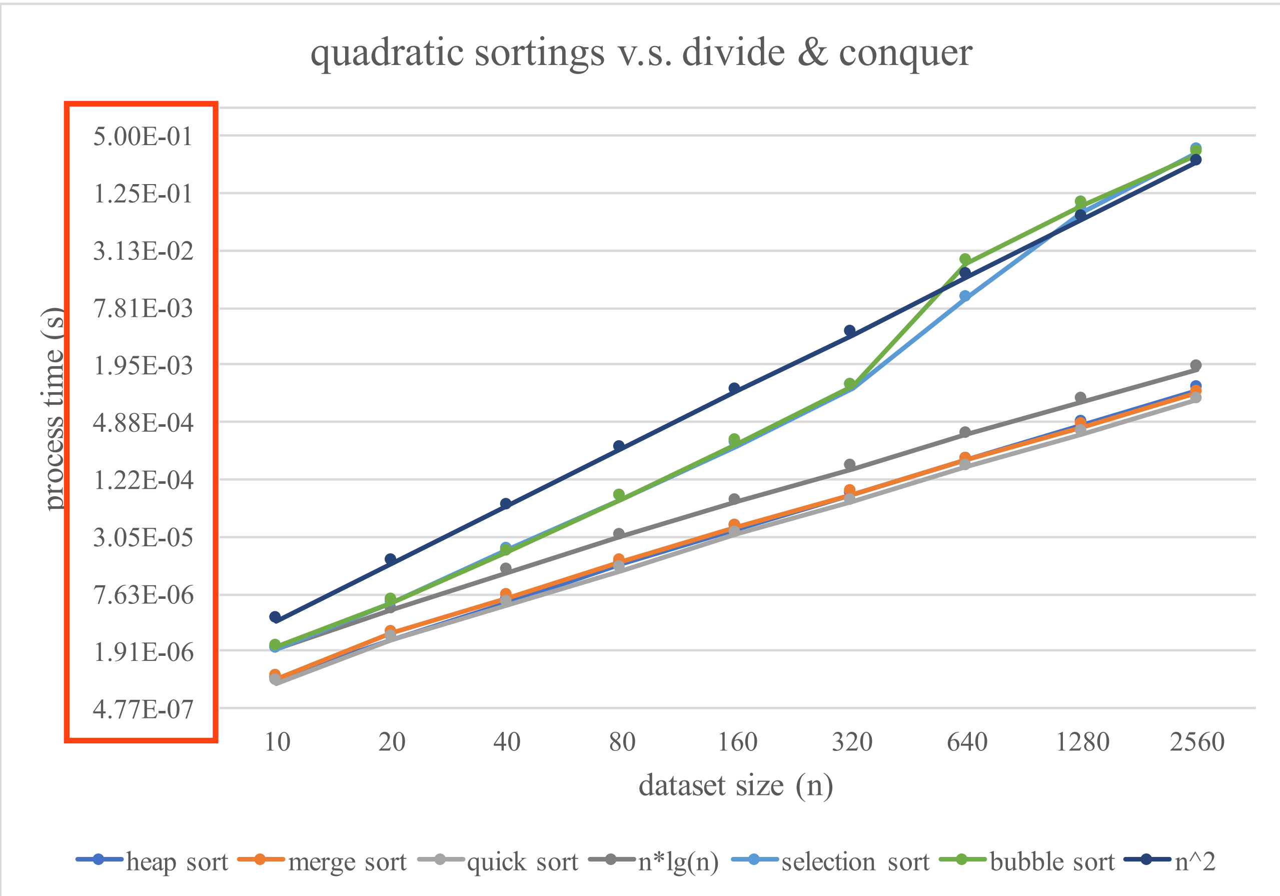
```
// Sort array in range low to high in nondecreasing order with quick sort method
// input: list[low:high], int low, high
// output: sorted list[low:high]
Algorithm QuickSort (list, low, high)
{
        if (low < high) then {                        // terminal condition
                mid := Partition(list, low, high + 1); // determine the separate point by partition()
                QuickSort(list, low, mid);             // solve left part
                QuickSort(list, mid + 1, high);        // solve right part
        }
}

/* let the array match the property that left part element is smaller than the first element, right part
element is bigger than first element, then return the first element's updated position
*/
// input: list[low:high], int low, high
// output: sorted list[low:high]
Algorithm Partition (list, low, high)
{
        v := list[low];                                // initializations of variable
        i := low;
        j := high;
        repeat {
                repeat i := i + 1; until(list[i] ≥ v);  // find i such that list[i] < v
                repeat j := j – 1; until(list[j] ≤ v);  // find j such that list[j] < v
                if(i < j) then swap(list, i, j);        // swap the two element
        }until (i ≥ j) ;
        list[low] := list[j];                          // insert v to the right position
        list[j] := v;
        return j ;                                     // return v's position
}
```

4. Results:

Data size (n)	heap sort (s)	merge sort (s)	quick sort (s)
10	9.62E-07	9.80E-07	8.94E-07
20	2.58E-06	2.92E-06	2.58E-06
40	6.29E-06	7.00E-06	5.86E-06
80	1.56E-05	1.66E-05	1.37E-05
160	3.76E-05	3.83E-05	3.27E-05
320	8.44E-05	8.55E-05	7.01E-05
640	1.96E-04	1.92E-04	1.62E-04
1280	4.56E-04	4.40E-04	3.65E-04
2560	1.06E-03	9.88E-04	8.35E-04

5. Observation and conclusion:

There are some extra requirements that has to be made to execute these algorithms, such as adding one extra array allocation for quick sort to compare the value of *list*[*N*], which is set to be ∞. In merge sort, there are also a required *temp* array to store the merged outcome and then store it back to the target array. In general, the results meet the analysis' expectation as we can see that the tendency of the three algorithms are *n*lg*n*. We can also feel the significant drop in process time just by sitting in front of the computer when compared to homework 1. The precise time difference can be found in the below chart, if R = 500 then the time of executing bubble sort toward s9.dat is about 10 ~ 20 seconds, while in this time's homework it only took less than a second. If we look carefully towards the time difference between these three algorithms, we can still find out that quick sort is a bit faster than merge sort. The possible reason of this result may be that quick sort doesn't require extra array to store outcome and merge sort calls too much recursive function, result in the time dominance in function calling with every function carrying out too few instructions.



	Heap sort	Merge sort	Quick sort
Time complexity(average)	O(*n*lg*n*)	O(*n*lg*n*)	O(*n*lg*n*)/O(n^2)(worst)
Space complexity(average)	O(1)	O(n)	O(n)(worst)/ O(lg*n*)

What are these?

```
$ gcc hw05.c
$ a.out < s7.dat
N = 640
HeapSort CPU time: 8.969784e-05 s
MergeSort CPU time: 8.294392e-05 s
QuickSort CPU time: 8.160162e-05 s
1 acarophobia
2 achillea
3 acre-foot
4 acrocarp
5 adelgidae
...
...
635 wondrous
636 woolly
637 worktable
638 worsening
639 yeleped
640 yellow
```

Can use variable size strings for better memory efficiency.
No need to malloc a string for every temp[i].

---

score: 70.0

- Overall report writing
  - English writing needs more practice.
  - Writing can be more logical – terms should be defined before referred to.
- Report format
  - Need double line spacing
- Time/Space complexity
  - Space complexity analysis needs to be more complete
- Results
  - Figures can still be improved.
- Program format can be improved

hw05.c

```c
// EE3980 HW05 Better Sorts
// 106061225, 楊宇羲
// 2021/4/5

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>

char **list;                                          // target array to sort
char **temp;                                          // for heap sort to store temp
char **a;                                             // original array
int N;                                                // size of array
int R;                                                // repitition

double GetTime(void);                                 // get local time in seconds
void ReadArray(void);                                 // store data in array
void CopyArray(void);                                 // copy array
void PrintArray(void);                                // display result
void HeapSort(char **list, int n);                    // perform heap sort
void MergeSort(char **list, int low, int high);       // perform merge sort
void QuickSort(char **list, int low, int high);       // perform quick sort
void Heapify(int i, int n);                           // find max heap
void Merge(int low, int mid, int high);               // for merge sort
int Partition(int low, int high);                     // swap value to right position

int main(void)
{
    int i;                                            // for loop counting
    double t;                                         // for CPU time counting

    scanf("%d", &N);                                  // size of array
    printf("N = %d\n", N);
    R = 500;                                          // repetition time

    ReadArray();                                      // read data

    // heap sort
    t = GetTime();                                    // start counting time
```

```
    for (i = 0; i < R; i++) {                   // repeat R times
        CopyArray();                            // call copy function
        HeapSort(list, N);                      // start heap sort
    }
    t = (GetTime() - t) / R;                    // stop and calculate time
    printf("HeapSort CPU time: %e s\n", t);

    // merge sort
    t = GetTime();                              // start counting time
    for (i = 0; i < R; i++) {                   // repeat R times
        CopyArray();                            // call copy function
        MergeSort(list, 0, N - 1);              // start merge sort
    }
    t = (GetTime() - t) / R;                    // stop and calculate time
    printf("MergeSort CPU time: %e s\n", t);

    // quick sort
    t = GetTime();                              // start counting time
    for (i = 0; i < R; i++) {                   // repeat R times
        CopyArray();                            // call copy function
        QuickSort(list, 0, N - 1);              // start quick sort
    }
    t = (GetTime() - t) / R;                    // stop and calculate time
    printf("QuickSort CPU time: %e s\n", t);

    PrintArray();                               // print array

    return 0;
}
```

Need a blank line here.

```
void HeapSort(char **list, int n)               // heap sort
{
    int i;                                      // for loop counting
    char *temp;                                 // for swapping

    for (i = (n - 1) / 2; i >= 0; i--) {        // turn whole array to max heap
        Heapify(i, n - 1);
    }
    for (i = n - 1; i >= 1; i--) {              // repeat n - 1 times
        temp = list[i];                         // swap list[i] with list[0]
        list[i] = list[0];
```

```
        list[0] = temp;
        Heapify(0, i - 1);                      // find the remaining max heap
    }
}
Need a blank line here.
void Heapify(int i, int n)                      // find max heap
{
    int j;                                      // left child
    char *item;                                 // parent is target
    int done;                                   // check if it is done

    j = 2 * i + 1;                              // statements
    item = list[i];
    done = 0;

    while (j <= n && !done) {                   // stop when it is done
        // if right child is bigger
        if (j < n && strcmp(list[j], list[j + 1]) < 0) j++;
        if (strcmp(item, list[j]) > 0) done = 1;// in right position
        if (strcmp(item, list[j]) > 0) done = 1;  // in right position
        else {
            list[(j - 1) / 2] = list[j];        // let child become parent
            j = 2 * j + 1;
        }
    }
    list[(j - 1) / 2] = item;                   // store item to child
}
Need a blank line here.
void MergeSort(char **list, int low, int high)  // merge sort
{
    int mid;
Need a blank line here.
    if (low < high) {                           // terminal condition
        mid = (low + high) / 2;                 // determine mid index
        MergeSort(list, low, mid);              // left part
        MergeSort(list, mid + 1, high);         // right part
        Merge(low, mid, high);                  // perform merge
    }
}
Need a blank line here.
void Merge(int low, int mid, int high)          // start merge to right position
```

```
{
    int h;                                  // compare left half
    int i;                                  // storing temp array
    int j;                                  // compare right half
    int k;                                  // loop variable

    h = low;                                // statements
    i = low;
    j = mid + 1;

    while (h <= mid && j <= high) {         // compare left & right
        if (strcmp(list[h], list[j]) <= 0) {  // if left is smaller, pick left
            temp[i] = list[h];              // store it to temp
            h++;                            // go to next one
        }else {                             // otherwise, pick right
        } else {                             // otherwise, pick right
            temp[i] = list[j];              // store it to temp
            j++;                            // go to next one
        }
        i++;                                // store next temp
    }

    if (h > mid) {                          // if left side has done
        for (k = j; k <= high; k++) {       // store all the right side
            temp[i] = list[k];
            i++;
        }
    }else {                                 // if right side done first
    } else {                                 // if right side done first
        for (k = h; k <= mid; k++) {        // store all the left side
            temp[i] = list[k];
            i++;
        }
    }
    for (k = low; k <= high; k++) {         // update correct array to list
        list[k] = temp[k];
    }
}
```

Need a blank line here.

```
void QuickSort(char**list, int low, int high)   // perform quick sort
{
```

```
    int mid;                                    // find mid index
Need a blank line here.
    if (low < high) {                           // terminal condition
        mid = Partition(low, high + 1);         // determine mid
        QuickSort(list, low, mid - 1);          // do left part
        QuickSort(list, mid + 1, high);         // do right part
    }
}
Need a blank line here.
int Partition(int low, int high)                // find mid to set partition
{
    char *v;                                    // target
    char *temp;                                 // for swapping
    int i;                                      // left element
    int j;                                      // right element
    int flag;                                   // check initial condition

    v = list[low];                              // initial conditions
    i = low;
    j = high;
    flag = 0;

    while (i < j) {                             // terminal condition
        // find i such that list[i] >= v
        while (flag == 0 || (strcmp(list[i], v) < 0 && i < N)){
        while (flag == 0 || (strcmp(list[i], v) < 0 && i < N)) {
            i++;
            if (flag == 0) flag = 1;
        }
        flag = 0;
        // find j such that list[j] <= v
        while (flag == 0 || (strcmp(list[j], v) > 0 && j < N)){
        while (flag == 0 || (strcmp(list[j], v) > 0 && j < N)) {
            j--;
            if (flag == 0) flag = 1;
        }
        if (i < j) {                            // swap list[i] & list[j]
            temp = list[i];
            list[i] = list[j];
            list[j] = temp;
        }
```

```
    }
    list[low] = list[j];                        // swap list[low] & v
    list[j] = v;
    return j;                                   // return j's position
}
Need a blank line here.
void ReadArray(void)                            // read in datas
{
    int i;                                              // for loop

    a = (char**)malloc((N + 1) * sizeof(char*));        // the original array
    list = (char**)malloc((N + 1) * sizeof(char*));     // the array to sort
    temp = (char**)malloc((N + 1) * sizeof(char*));     // temporary array

    for (i = 0; i < N; i++) {
        a[i] = (char*)malloc(50 * sizeof(char));        // dynamic arrays
        list[i] = (char*)malloc(50 * sizeof(char));
        temp[i] = (char*)malloc(50 * sizeof(char));
        scanf("%s", a[i]);                              // store strings
    }
    a[N] = "zzzzzz";                                    // for quick sort
    list[N] = "zzzzzz";                                 // for quick sort
    temp[N] = "zzzzzz";                                 // for quick sort
}
Need a blank line here.
void CopyArray(void)                            // copy data
{
    int i;                                      // loop counting
Need a blank line here.
    for(i = 0; i < N; i++) {
    for (i = 0; i < N; i++) {
        list[i] = a[i];                         //copy array 'a' to 'list'
        list[i] = a[i];                         // copy array 'a' to 'list'
    }
}
Need a blank line here.
void PrintArray(void)                           // display result
{
    int i;                                      // loop counting
Need a blank line here.
    for (i = 0; i < N; i++) {                   // Display sorted array
```

```
        printf("%d %s\n", i + 1, list[i]);
    }
}
Need a blank line here.
double GetTime()                                // get time
double GetTime(void)                                // get time
{
    struct timeval tv;

    gettimeofday(&tv, NULL);
    return tv.tv_sec + 1e-6 * tv.tv_usec;       // sec + micro sec
}

```