Last week

• Time complexity
• Linear and binary search
Agenda

- Space complexity
- Sorting examples
Space complexity
Complexity measurements

• In the last tutorial that a program demands two types of resources:

  Memory  Time
Space complexity

• When talking about space complexity, we’ll refer to the space that an algorithm takes up to solve a problem.
• Just like time complexity, space complexity will not be measured in bits or bytes, but rather space as it relates to the size of the input.
• Even though the input itself is actually stored in memory, we will not count it in referring to our algorithm’s space complexity.
What is the size?

- What is the space complexity of the function?

```c
int example1(int a, int b) // O(1)
{
    int c = a + b;
    return c;
}
```

```c
int example2(int a[], int n) // O(1)
{
    int i, c = a[0];
    for (i = 1; i < n; ++i)
        c += a[i];
    return c;
}
```

```c
#define M ...
#define N ...
void ex3(int a[N][M]) // O(M*N)
{
    int i, j, d[M][N];
    for (i = 0; i < M; ++i)
        for (j = 0; j < N; ++j)
            d[i][j] = a[j][i];
    for (i = 0; i < M; ++i)
        for (j = 0; j < N; ++j)
            printf("%d\n",d[i][j]);
}
```

```c
#define MAX 100
void ex4(int a[], int n) // O(1)
{
    int i, c[MAX];
    for (i = 0; i < n; ++i)
        c[a[i] % MAX] += a[i];
    for (i = 0; i < MAX; ++i)
        printf("%d\n",c[i]);
}
```

M and N are established from #define
Example of space complexity O(n)

• Printing the top 10% of grades in the array:

```c
#define NUM_GRADES ...

int find_min(int arr[], int m) {
    int i, min = 0;
    for (i=1; i<m; ++i)
        if (arr[min] > arr[i])
            min = i;
    return min;
}

void print_best10(int grades[]) {
    int best[NUM_GRADES/10]={0}, i, min, m = NUM_GRADES/10;
    for (i=0; i < m; ++i)
        best[i] = grades[i];
    min = find_min(best, m);
    for (i=m; i<n; ++i)
        if (grades[i] > best[min]) {
            best[min] = grades[i];
            min = find_min(best, m);
        }
    for (i=0; i < m; ++i)
        printf("%d ", best[i]);
}
```
Examples: sorting
Question 1

• Write a function that accepts an array of $n$ integers, and sorts it in non-increasing order.
• Thus, for every $i$: arr[$i$] $\geq$ arr[$i+1$].
• The algorithm must use $O(1)$ space.
void swap(int* x, int* y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

void sort(int* arr, int* arr2 int n) {
    /*bubble sort*/
    int i, size = n, swapped = 1;
    while(size>1 && swapped){
        swapped = 0;
        for(i=1; i<size; i++)
            if (arr[i-1] < arr[i]){
                swap(arr+i, arr+i-1);
                swapped = 1;
            }
        size--;
    }
}

Question 2

- Write a function that gets an array of 10 integers, and returns the amount of the total repetitions of elements in the array.
  - This number will not include the elements that only appear once in the array.
- For example, for the input:

  8 1 8 4 8 6 4

  The output will be 5.

- Next example:

  1 2 3 4 6 -9 8

  The output will be zero.
- We can do this in O(N) space.
- N is established with a #define
- This should be done without changing the input array.
Solution idea

• We’ll iterate through the array and execute:
  – If a number appears once, it’s not yet clear if it will appear again. Thus we can skip this occurrence of the number.
  – When we find the second instance of the number, we’ll start counting the number of times it appears. Since we did not count it after its first appearance, we count it twice.
  – If it appears more than twice, we continue counting.
• How do you check if the number was in the array in the past?
Solution idea – O(1) space

- In order to check if the number appeared in the past:
  - We can pass through the array from the left of its location to the beginning of the array...
  - And count how many times it appears.

- How much does this cost us?
  - Checking for an appearance is an O(N) operation.
  - Counting numbers: N times O(N), thus a total O(N^2)

- Can we be more efficient?
Solution idea – O(N) memory

• We will count in mostly the same way, but we’ll check for past appearances in O(1), meaning we’ll only check one element (the preceding element).
  – Therefore, we need all of the same numbers to be next to each other.
  – We’ll sort them first!
  – But...we can’t change the array!
  – So we’ll define a “helper” array of size N - space complexity is now O(N)
Solution idea – O(N) memory

• We’ll copy the elements of the array to the helper array and sort them.
• Then, for each element, we’ll check the element to its left.
  – If it’s different, then we won’t count it, but we’ll record that we encountered the new number.
  – If it’s equal, then we’ll add 1 or 2, depending on whether it’s a new value or not.
• How much does this cost us?
  – Copying the array - O(N)
  – Sorting the array - O(N^2)
  – Checking an appearance O(1)
  – Counting numbers :N times O(1) ,so O(N)
  – Thus, the algorithm’s time complexity is the following: O(N+N^2+N) = O(N^2)
• Why weren’t we able improve from last time?
int calcMoreThanOne(int *arr)
{
    int tmpArr[N], i, dupNum = 0, dup = 0; /* copy the original array */
    for(i=0; i<N; i++) {
        tmpArr[i] = arr[i];
    }
    /* sort */
    sort(tmpArr, N);
    /* count duplicated elements */
    dup = 0;
    for(i=1; i<N; i++) {
        if (tmpArr[i-1]==tmpArr[i]) {
            dupNum += (dup==0)?2:1;
            dup = 1;
        } else {
            dup = 0;
        }
    }
    return dupNum;
}
Question 3 – Part A

• Implement the function
  ```c
  int permuEQ(char *str1, char *str2)
  ```
  which receives two strings and checks if it’s possible to change the order of the characters in one string such that it will be the same as the second string?

• The function should return 1 if it’s possible, and 0 otherwise

• For example
  – The call `permuEQ ("computer", "mutorpec")` returns 1.
  – The call `permuEQ ("abc", "bcd")` returns 0.
  – The call `permuEQ ("a", "aa")` returns 0.

• You can change content of the input arrays
• You can use the library functions from `string.h`
• It’s required to use space complexity of O(1)
Solution idea

• Solution 1
  – For every character that appears in str1 check if there is such a character in str2.
    • But we must be wary of duplicate characters i.e. "aacc" and "accc."
  – Also, we’ll check if the lengths of the strings are the same

• Solution 2
  – Sort the letters in the string
  – Check if the strings are identical
    • We’ll use strcmp for help
void swap(int* x, int* y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

void maxSort(char *str) {
    int i, j, max;
    for(i = 0; str[i+1] != \0; i++) {
        max = i;
        for(j = i+1; str[j] != \0; j++)
            if(str[j] > str[max])
                max = j;
        swap(str + i, str + max);
    }
}

int permuEQ(char *s1, char *s2) {
    sort(s1, strlen(s1));
    sort(s2, strlen(s2));
    //maxSort(s1);
    //maxSort(s2);
    return strcmp(s1, s2) == 0;
}
Write a function `int checkPair(char *a1, char *a2, char *b1, char *b2)` that accepts four strings and checks if any sequence of all of the characters from the first two strings can be equivalent to any sequence of characters from the last two strings.

Return 1 if it’s possible, 0 otherwise.

For example:
- The call `checkPair("cat", "hat", "aahc", "tt")` returns 1.
- The call `checkPair("cat", "hat", "aaxy", "tt")` returns 0.
- The call `checkPair("cat", "hat", "aahh", "ctt")` returns 0.

You may not change the content of the input strings.

You can use a space complexity that is linear to the input.

The size of each string will not exceed K characters. K is established with a K #define.
Solution idea

- We’ll take advantage of the work that we did in Part A.
- Part A compares between two strings but not sequences of strings.
- Thus, we’ll combine the pairs and use part A.
- Notice that we are allowed to define additional array(s), but we are not allowed to change the original strings.
- We’ll define two helper arrays in order to build two strings as a sequence of other pairs.
#include <string.h>

int checkPair(char *a1, char *a2, char *b1, char *b2) {
    char temp1[K+K+1] = {0}, temp2[K+K+1] = {0};
    int res;
    strcpy(temp1, a1);
    strcat(temp1, a2);
    strcpy(temp2, b1);
    strcat(temp2, b2);
    res = permuEQ(temp1, temp2);
    return res;
}

Why do we need to add 1?
Question 4

• Write a function void sort3(int array[], int n)
• Which accepts an array of integers and its size, and sorts it according to the remainder after dividing it by 3.
• Thus, in the beginning of the sorted array there will be numbers that divide evenly by 3, and then numbers that have a remainder of 1 after being divided, and then numbers that have a remainder of 2.
• Use space complexity of O(1).
Sorting example

• This example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>7</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

• Will change to the array:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>7</td>
<td>7</td>
<td>13</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

• Note that order is not maintained within a set. For example, 2, 11, and 5 have a remainder of 2 when divided by 3, but they are not written in increasing order.

• We have must also be aware of negative numbers
Solution idea

• Solution 1;
  – Sort the array according to modulo 3

• Solution 2
  – Make 3 passes through the array
  – Each time we’ll sort all of the elements with the same value of modulo 3, so that we maintain the order of what has been previously sorted.
Solution (2)

```c
void sort3(int a[], int n) {
    int done = 0, modulo, i;
    for (modulo = 0; modulo < 3; modulo++) {
        for (i = 0; i < n; i++)
            if ((a[i] % 3 == modulo) ||
                a[i] % 3 == -modulo) {
                int temp = a[i];
                a[i] = a[done];
                a[done] = temp;
                done++;
            }
    }
}
```

What will happen if we replace 3 with 2?

What is the time complexity of this algorithm?
Question 5

- We can represent a simple fraction using two integers: one for the numerator and one for the denominator.
  - For example, one third (1/3) will be represented by the pair 1 and 3.
- Write a function `void frac_sort(int a[], int b[], int n)` that accepts two arrays of the same size `n`, where `a` contains the numerator and `b` contains the corresponding denominator.
- The function sorts the fractions from small to large.
- The order of equivalent fractions with different representations is unimportant. For example 1/3 and 2/6.
- We will assume that 0 does not appear as a numerator or denominator.
Sorting examples

For example, these arrays:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>-2</td>
<td>4</td>
<td>1</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>-4</td>
<td>12</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

After calling frac_sort(a,b,5) become:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>1</td>
<td>4</td>
<td>-2</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>12</td>
<td>-4</td>
<td>3</td>
</tr>
</tbody>
</table>
Solution idea

• We’ll sort according to the maximum
• We must account for negative numbers!
• Therefore, we’ll calculate the decimal value of each fraction in order to compare fractions.
• When we switch the location of a fraction, we will have to switch them in pairs. (We’ll switch the same elements in array a and b).
```c
int find_max(int a[], int b[], int n) {
    int i, i_max = 0;
    float max = (float)a[0] / b[0];
    for (i=1; i<n; i++) {
        float curr = (float)a[i] / b[i];
        if (curr > max) {
            i_max = i;
            max = curr;
        }
    }
    return i_max;
}
```
Continuation of the solution

```c
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

void frac_sort(int a[], int b[], int n) {
    int length;
    for (length = n; length > 1; length--) {
        int i_max = find_max(a, b, length);
        swap(&a[length-1], &a[i_max]);
        swap(&b[length-1], &b[i_max]);
    }
}
```
void swap(int* x, int* y)
{
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

void sort(int* a, int* b, int n)
{
    /*bubble sort*/
    int i, size = n, swapped = 1;
    while(size>1 && swapped)
    {
        swapped = 0;
        for(i=1; i<size; i++)
            if ( ((float)a[i-1]) / ((float)b[i-1])
                > ((float)a[i]) / ((float)(b[i])) )
            {
                swap(a+i,a+i-1);
                swap(b+i,b+i-1);
                swapped = 1;
            }
        size--;
    }
}

int main(void)
{
    int a[] = {1, 4,6,1};
    int b[] = {2,10,7,8};

    sort(a,b,4);
    /*4 is the array size */
    int i;
    for (i = 0; i <4; i++)
    {
        printf("%d ",a[i]);
    }
    printf("\n");

    for (i = 0; i <4; i++)
    {
        printf("%d ",b[i]);
    }
    return 0;
}