מ之道 לשפת C

Tutorial 13: Recursion, Sorting, and Review
Recursive merge
Remainder: Recursive Merge

• Receive two \textbf{sorted} arrays $a[ ]$, $b[ ]$ and their sizes: $sizeA$, $sizeB$
• Return a single \textbf{sorted} array $c[ ]$ of size $sizeA + sizeB$ which contains all the elements from $a[ ]$ and from $b[ ]$. 
Merging arrays: the recursive algorithm

• In each step, we’ll copy a single cell and call the recursive function again, without the cell we copied.

```
16 19 25 48 50 67
```

```
14 22 37 45 56 61
```

```
14 16 19 22 25 37 45 48 50 56 61 67
```
void merge(int a[], int sizeA, int b[], int sizeB, int c[]) {
    int takeFromA;
    if (sizeA == 0 && sizeB == 0) {
        return; /*both a and b are finished- our job is done */
    }

    if (sizeA == 0) { /*a is finished- copy cells from b */
        takeFromA = 0;
    } else if (sizeB == 0) { /*b is finished- copy cells from a */
        takeFromA = 1;
    } else { /* sizeB != 0 and sizeA != 0 */
        takeFromA = (a[0] < b[0]);
    }

    if (takeFromA) {
        c[0] = a[0];
        merge(a+1, sizeA-1, b, sizeB, c+1);
    } else { /* take from B */
        c[0] = b[0];
        merge(a, sizeA, b+1, sizeB-1, c+1);
    }
}
Recursive Search
Recursive Binary Search

• Lets implement a recursive binary search

• **Goal:** find the index of a the cell whose value is $X$ (If such cell exists)

• **Return:**
  – If such cell exists- return it’s index
  – Otherwise: return -1
Recursive Binary Search

1. Stopping conditions:
   a) If the size of the array is 0: return -1
   b) If the cell in the middle equals to our goal: return its index

2. If the cell in the middle is bigger than our goal:
   ▪ **Recursive step**: call the function again, but only with the first half of the array, and return the same

3. If the cell in the middle is smaller than our goal:
   ▪ **Recursive step**: call the function again, but only with the second half of the array. Return:
     ➢ If the recursive call returned -1: return the same
     ➢ Otherwise, add to the return value the amount of omitted cells (the first half).
Recursive Binary Search

```c
int binsearch(int a[], int size, int val)
{
    if (size == 0)
        return -1;
    if (a[size/2] == val)
        return size/2;
    if (a[size/2] > val)
        return binsearch(a, size/2, val);
    else {
        int pos = binsearch(a+(size/2)+1, size-(size/2)-1, val);
        if (pos == -1)
            return -1;
        return pos + size/2 + 1;
    }
}
```
Recursive Bubble Sort
### Remainder: bubble sort

<table>
<thead>
<tr>
<th>Iteration 1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>9</td>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Iteration 2</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>Iteration 3</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>Iteration 4</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>

```plaintext
7 9 2 5 3
```
void swap(int* x, int* y)
{
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

void sort(int* arr, 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--;
    }
}
Recursive Bubble Sort

• The idea:
  – Replace the outer loop with a recursion
  – As we all remember- in the end of each iteration the biggest value is already in the end of the array
  – Thus, in the next iteration, we can sort only what’s left from the array
  – Each recursive call we’ll pass as parameter only what's left from the array
  – Stopping condition will be an array of size 1 (because it’s already sorted).
Helper function: swap

• Our old friend:

```c
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```
Recursive Bubble Sort

```c
void rbs(int a[], int size)
{
    int i;
    if (size <= 1) {
        return; /* Already sorted */
    }

    for(i = 0; i < size-1; i++) {
        if (a[i] > a[i+1]) {
            swap(&a[i], &a[i+1]);
        }
    }

    rbs(a, size-1);
}
```

How can we replace this loop with a recursive function?
Execute the implementation on the example

```c
void rbs(int a[], int size)
{
    int i;
    if (size <= 1) {
        return; /*Already sorted*/
    }
    for (i = 0; i < size-1; i++) {
        if (a[i] > a[i+1]) {
            swap(&a[i], &a[i+1]);
        }
    }
    rbs(a, size-1);
}
```
Riddle

What will happen if we switch the execution order in bubble sort?

```c
void rbs(int a[], int size)
{
    int i;
    if (size <= 1) {
        return; /* Already sorted */
    }
    rbs(a, size-1);
    for (i = 0; i < size-1; i++) {
        if (a[i] > a[i+1]) {
            swap(&a[i], &a[i+1]);
        }
    }
}
```
Execute the new implementation on the example:

- The new implementation is wrong.

Conclusion:
The order is important.
Why didn’t it work?

• In order to understand why it didn’t work, we’ll try to figure out the logic of this program:
  – Array of size one is sorted - correct.
  – If we sort the array without its last cell, and then execute a single bubble iteration on the entire array. Will we receive a sorted array? NO!

– For example:

```
void rbs(int a[], int size) {
    int i;
    if (size <= 1) {
        return; /*Already sorted*/
    }
    rbs(a, size-1);
    for (i = 0; i < size-1; i++) {
        if (a[i] > a[i+1]) {
            swap(&a[i], &a[i+1]);
        }
    }
}
```

2 5 7 6 9

Not sorted

bubble

2 5 7 9 6

sorted
Small fix

- Notice that the logic is correct if we add the new cell in the beginning

| 2 | 5 | 6 | 7 | 9 |

- So, if we change the program according to the new logic, it will work

```c
void rbs(int a[], int size)
{
    int i;
    if (size <= 1) {
        return; /*Already sorted*/
    }
    rbs(a+1, size-1);
    for (i = 0; i < size-1; i++) {
        if (a[i] > a[i+1]) {
            swap(&a[i], &a[i+1]);
        }
    }
}
```
Review question 1

• 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.
Idea #1

• 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?
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?
Idea #2

- 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)
Idea #2

• 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)
  – There are sorting algorithms which work in O(N*log(N)) so this solution could be as fast as O(N*log(N))
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;
}
Review Question 2

• 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;
}
GOOD LUCK