Build X: Algorithms Week 2

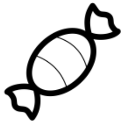
- Functions/Recursion
- Merging 2 sorted lists
- Divide and Conquer ("Divide et Impera")
- Binary Search
- Merge Sort, Quick Sort, Radix Sort

01st of February 2016

Functions

Functions in CS - Quick Intro

- What is a function?
  - A **function** is a type of procedure or routine.
- Advantages of using a function:
  - We can write our program as a bunch of sub-steps
  - ! We can reuse the code instead of rewriting it.
  - Helps with the memory consumption
    - ☐E.g. variables inside the function live as long as the function does
  - Test only small parts of our program



Functions in CS

a. Void functions (Procedures)

- Functions that does not return any data

```
public static void printElements(int[] elementsToPrint) {

    //Goes through the list and prints each element
    for (int i = 0; i < elementsToPrint.length; ++i) {
        System.out.print(elementsToPrint[i] + " ");

    }
}
```

Functions in CS

b. Return Functions

```
public static int[] readElements(int n) {

    int[] a = new int[10];

    //Creates a Scanner that will help us read the input
    Scanner in = new Scanner(System.in);

    for (int i = 0; i < n; ++i) {
        a[i] = in.nextInt();
    }

    //Close the Scanner if it's no longer needed
    in.close();

    return a;
}
```

Recursion

Recursion

- when method calls <u>itself</u>

- *common example:* Factorial function:

  $$n! = 1*2*3*\ldots(n-1)*n$$

- (recursive)Java Implementation of Factorial function

```
public static int recursiveFactorial(int n)
{
    if(n<=0)
    {
        return 1;
    }

    return n*recursiveFactorial(n-1);
}
```

Components of a recursive function

- Basis case
- Recursive case

```
public static int recursiveFactorial(int n)
{
    if(n<=0)
    {
        return 1;
    }

    return n*recursiveFactorial(n-1);
}
```

Recursion

- Linear recursion
- Tail recursion
- Binary recursion

Linear recursion

- Test for base cases
- Recur only <u>once</u>
  - perform only one recursive call
  - this step may have a test that decides which possible recursive call to make; should ultimately make only 1
  - define it so that each call makes progress towards the base case

example: linear recursion

- *algorithm*: LinearSum(A,n)
  input: an array A; and an integer n>1 such that A has at least n elements
  output: Sum of the first n integers in A

- *pseudocode:*

```
if(n=0)
    return A[0]
else
    return LinearSum(A, n-1) +A[n-1]
```

Tail recursion

- form of linear recursion
- BUT the recursive call is the last thing the function does
- tail recursive function can be easily implemented in an iterative manner; => replace the recursive call with a loop

Binary recursion

- occurs whenever there are 2 recursive calls for each non-base case

example: add all numbers in an integer Array

- *algorithm*: BinarySum(A,i,n)
  input: an array A of integers; i>=0; n>=1
  output: sum of integers in A starting at i

- *pseudocode*:
  if n=1
  return A[i]
  else
  return BinarySum(A,i,[n/2])+
BinarySum(A,i+[n/2], [n/2])

Merging 2 sorted lists

Merging 2 sorted lists/arrays

- Used to obtain a sorted list by combining 2 sorted arrays

- Inefficient way :
  - Add the second array to the first one and then use a sorting algorithm (O(n*n) or O(n*log(n))
- Efficient : Use Collation
  - Complexity : O(n+m)

Merging- Example

- First List: 1 3 4 5 7 9
- Second List: 2 6 8 10 11 12

- Output : 1 2 3 4 5 6 7 8 9 10 11 12

Time to code

-Collation of 2 sorted lists-

Divide and Conquer

Divide and Conquer - Technique

- A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same type.

- The algorithm stops when the sub-problems are simple enough to be solved.

Divide and Conquer - Advantages

- Useful in solving complex problems by splitting it in multiple sub-problems.
- It can be used to optimize different algorithms
  - Time
    - E.g
    - From O(n) to O(log n) – Binary Search
    - From O(n*n) to O(n * log(n)) – Merge Sort

Divide and Conquer - Disadvantages

- Uses recursion
  - ! Make sure there is enough stack memory

- Make sure base cases are properly selected

Divide and Conquer - Steps

- 1. **Divide** the problem into sub-problems as many times as you need.
- 2. **Conquer** the sub-problems by solving them.
- 3. Combine the solutions of the sub-problems into a solution for the initial problem.

Divide and Conquer - Example

Binary Search :

- Input : Given a sorted array find the position X in the list.
  - O(n)
  - O(log n)
- 3 7 8 11 21 39 40 100 101 , X
  - X = element to look for;

- Output : Position if the element is found, -1 otherwise

Time to code

-Binary Search-

Merge Sort

Merge sort

-on sequence S with n elements:

Divide: if S has 0 elements=> return S
otherwise remove all elements from S and put them into 2 sequences S1, S2

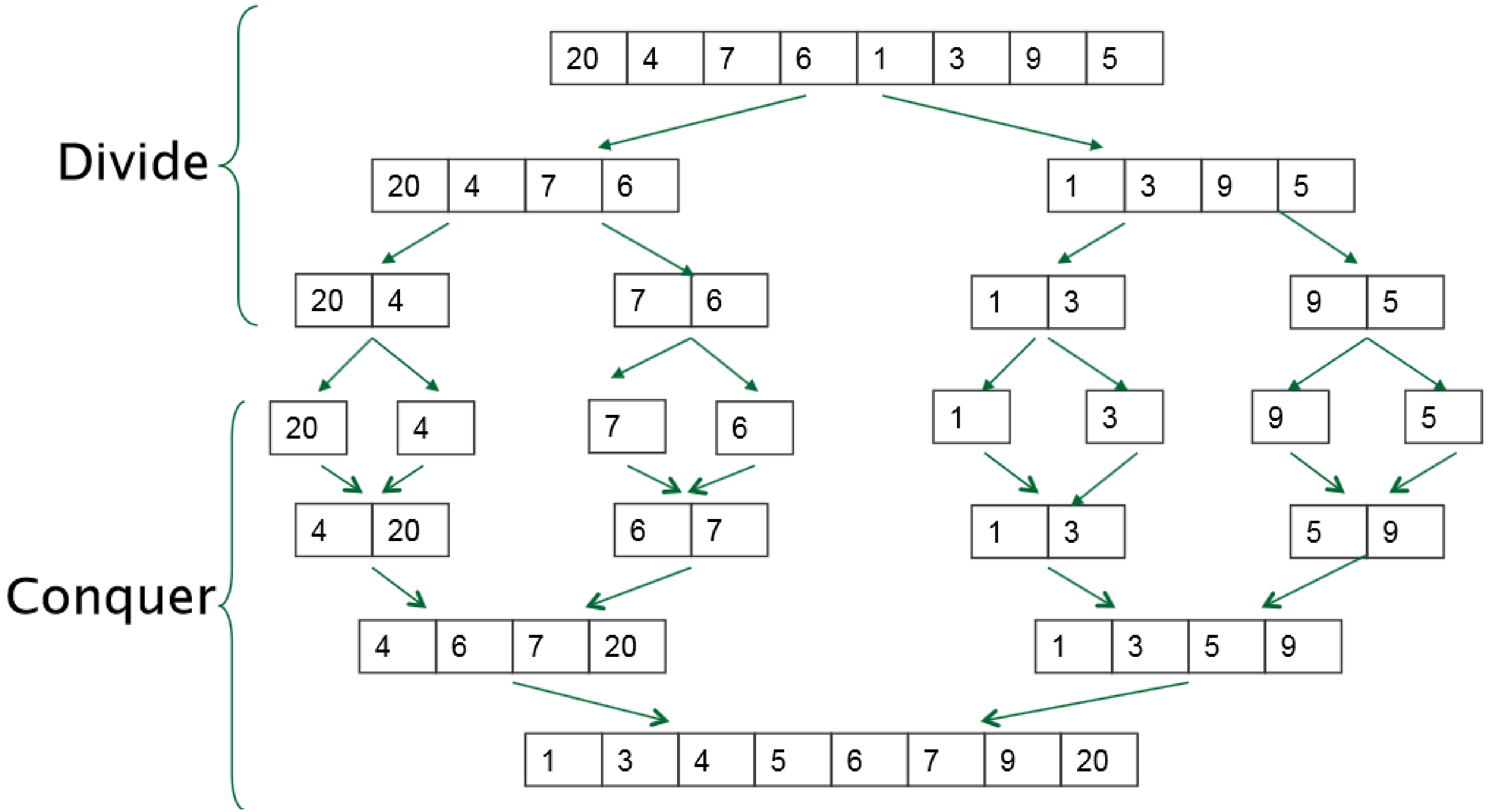
Recur: do the same with S1 and S2

Conquer: merge S1 and S2 into sorted sequence S

Merge-Sort

- <u>running time</u>: O(n logn)
- height h of a Merge-Sort tree is O(logn)
- for the i-th merging iteration, the complexity of the problem is O(n)

Merge Sort - Example



Time to code

-Merge Sort-

Quick Sort

Quick Sort – Description

- Efficient sorting algorithm

- Implement it well and you will get a 2-3 better <u>time complexity</u> than *Merge sort* and *Heapsort*.
- Efficient use of <u>memory consumption</u>

- It is a comparison sort which means it can sort anything that uses the “< , >” operands.

Quick Sort – Algorithm

- It is based on “Divide and Conquer” technique
- Steps similar to Merge Sort
  - Difference : Instead of splitting the list in 2 halves, we chose an element to be our “pivot”.
- Depending on the pivot selection and the partitioning steps our <u>time complexity</u> and <u>memory consumption</u> will change.

Quick Sort - **Lomuto partition scheme**

- Choses the pivot as the last element
- Inefficient as in most cases
  - Time Complexity
    - O(n * log n)
    - O(n^2) when the array is sorted or all elements are equal
  - Memory Consumption –
    - General : O(log n)
    - Worst case: O(n)

Quick Sort Lomuto - Example:

- Input : 3 4 1 2 6 3 Output : 1 2 3 3 4 6
- Steps :
  - 3 4 1 2 6 3
  - 3 4 1 2 6 3
  - 1 4 3 2 6 3
  - 1 2 3 4 6 3
  - 1 2 3 4 6 3
  - 1 2 3 4 6 3
  - 1 2 3 4 6 3
  - 1 2 3 4 6 3
  - 1 2 3 3 6 4
  - 1 2 3 3 4 6

Time to code

-Quick Sort (Lomuto)-

Quick Sort – **How to improve it**

- Selecting the last element as a pivot is inefficient

=> Chose a random pivot

- Recurse first into the smaller side of the 'partition'
- When the number of elements is below a bar(set by us), switch to a non-recursive sorting algorithm such as insertion-sort.

Counting Sort (Recap)

Counting Sort – Example

Input :
1 5 2 1 0 9 8 2 4 5 2 3 5 1 1 1

Step 1:
0 1 2 3 4 5 6 7 8 9
1 5 3 1 1 3 0 0 1 1
In our initial input "0" appears once, "1" appears 5 times, "2" appears 3 times, etc.

Step 2 :
0 1 1 1 1 1 2 2 2 3 4 5 5 5 8 9

Counting Sort

- Time complexity
  - O(n) – read and print the values
- Memory consumption
  - O (max-min) – keep another data structure for counting
    - Max = highest element
    - Min = minimum element
- When can we use it?
  - Good if our elements are in a specified interval with no high variations
  - Usually used as a sub-routine of other algorithms (Radix Sort)

Radix Sort

Radix Sort – Description

- ! Not a comparison sort.
- It uses keys of the elements to sort the list and not comparisons of the whole elements.
- Time complexity : O(w*n)
  - W – word size (fixed space of size used by a processor design)
  - N – number of keys

LSD Radix Sort - Description

- It uses counting sort as a sub-routine
- It is a fast and stable sorting algorithm
- Uses keys in integer representation to sort
  - Keys may be a string of characters or numerical digits
- Time Complexity : O(n * k)
  - N – number of keys
  - K – average length of keys

LSD Radix Sort - Steps

- LSD = Least significant digit;
- 1. Take the least significant digit of each key
- 2. Group the keys based on that digit
  - If 2 keys has the same digit then we keep the initial order
- 3. Repeat the process until there are no more digits

LSD Radix Sort - Example

- Input: 170, 45, 75, 90, 802, 2, 24, 66
- Output: 2, 24, 45, 66, 75, 90, 170, 802

MSD Radix Sort - Example

- MSD = Most significant digit;
- Same procedure as LSD but we start from left;

Q&A

We are not done

Algorithms Competition

- hackerarena.co.uk
- Challenges will be released every Thursday and will be based on the topics from that week
- Winner of the stage will be announced every Monday.
- Top students will qualify for a final competition