Week 02 Tutorial

Recursion, Analysis of Algorithms

1. (Recursion)

   Write both recursive and iterative functions to compute the length of a linked list, where the data structures are defined as:

   ```
   typedef struct _node {
      int data;
      struct _node *next;
   } Node;
   typedef Node *List;
   ```

   Both functions should have the same interface:

   ```
   int length(List L) { ... }
   ```

2. (Recursion)

   Analyse the behaviour of the following function, which returns the nth Fibonacci number:

   ```
   int fib(int n)
   {
      assert(n > 0);
      if (n == 1 || n == 2)
         return 1;
      else
         return fib(n-1) + fib(n-2);
   }
   ```

   Examine how it computes `fib(3)`, `fib(5)`, `fib(7)`. Is this method feasible for larger values of `n`?

3. (Recursion)

   Consider the problem of finding the mean (average) value in a linked list. Write iterative and recursive functions to do this. The recursive solution may require auxiliary functions.

   Both functions have the same interface:

   ```
   float mean(List L) { ... }
   ```

   Assume the same list types as in the earlier question.

4. (Recursion)

   Consider the problem of computing x^n for two integers x and n, where n is non-negative. Since the result of this grows very large, even for moderate values of n, assume that we are not worried about overflows.

The standard iterative solution to this is $O(n)$:

```
int pow(int x, unsigned int n)
{
   int res = 1;
   for (int i = 1; i <= n; i++)
      res = res * x;
   return res;
}
```

It is possible to write a recursive solution to this which is $O(log\, n)$. Write one.

5. (Pseudocode)

   Express the following verbal description in pseudocode:

   In the first phase, we iteratively pop all the elements from stack S and enqueue them in queue Q, then dequeue the element from Q and push them back onto S.

   As a result, all the elements are now in reversed order on S.

   In the second phase, we again pop all the elements from S, but this time we also look for the element x.

   By again passing the elements through Q and back onto S, we reverse the reversal, thereby restoring the original order of the elements on S.

6. (Pseudocode)

   Express the following pseudcode fragments in C:

   a. **A** is an array of **integer** values

      ...
      swap A[i] and A[j]
      ...

   b. **head** points to beginning of linked list

      ...
      swap the first and second elements in the list
      ...

   c. **S** is a stack

      ...
      swap the top two elements on S
      ...

   Define any data structures that you need in C.

7. (Counting primitive operations)

   We say that there are 5 primitive operations in

   C[i,j] = C[i,j] + A[i,k]·B[k,j]

   Explain how this is derived?

8. (Counting primitive operations)

   We quote the number of operations for

   for all i **in** 1..n **do** ...

   as *n + (n+1)* or *2n + 1*

   Explain how this is derived?

9. (Counting primitive operations)

   The following algorithm

   - takes a sorted array A[1..*n*] of characters, and
   - outputs, in reverse order, all 2-letter words νω such that ν ≤ ω.

   Count the number of primitive operations (evaluating an expression, indexing into an array). What is the time complexity of this algorithm in big-O notation?

```
for all i = n down to 1 do
    for all j = n down to i do
        print A[i] A[j]
    end for
end for
```

10. (Algorithms and complexity)

    Determine the complexity of the following algorithm:

```
splitList(L):
   input: non-empty linked list L
   output: L split into two halves
   // use slow and fast pointers to traverse L
   slow=head(L), fast=head(L).next
   while fast≠NULL and fast.next≠NULL do
      // advance pointers
      slow=slow.next, fast=fast.next.next
   end while
   cut L between slow and slow.next
```

11. (Algorithm Complexity)

    What is the complexity of the following algorithm?

```
binaryConversion(n):
    **input**: positive integer n
    **output**: binary representation of n on a stack

    create empty stack S
    **while** n>0 **do**
        push (n mod 2) onto S
        n = ⌊ n/2 ⌋
    **end while**
    **return** S
```

Assume that creating a stack and pushing an element are both $O(1)$ operations (i.e. "constant")

12. (Algorithms and complexity)

    Develop an algorithm to determine if a character array of length *n* encodes a *palindrome* — that is, it reads the same forward and backward. For example, "racecar" is a palindrome.

    a. Write the algorithm in pseudocode.

    b. Analyse the time complexity of your algorithm.

    c. Implement your algorithm in C. Your program should accept a single command line argument, and check whether it is a palindrome. Examples of the program executing are

    ```
    $ ./palindrome racecar
    yes
    $ ./palindrome reviewer
    no
    ```

    Hint: You may use the standard library function *strlen(3)*, which has prototype `size_t strlen(char *)`, is defined in `<string.h>`, and which computes the length of a string (without counting its terminating `'\0'`-character).

13. (Algorithms and complexity)

    Let $p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ be a polynomial of degree n. Design an $O(n)$-time algorithm for computing $p(x)$.

    Hint: Assume that the coefficients a_i are stored in an array A[0..n].

14. (Algorithms and complexity)

    A vector V is called *sparse* if most of its elements are 0. In order to store sparse vectors efficiently, we can use a list `L` to store only its non-zero elements. Specifically, for each non-zero element $V[i]$, we store an index-value pair $(i, V[i])$ in `L`. We call `L` the *compact form* of V.

    For example, the 8-dimensional vector $V = (2.3, 0, 0, 0, -5.61, 0, 0, 1.8)$ could be stored in a list `L` of size 3: $[(0, 2.3), (4, -5.61), (7, 1.8)]$.

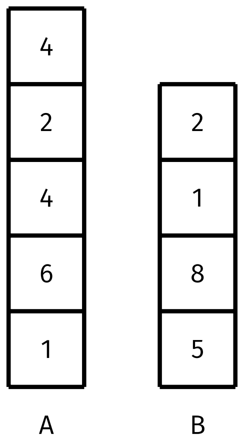
Describe an efficient algorithm for adding two sparse vectors V_1 and V_2 of equal dimension, but given in compact form. The result should be in compact form too. What is the time complexity of your algorithm, depending on the sizes m and n of the compact forms of V_1 and V_2, respectively?

Hint: The sum of two vectors V_1 and V_2 is defined as usual; for example, (2.3,-0.1,0,0,1.7,0,0,0) + (0,3.14,0,0,-1.7,0,0,-1.8) = (2.3,3.04,0,0,0,0,0,-1.8).

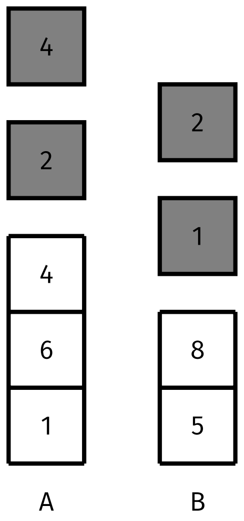
15. **Challenge Exercise**

Suppose that you are given two stacks of non-negative integers A and B and a target threshold $k \geq 0$. Your task is to determine the maximum number of elements that you can pop from A and B so that the sum of these elements does not exceed k.

For example, given the stacks



The maximum number of elements that can be popped without exceeding $k = 10$ is 4:



If $k = 7$, then the answer would be 3: the top element of A and the top two elements of B.

a. Write an algorithm (in pseudocode) to determine this maximum for any given stacks A and B and threshold k. As usual, the only operations you can perform on the stacks are **pop** and **push**. You *are* permitted to use a third "helper" stack, but no other aggregate data structure.

b. Determine the time complexity of your algorithm depending on the sizes m and n of input stacks A and B.

Hints:

- A so-called greedy algorithm would simply take the smaller of the two elements currently on top of the stacks and continue to do so as long as you haven't exceeded the threshold. This won't work in general for this problem.
- Your algorithm only needs to determine the number of elements that can maximally be popped without exceeding the given *k*. You do not have to return the numbers themselves nor their sum. Also you do not need to restore the contents of the two stacks; they can be left in any state you wish.