# Design and Analysis of Algorithms — Lab

R S Milton, C Aravindan, T T Mirnalinee
Department of CSE, SSN College of Engineering

Session 1: Programming in Python – Review

## 1 Fast Exponentiation

1. Design an efficient algorithm to compute $M^n$ where $M$ is a matrix, using the idea:
   - ▷ Convert $n$ to a binary bit string $b_{k\cdots 0}$.
   - ▷ Initialize `term` to $M$
   - ▷ Initialize `result` to $M$ if $b_0 = 1$, otherwise initialize to $1$
   - ▷ Iterate over the bits $b_1$ to $b_k$:
     - `term` = `term` * `term`
     - If $b_i = 1$, then `result` = `result` * `term`
2. Write a function to compute $f_n$, the $n$th Fibonacci number, using your fast matrix exponentiation algorithm.

## 2 Selection sort

1. Write a function `print_array(a, low, high)` to print the items of a subarray `a[low:high]` on the screen. Test it.
2. Write a function `read_array()` to read a list of integers from the keyboard and return a list `a` those integers and the count `n` of integers read.
3. Write a function `minimum(a)` that finds the smallest number in an array `a`. Input is an array `a[0:n]` of `n` comparable items. Output should be the index of the smallest item in `a[0:n]`. Test the function interactively and from a `main()` function. Test it for several lists of numbers where each test should read a list of numbers from the keyboard.

4. Modify `minimum(a)` to `minimum(a, low, high)`. You are given an array `a[0:n]` of `n` comparable items. Define the function `minimum(a, low, high)` that returns the index of the smallest item in the subarray `a[low:high]`.

5. **Selection sort**: Selection sort is an algorithm for sorting an array of items, say `a[0:n]`. The idea of the algorithm is expressed below:

   ```
   bring the smallest item to 0
   bring the next smallest item to 1
   bring the next smallest item to 2
   ...
   bring the largest item to n-1
   ```

   This roughly translates to

   ```
   swap minimum(a,0,n) with 0
   swap minimum(a,1,n) with 1
   swap minimum(a,2,n) with 2
   ...
   swap minimum(a,n-2,n) with n-2
   ```

   which uses `minimum(a, i, n)` to find the minimum of a subarray `a[i:n]`. Implement selection sort, using `minimum()` function. Note: remember that when a function changes the items of an array parameter, the changes are effected in the items of the actual array argument also.

   Test the function from a `main()` for several lists of numbers. Each test should read a list of numbers from stdin.

# 3 Insertion sort

Develop functions useful for implemeting `insertion_sort`.

1. We are given a list of sorted items. Suppose we want to insert a new item, called the *target*, to the list and maintain the resulting list sorted. To achieve this, we have to insert the target after a smaller item and before an item no smaller than the target. Items of a list are indexed from 0. Therefore, starting from index 0, the target should be inserted before the first item no smaller than the target. At what index should we insert the target in the list? We only want the index and do not actually insert the target. Construct a recursive algorithm. Name it `linear_locate`.

```
linear_locate (15, [5, 10, 20, 35, 50])
2
linear_locate (35, [5, 10, 20, 35, 50])
3
linear_locate (2, [5, 10, 20, 35, 50])
0
linear_locate (25, [])
0
```

2. Design `linear_locate` of Question 1 as a tail recursive algorithm.

3. Design `linear_locate` of Question 2 as an iterative algorithm equivalent to the tail recursive algorithm.

4. Algorithm `linear_locate` returns the right position for the target. Consider a slightly different problem: Linear search. If the target is in the list, the output should be its index in the list. Otherwise, the output should be any invalid index, say $-1$. Construct `linear_search` using `linear_locate`.

```
linear_search (15, [5, 10, 20, 35, 50])
-1
linear_search (35, [5, 10, 20, 35, 50])
3
linear_search (2, [5, 10, 20, 35, 50])
0
linear_search (25, [])
0
```

5. Algorithm, `ordered_insert`, should insert the target at its right position in a sorted list and create a new sorted list. Implement `ordered_insert (u, v)` as a recursive function. `u` is an item, and `v` is a sorted list. The algorithm will have the same outline as `linear_locate`, and differ only in the way the solution is constructed from the subsolution.

```
ordered_insert (15, [5, 10, 20, 35, 50])
[5, 10, 15, 20, 35, 50]
ordered_insert (35, [5, 10, 20, 35, 50])
[5, 10, 20, 35, 35, 50]
ordered_insert (2, [5, 10, 20, 35, 50])
[2, 5, 10, 20, 35, 50]
ordered_insert (25, [])
[25]
```

6. Construct an iterative algorithm for ordered insert of Question 5.

7. Implement `insertion_sort (v)` as a function using the idea

$$\text{sort (v)} = \begin{cases} [\,] & \text{v = [ ]} \\ \text{ordered\_insert (head (v), sort (tail (v)))} & \text{otherwise} \end{cases}$$

8. Algorithm of Question 6 inserts an item `t` into a sorted array `a[0:n]`. Modify it to algorithm `oinsert (a, j)` which requires that *subarray* `a[0:j]` is sorted and inserts item `a[j-1]` to `a[0:j]`. After insertion, it ensures that subarray `a[0:j+1]` is sorted.

   How do we do it? Let us refer to `a[j-1]` as the target. Starting from `a[j-2]`, keep swapping the target with the previous items of the array until the target comes to its correct position.

9. Solve Question 8, but use a slightly different iterative step: back up `a[j-1]` in a variable `target`. Then, starting from `a[j-2]`, keep shifting the items of the array to the right until the correct position for `target` is found; then, in that position, insert `target`.

10. Implement algorithm of Question 7 using array. You are given as input an array `a[0:n]` of `n` numbers. The output should be sorted `a[0:n]`. Decompose the problem into subarray `a[0:n-1]` and item `a[n-1]`. Sort `a[0:n-1]` recursively. Then, insert `a[n-1]` in the right place in `a[0:n-1]` using `oinsert(a, n)`. Imlement a recursive version and an iterative verion.

11. Implement the iterative algorithm of Question 7 with two nested loops, the inner loop being the functionality of `oinsert(a, n)`.

12. Algorithm of Question 8 inserts item `a[j-1]` into a sorted subarray `a[0:j-1]`. Instead, suppose subarray `a[i+1:n]` is sorted and we need to insert item `a[i]` into `a[i+1:n]` so that `a[i:n]` becomes sorted. Modify the algorithm.

# 4 Heapsort

We will develop functions useful for implemeting `heapsort`.

1. The input is an array of sorted items. The items are stored only at positions 1, 2, 4, 8, 16, ...(powers of 2). We do not store anything in between. The items of the array are sorted except `a[1]`. Modify algorithm of Section 3, Question 12 so that the items of the array become sorted.

# 5 Mergesort

We will develop functions useful for implemeting `mergesort`.

1. Two sorted lists `u` and `v` are given as input. Construct a recursive algorithm `ordered_merge (u, v)` which inserts each item of `u` in its right position in `v` and constructs a new sorted list as the output. Design `ordered_merge (u, v)` using `ordered_insert` Section 3 Question 5.

   ```
   ordered_merge ([15, 40, 45], [5, 10, 20, 35, 50])
   [5, 10, 15, 20, 35, 40, 45, 50]
   ordered_merge ([30, 60], [20, 35, 50])
   [20, 30, 35, 50, 60]
   ordered_merge ([], [20, 35, 50])
   [20, 35, 50]
   ```

2. Construct an iterative version of the algorithm for Question 1