Fast Sorting

Topics for this week:
- Merge Sort
- Quick Sort
- their time-complexity (introducing $O(n \log n)$)

Announcements about the project:
- How to test
- Deepcopy issues

Midterm:
- Solutions of problems 4 and 5
Problem: Creating a book index

We need to create a book index, i.e., an alphabetically sorted list of words from the book, along with the page numbers.

Note: we’ll work with text files and instead of page numbers, we will print out line numbers.
We redefined the problem: having a (very long) list of words, how do we efficiently sort them alphabetically?

A **divide-and-conquer** idea:

```python
def merge_sort(list):
    n = len(list)
    if n == 1:
        return list
    c1, c2, n = steps
    list_A, list_B = split_list_into_two_lists_of_equal_size(n)
    sorted_list_A = merge_sort(list_A)
    sorted_list_B = merge_sort(list_B)
    return merge(sorted_list_A, sorted_list_B)
```

To solve a problem by cutting it into (usually 2) smaller problems, solve these problems and put the solutions together.

Let \( c = c_1 + c_2 \)
Merge Sort

Stack trace diagram for input [5,7,1,3,4,8,6,2]:
Merge Sort

Merge function:

```python
def merge(A, B):
    # pre-condition: listA and listB are sorted
    i = 0
    j = 0
    k = 0
    while i < len(A) and j < len(B):
        if A[i] < B[j]:
            C[k] = A[i]
            i += 1
        else:
            C[k] = B[j]
            j += 1
        k += 1
    while i < len(A):
        C[k] = A[i]
        i += 1
        k += 1
    while j < len(B):
        C[k] = B[j]
        j += 1
        k += 1
    return C
```

Example:

- A = [2, 3, 9, 10]
- B = [5, 6, 7, 8]
- C = merge(A, B)
  - C = [2, 3, 5, 6, 7, 8, 9, 10]

- A = [2, 3, 3, 5]
- B = [1, 3, 5, 5]
- C = merge(A, B)
  - C = [1, 2, 3, 3, 5, 5]

Works even for duplicate numbers.
Merge Sort

Running time:

- will spend $c_1 \cdot n$ steps + then recursion step
- will spend $2 \cdot c_1 \cdot \frac{n}{2}$ steps + recursion
  - $4 \cdot c_1 \cdot \frac{n}{4}$
  - $8 \cdot c_1 \cdot \frac{n}{8}$
  - $16 \cdot c_1 \cdot \frac{n}{16}$
  - ...
- merge takes $2 \cdot c_2 \cdot \frac{n}{2}$
- merge takes $c_2 \cdot n$ steps

all together: $c_1 \cdot n + c_1 \cdot n + \ldots + c_1 \cdot n + c_2 \cdot n + c_2 \cdot n + \ldots + c_2 \cdot n$

$= c \cdot n \cdot \#levels = O(n \log n)$

$\#levels = \log_2 n$
Quick Sort

- an alternative (the most commonly used) sorting algorithm

\[ x = 6, 3, 1, 7, 8, 5, 2, 10, 14, 2, 8, 5 \]

for pivot 6:
\[ A = 3, 1, 5, 2, 2, 5 \]
\[ B = 6 \]
\[ C = 7, 8, 10, 14, 8 \]

idea:
choose a pivot \( x \)
rarrange the list into three lists: smaller than \( x \), equal \( x \), larger than \( x \)
return quickSort(\( A \)) + \( B \) + quickSort(\( C \))

\[ \text{concatenation} \]
Quick Sort

Running time:

if pivot splits the list into A and C of roughly equal size, then the running time is like MergeSort:

\[ O(n \log n) \]

However, if pivot is the smallest element, then A is empty and C is of size \( n-1 \). If this happens repeatedly, then the running time is:

\[
c \cdot n + c \cdot (n-1) + c \cdot (n-2) + c \cdot (n-3) + \ldots + c \cdot 2 + c \cdot 1 =
\]

\[
= c \left( n + n-1 + n-2 + \ldots + 2 + 1 \right) = c \cdot (n+1) \cdot \frac{n}{2} = O(n^2)
\]

In the worst case, the running time is \( O(n^2) \). \( \Leftarrow \) bad news

Good news: typically, when pivot chosen as a random element from the list, the running time is: \( O(n \log n) \)