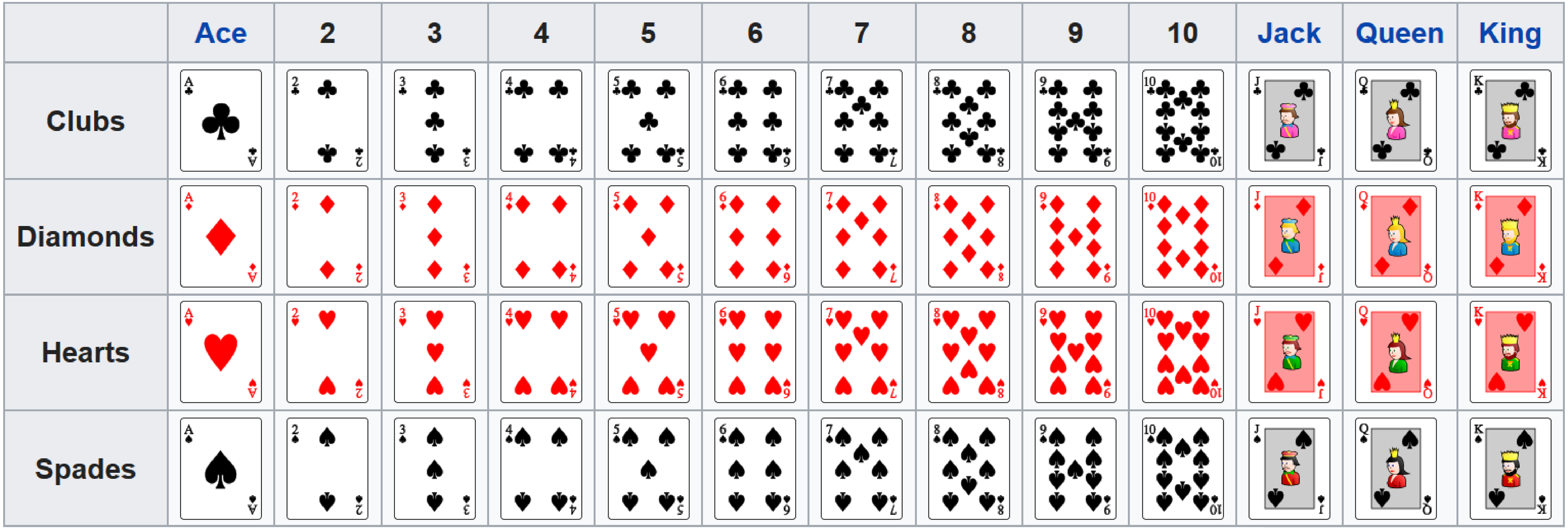
Recursion and iteration

- algorithm examples

Standard 52-card deck

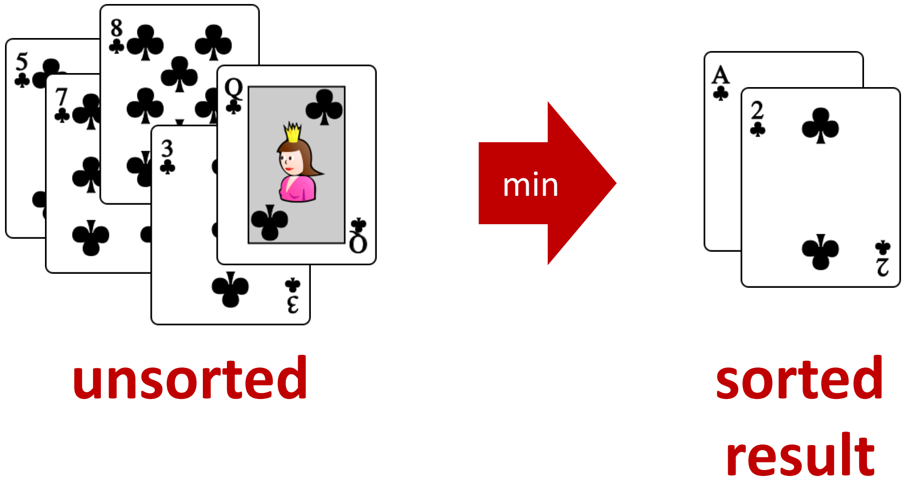


en.wikipedia.org/wiki/Standard_52-card_deck

Selection sort

selection_sort.py

```python
def selection_sort(L):
    unsorted = L[:]
    result = []

    while unsorted:
        e = min(unsorted)
        unsorted.remove(e)
        result.append(e)

    return result
```

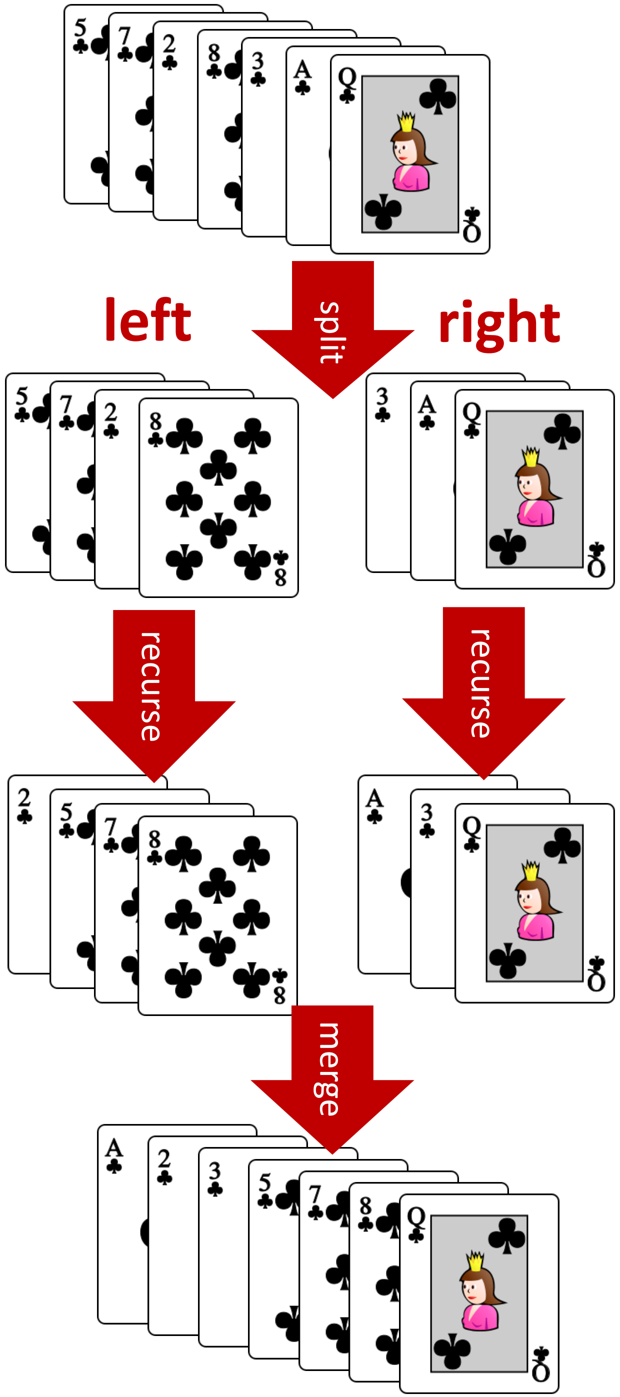


- `min` and `.remove` scan the remaining `unsorted` list for each element moved to `result`
- order $|L|^2$ comparisons

Sorting a pile of cards (Merge sort)

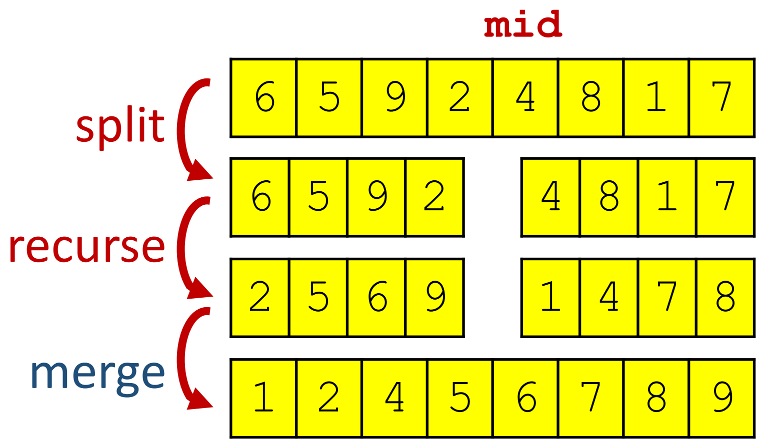
- If one card in pile, i.e. pile is sorted
- Otherwise

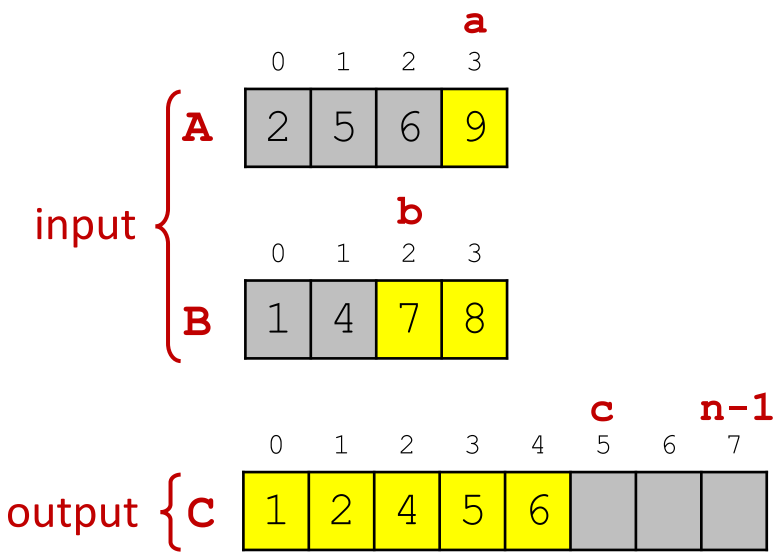
  1) Split pile into two piles, **left** and **right**, of approximately same size
  2) Sort **left** and **right** recursively (independently)
  3) Merge **left** and **right** (which are sorted)



merge_sort.py

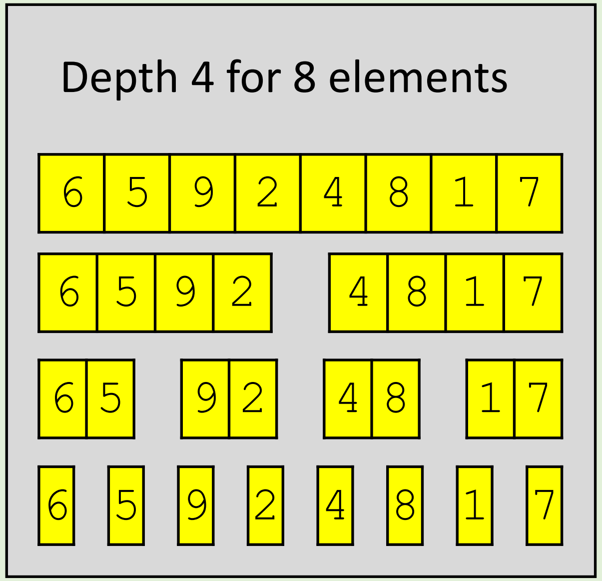
```
def merge_sort(L):
    n = len(L)
    if n <= 1:
        return L[:]
    mid = n // 2
    left, right = L[:mid], L[mid:]
    return merge(merge_sort(left), merge_sort(right))

def merge(A, B):
    n = len(A) + len(B)
    C = n * [None]
    a, b = 0, 0
    for c in range(n):
        if a < len(A) and (b == len(B) or A[a] < B[b]):
            C[c] = A[a]
            a = a + 1
        else:
            C[c] = B[b]
            b = b + 1
    return C
```





Question – Depth of recursion for 52 elements

a) 1
b) 2
c) 3
d) 4
e) 5
f) 6
g) 7
h) 8
i) 9
j) 10
k) Don't know



Question – Order of comparisons by Merge sort ?

a) $\sim n$

b) $\sim n\sqrt{n}$

c) $\sim n \log_2 n$

d) $\sim n^2$

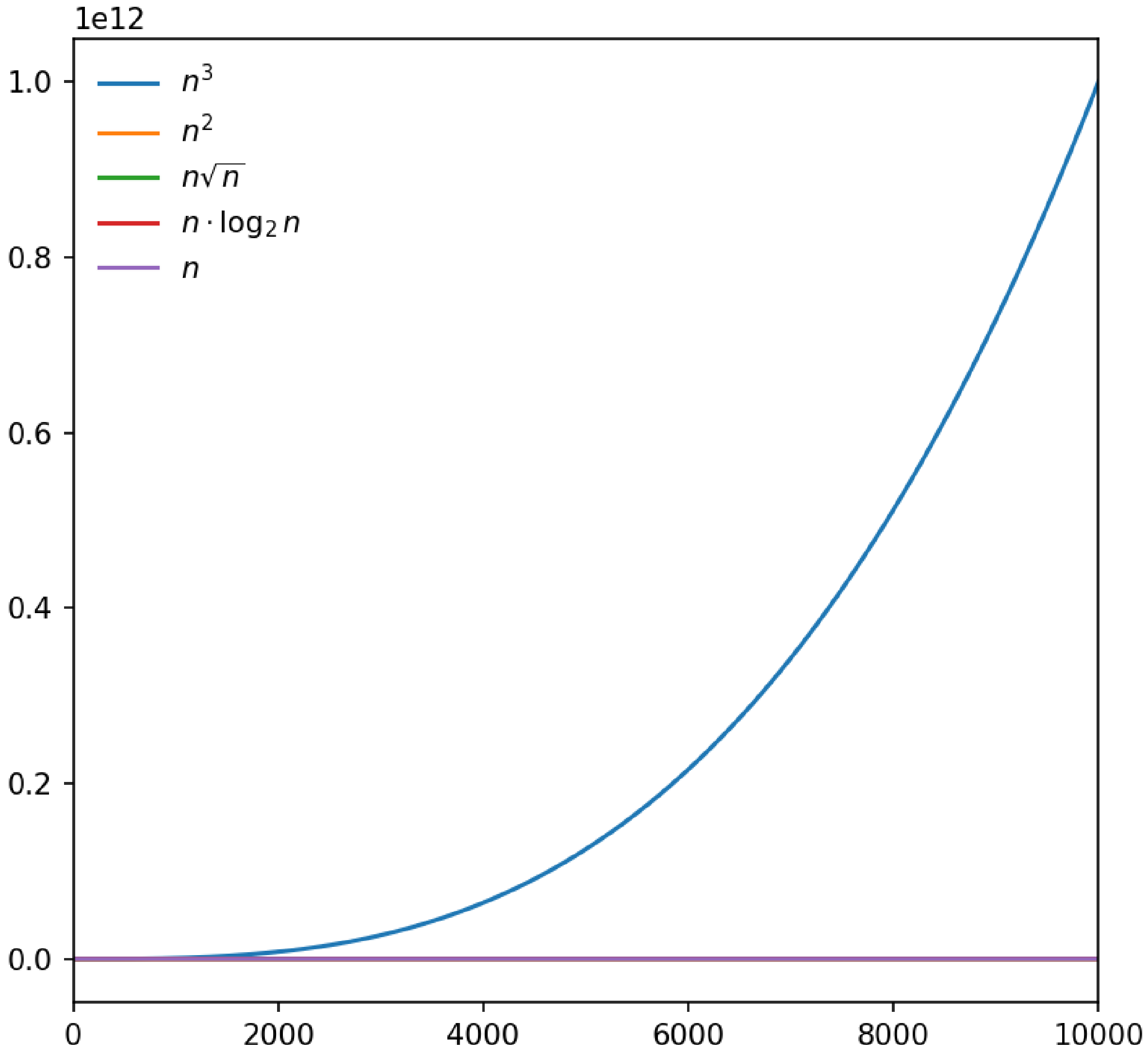
e) $\sim n^3$

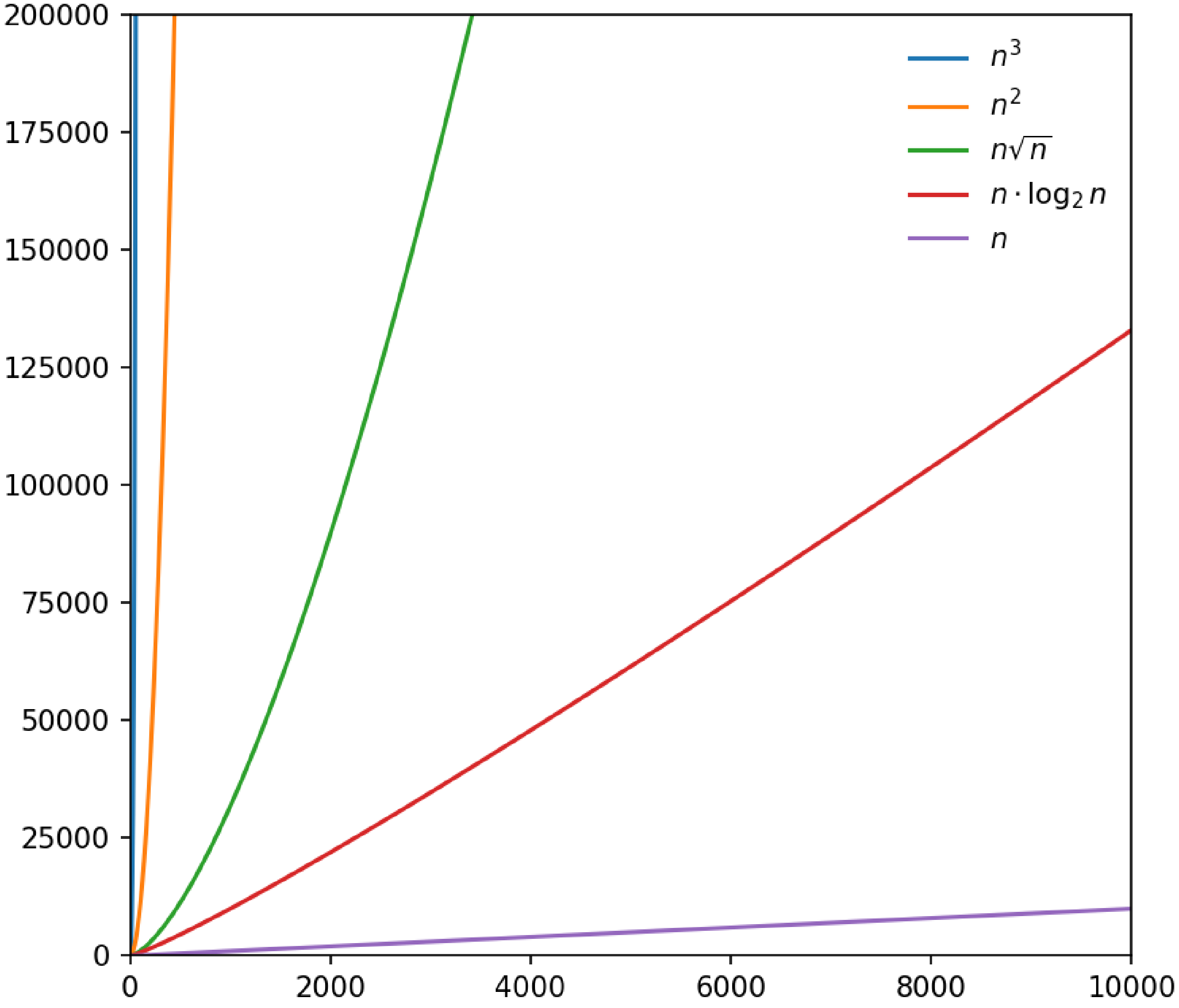
f) Don't know

merge_sort.py

```
def merge_sort(L):
    n = len(L)
    if n <= 1:
        return L[:]
    else:
        mid = n // 2
        left, right = L[:mid], L[mid:]
        return merge(merge_sort(left), merge_sort(right))

def merge(A, B):
    n = len(A) + len(B)
    C = n * [None]
    a, b = 0, 0
    for c in range(n):
        if a < len(A) and (b == len(B) or A[a] < B[b]):
            C[c] = A[a]
            a = a + 1
        else:
            C[c] = B[b]
            b = b + 1
    return C
```

Growth of some functions





Merge sort without recursion

- Start with piles of size one
- Repeatedly merge two smallest piles

merge_sort.py

```python
def merge_sort_iterative(L):
    Q = [[x] for x in L]
    while len(Q) > 1:
        Q.insert(0, merge(Q.pop(), Q.pop()))
    return Q[0]

from collections import deque

def merge_sort_deque(L):
    Q = deque([[x] for x in L])
    while len(Q) > 1:
        Q.appendleft(merge(Q.pop(), Q.pop()))
    return Q[0]
```

⚠ insert at front of list inefficient

deques are a generalization of lists with efficient updates at both ends

```
merge_sort_iterative([7,1,9,3,-2,5])
```

Values of Q in while-loop

```
[[7], [1], [9], [3], [-2], [5]]
[[-2, 5], [7], [1], [9], [3]]
[[3, 9], [-2, 5], [7], [1]]
[[1, 7], [3, 9], [-2, 5]]
[[-2, 3, 5, 9], [1, 7]]
[[-2, 1, 3, 5, 7, 9]]
```

Note: Lists in Q appear in non-increasing length order, where longest ≤ 2· shortest

Question – Number of iterations of while-loop ?

```
merge_sort_iterative([7, 1, 9, 3, -2, 5])
```

a) 1
b) 2
c) 3
d) 4
e) 5
f) 6
g) 7
h) Don't know

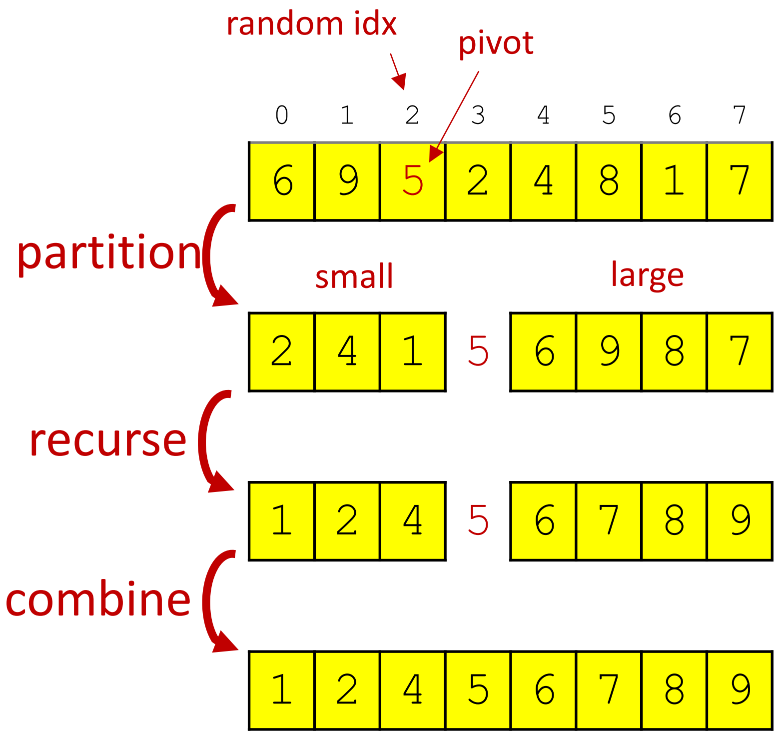
merge_sort.py

```
def merge_sort_iterative(L):
    Q = [[x] for x in L]
    while len(Q) > 1:
        Q.insert(0, merge(Q.pop(), Q.pop()))
    return Q[0]
```

Quicksort (randomized)

quicksort.py

```
import random

def quicksort(L):
    if len(L) <= 1:
        return L[:]

    idx = random.randrange(len(L))
    pivot = L[idx]

    small = [e for e in L if e < pivot]
    equal = [e for e in L if e == pivot]
    large = [e for e in L if e > pivot]
    return quicksort(small) + equal + quicksort(large)
```



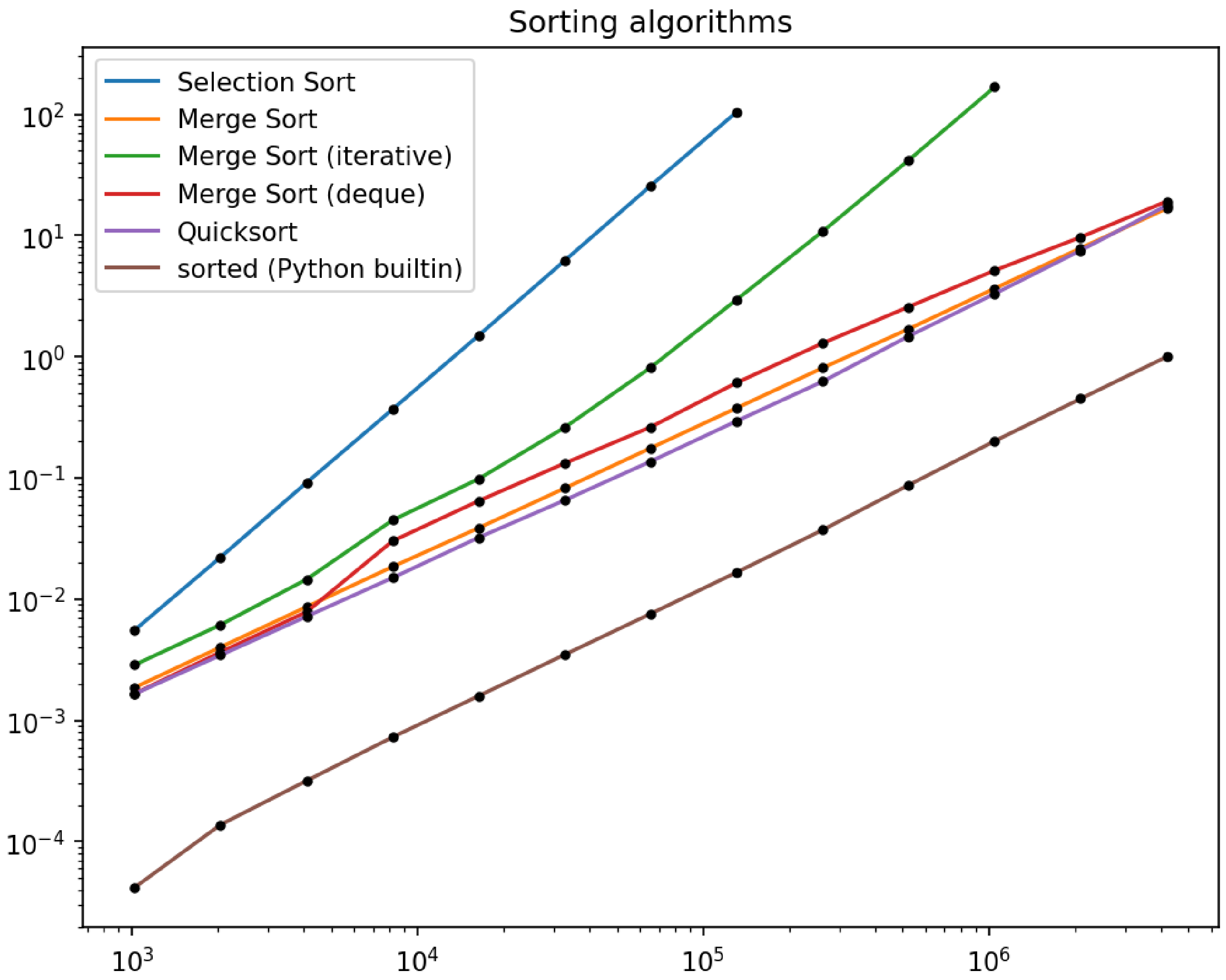
order $|L| \cdot \log_2 |L|$ comparisons, expected

Sorting comparison (single run)

tuned merge-sort (Tim-sort) implementation in C

$\|L\|$	Selection sort	Merge sort Recursive	Merge sort Iterative	Merge sort Deque	Quicksort	sorted (Python builtin)
2^{10}	0.006	0.002	0.003	0.002	0.002	0.00004
2^{11}	0.02	0.004	0.006	0.000	0.003	0.0001
2^{12}	0.09	0.008	0.01	0.008	0.007	0.0003
2^{13}	0.37	0.02	0.04	0.03	0.02	0.0007
2^{14}	1.50	0.04	0.10	0.06	0.03	0.002
2^{15}	6.19	0.08	0.26	0.13	0.07	0.003
2^{16}	25.67	0.18	0.81	0.26	0.14	0.008
2^{17}	104.20 (x 4)	0.38	2.96	0.61	0.29	0.02
2^{18}		0.81	10.78	1.29	0.62	0.04
2^{19}		1.69	41.71	2.58	1.48	0.09
2^{20}		3.65	167.31 (x 4)	5.15	3.30	0.20
2^{21}		7.85		9.68	7.53	0.45
2^{22}		16.69 (x 2)		19.09 (x 2)	17.6 (x 2)	1.00 (x 2)

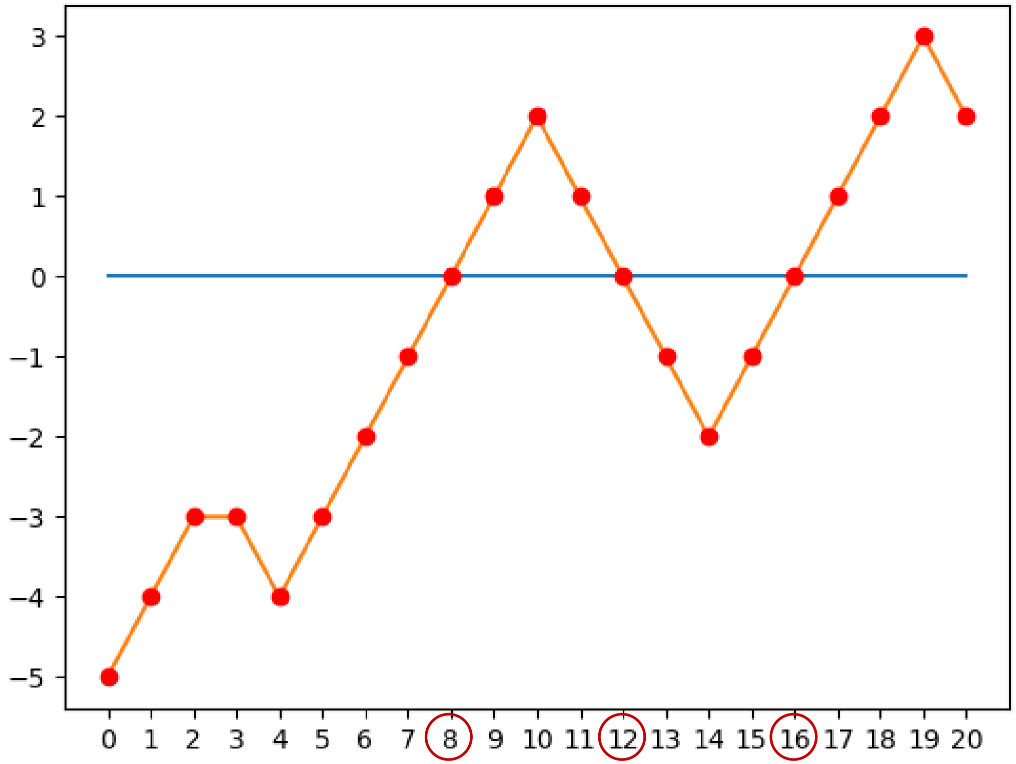
Sorting comparison



Find zero

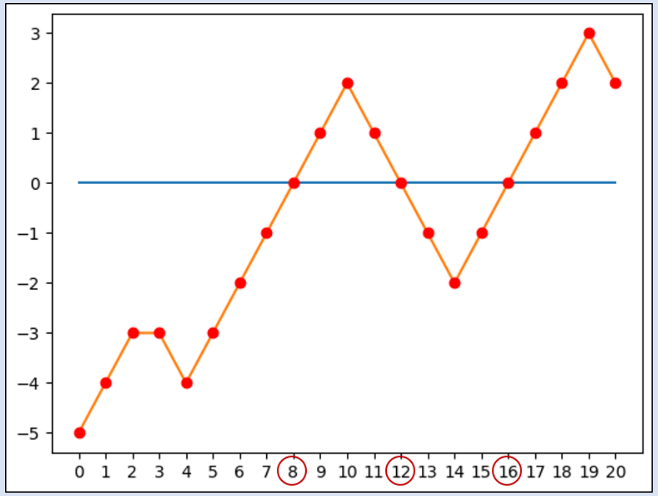
- Given a list L of integers starting with a negative and ending with a positive integer, and where |L[i+1] - L[i]| ≤ 1, find the position of a zero in L.

L = [-5, -4, -3, -3, -4, -3, -2, -1, 0, 1, 2, 1, 0, -1, -2, -1, 0 , 1, 2, 3, 2]



find_zero.py

```python
def find_zero_loop(L):
    i = 0
    while L[i] != 0:
        i += 1
    return i

def find_zero_enumerate(L):
    for i, e in enumerate(L):
        if e == 0:
            return i

def find_zero_index(L):
    return L.index(0)
```




```python
def find_zero_binary_search(L):
    low = 0
    high = len(L) - 1
    while True:  # L[low] < 0 < L[high]
        mid = (low + high) // 2
        if L[mid] == 0:
            return mid
        elif L[mid] < 0:
            low = mid
        else:
            high = mid

def find_zero_recursive(L):
    def search(low, high):
        mid = (low + high) // 2
        if L[mid] == 0:
            return mid
        elif L[mid] < 0:
            return search(mid, high)
        else:
            return search(low, mid)

    return search(0, len(L) - 1)
```

find_zero.py

```python
def find_zero_loop(L):
    i = 0
    while L[i] != 0:
        i += 1
    return i

def find_zero_enumerate(L):
    for i, e in enumerate(L):
        if e == 0:
            return i

def find_zero_index(L):
    return L.index(0)
```

Function ($\|L\| = 10^6$)	Time, sec
find_zero_loop	0.13
find_zero_enumerate	0.10
find_zero_index	0.015
find_zero_binary_search	0.000015
find_zero_recursive	0.000088

```python
def find_zero_binary_search(L):
    low = 0
    high = len(L) - 1
    while True:  # L[low] < 0 < L[high]
        mid = (low + high) // 2
        if L[mid] == 0:
            return mid
        elif L[mid] < 0:
            low = mid
        else:
            high = mid

def find_zero_recursive(L):
    def search(low, high):
        mid = (low + high) // 2
        if L[mid] == 0:
            return mid
        elif L[mid] < 0:
            return search(mid, high)
        else:
            return search(low, mid)

    return search(0, len(L) - 1)
```

Greatest Common Divisor (GCD)

Notation $x \uparrow y$ denotes y is divisible by x, e.g. $3 \uparrow 12$
i.e. $y = a \cdot x$ for some integer a

Definition $\gcd(m, n) = \max \{ x \mid x \uparrow m \text{ and } x \uparrow n \}$

Fact if $x \uparrow y$ and $x \uparrow z$ then $x \uparrow (y + z)$ and $x \uparrow (y - z)$

Observation
(recursive definition)

$$\gcd(m, n) = \begin{cases} m & \text{if } m = n \\ \gcd(m, n - m) & \text{if } m < n \\ \gcd(m - n, n) & \text{if } m > n \end{cases}$$

gcd(90, 24)

m	n
90	24
66	24
42	24
18	24
18	6
12	6
6	6

Greatest Common Divisor (GCD)

gcd_slow.py

```
def gcd(m, n):
    while m != n:
        if n > m:
            n = n - m
        else:
            m = m - n
    return m
```

gcd.py

```
def gcd(m, n):
    while n != 0:
        m, n = n, m % n
    return m
```

gcd_slow_recursive.py

```
def gcd(m, n):
    if m == n:
        return m
    elif m > n:
        return gcd(m - n, n)
    else:
        return gcd(m, n - m)
```

gcd_recursive.py

```
def gcd(m, n):
    if n == 0:
        return m
    else:
        return gcd(n, m % n)
```

gcd_recursive_one_line.py

```
def gcd(m, n):
    return m if n == 0 else gcd(n, m % n)
```

Permutations

- Generate a list L of all permutations of a tuple

Python shell

```
> permutations(('a', 'b', 'c'))
| [('a', 'b', 'c'), ('b', 'a', 'c'), ('b', 'c', 'a'),
   ('a', 'c', 'b'), ('c', 'a', 'b'), ('c', 'b', 'a')]
```

permutations.py

```python
def permutations(L):
    if len(L) == 0:
        return [L[:]]  # empty tuple (ensures same type as L)
    else:
        P = permutations(L[1:])
        return [p[:i] + L[:1] + p[i:] for p in P for i in range(len(L))]
```

- An implementation of `permutations` exists in the itertools module

Maze solver

Input

- First line #rows and #columns
- Following #rows lines contain strings containing #column characters
- There are exactly one 'A' and one 'B'
- '.' are free cells and '#' are blocked cells

Output

- Print whether there is a path from 'A' to 'B' or not

maze input

```
11 19
#######A###########
#.......#.....#...#
#.###.###...#.#.#.#
#...#.....#.#...#.#
#.#.###.#.#.#.###.#
#.#.....#...#.#...#
#.###########.#.#.#
#.#.#.....#...#.#.#
#.#.#####.#####.#.#
#.........#.....#.#
###############B###
```

Maze solver (recursive)

maze_solver.py

```python
def explore(i, j):
    global solution, visited

    if (0 <= i < n and 0 <= j < m and
        maze[i][j] != '#' and not visited[i][j]):

        visited[i][j] = True

        if maze[i][j] == 'B':
            solution = True

        explore(i - 1, j)
        explore(i + 1, j)
        explore(i, j - 1)
        explore(i, j + 1)
```

```python
def find(symbol):
    for i, row in enumerate(maze):
        j = row.find(symbol)
        if j >= 0:
            return (i, j)

n, m = [int(x) for x in input().split()]
maze = [input() for i in range(n)]

solution = False
visited = [m * [False] for i in range(n)]

explore(*find('A'))

if solution:
    print('path from A to B exists')
else:
    print('no path')
```

maze input

```
11 19
#######A###########
#.......#.....#...#
#.###.###...#.#.#.#
#...#.....#.#...#.#
#.#.###.#.#.#.###.#
#.#.....#...#.#...#
#.###########.#.#.#
#.#.#.....#...#.#.#
#.#.#####.#####.#.#
#.........#.....#.#
###############B###
```

Maze solver (iterative)

maze_solver_iterative.py

```
def explore(i, j):
    global solution, visited

    Q = [(i, j)]  # cells to visit

    while Q:
        i, j = Q.pop()
        if (0 <= i < n and 0 <= j < m and
            maze[i][j] != '#' and not visited[i][j]):

            visited[i][j] = True

            if maze[i][j] == 'B':
                solution = True

            Q.append((i - 1, j))
            Q.append((i + 1, j))
            Q.append((i, j - 1))
            Q.append((i, j + 1))
```

```
def find(symbol):
    for i, row in enumerate(maze):
        j = row.find(symbol)
        if j >= 0:
            return (i, j)

n, m = [int(x) for x in input().split()]
maze = [input() for i in range(n)]

solution = False
visited = [m*[False] for i in range(n)]

explore(*find('A'))

if solution:
    print("path from A to B exists")
else:
    print("no path")
```