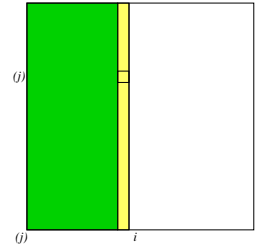
This week (Chapter 2):

- Abstract data types (Today)
- Data Structures (Wednesday and Friday)
- Programming practices (Friday)

Today:

- Analyses of merge sort and quick sort
- Exercises
- Definition *abstract data type*, especially in contrast with *data structure*
- The "canonical" ADTs
- Start data structures (time permitting)

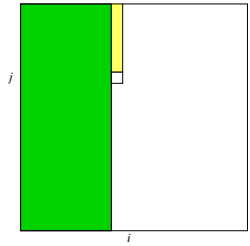
```
def mat_find1(M, x):
    i = 0
    found = False
    while not found and i < len(M):
        j = 0
        while not found and j < len(M[i]) :
            found = M[i][j] == x
            j += 1
        i += 1
    if found :
        return (i-1, j-1)
    else :
        return None
```



Invariant (Outer loop of `mat_find1`)

1. $\forall\ a \in [0, i-1), \forall\ b \in [0, m), \mathtt{M}[a][b] \neq x$
2. $\sim$ `found` *iff* $\forall\ b \in [0, m), \mathtt{M}[i-1][b] \neq x$
3. `found` *iff* $\mathtt{M}[i-1][j-1] = x$
4. *i is the number of iterations of the outer loop completed.*

```
def mat_find1(M, x):
    i = 0
    found = False
    while not found and i < len(M):
        j = 0
        while not found and j < len(M[i]) :
            found = M[i][j] == x
            j += 1
        i += 1
    if found :
        return (i-1, j-1)
    else :
        return None
```



Invariant (Inner loop of `mat_find1`)

1. $\forall\ b \in [0, j-1), \texttt{M}[i][b] \neq x$
2. `found` *iff* $\texttt{M}[i][j-1] = x$
3. *j is the number of iterations of the inner loop completed on the current iteration of the outer loop.*

```
def mat_find1(M, x):
    i = 0
    found = False
    while not found and i < len(M):
        j = 0
        while not found and j < len(M[i]) :
            found = M[i][j] == x
            j += 1
        i += 1
    if found :
        return (i-1, j-1)
    else :
        return None
```

In the worse case, each position in the array is read once, hence $\Theta(m^2)$ or $\Theta(n)$.

```
def mat_find2(M, x):
    i = len(M) - 1
    j = 0
    found = False
    while not found and i >= 0  and j < len(M[i]):
        while i >= 0 and M[i][j] > x :
            i -= 1
        while i >= 0 and j < len(M[i]) and M[i][j] < x :
            j += 1
        if i >= 0 and j < len(M[i]) :
            found = M[i][j] == x

    if found :
        return (i, j)
    else :
        return None
```

1	2	8	21	43	57	92	103
3	5	9	23	44	61	93	105
17	22	27	30	46	62	95	106
37	39	42	47	48	69	99	108
64	67	71	75	76	77	101	110
73	74	81	88	89	91	107	119
92	96	100	102	103	106	111	121
115	116	126	131	138	146	152	160

```
def mat_find2(M, x):
    i = len(M) - 1
    j = 0
    found = False
    while not found and i >= 0  and j < len(M[i]):
        while i >= 0 and M[i][j] > x :
            i -= 1
        while i >= 0 and j < len(M[i]) and M[i][j] < x :
            j += 1
        if i >= 0 and j < len(M[i]) :
            found = M[i][j] == x

    if found :
        return (i, j)
    else :
        return None
```

On any iteration of the outer loop, at least one of the inner loops must have at least one iteration, or else we have found the item at position (i, j).

Thus the number of iterations of the outer loop is less than or equal to the sum of the total number of iterations of the inner loops plus one. Each inner loop will have at most m **total** iterations.

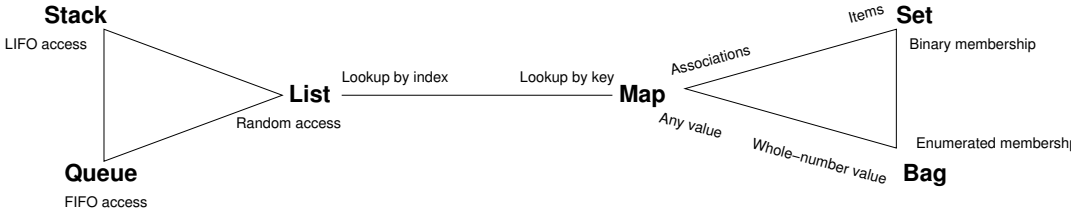
Hence worst case $\Theta(m)$ or $\Theta(\sqrt{n})$.

An abstract data type (ADT) is a data type whose representation is hidden from the client. Implementing an ADT as a Java class is not very different from implementing a function library as a set of static methods. The primary difference is that we associate data with the function implementations and we hide the representation of the data from the client. When using an ADT, we focus on the operations specified in the API and pay no attention to the data representation; when implementing an ADT, we focus on the data, then implement operations on that data.

[Sedgewick and Wayne, *Algorithms*, Pg 64; also cf pg 84]

The “canonical ADTs”:

- **List.** Linear collection with sequential and random access.
- **Stack.** Linear collection with LIFO access.
- **Queue.** Linear collection with FIFO access.
- **Set.** Unordered collection with binary membership.
- **Bag.** Unordered collection with enumerated membership.
- **Map.** Unordered collection of associations between keys and values.


Stack
LIFO access
Queue
FIFO access
List
Random access
Lookup by index
Lookup by key
Map
Items
Set
Binary membership
Associations
Any value
Whole−number value
Enumerated membershp
Bag

The four basic ways to implement an ADT:

- Use an array
- Use a linked structure
- Use an "advanced" data structure, varying and/or hybridizing linked structures and arrays
- Adapt an existing implementation of another ADT.

Coming up:

Due Tues, Jan 25:
Finish reading Section 2.1
Do Ex 1.11
Take ADT quiz

Due Fri, Jan 28:
Read Section 2.(2, 4, & 5)
Take data structures quiz

Also:
Do "basic data structures" practice problems (suggested by Mon, Jan 31)
Do **"Implementing ADTs"** *project (suggested by Wed, Feb 2)*