Algorithm analysis

... with focus on Big O analysis of recursive algorithms

Recap

$$\begin{array}{l} \text{Algoritme loop1}(n) \\ \quad i = 1 \\ \quad \text{while } i \leq n \\ \qquad i = 3 * i \end{array}$$

In [ ]:

```
int n = 82;
int i = 1;

while (i <= n) {
    System.out.println(i);
    i=3*i;
}
```

$$\begin{array}{l} \text{Algoritme loop2}(n) \\ \quad s = 1 \\ \quad \text{for } i = 1 \text{ to } n \\ \qquad \text{for } j = 1 \text{ to } n \\ \qquad\quad s = s + 1 \end{array}$$

In [ ]:

```
int n = 3;
int s = 1;

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        s = s + 1;
        System.out.println("i = " + i + " and j = " + j);
    }
}
```

$$
\begin{aligned}
&\text{Algoritme loop3}(n)\\
&\quad i = 1\\
&\quad \text{while } i \leq n\\
&\qquad i = 2 * i
\end{aligned}
$$

In [ ]:

```
int n = 32;
int i = 1;

while (i <= n) {
    System.out.println(i);
    i=2*i;
}
```

$$
\begin{aligned}
&\text{Algoritme loop4}(n)\\
&\quad i = 1\\
&\quad \text{while } i \leq n * n\\
&\qquad i = 3 * i
\end{aligned}
$$

In [ ]:

```
int n = 256;
int i = 1;

while (i <= n*n) {
    System.out.println(i);
    i=3*i;
}
```

$$
\begin{array}{l}
\text{Algoritme loop5}(n)\\
\quad i = 1\\
\quad \text{while } i \leq n\\
\qquad j = 0\\
\qquad \text{while } j \leq n\\
\qquad\quad j = j + 1\\
\qquad i = 2 * i
\end{array}
$$

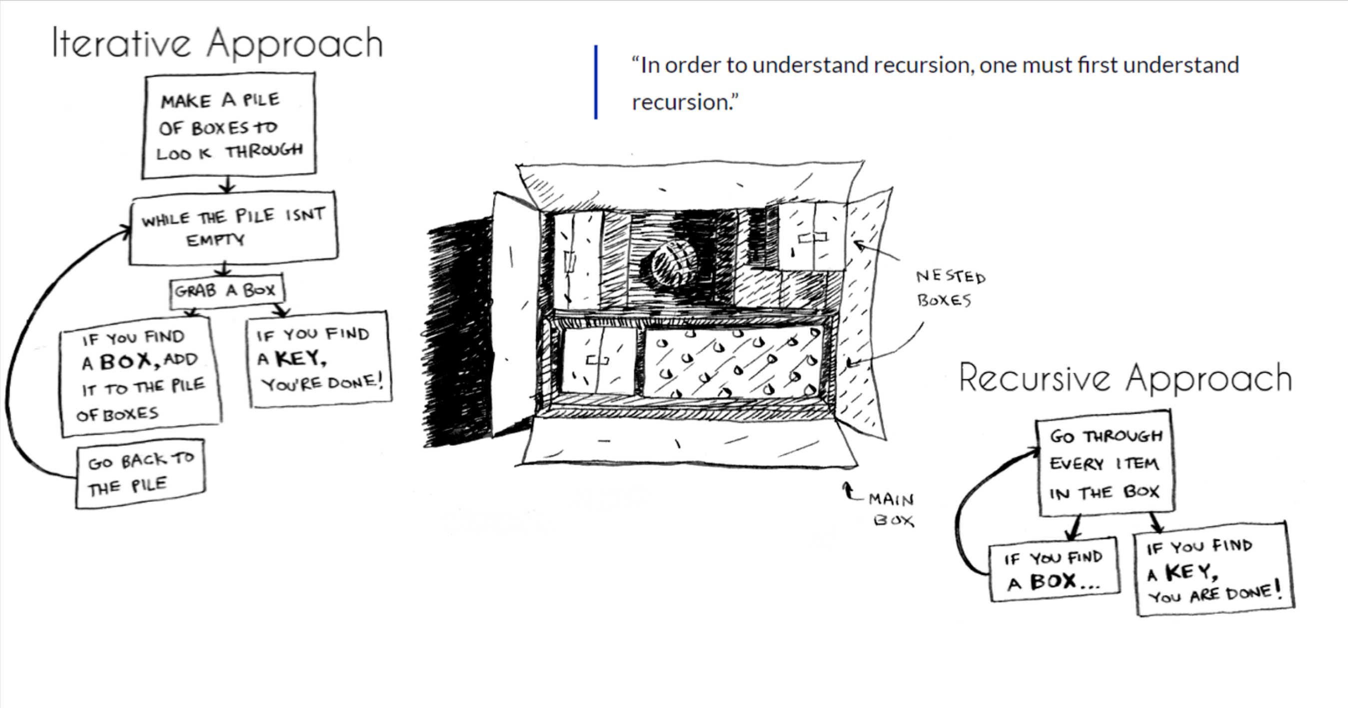
In [ ]:

```
int n = 8;
int i = 1;

while (i <= n) {
    int j = 0;
    while (j <= n) {
        System.out.println("i = " + i + " and j = " + j);
        j = j +1;
    }
    i=2*i;
}
```

$$
\begin{array}{l}
\text{Algoritme loop6}(n)\\
\quad i = 1\\
\quad \text{while } i \leq n\\
\qquad j = i\\
\qquad \text{while } j \leq n\\
\qquad\quad j = j + 1\\
\qquad i = 2 * i
\end{array}
$$

In [ ]:

```
int n = 32;
int i = 1;

while (i <= n) {
    int j = i;
    while (j <= n) {
        System.out.println("i = " + i + " and j = " + j);
        j = j +1;
    }
    i=2*i;
}
```



Source: https://www.freecodecamp.org/news/how-recursion-works-explained-with-flowcharts-and-a-video-de61f40cb7f9/

Recursive functions can be really elegant

A simple example:

```
def factorial_iterative(n):
    value = 1
    while n > 1:
        value = value*n
        n = n-1
    return value
```

```
def factorial_recursive(n):

    if n == 1:
        return 1

    return n*factorial_recursive(n-1)
```

Recursive functions can be really elegant

A better example:

```
def power_iterative(x, n):
    value = 1;
    for i in range(n):
        value = value*x
    return value
```

This correspond to “floor”, i.e. rounding down. In java, this is done automatically when two ints are divided.

```
def power_recursive(x, n):

    if n == 0:
        return 1

    value = power_recursive(x, int(n/2))

    if n % 2 == 0:
        return value*value
    else:
        return x*value*value
```

A Big $\mathcal{O}$ analysis will show us why the recursive algorithm is superior!

Finding the time complexity of recursive algorithms

Try to find the time complexity of this (very simple!) recursive algorithm:

```python
def factorial_recursive(n):

    if n == 1:
        return 1

    return n*factorial_recursive(n-1)
```

Finding the time complexity of recursive algorithms

… how about this one?

```
def recursive_function(n):

    if n == 1:
        print("This is the base case!")
        return 0

    for i in range(n):
        print("Wooo, recursion :-) ")

    return recursive_function(int(n/2))+recursive_function(int(n/2))
```

Finding the time complexity of recursive algorithms

It gets complicated pretty fast!

We will go through 2 ways of finding the time complexity:

1) Solving the recurrence relation
2) Using the "Master Theorem"

For both methods, you first need to find the **recurrence relation** for the algorithm.

Finding the recurrence relation for a recursive algorithm

This is the function, T, that describes the number of time units used by the algorithm as a function of the input n. Let's try on "factorial_recursive":

1. Figure out how many time units are used for the base case:

$$T(1) = 2$$

2. Figure out how many time units are used for cases which are *not* the base case:

$$T(n) = 3 + T(n-1)$$

```
def factorial_recursive(n):

    if n == 1:
        return 1

    return n*factorial_recursive(n-1)
```

1 time unit (base case and recursive case)

1 time unit (base case)

3 + T(n-1) time units (recursive case)
(1 return, 1 multiplication, 1 subtraction, 1 recursive call)

Solving the recurrence relation

Let's figure out how to write $T(n)$ as an explicit function of n:

$$T(n) = 3 + T(n-1) \quad \text{with } T(1) = 2$$

We can e.g. do it by writing down the first few terms until we recognize the pattern ("forwards substitution"):

$$T(1) = 2$$
$$T(2) = 3 + T(1) = 3 + 2$$
$$T(3) = 3 + T(2) = 3 + 3 + 2 = 3(3-1) + 2$$
$$T(4) = 3 + T(3) = 3 + 3 + 3 + 2 = 3(4-1) + 2$$
...
$$T(n) = 3 \cdot (n-1) + 2 = 3n - 1$$

From this we can easily determine the time complexity: $3n - 1 = \mathcal{O}(n)$

Let’s find the recurrence relation for this function:

```python
def recursive_function(n):

    if n == 1:
        print("This is the base case!")
        return 0

    for i in range(n):
        print("Wooo, recursion :-) ")

    return recursive_function(int(n/2))+recursive_function(int(n/2))
```

1 time unit (base case and recursive case)

1 time unit (base case)

1 time unit (base case)

2n+2 time units (recursive case)
(1 initialization, n+1 tests, n increments)

n time units (recursive case)

4 + 2*T(n/2) time units (recursive case)
(1 return, 1 addition, 2 divisions and 2 recursive calls)

$$T(1) = 3$$
$$T(n) = 3n + 7 + 2 \cdot T\left(\frac{n}{2}\right)$$

$\frac{n}{2}$ is actually rounded down (so we always get a whole number) but this is often not written explicitly.

... and since “Big O” isn’t an exact measure anyway, it’s no big deal ☺

Solving the recurrence relation

$$T(n) = 3n + 7 + 2 \cdot T\left(\frac{n}{2}\right) \quad \text{with} \quad T(1) = 3$$

Its hard to recognize the pattern from forwards substitution in this case (try ☺)! Let's try to do it backwards:

$T(n) = 3n + 7 + 2 \cdot T\left(\frac{n}{2}\right)$ We end up with $1 = 2^0$ of these

$T\left(\frac{n}{2}\right) = 3 \cdot \frac{n}{2} + 7 + 2 \cdot T\left(\frac{n}{4}\right)$ and $2 = 2^1$ of these

$T\left(\frac{n}{4}\right) = 3 \cdot \frac{n}{4} + 7 + 2 \cdot T\left(\frac{n}{8}\right)$ and $4 = 2^2$ of these

...

$T\left(\frac{n}{2^i}\right) = 3 \cdot \frac{n}{2^i} + 7 + 2 \cdot T\left(\frac{n}{2^{i+1}}\right)$ and 2^i of these

... continues until $\frac{n}{2^i} = 1 \Leftrightarrow i = \log_2(n)$, then:

$T(1) = 3$ and $2^{\log_2(n)} = n$ of these

Total number of time units: $T(n) = n \cdot 3 + \sum_{i=0}^{\log_2(n)-1} 2^i \cdot \left(3 \cdot \frac{n}{2^i} + 7\right)$

Evaluating the sum

$$T(n) = n \cdot 3 + \sum_{i=0}^{\log_2(n)-1} 2^i \cdot \left(3 \cdot \frac{n}{2^i} + 7\right)$$

$$= 3n + \sum_{i=0}^{\log_2(n)-1} (3n) \quad + \quad \sum_{i=0}^{\log_2(n)-1} \left(2^i \cdot 7\right)$$

$$= 3n \; + \; 3n \cdot \log_2(n) \quad + \quad 7 \cdot (n-1)$$

$$= 10n + 3n \cdot \log_2(n) - 7 \quad = \boldsymbol{O(n \cdot log(n))}$$

TABLE 2 Some Useful Summation Formulae.

Sum	*Closed Form*
$\sum_{k=0}^{n} ar^k \ (r \neq 0)$	$\frac{ar^{n+1} - a}{r-1}, r \neq 1$

The Master theorem

When the structure of the recursion is of the form $T\left(\frac{n}{2}\right)$ - as it was for the function we just considered – solving the recurrence relation is quite hard!

Luckily there's another method: We can use "The Master Theorem" ("Divide and conquer") in such cases:

$$T(N) = a \cdot T\left(\frac{N}{b}\right) + \underbrace{\Theta\left(N^k\right)}_{f(n)}$$

$$T(N) = \begin{cases} \Theta\left(N^{\log_b(a)}\right) & \text{if } a > b^k \\ \Theta\left(N^k \cdot \log N\right) & \text{if } a = b^k \\ \Theta\left(N^k\right) & \text{if } a < b^k \end{cases}$$

Let's try using the Master Theorem on recursive_function:

$T(n) = 3n + 7 + 2 \cdot T\left(\frac{n}{2}\right)$ with $T(1) = 3$

$$T(N) = a \cdot T\left(\frac{N}{b}\right) + \Theta\left(N^k\right)$$

$$T(N) = \begin{cases} \Theta\left(N^{\log_b(a)}\right) & \text{if } a > b^k \\ \Theta\left(N^k \cdot \log N\right) & \text{if } a = b^k \\ \Theta\left(N^k\right) & \text{if } a < b^k \end{cases}$$

Since $3n + 7 = \Theta(n^1)$, we can use this with $a = 2,\ b = 2$ and $k = 1$. Then $a = b^k$, so

$$T(n) = \Theta(n \cdot \log(n))$$

The Master theorem

- You cannot use the Master Theorem if
  - $T(n)$ is not monotone, ex: $T(n) = \sin n$
  - $f(n)$ is not a polynomial, ex: $T(n) = 2T\left(\frac{n}{2}\right) + 2^n$
  - b cannot be expressed as a constant, ex: $T(n) = T(\sqrt{n})$ or $T(n) = n^2 + T(n-1)$

- Does the base case remain a concern?

$$T(N) = a \cdot T\left(\frac{N}{b}\right) + \Theta\left(N^k\right)$$

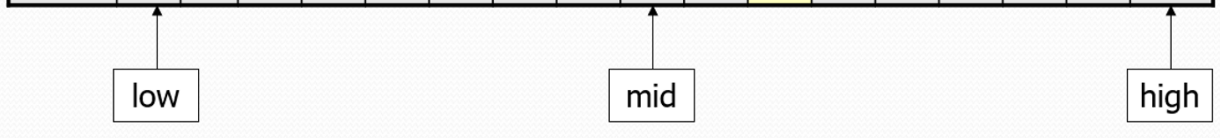
$$T(N) = \begin{cases} \Theta\left(N^{\log_b(a)}\right) & \text{if } a > b^k \\ \Theta\left(N^k \cdot \log N\right) & \text{if } a = b^k \\ \Theta\left(N^k\right) & \text{if } a < b^k \end{cases}$$

	$\Theta(\log n)$	$\Theta(\sqrt{n})$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^2 \log n)$	$\Theta(n^3)$
$T(n) = 4 \cdot T(n/2) + n^2$	A	B	C	D	E	F	G
$T(n) = T(n/5) + 5$	A	B	C	D	E	F	G
$T(n) = 3 \cdot T(n/4) + n^3$	A	B	C	D	E	F	G
$T(n) = 3 \cdot T(n/4) + n$	A	B	C	D	E	F	G
$T(n) = 2 \cdot T(n/4) + 1$	A	B	C	D	E	F	G

Application of recursion: Binary search

- **binary search**: Locates a target value in a *sorted* array/list by successively eliminating half of the array from consideration.
  - How many elements will it need to examine?
  - Can be implemented with a loop or recursively
  - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103



Consider the code on the right. Let

```
arr = [1,4,6,19,42]
low = 0
high = 4
x = 19
```

What does the function binary_search return?

Try to answer without coding! ☺

```
def binary_search(arr, low, high, x):

    if high >= low:

        mid = int((high + low)/2)

        if arr[mid] == x:
            return mid

        elif arr[mid] > x:
            return binary_search(arr, low, mid-1, x)

        else:
            return binary_search(arr, mid+1, high, x)

    else:
        return -1
```

In [ ]:

```
IFrame(src="https://content.screencast.com/users/brooks5283/folders/Snagit/media/c96a12fa-ef69-40ed-a9bf-8824cd6f0336/03.31.2022-21.26.GIF",
       width='100%',
       height='600px')
```

In [ ]:

```
IFrame(src="https://content.screencast.com/users/brooks5283/folders/Snagit/media/c96a12fa-ef69-40ed-a9bf-8824cd6f0336/03.31.2022-21.26.GIF",
       width='100%',
       height='800px')
```

In [ ]:

```
// JAVA iterative binary search

// find out if a key x exists in the sorted array A
// or not using binary search algorithm

public static int binarySearchIter(int[] a, int x)
{
    // search space is A[low..high]
    int low = 0, high = a.length - 1;

    // till search space consists of at-least one element
    while (low <= high)
    {
        // we find the mid value in the search space and
        // compares it with key value

        int mid = (low + high) / 2;

        // overflow can happen. Use:
        // int mid = low + (high - low) / 2;

        // key value is found
        if (x == a[mid]) {
            return mid;
        }

        // discard all elements in the right search space
        // including the mid element
        else if (x < a[mid]) {
            high = mid - 1;
        }

        // discard all elements in the left search space
        // including the mid element
        else {
            low = mid + 1;
        }
    }

    // x doesn't exist in the array
    return -1;
}
```

In [ ]:

```
int[] a = {1, 4, 56, 68, 69, 72, 222, 235, 674};
int target = 68;

int index = binarySearchIter(a, target);

if (index != -1) {
    System.out.println("Element found at index " + index);
}
else {
    System.out.println("Element not found in the array");
}
```

In [ ]:

```
// JAVA recursive binary search

// Find out if a key x exists in the sorted array
// A[low..high] or not using binary search algorithm
public static int binarySearchRecur(int[] a, int low, int high, int x)
{
    // Base condition (search space is exhausted)
    if (low > high) {
        return -1;
    }

    // we find the mid value in the search space and
    // compares it with key value

    int mid = (low + high) / 2;

    // overflow can happen. Use
    // int mid = low + (high - low) / 2;

    // Base condition (key value is found)
    if (x == a[mid]) {
        return mid;
    }

    // discard all elements in the right search space
    // including the mid element
    else if (x < a[mid]) {
        return binarySearchRecur(a, low,  mid - 1, x);
    }

    // discard all elements in the left search space
    // including the mid element
    else {
        return binarySearchRecur(a, mid + 1,  high, x);
    }
}
```

In [ ]:

```
int[] a = {1, 4, 56, 68, 69, 72, 222, 235, 674};
int target = 68;

int low = 0;
int high= a.length - 1;
int index = binarySearchRecur(a, low, high, target);


if (index != -1) {
    System.out.println("Element found at index " + index);
}
else {
    System.out.println("Element not found in the array");
}
```