Introduction to computer programming with python

Complexity Part 2

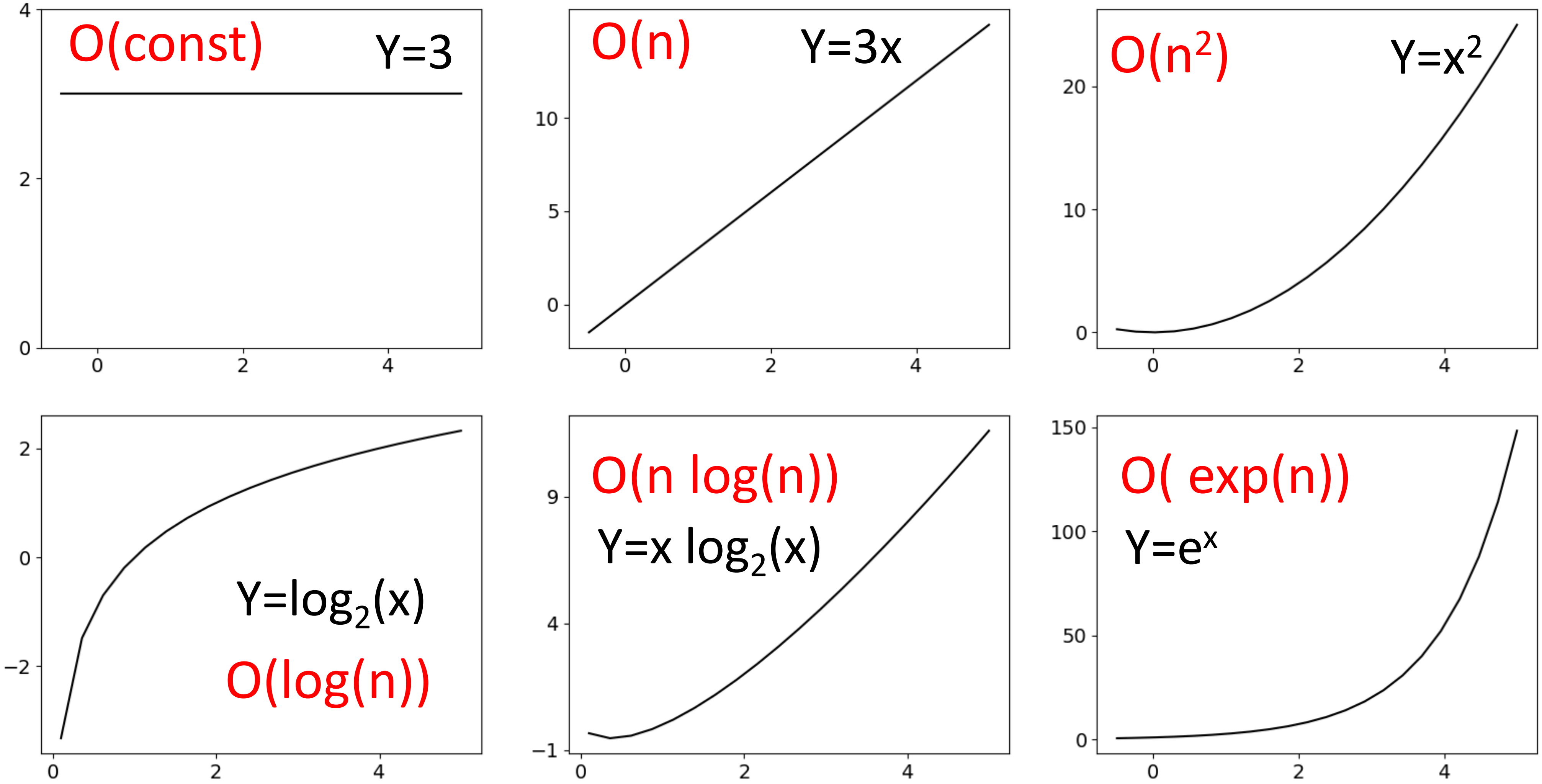
Spring Semester, 2023

Contents

Complexity:

- Recap Os of growth
- Logarithmic O of growth (binary search)
- Log-linear O of growth (merge sort)

Types of orders of growth



Constant Functions – O(c)

```python
def c_to_f(c):
    return (c*9/5) + 32 # 3 operations
```

C=10 or c=1000000 run @ constant time & same # of operations

Linear O of growth– single loops O(n)

```python
def search(L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
        if L[i] > e:
            return False
    return False
```

$\boxed{\text{len(L)}}$ ← O(n)

Orders of growth estimation

O(n+(<n)) => O(n)

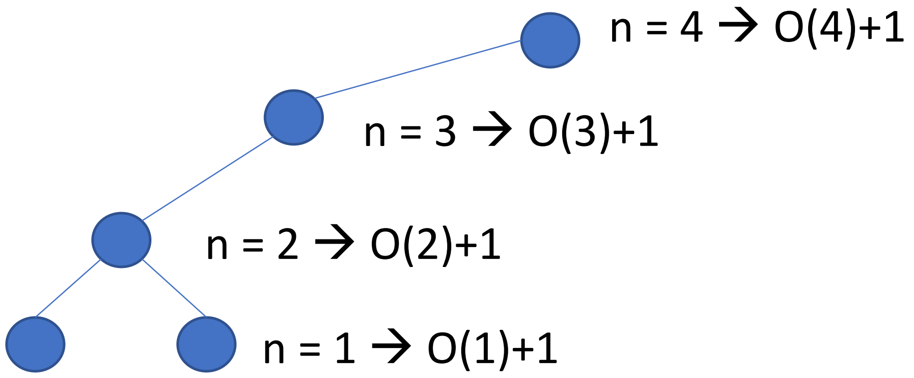
By dominant term

Linear O of growth – O(n) recursions w/ memoization

```
def fib_mem(n,d):
    if n in d:
        return d[n]
    else:
        d[n] = fib_mem(n-1,d)+fib_mem(n-2,d) # memoization
        return d[n]
print(fib_mem(6,{0:0,1:1}))
```

O(n)

O(n)+1→ O(n)



Polynomial O of growth - Nested loops
$O(n^c)$

```
for hours in range(24):  O(n)
    for minutes in range(60):  O(n)
        for seconds in range(60):  O(n)
            print(hours, ':', minutes, ':', seconds)
```

$O(n)*O(n)*O(n) = O(n^3)$

Exponential O of growth - C^n
n recursive calls (w/out memoization)

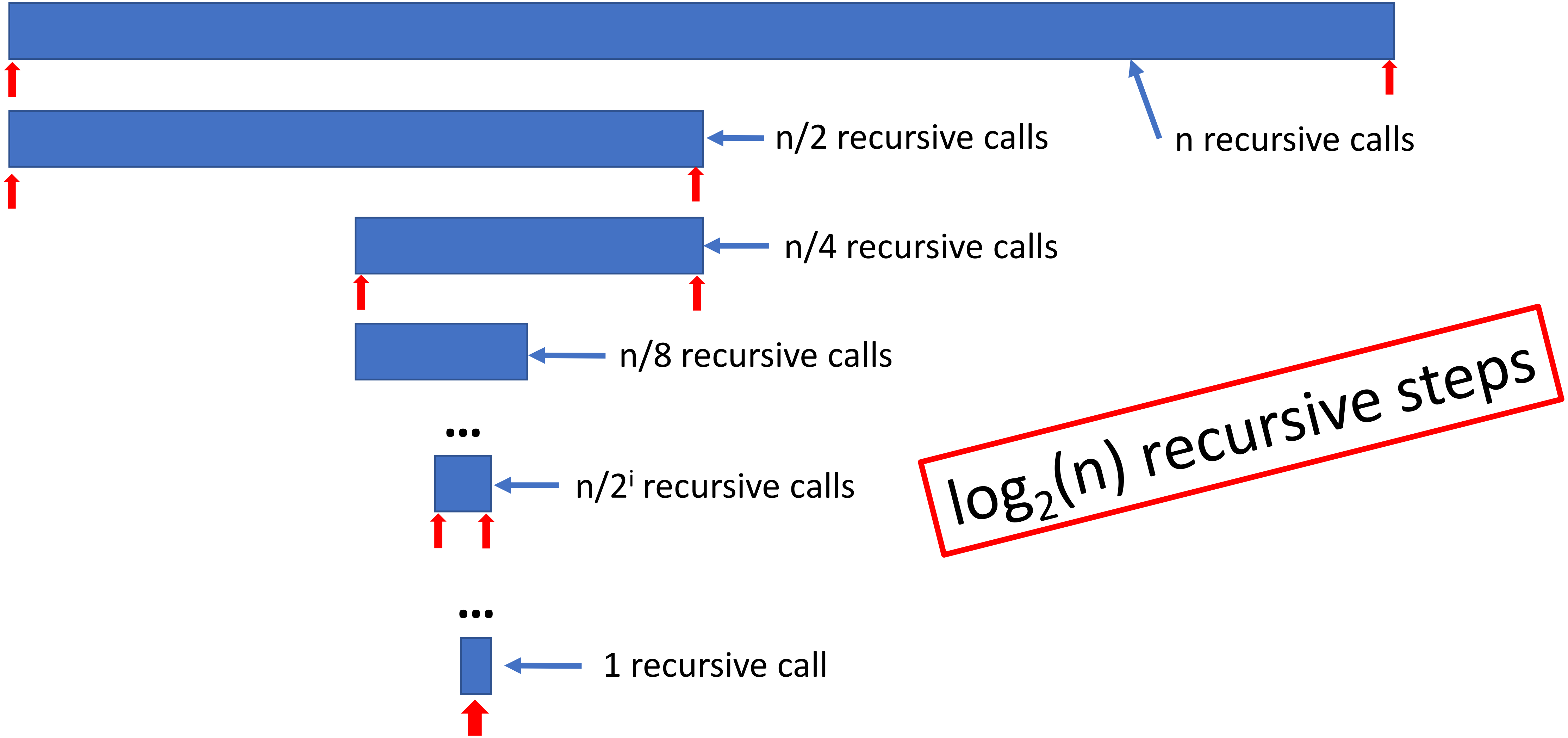
```python
def fib(n):
    if n <=1:
        return n
    else:
        return fib(n-1)+fib(n-2)
print(fib(6))
```

$\boxed{O(2^n)}$

Complexity growth

notation	complexity	n=10	n=100	n=1000	n=1000000
constant	O(1)	1	1	1	1
logarithmic	O(log(n))	1	2	3	6
linear	O(n)	10	100	1000	1000000
log-linear	O(n log n)	10	200	3000	6000000
n^c - Polynomial n^2 - quadratic	$O(n^2)$	100	10000	1000000	10^{12}
C^n - Exponential	$O(2^n)$	1024	12676506 00228229 40149670 3205376	1071508607186267320948425049060001810561404811705533607443750388370351051124936122493198378815695858127594672917553146825187145285692314043598457757469857480393456777482423098542107460506237114187795418215304647498358194126739876755916554394607706291457119647768654216766042983165262438683720566806937 6	TLDR (אמל"ק)

Bisection (binary) search
logarithmic O of growth O(log(n))



Binary search illustration

Is x=7 in array?

arr = [1,3,4,5,7,9,12,13,15,18,20]

low mid high

arr[mid]>x → [1,3,4,5,7,9]

low mid high

arr[mid]<x → [5,7,9]

low high

mid

arr[mid]==x → return True

Bisection (binary) search

Iterative version

```
def binary_search(arr, x):
    low = 0
    high = len(arr) - 1
    mid = 0

    while low <= high:

        mid = (high + low) // 2

        # If x is greater, ignore left half
        if arr[mid] < x:
            low = mid + 1

        # If x is smaller, ignore right half
        elif arr[mid] > x:
            high = mid - 1

        # means x is present at mid
        else:
            return mid

    # If we reach here, then the element was not present
    return -1
```

Returns the index of x in sorted array if exists otherwise returns -1

Bisection (binary) search
recursive version

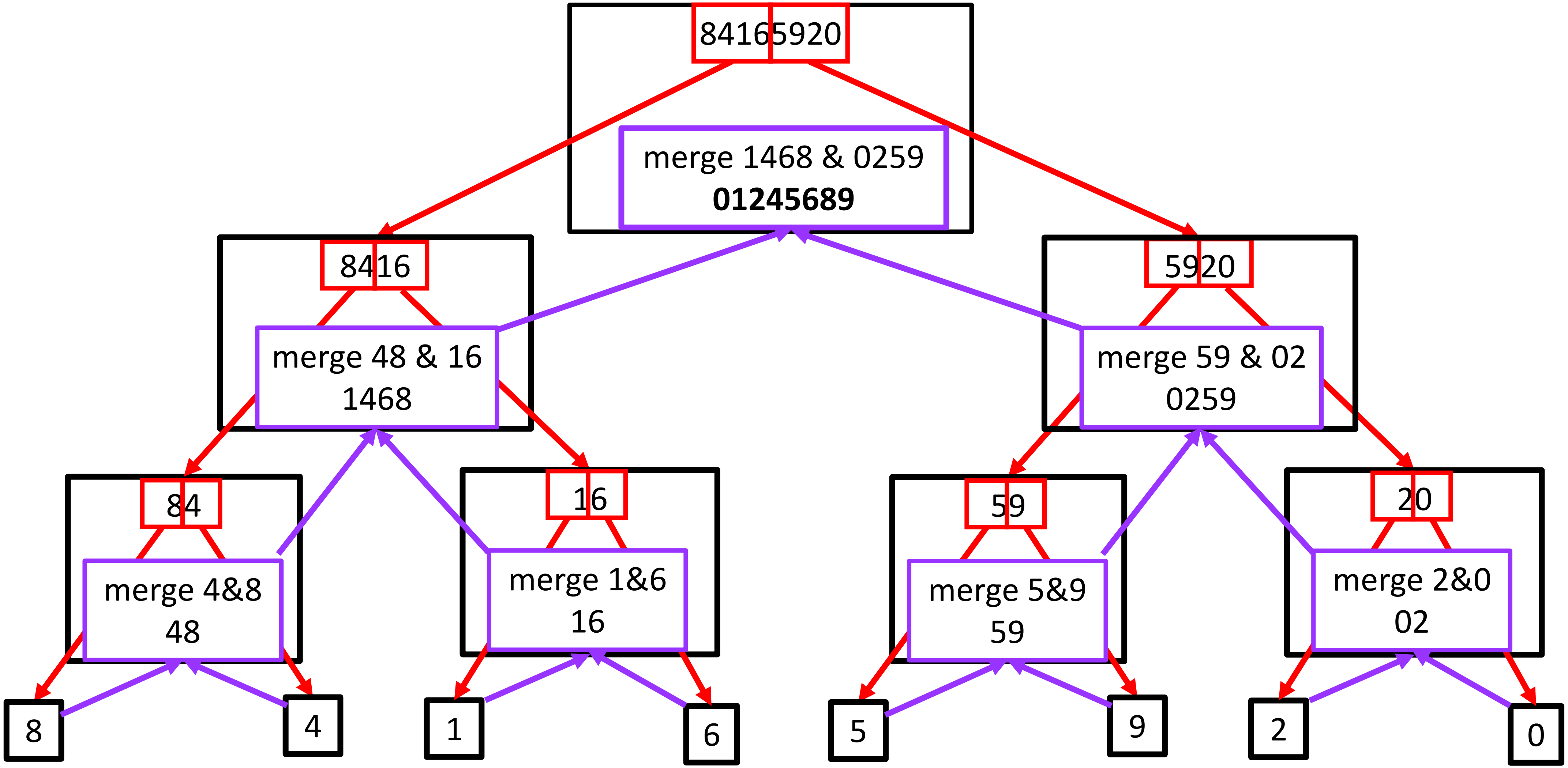
```
def binary_search(arr, low, high, x):
    # Check base case
    if high >= low:

        mid = (high + low) // 2

        # If element is present at the middle itself
        if arr[mid] == x:
            return mid

        # If element is smaller than mid, then it can only
        # be present in left sub-array
        elif arr[mid] > x:
            return binary_search(arr, low, mid - 1, x)

        # Else the element can only be present in right subarray
        else:
            return binary_search(arr, mid + 1, high, x)

    else:
        # Element is not present in the array
        return -1
```

Returns the index of x in sorted array if exists otherwise returns -1

Complexity growth

notation	complexity	n=10	n=100	n=1000	n=1000000
constant	O(1)	1	1	1	1
logarithmic	O(log(n))	1	2	3	6
linear	O(n)	10	100	1000	1000000
log-linear	O(n log n)	10	200	3000	6000000
n^c - Polynomial n^2 - quadratic	$O(n^2)$	100	10000	1000000	10^{12}
C^n - Exponential	$O(2^n)$	1024	12676506 00228229 40149670 3205376	10715086071862673209484250 49060001810561404811705533 60744375038837035105112493 61224931983788156958581275 94672917553146825187145285 69231404359845775746985748 03934567774824230985421074 60506237114187795418215304 64749835819412673987675591 65543946077062914571196477 68654216766042983165262438 6837205668069376	TLDR (אמל"ק)

Merge Sort: log-linear O of growth O(n log n) illustration



Merge Sort Code
log-linear O of growth O(n log n)

Outer recursion

```python
def merge_sort(L):
    print('merge sort: ' + str(L))
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```

O(log(n))

O(n)

O(n*log(n))

Merge Sort Code
log-linear O of growth O(n log n)

Merging sorted paired lists

```
def merge(left, right):
    result = []
    i,j = 0,0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
        result.append(right[j])
        j += 1
    print('merge: ' + str(left) + '&' + str(right) + ' to ' +str(result))
    return result
```

O(n)

O(n)

O(n)

3*O(n) → O(n)

Summary of Complexity growth

notation	complexity	Example
constant	O(1)	Input independent functions (c to f)
logarithmic	O(log(n))	Binary search in a sorted list
linear	O(n)	Single loops
log-linear	O(n log n)	Merge sort
n^c - Polynomial n^2 - quadratic	O(n^2)	Nested loops
exponential	O(2^n)	Recursions with two/more recursive calls (Fibonacci)