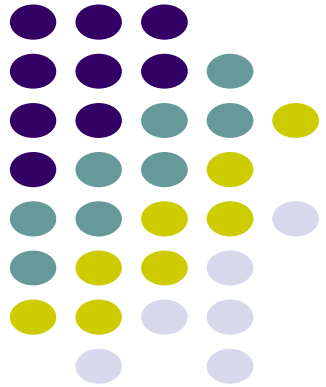
Analysis of algorithms (notion) and last slide this semester!



Intuition:
Calculating order of algorithms

- Based on the Math definition of big O (not seen in this course) and related theorems, when calculating the order of an algorithm we:
  - Count the **number of times a critical operation is executed** (it's important to be clear about the order of execution!!)
  - Considering the exact result of operations:
    - Disregard "constants"
    - Disregard "lower exponent terms"
  - Let's see some examples…

Calculate the order of an algorithm

```
x =0
y = 10
x = x+1
for i in range (n)
    x = x + y
    y = y + 1000
```

Critical operation: addition

How many exact additions are there in the code? 3

How many additions are executed 1 + 2n

Order of additions? O(n)

Calculate the order of an algorithm

```
count = 0
for i in range (n)
    count = count +10
for k in range (n)
    count = count + k
```

Critical operation: addition

How many exact additions are there in the code? 2

How many additions are executed? 2n

Order of additions? O(n)

Calculate the order of an algorithm

```
count = 0
for j in range (n) :
    for k in range (n) :
        count = count + 10
```

Critical operation: addition

How many exact additions are there in the code?

How many additions are executed? n^2

Order of additions? $O(n^2)$

Calculate the order of an algorithm

```
count = 0
for j in range (n)
    for k in range (n)
        count = count + 10
for k in range(n)
    count += 2
```

Critical operation: addition

How many exact additions are there in the code?

How many additions are executed? $n^2 + n$

Order of additions? $O(n^2+n) = O(n^2)$

Calculate the order of an algorithm:
when there are variations in the input data

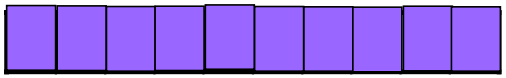
It is usually calculated:

best, average and worse cases

- Linear Search
- Binary search

Linear Search

- Check/visit item after item, from beginning to end



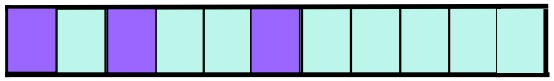
Best Case, Average Case, Worst Case: Linear search

- *Best case* - the first element we check is the element we are looking for $O(1)$
- *Worst case* - the last element we search is the one we are looking for (or the element is not there!) $O(n)$
- *Average case* $O(n/2) = O(n)$

Intuition of an O(logn) algorithm:

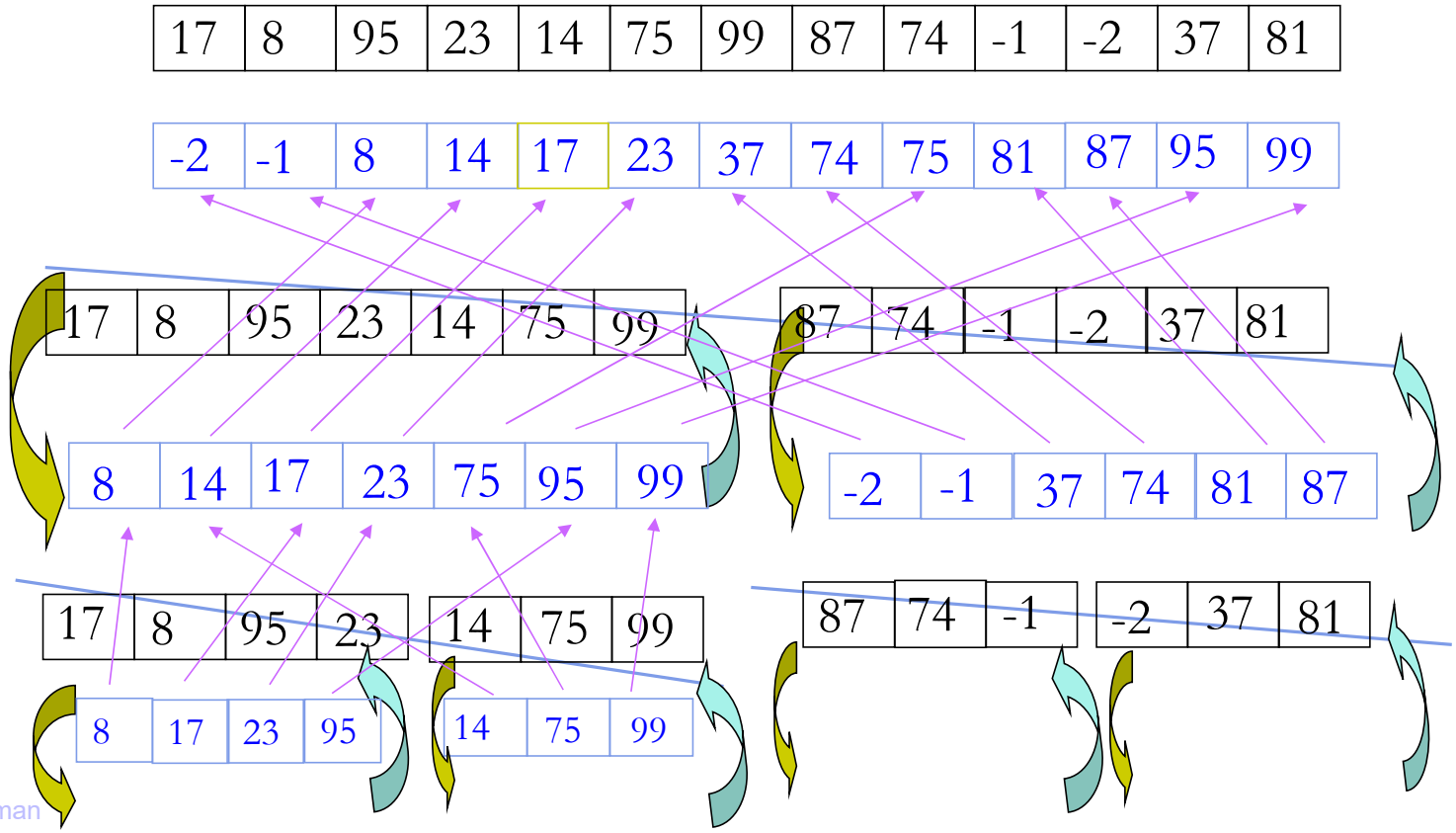
In each stage the algorithm processes one half of the previous stage

Binary Search



$O(\log n)$

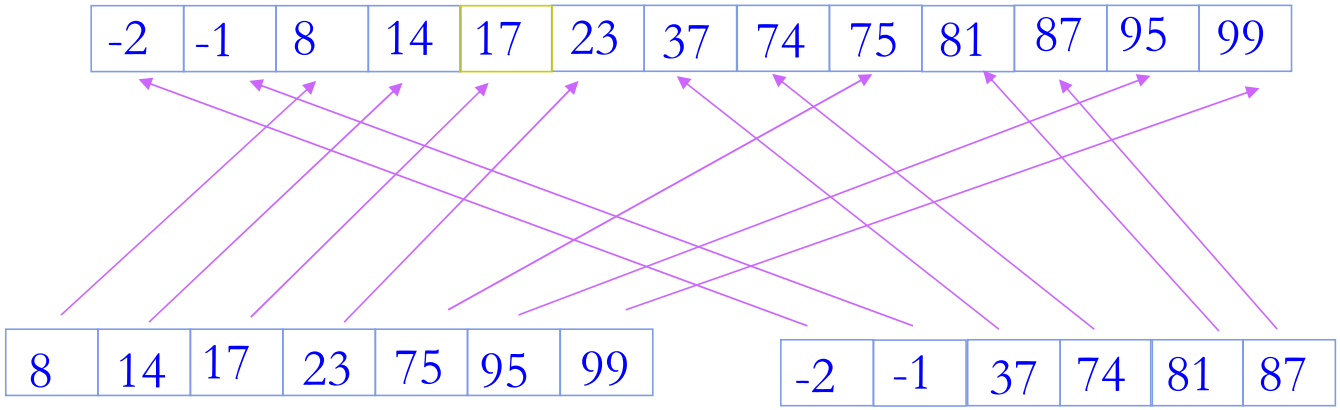
Merge Sort


17 8 95 23 14 75 99 87 74 -1 -2 37 81
-2 -1 8 14 17 23 37 74 75 81 87 95 99
17 8 95 23 14 75 99
87 74 -1 -2 37 81
8 14 17 23 75 95 99
-2 -1 37 74 81 87
17 8 95 23
14 75 99
87 74 -1
-2 37 81
8 17 23 95
14 75 99

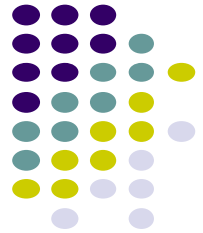
Mergesort: pseudocode

```
def mergeSort(alist):
    if (alist has 2 or more elements):
        sortedLeft = mergeSort(left half of alist)
        sortedRight = mergeSort(right half of alist)
        result = merge(sortedLeft , sortedRight)
    else:
        result = alist     # the list is already sorted,
                           # 1 element only
    return result
```

Example: Merging part only



Merging from both lists…



```
def merge2sorted(la,lb):
    j = 0
    k = 0
    res = []
    while j < len(la) and k < len(lb):
        if la[j] < lb[k]:
            res.append(la[j])
            j += 1
        else:
            res.append(lb[k])
            k += 1
        print("res so far. Both lists had elements",res)
```

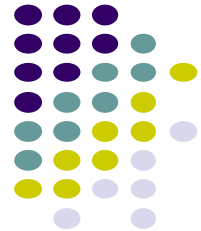
next need to consider elements remaining in one of the lists, if there are such

```
def merge2sorted(la,lb):
    j = 0
    k = 0
    res = []
    while j < len(la) and k < len(lb):
        if la[j] < lb[k]:
            res.append(la[j])
            j += 1
        else:
            res.append(lb[k])
            k += 1
        print("res so far. Both lists had elements",res)

    while j < len(la):
        res.append(la[j])
        j += 1
        print("res so far. la had elements",res)
    while k < len(lb):
        res.append(lb[k])
        k += 1
        print("res so far. lb had elements",res)
    return res

la = [8,14,17,23,75,95,99]
lb = [-2,-1,37,74,81,87]
print("\nBefore calling function....\n")
print("la",la)
print("lb",lb)
print()

lres = merge2sorted(la,lb)
print("\nFinal result",lres)
```



Complexity (order) of mergesort?

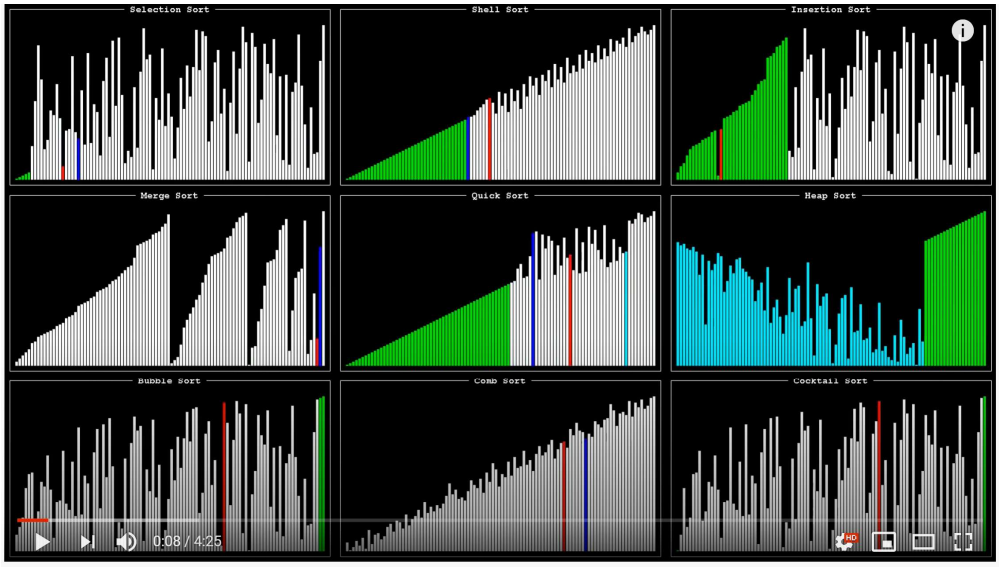
O(nlogn)

- *Intuitively, considering various operations:*
  - Split in two halves (just as in binary search) ⇒ **log n** times
  - But all branches are executed, and the merge operation for all the elements (considering all levels) ⇒ **n** operations

- **Note: Extra space is needed to create the merged lists** at each level. This may be problematic for large n.
  - (Other sorts do all the manipulations in place for ex with swapping with just a temporary variable extra for the swap)

Visualizing different sorts!

https://www.youtube.com/watch?v=ZZuD6iUe3Pc



Last slide this semester!!!

- Good luck with projects and exams!!
- Final Reflections TBP
- Review and… Sleep well!!
- See you maybe in the consultation class!
- See you in the exam!!! Arrive early.
  - Pencil, SFU ID, and eraser (maybe 1 page 1 side cheat sheet)
- Watch for announcements!