Write a program to ask the user for a value and output the first line in a file where the value occurs.

Searching and Sorting

Aslam safla
aslam@cs.uct.ac.za

Problems

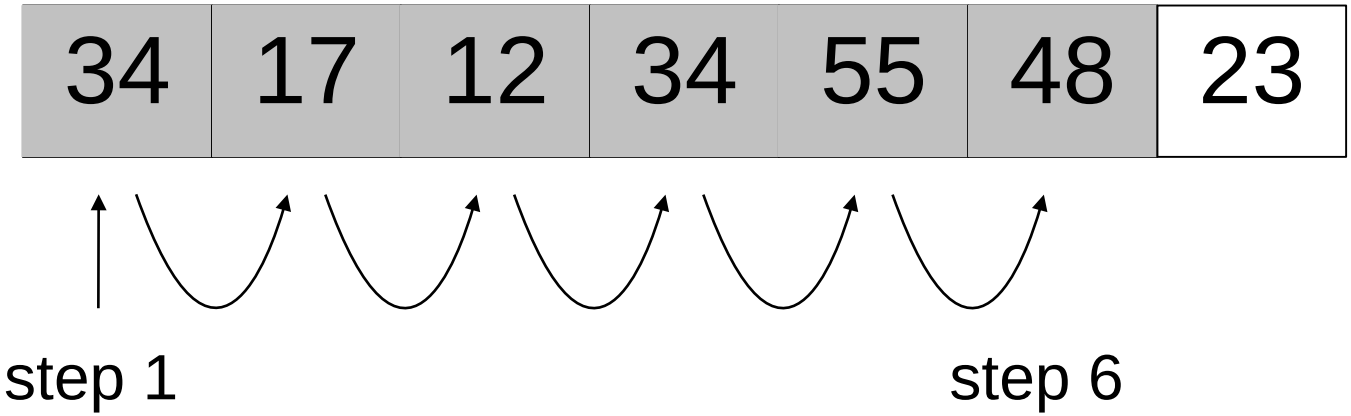
Write a program to search for a value in a given list.

- Example 1:
  - Search for an ID number when a person wishes to vote.
- Example 2:
  - Search for a person's record when a person inserts a card into an ATM.
- Example 3:
  - Search for a student record when you log into Vula.

Linear Search

Start at one end of a list and check each element until either it is found or the list ends.

- Given list below, find 48



Linear Search

```
def linear_search (arr, query ):
    """Search sequentially through values for query."""
    for i in range(len(arr)):
        if arr[i]==query: return i
    return -1
```

Binary Search

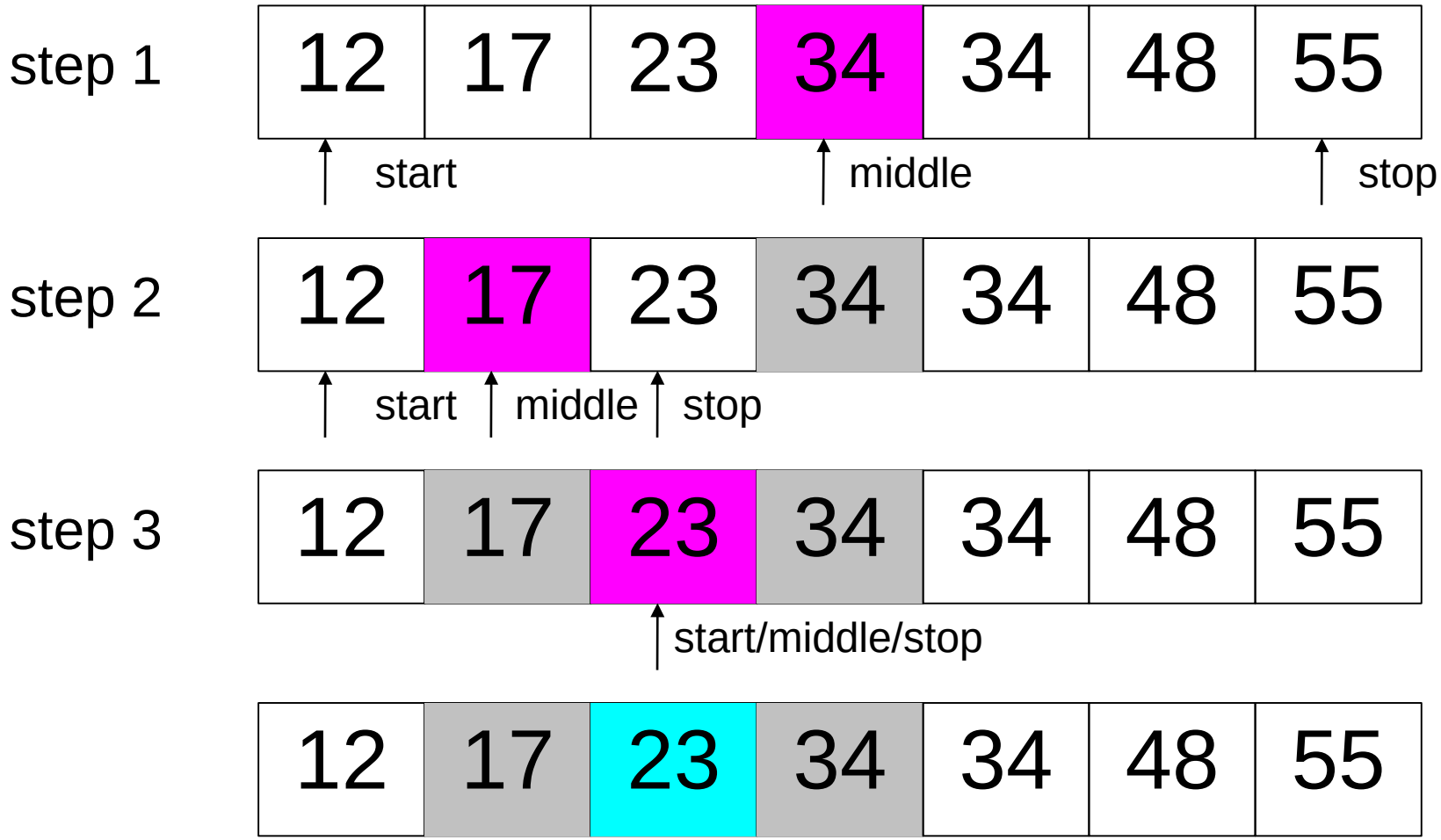
If list **is sorted**:

- determine which half contains search term
- **recursively** search that half
- stop when item is found or list is empty

- Given list below, find 23

12	17	23	34	34	48	55

Binary Search



Binary Search Algorithm

```
def binary_search2 ( values, query, start, stop ):
    """Binary search through values[start:stop+1] for query."""
    # check if list is empty
    if start > stop:
        return -1
    # find midpoint
    middle = (start + stop) // 2
    # check if value is at midpoint
    if values[middle] == query:
        return middle
    # check if value is in second half
    if values[middle] < query:
        # recursively search second half
        return binary_search2 (values, query, middle+1, stop)
    # assume value is in first half and recurse
    return binary_search2 (values, query, start, middle-1)
```

Binary Search

Binary search uses a **recursive method** to search an array to find a specified value

The array must be a **sorted array**:

`a[0]≤a[1]≤a[2]≤. . . ≤ a[finalIndex]`

If the value is found, its index is returned

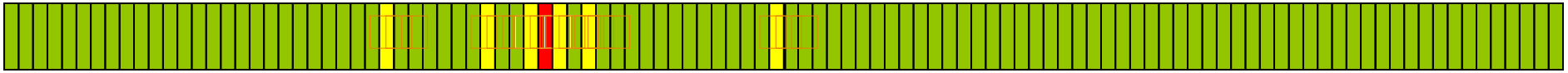
If the value is not found, -1 is returned

Note: Each execution of the recursive method reduces the search space by about a half

Binary search animation

Visualization:

searching for number *k* in a sorted list



Efficiency of Binary Search

The binary search algorithm is extremely fast compared to an algorithm that tries all array elements in order

- About half the array is eliminated from consideration right at the start
- Then a quarter of the array, then an eighth of the array, and so forth

Given an array with **1,000 elements**, on average the binary search will only need to compare about **10 array elements** to the key value, as compared to an **average of 500** for a serial search algorithm

Comparing algorithms: Running Times

We want a machine-independent estimate formula of how long a program will take to run as a function of input size

SO, we count the number of **operations** (+,-,x,comparison etc.)

and assume that everything else takes no time at all

(a simplifying assumption).

Comparing algorithms:Running Times and Big-O Notation

Estimates are usually expressed in **big-O notation**

upper bound estimate;

not an exact count,

but correct to a constant multiple.

Only include the **term with the highest exponent** and do **not** pay any attention to **constant multiples.**

Efficiency of Algorithms

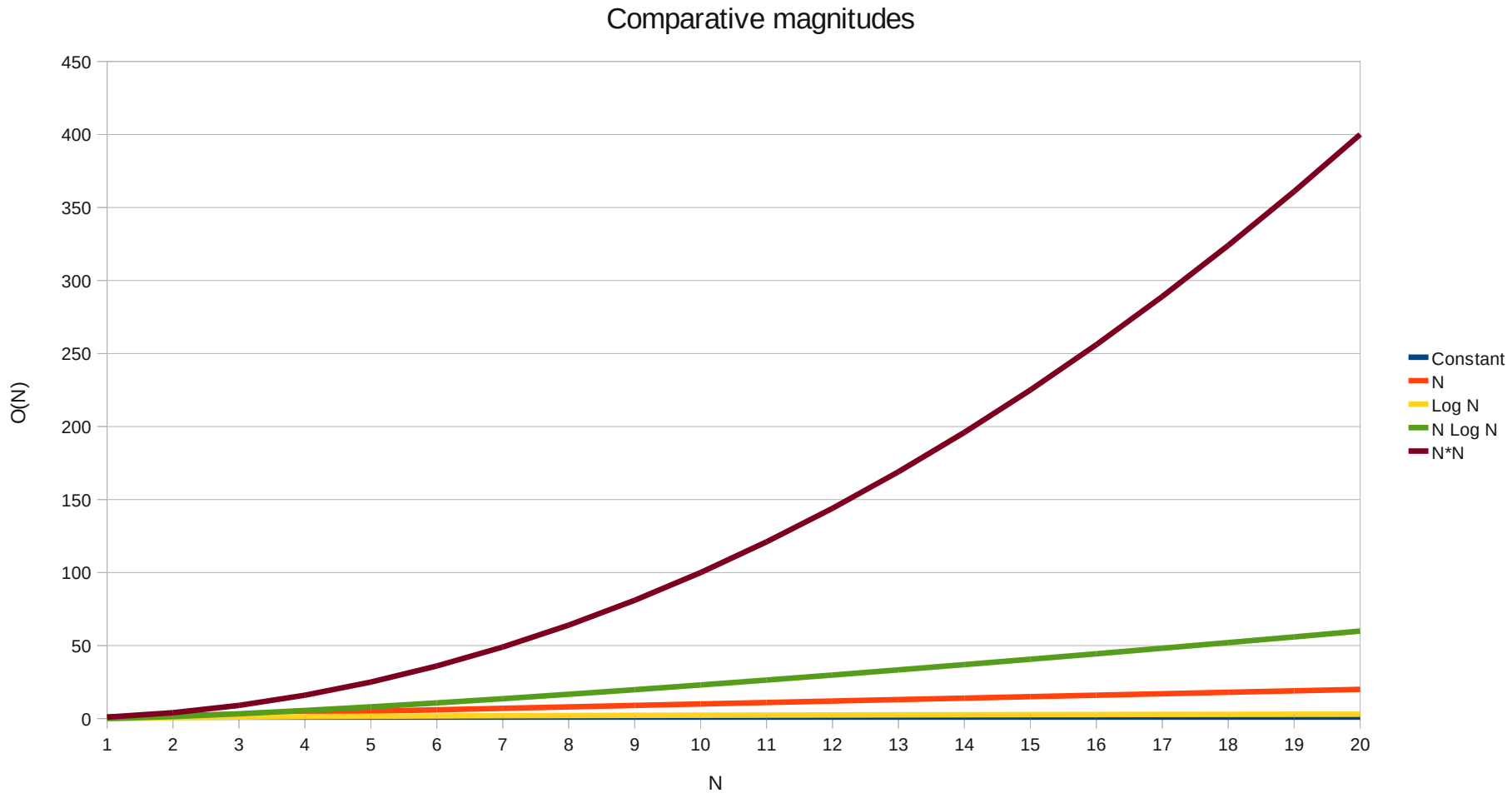
We measure efficiency of searching algorithms in terms of the upper bound of number of comparisons for n items (because comparisons take more CPU time than other operations), assuming large values of n.

In computer science, big O notation is used to classify algorithms according to how they respond to changes in input size

Typical algorithmic efficiencies:

- O(1) = constant time
- O(log n) = time proportional to log (n)
- O(n) = linear time, proportional to n
- O(n log n) = time proportional to n x log (n)
- $O(n^2)$ = time proportional to n x n

Comparative Magnitudes



Order of Growth

How much faster will algorithm run on a computer that is twice as fast?
How much longer does it take to solve problem of double input size?

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Comparing Algorithms

When comparing algorithms we talk about the:

- Best case
- Worst case
- Average case

with respect to the input size, n.

Comparing Search Algorithms

Linear Search:

- Best case: O(1)
- Average case: O(n)
- Worst case: O(n)

Binary Search:

- Best case: O(1)
- Average case: O(log n)
- Worst case: O(log n)

Comparing Search Algorithms

Linear Search:

- Best case: O(1)
- Average case: O(n)
- Worst case: O(n)

Binary Search:

- Best case: O(1)
- Average case: O(log n)
- Worst case: O(log n)

- Python Dictionary: O(1) in all cases (but no list order)!

Sorting

Problems

Write a program to sort a list of items.

- Example 1: Sort items to enable binary search.
- Example 2: Sort list of student names to print out final results for noticeboard.
- Example 3: Sort marks to find median mark.
- Example 4: Sort names to find unique names.

Selection Sort

Select smallest item in list and place at beginning (swap with first item), then repeat for rest of list, until list is completely processed.

- Sort the following list:

34	17	12	34	55	48	23

Selection Sort

12	17	34	34	55	48	23
12	17	34	34	55	48	23
12	17	23	34	55	48	34
12	17	23	34	55	48	34
12	17	23	34	34	48	55
12	17	23	34	34	48	55
12	17	23	34	34	48	55

Selection Sort Algorithm

```
def selection_sort ( values ):
    """Sort values using selection sort algorithm."""
    # iterate over outer positions in list
    for outer in range (len (values)):
        # assume first value is minimum
        minimum = outer
        # compare minimum to rest of list and update
        for inner in range (outer+1, len (values)):
            if values[inner] < values[minimum]:
                minimum = inner
        # swap minimum with outer position
        values[minimum],values[outer]= values[outer],values[minimum]
```


Dancing algorithms…

https://keet.wordpress.com/2015/02/24/dancing-algorithms/

Selection sort

Basic operation is comparison (again).

- Best case?
- Worst case?
- Average case?

Selection sort

Basic operation is comparison (again).

Best case?

Worst case?

Average case?

best case = worst case = average case

Number of comparisions depends only on input data size (n), *not* type of data

Selection sort

Basic operation is comparison (again).

Best case?

Worst case?

Average case?

best case = worst case = average case

Selecting the lowest element requires scanning all n elements ($n - 1$ comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining $n - 1$ elements and so on, for $(n - 1) + (n - 2) + ... + 2 + 1 = n(n - 1) / 2$

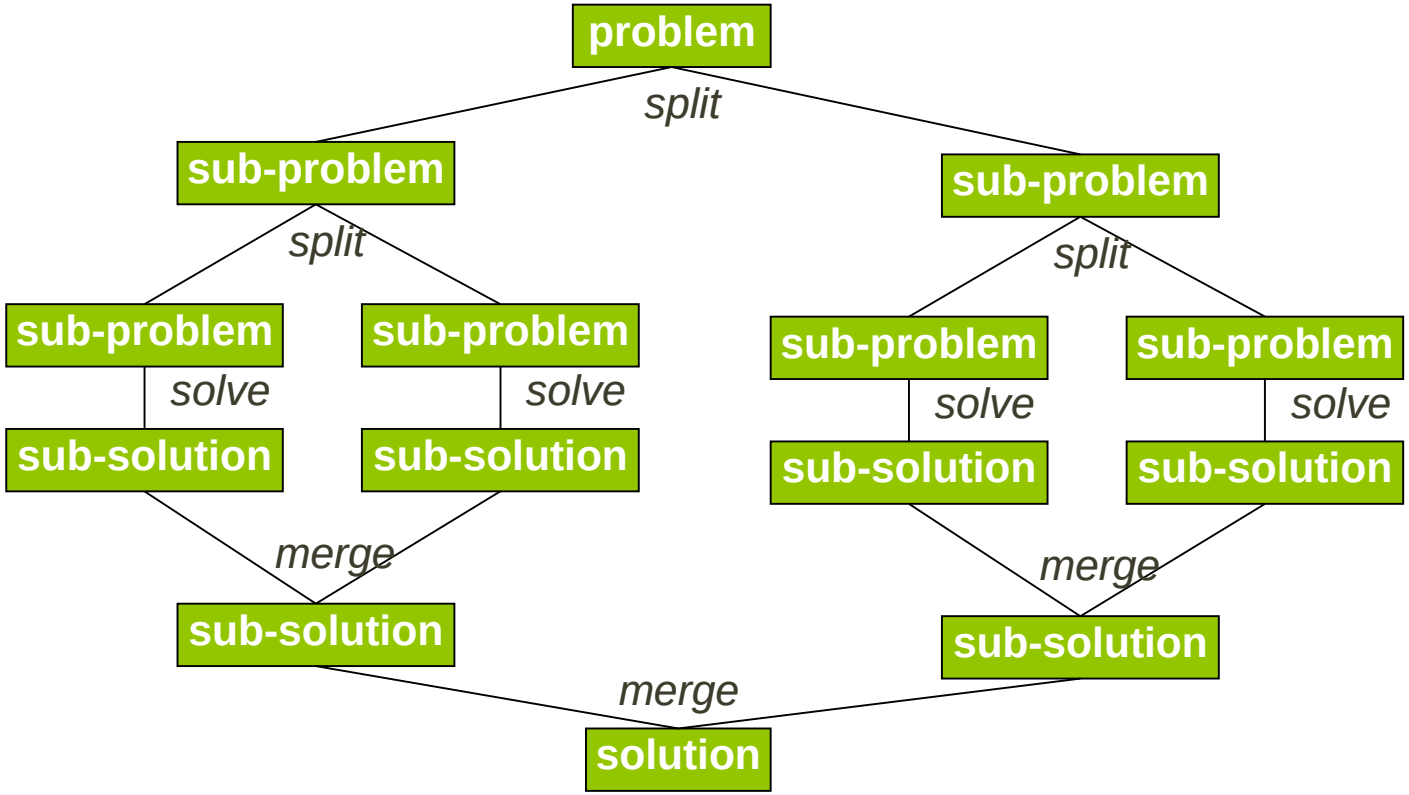
i.e $O(n^2)$ comparisons

Simplified using arithmetic progression

A Sorting Pattern

The **most efficient** sorting algorithms all seem to follow a **divide-and-conquer** strategy

- Naturally recursive



Divide-and-conquer algorithms

Characterized by dividing problems into sub-problems that are of the same form as the larger problem.

Problems are solved independently, and then merged into a solution for the whole problem.

e.g. Mergesort and quicksort

Mergesort

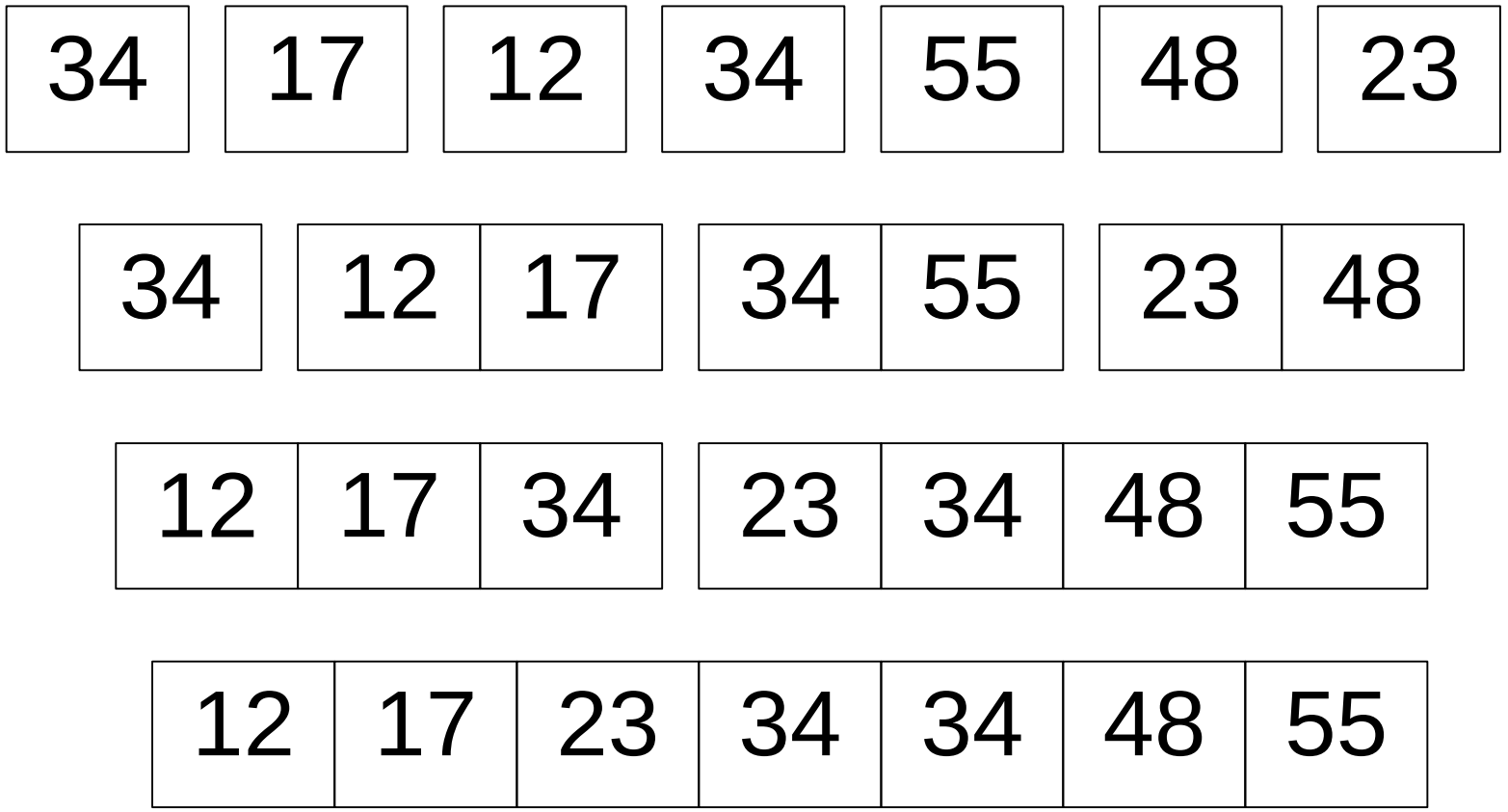
Partition list arbitrarily into 2, then sort each (recursively) and merge the 2 sorted lists into a final solution.

- Sort the following list:

34	17	12	34	55	48	23

Merge sorting was one of the first methods proposed for computer sorting, invented by John von Neumann in 1945.

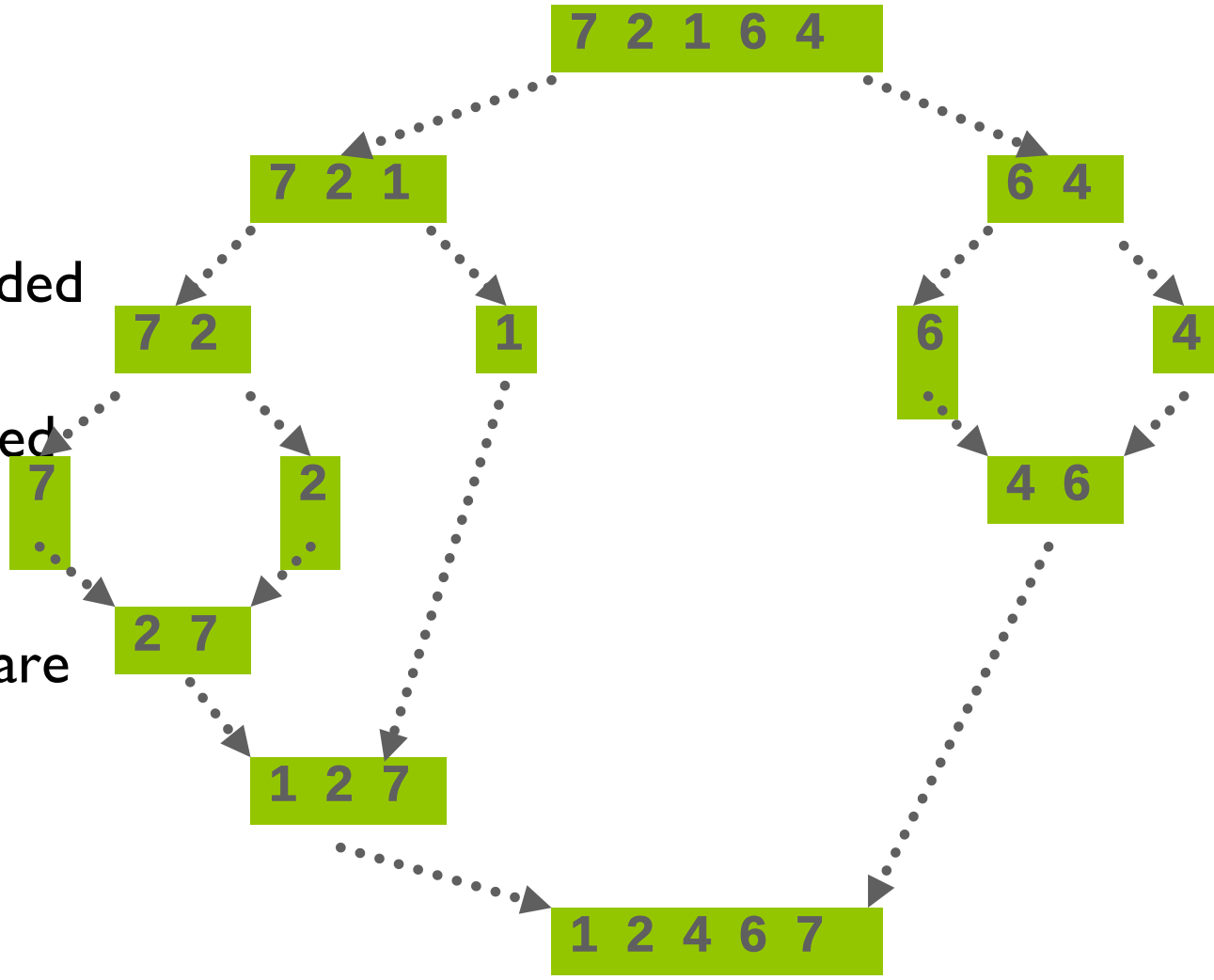
Mergesort


34 17 12 34 55 48 23
34 12 17 34 55 23 48
12 17 34 23 34 48 55
12 17 23 34 34 48 55

Mergesort

A classical sequential divide-and-conquer sorting algorithm.

- Unsorted list first divided into half.
- Each half is again divided into two.
- Continued until individual numbers are obtained.
- pairs of numbers combined (merged) until the one fully sorted list is obtained.



Mergesort Algorithm 1

```
def merge ( list1, list2 ):
    """Merge 2 sorted lists."""
    new_list = []
    while len(list1)>0 and len(list2)>0:
        if list1[0] < list2[0]:
            new_list.append (list1[0])
            del list1[0]
        else:
            new_list.append (list2[0])
            del list2[0]
    return new_list + list1 + list2
```


MergesortAlgorithm 2

```
def merge_sort ( values ):
    """Sort values using merge sort algorithm."""
    if len(values)>1:
        sorted1 = merge_sort (values[:len(values)//2])
        sorted2 = merge_sort (values[len(values)//2:])
        return merge (sorted1, sorted2)
    else: return values
```

Mergesort analysis

Basic operation is comparison (again).

- Best case?
- Worst case?
- Average case?

Mergesort analysis

Basic operation is comparison (again).

- Best case?
- Worst case?
- Average case?

The divide step takes constant time, regardless of the subarray size. After all, the divide step just computes the midpoint qq of the indices pp and rr- Θ(1).

The merging step requires recurrence relations for analysis, but basically is n * (number of levels in the “tree”) = n*2(log(n) = O(n log n)

The combine step merges a total of n elements, taking O(n) time.

Mergesort makes a copy of the list – it is not **in place.** This matters if space is an issue.

Order of Growth

How much faster will algorithm run on a computer that is twice as fast?
How much longer does it take to solve problem of double input size?

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Efficiency class is important

My laptop sorts an array of 100 million items in 30 seconds using an $O(n \log_2 n)$ algorithm

- How long would it take with an n^2 algorithm?

How long to sort 100m items?

$n * \log_2 n = 2{,}657{,}542{,}476$

$n^2 = 10{,}000{,}000{,}000{,}000 = 10^{16}$

$10^{16} / 2{,}657{,}542{,}476 = 3{,}762{,}875$

3,762,875 x 30 seconds = 112,886,248s = 3.5 years

So selection sort is utterly impractical for large data sets!

Quicksort

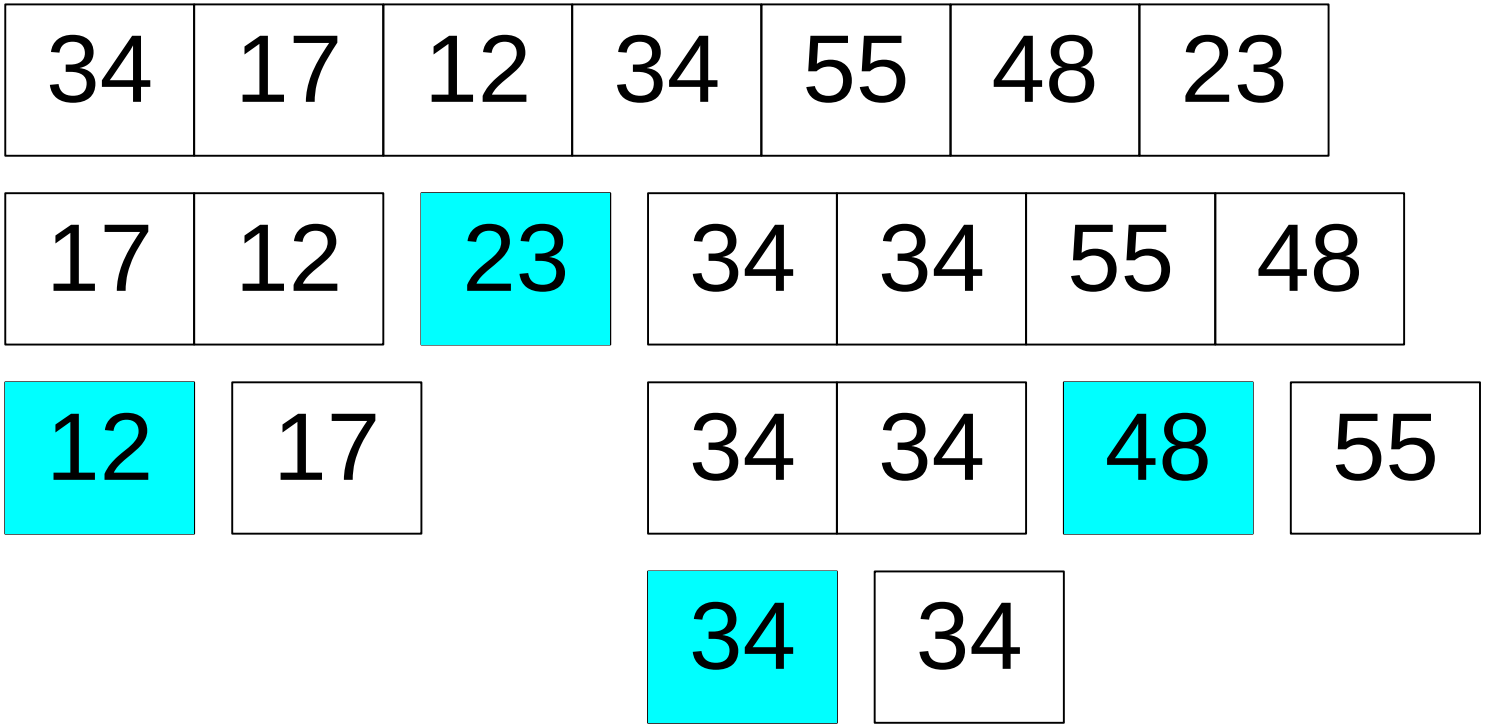
Partition list based on pivot value in correct middle position, then sort each partition (recursively) for a final solution.

- Sort the following list:

34	17	12	34	55	48	23

Sir Charles Anthony Richard Hoare
British computer scientist best known for the development (in 1960, at **age 26**) of Quicksort, one of the world's most widely used sorting algorithms.

Quicksort



Quicksort Algorithm 1

```
def swap ( values, source, dest ):
   """Exchange source and dest values in list."""
   values[source], values[dest] = values[dest], values[source]


def partition ( values, start, stop ):
   """Partition list in-place based on last value as pivot."""
   pivot = values[stop]
   midpoint = start
   for position in range (start, stop):
      if values[position] <= pivot:
         swap (values, position, midpoint)
         midpoint += 1
   swap (values, midpoint, stop)
   return midpoint
```


Quicksort Algorithm 2

```
def quick_sort2 ( values, start, stop ):
    """Sort values from start to stop using
    quicksort algorithm."""
    if stop > start:
        pivot = partition (values, start, stop)
        quick_sort2 (values, start, pivot-1)
        quick_sort2 (values, pivot+1, stop)

def quick_sort ( values ):
    """Sort values using quicksort algorithm."""
    quick_sort2 (values, 0, len(values)-1)
    return values
```


Efficiency of Quicksort

Depends on input:

- Worst case: *when does this occur?*
- Best case:
- Average case:

Efficiency of Quicksort

Depends on input:

- Worst case: O(n^2) *when does this occur?*
- Best case O(*n* log *n*)
- Average case: O(*n* log *n*)

Improvements to Quicksort (better pivot selection, switching to simpler sort on small subfiles, recursion elimination) can cut running time by 20-25%

Sorting and searching

Sorts

- Selection $O(n^2)$
- Merge O(n logn)
- Quick O (n logn)

Searching

- Linear search O(n)
- Binary search O(logn)

Stability and Comparisons

Comparisons can apply to any type of data:

- integers
  - not integers stored as strings ('2' > '10')!
- strings
- lists/dictionaries/etc
  - must define how to compare 2 items
  - sort ([['Ntwa',15'],['Chao',12],['Grace',20],['Ntwa',16]])

A **stable sort** is when the relative positions of items with the same key does not change.

- [['Chao',12],['Grace',20],['Ntwa',15'],['Ntwa',16]] is stable
- [['Chao',12],['Grace',20],['Ntwa',16'],['Ntwa',15]] is not stable

Summary

Sorting and searching are classic computation problems

We only look at **comparison sorts** in this section

There are many different algorithms, with different advantages

- Eg. Sorting:
  - Bubble Sort
  - Heap Sort
  - **Selection Sort**
  - Insertion Sort
  - **Merge Sort**
  - **Quick Sort**
  - Shell Sort

Comparison of sorting algorithms

There are many different algorithms, with different advantages

- Some are **fast**
  - Lower computational complexity
  - *Mergesort, Quicksort*
- some sorting algorithms are **in place**
  - Use no, or very little more, extra memory
  - *Selection sort, Quicksort*
- Some are **stable**
  - maintain the relative order of records with equal keys (i.e., values)
  - e.g. *Mergesort, Insertion sort, Bubble sort, Radix sort*
- Some are **adaptive**
  - Do less work if array is already sorted
  - e.g *Insertion sort*

Basic Efficiency Classes

1	constant	Outside best-case, few examples
log *n*	logarithmic	Algorithms that decrease by a constant
n	linear	Algorithms that scan an *n*-sized list
n log *n*	*n* log *n*	Algorithms that divide and conquer, e.g., quicksort
n^2	quadratic	Typically two embedded loops
n^3	cubic	Typically three embedded loops
2^n	exponential	Algorithms that generate all subsets of an *n*-element list
n!	factorial	Algorithms that generate all permutations of an *n*-element list