



UGANDA CHRISTIAN  
UNIVERSITY

A Center of Excellence in the Heart of Africa

# CSC2107: Design and Analysis of Algorithms

Chapter 3: Search and Sort Algorithms

# Sequences of values



- Two basic ways of storing a sequence of values
  - Arrays
  - Lists
- What's the difference?



# Arrays



- Single block of memory
  - Typically, fixed size
- Indexing is fast
  - Access  $A[i]$  in constant time for any  $i$
- Inserting an element between  $A[i]$  and  $A[i+1]$  is expensive
- Contraction is expensive



# Lists



- Values scattered in memory
  - Each element points to the next—“linked” list
  - Flexible size
- Follow  $i$  links to access  $A[i]$ 
  - Cost proportional to  $i \leq n$
- Inserting or deleting an element is easy
  - “Plumbing”



# Operations



- Exchange  $A[i]$  and  $A[j]$ 
  - Constant time in array  $O(1)$ , linear time in lists  $O(n)$
- Delete  $A[i]$  or Inserting a value after  $A[i]$ 
  - Constant time in lists (if we are already at  $A[i]$ )
  - Linear time in array
- Algorithms on one data structure may not transfer to another
  - Example: Binary search



# Search problem



- Is a value  $K$  present in a collection  $A$ ?
- Does the structure of  $A$  matter?
  - Array vs list
- Does the organization of the information matter?
  - Values sorted/unsorted



# The unsorted case

```
function search(A, K)
```

```
    i = 0;
```

```
    while i < n and A[i] != K do i = i+1;
```

```
    if i < n
```

```
        return i;
```

```
    else
```

```
        return -1;
```



# Python Code

```
# function search(A,K)
def search(A,K):
    # i = 0;
    i = 0
    # while i < n and A[i] != K do i = i+1;
    while i < len(A) and A[i] != K:
        i = i+1

    # if i < n return i;
    if i < len(A):
        return i
    # else return -1;
    else:
        return -1
```

```
lst = [74,32, 89, 55, 21, 64]
k = 21
search(lst,k) # output: 4
```





# Worst-case



- Need to scan the entire sequence A
  - $O(n)$  time for input sequence A of size n
- Does not matter if A is an array or a list



# Search a sorted sequence

- What if A is sorted?
  - Compare K with midpoint of A
  - If midpoint is K, the value is found
  - If  $K < \text{midpoint}$ , search left half of A
  - If  $K > \text{midpoint}$ , search right half of A
- Binary search



# Binary search ...

```
bsearch(K,A,l,r) // A sorted, search for K in A[l..r-1]
if (r - l == 0)
    return(false)

mid = (l + r) div 2 // integer division

if (K == A[mid])

    return (true)
if (K < A[mid])
    return (bsearch(K,A,l,mid))
else
    return (bsearch(K,A,mid+1,r))
```



# Binary Search ...



- How long does this take?
  - Each step halves the interval to search
  - For an interval of size 0, the answer is immediate
- $T(n)$ : time to search in an array of size  $n$ 
  - $T(0) = 1$
  - $T(n) = 1 + T(n/2)$



# Binary Search ...

- $T(n)$ : time to search in a list of size  $n$ 
  - $T(0) = 1$
  - $T(n) = 1 + T(n/2)$
- Unwind the recurrence
  - $T(n) = 1 + T(n/2) = 1 + 1 + T(n/2^2) = \dots$   
 $= 1 + 1 + \dots + 1 + T(n/2^k)$   
 $= 1 + 1 + \dots + 1 + T(n/2^{\log n}) = O(\log n)$



# Binary Search ...



- Works only for arrays
  - Need to be looked up  $A[i]$  in constant time
- By seeing only a small fraction of the sequence, we can conclude that an element is not present!



# Sorting

- Searching for a value
  - ✓ Unsorted array — linear scan,  $O(n)$
  - ✓ Sorted array — binary search,  $O(\log n)$
- Other advantages of sorting
  - ✓ Finding **median** value: midpoint of sorted list
  - ✓ Checking for duplicates
  - ✓ Building a frequency table of values



# How to sort?



- You are a Teaching Assistant for a course
- The instructor gives you a stack of exam answer papers with marks, ordered randomly
- Your task is to arrange them in ascending/descending order





# Strategy 1



- Scan the entire stack and find the paper with minimum marks
- Move this paper to a new stack
- Repeat with remaining papers
- ✓ Each time, add next minimum mark paper on top of new stack
- Eventually, new stack is sorted in descending order



# Strategy 1 ...



74

32

89

55

21

64



# Strategy 1 ...



74

32

89

55

~~21~~

64

21



# Strategy 1 ...



74

~~32~~

89

55

~~21~~

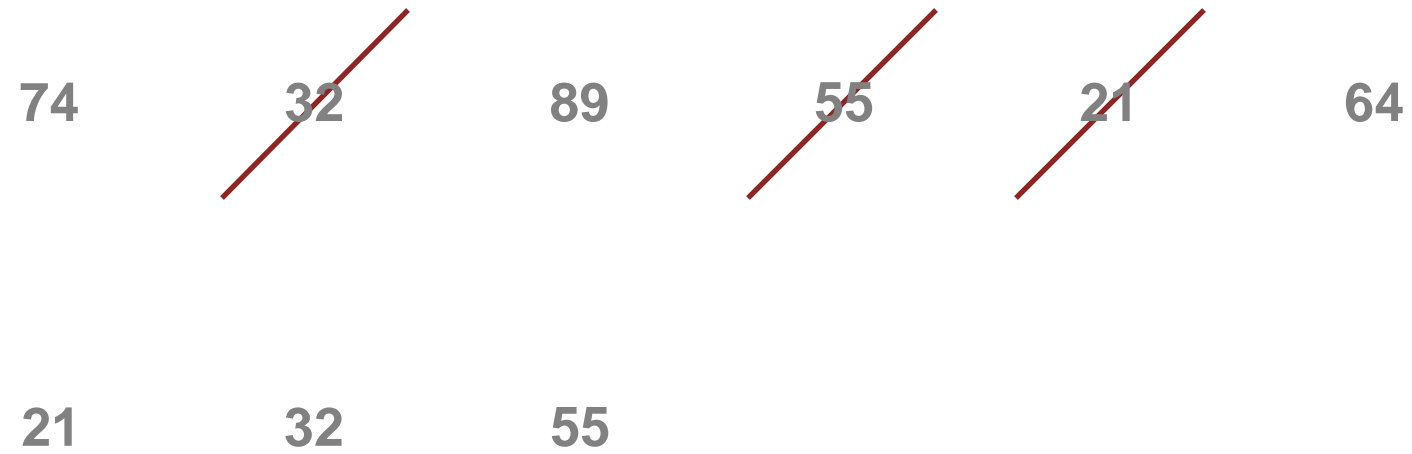
64

21

32



# Strategy 1 ...



# Strategy 1 ...

74

~~32~~

89

~~55~~

~~21~~

~~64~~

21

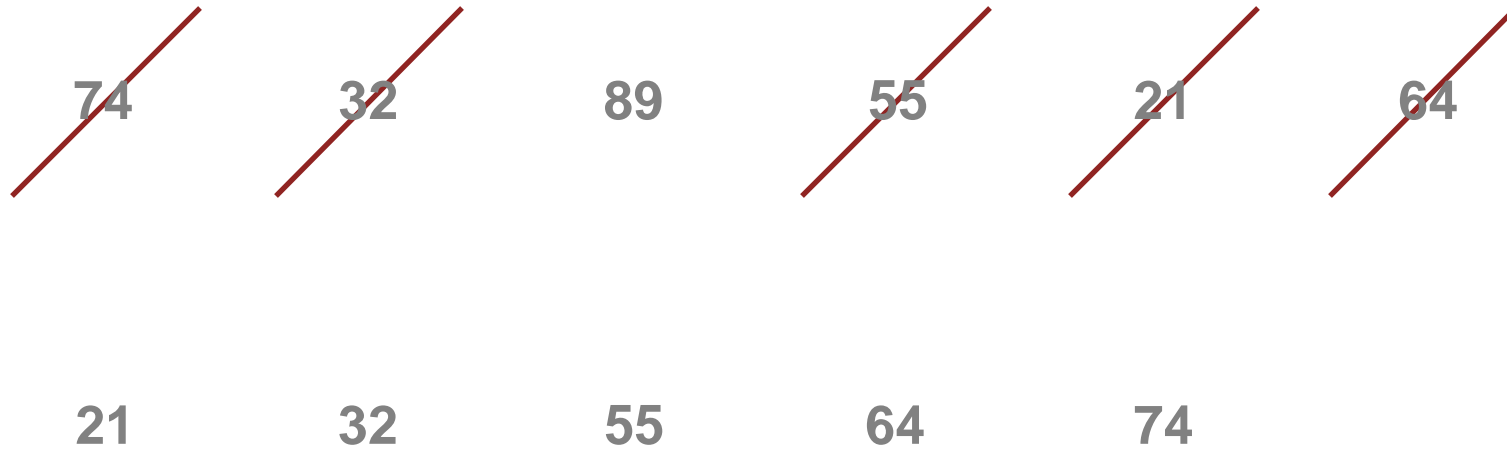
32

55

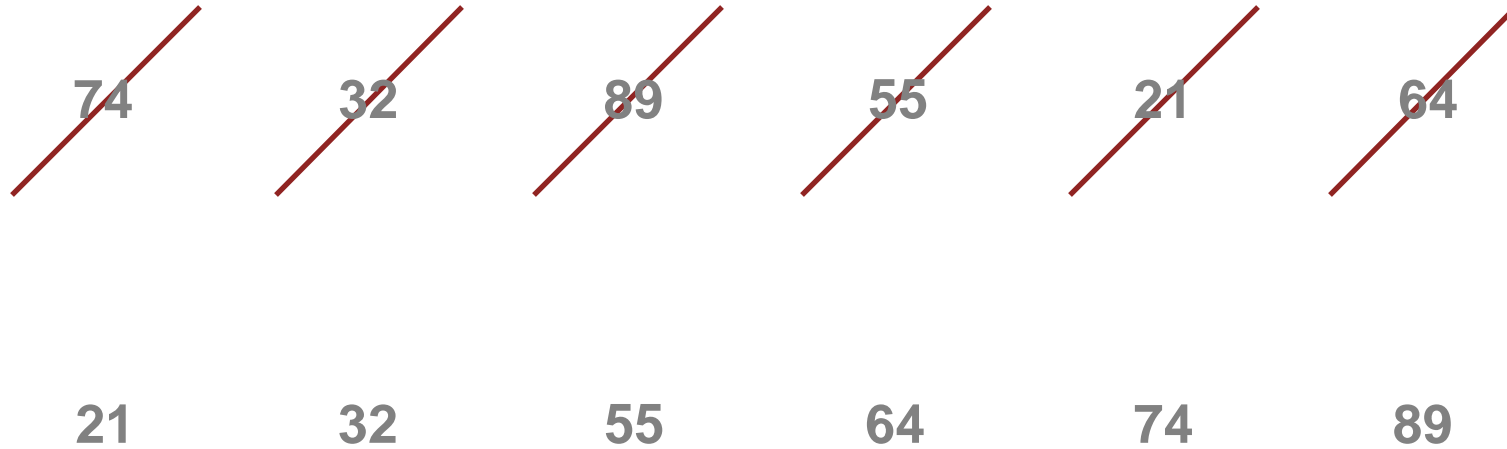
64



# Strategy 1 ...



# Strategy 1 ...





# Strategy 1 ...



- Selection Sort

- **Select** the next element in sorted order
- Move it into its correct place in the final sorted list



# Selection Sort



- Avoid using a second list
  - Swap minimum element with value in first position
  - Swap second minimum element to second position
  - ...



# Selection sort example

- Initial list:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

- After 1st, 2nd, and 3rd passes:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	18	12	22	27	30	36	50	7	68	91	56	2	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	12	22	27	30	36	50	7	68	91	56	18	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	22	27	30	36	50	12	68	91	56	18	85	42	98	25



# Selection Sort

```
SelectionSort(A,n) // Sort A of size n
for (startpos = 0; startpos < n; startpos++)
    // Scan segments A[0]..A[n-1], A[1]..A[n-1], ...

    // Locate position of minimum element in current segment
    minpos = startpos;
    for (i = minpos+1; i < n; i++)
        if (A[i] < A[minpos])
            minpos = i;

    // Move minimum element to start of current segment
    swap(A,startpos,minpos)
```



# Python code | Selection Sort

```
# SelectionSort(A,n) // Sort A of size n
def selectionSort(A):
    # for (startpos = 0; startpos < n; startpos++)
    for i in range(len(A)):
        # minpos = startpos;
        minpos = i
        # for (i = minpos+1; i < n; i++)
        for j in range(i+1, len(A)):
            # if (A[i] < A[minpos])
            if A[j] < A[minpos]:
                minpos = j
        # swap(A,startpos,minpos)
        swap(A, i, minpos)
def swap(lst, i, j):
    temp = lst[i]
    lst[i] = lst[j]
    lst[j] = temp
    return lst
A=[22, 18, 12, -4, 27, 30, 36, 50, 7, 68, 91, 56, 2, 85, 42, 98, 25]
selectionSort(A)
A
```



# Analysis of Selection Sort



- Finding minimum in unsorted segment of length  $k$  requires one scan,  $k$  steps.
- In each iteration, segment to be scanned reduces by 1
- $t(n) = n + (n-1) + (n-2) + \dots + 1 = n(n+1)/2 = O(n^2)$



# Recursive formulation



- To sort  $A[i .. n-1]$ 
  - Find minimum value in segment and move to  $A[i]$
  - Apply Selection Sort to  $A[i+1..n-1]$
- Base case
  - Do nothing if  $i = n-1$



# Selection Sort, recursive



```
SelectionSort(A, start, n) // Sort A from start to n-1
if (start >= n-1)
    return;
// Locate minimum element and move to start of segment
minpos = start;
for (i = start+1; i < n; i++)
    if (A[i] < A[minpos])
        minpos = i;
swap(A, start, minpos)
// Recursively sort the rest
SelectionSort(A, start+1, n)
```

Find the code here: <https://www.geeksforgeeks.org/recursive-selection-sort/amp/>





# Alternative calculation

- $t(n)$ , time to run selection sort on length  $n$ 
  - $n$  steps to find minimum and move to position 0
  - $t(n-1)$  time to run selection sort on  $A[1]$  to  $A[n-1]$
- **Recurrence**
  - $t(n) = n + t(n-1)$   
 $t(1) = 1$
  - $t(n) = n + t(n-1) = n + ((n-1) + t(n-2)) = \dots = n + (n-1) + (n-2) + \dots + 1 =$   
 $n(n+1)/2 = O(n^2)$



# How to sort?



- You are a Teaching Assistant for a course
- The instructor gives you a stack of exam answer papers with marks, ordered randomly
- Your task is to arrange them in ascending/descending order



# Strategy 2



- First paper: put in a new stack
- Second paper:
  - ✓ Lower marks than first? Place below first paper
  - ✓ Higher marks than first? Place above first paper
- Third paper
  - **Insert** into the correct position with respect to first two papers
- Do this for each subsequent paper:
  - **insert** into correct position in new sorted stack



# Strategy 2 ...



74

32

89

55

21

64



# Strategy 2 ...



~~74~~

32

89

55

21

64

74



# Strategy 2 ...

~~74~~

~~32~~

89

55

21

64

32

74



# Strategy 2 ...



~~74~~

~~32~~

~~89~~

55

21

64

32

74

89



# Strategy 2 ...

~~74~~

~~32~~

~~89~~

~~55~~

21

64

32

55

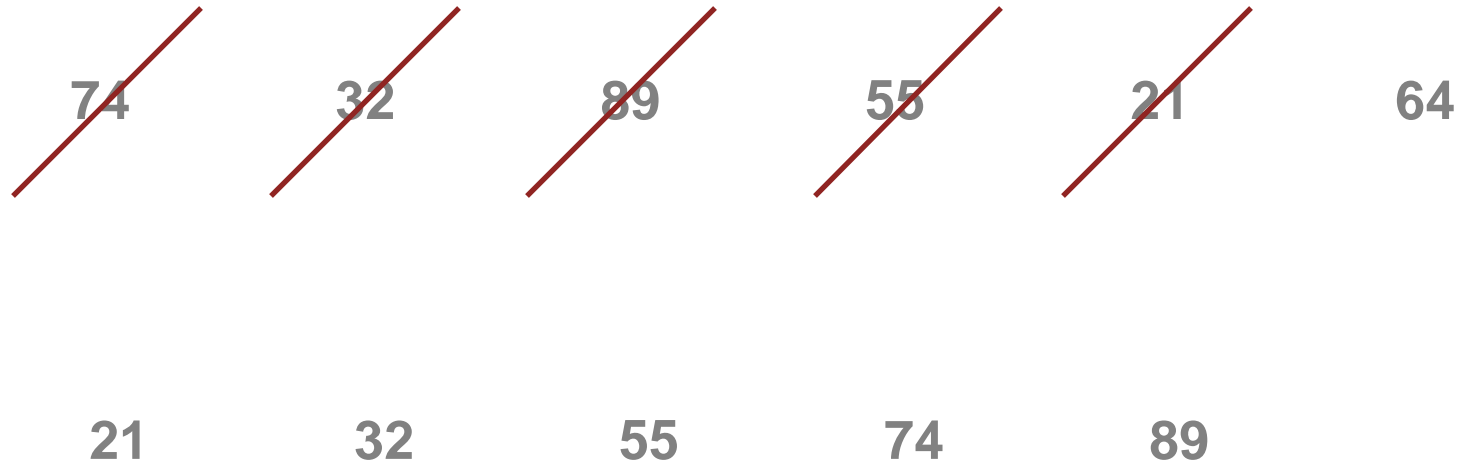
74

89





# Strategy 2 ...



# Strategy 2 ...

~~74~~

~~32~~

~~89~~

~~55~~

~~21~~

~~64~~

21

32

55

64

74

89



# Strategy 2 ...



- Insertion Sort
  - Start building a sorted sequence with one element
  - Pick up next unsorted element and insert it into its correct place in the already sorted sequence



# Insertion Sort

```
InsertionSort(A,n) // Sort A of size n
for (pos = 1; pos < n; pos++)
    // Build longer and longer sorted segments
    // In each iteration A[0]..A[pos-1] is already sorted

    // Move first element after sorted segment left
    // till it is in the correct place
    nextpos = pos
    while (nextpos > 0 &&
           A[nextpos] < A[nextpos-1])
        swap(A,nextpos,nextpos-1)
        nextpos = nextpos-1
```



# Python code | Insertion Sort

```
def insertionSort(A,n): # Sort A of size n
    for pos in range(1, n):
        nextpos = pos
        while nextpos > 0 and A[nextpos] < A[nextpos-1]:
            swap(A,nextpos,nextpos-1)
            nextpos = nextpos-1
    return A
```

```
def swap(lst, i, j):
    temp = lst[i]
    lst[i] = lst[j]
    lst[j] = temp
    return lst
```

```
lst = [74,32, 89, 55, 21, 64]
print(insertionSort(lst, len(lst))) # output: [21, 32, 55, 64, 74, 89]
```



# Insertion Sort



74

32

89

55

21

64



# Insertion Sort



74

32

89

55

21

64



# Insertion Sort



32

74

89

55

21

64





# Insertion Sort



32

74

89

55

21

64



# Insertion Sort



32

74

55

89

21

64



# Insertion Sort



32

55

74

89

21

64



# Insertion Sort



32

55

74

21

89

64



# Insertion Sort



32

55

21

74

89

64



# Insertion Sort



32

21

55

74

89

64



# Insertion Sort



21

32

55

74

89

64



# Insertion Sort



21

32

55

74

64

89





# Insertion Sort



21

32

55

64

74

89



# Analysis of Insertion Sort

- Inserting a new value in sorted segment of length  $k$  requires up to  $k$  steps in the worst case
- In each iteration, sorted segment in which to insert increased by 1
- $t(n) = 1 + 2 + \dots + n-1 = n(n-1)/2 = O(n^2)$



# Recursive formulation



- To sort  $A[0..n-1]$ 
  - Recursively sort  $A[0..n-2]$
  - Insert  $A[n-1]$  into  $A[0..n-2]$
- Base case:  $n = 1$



# Insertion Sort, recursive



```
InsertionSort(A,k) // Sort A[0..k-1]
    if (k == 1)
        return;

    InsertionSort(A,k-1);

    Insert(A,k-1);

    return;

Insert(A,j) // Insert A[j] into A[0..j-1]
    pos = j;
    while (pos > 0 && A[pos] < A[pos-1])
        swap(A,pos,pos-1);
    pos = pos-1;
```

Find the code here: <https://www.geeksforgeeks.org/recursive-insertion-sort/amp/>



# Recurrence



- $t(n)$ , time to run insertion sort on length  $n$ 
  - Time  $t(n-1)$  to sort segment  $A[0]$  to  $A[n-2]$
  - $n-1$  steps to insert  $A[n-1]$  in sorted segment
- Recurrence
  - $t(n) = n-1 + t(n-1)$   $t(1) = 1$
  - $t(n) = n-1 + t(n-1) = n-1 + ((n-2) + t(n-2)) = \dots = (n-1) + (n-2) + \dots + 1 = n(n-1)/2 = O(n^2)$



# $O(n^2)$ sorting algorithms



- Selection sort and insertion sort are both  $O(n^2)$
- So is bubble sort, which we will not discuss here
- $O(n^2)$  sorting is infeasible for  $n$  over 10000
- Among  $O(n^2)$  sorts, insertion sort is usually better than selection sort and both are better than bubble sort
  - What happens when we apply insertion sort to an already sorted list?



# $O(n^2)$ sorting algorithms

- Selection sort and insertion sort are both  $O(n^2)$
- $O(n^2)$  sorting is infeasible for  $n$  over 100000



# A different strategy?

- Divide array in two equal parts
- Separately sort left and right half
- Combine the two sorted halves to get the full array sorted





# Combining sorted lists



- Given two sorted lists A and B, combine into a sorted list C
  - Compare first element of A and B
  - Move it into C
  - Repeat until all elements in A and B are over
- **Merging** A and B



# Merging two sorted lists

32                      74                      89

21                      55                      64



# Merging two sorted lists

32

74

89

~~21~~

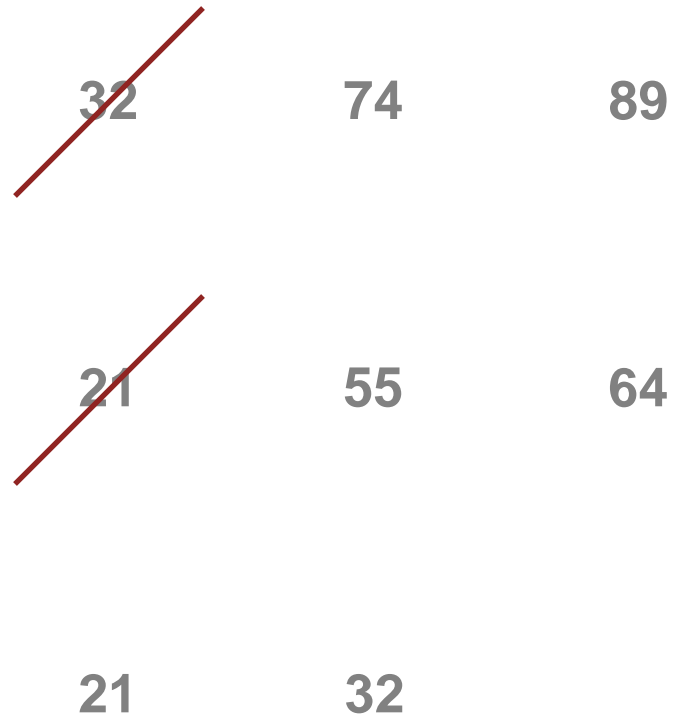
55

64

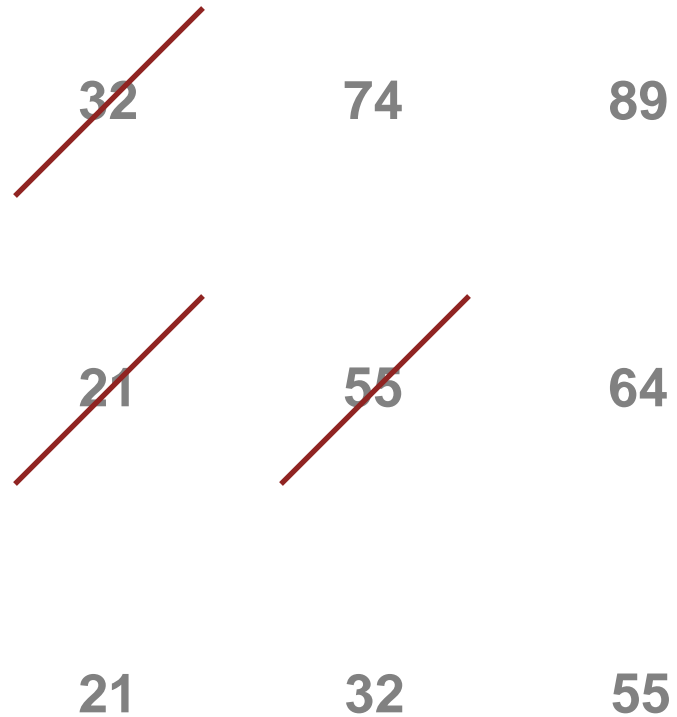
21



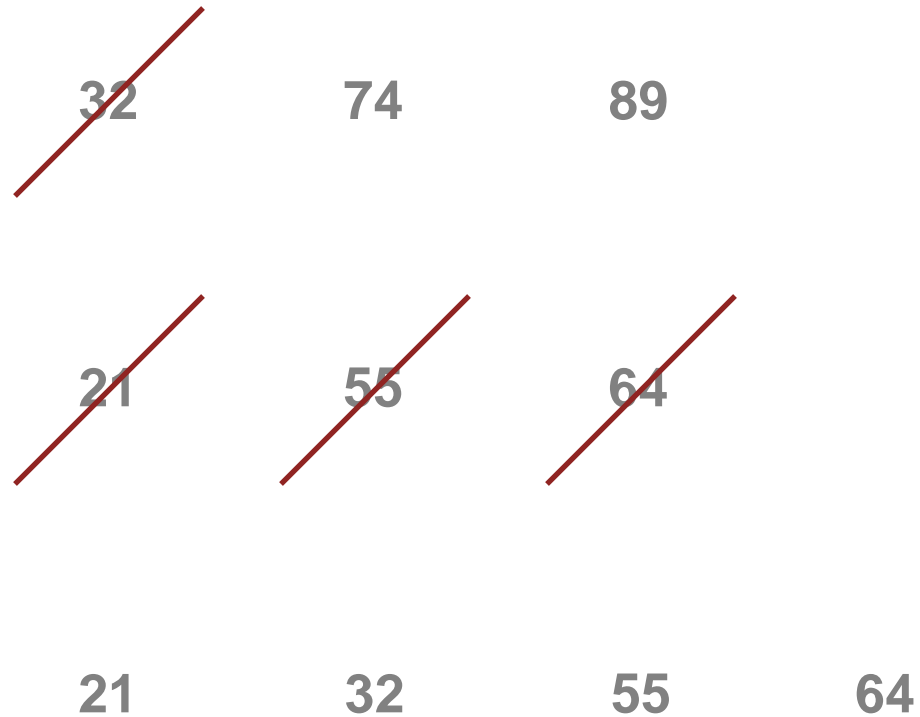
# Merging two sorted lists



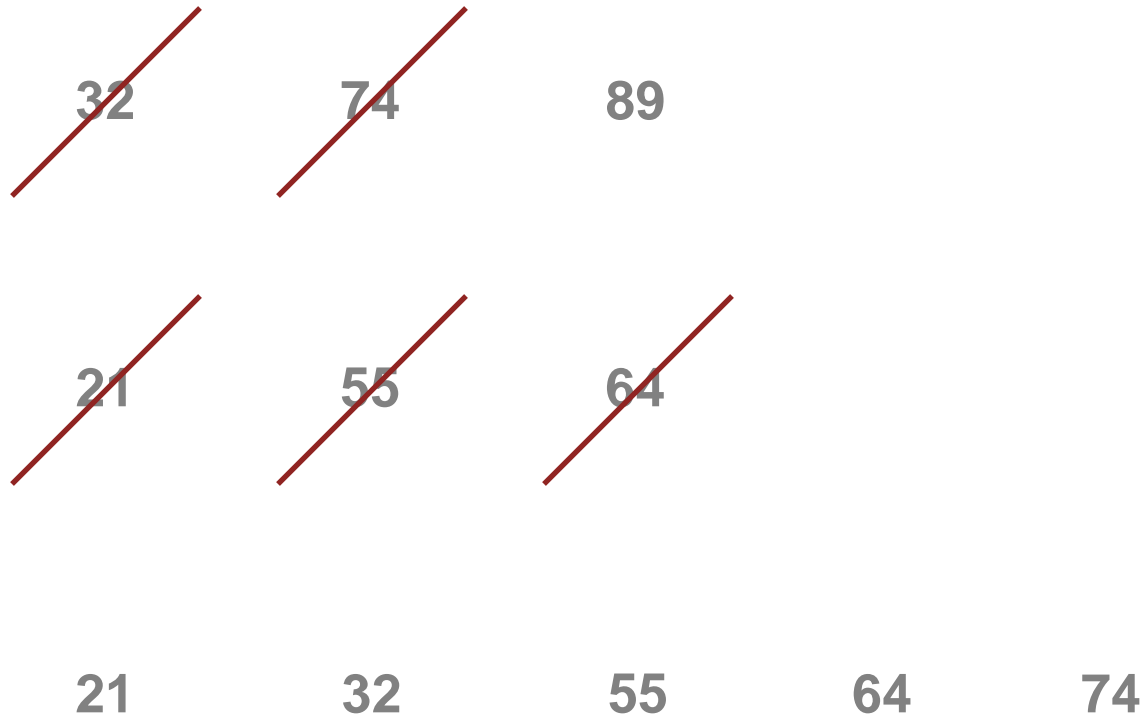
# Merging two sorted lists



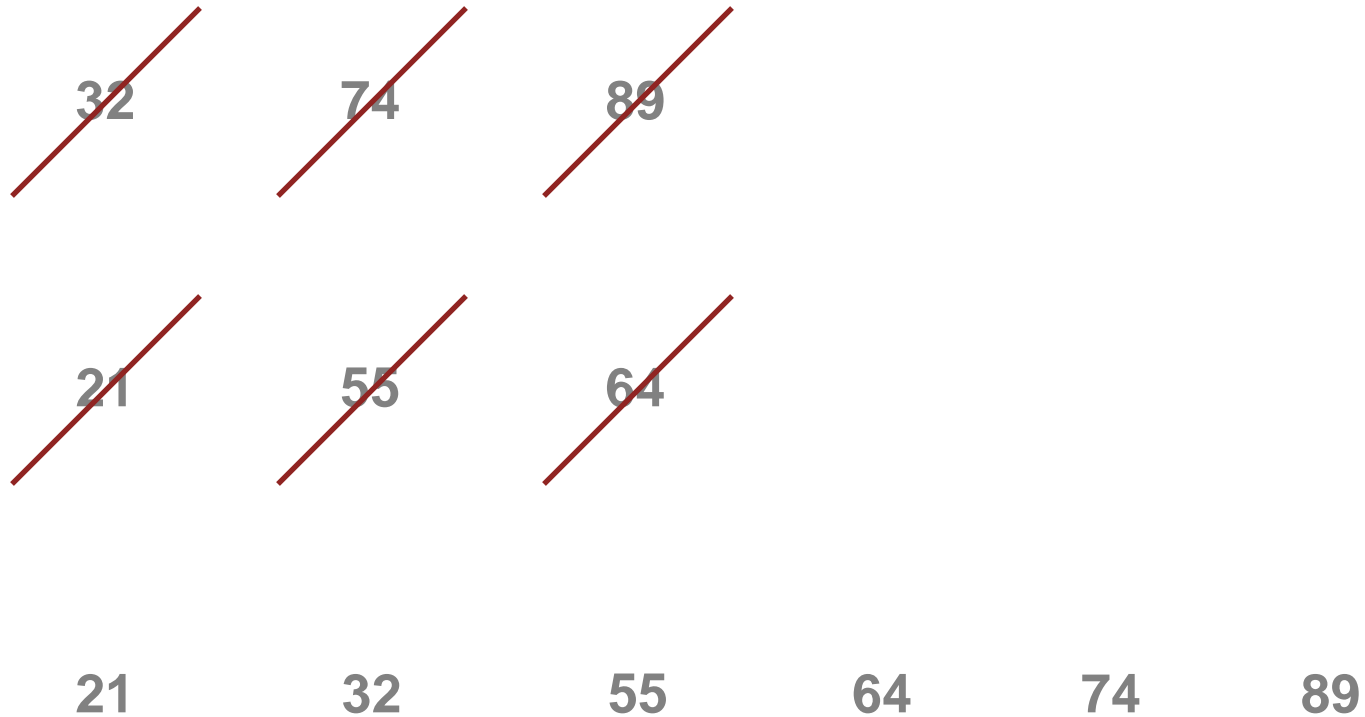
# Merging two sorted lists



# Merging two sorted lists



# Merging two sorted lists





# Merge Sort



- Sort  $A[0]$  to  $A[n/2-1]$
- Sort  $A[n/2]$  to  $A[n-1]$
- Merge sorted halves into  $B[0..n-1]$
- How do we sort the halves?
- Recursively, using the same strategy!



# Merge Sort

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----



# Merge Sort

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

43	32	22	78
----	----	----	----



# Merge Sort

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

43	32	22	78
----	----	----	----

63	57	91	13
----	----	----	----



# Merge Sort

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

43	32	22	78
----	----	----	----

63	57	91	13
----	----	----	----

43	32
----	----

22	78
----	----



# Merge Sort

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

43	32	22	78
----	----	----	----

63	57	91	13
----	----	----	----

43	32
----	----

22	78
----	----

63	57
----	----

91	13
----	----



# Merge Sort

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

43	32	22	78
----	----	----	----

63	57	91	13
----	----	----	----

43	32
----	----

22	78
----	----

63	57
----	----

91	13
----	----

43
----

32
----



# Merge Sort

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

43	32	22	78
----	----	----	----

63	57	91	13
----	----	----	----

43	32
----	----

22	78
----	----

63	57
----	----

91	13
----	----

43
----

32
----

22
----

78
----





# Merge Sort

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

43	32	22	78
----	----	----	----

63	57	91	13
----	----	----	----

43	32
----	----

22	78
----	----

63	57
----	----

91	13
----	----

43
----

32
----

22
----

78
----

63
----

57
----



# Merge Sort

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

43	32	22	78
----	----	----	----

63	57	91	13
----	----	----	----

43	32
----	----

22	78
----	----

63	57
----	----

91	13
----	----

43
----

32
----

22
----

78
----

63
----

57
----

91
----

13
----



# Merge Sort

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

43	32	22	78
----	----	----	----

63	57	91	13
----	----	----	----

32	43
----	----

22	78
----	----

63	57
----	----

91	13
----	----

43
----

32
----

22
----

78
----

63
----

57
----

91
----

13
----



# Merge Sort

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

43	32	22	78
----	----	----	----

63	57	91	13
----	----	----	----

32	43
----	----

22	78
----	----

63	57
----	----

91	13
----	----

43
----

32
----

22
----

78
----

63
----

57
----

91
----

13
----



# Merge Sort

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

43	32	22	78
----	----	----	----

63	57	91	13
----	----	----	----

32	43
----	----

22	78
----	----

57	63
----	----

91	13
----	----

43
----

32
----

22
----

78
----

63
----

57
----

91
----

13
----



# Merge Sort

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

43	32	22	78
----	----	----	----

63	57	91	13
----	----	----	----

32	43
----	----

22	78
----	----

57	63
----	----

13	91
----	----

43
----

32
----

22
----

78
----

63
----

57
----

91
----

13
----



# Merge Sort

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

22	32	43	78
----	----	----	----

63	57	91	13
----	----	----	----

32	43
----	----

22	78
----	----

57	63
----	----

13	91
----	----

43
----

32
----

22
----

78
----

63
----

57
----

91
----

13
----



# Merge Sort

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

22	32	43	78
----	----	----	----

13	57	63	91
----	----	----	----

32	43
----	----

22	78
----	----

57	63
----	----

13	91
----	----

43
----

32
----

22
----

78
----

63
----

57
----

91
----

13
----





# Merge Sort

13	22	32	43	57	63	78	91
----	----	----	----	----	----	----	----

22	32	43	78
----	----	----	----

13	57	63	91
----	----	----	----

32	43
----	----

22	78
----	----

57	63
----	----

13	91
----	----

43
----

32
----

22
----

78
----

63
----

57
----

91
----

13
----



# Divide and conquer

- Break up problem into disjoint parts
- Solve each part separately
- Combine the solutions efficiently



# Merging sorted lists

- Combine two sorted lists A and B into C
- If A is empty, copy B into C
- If B is empty, copy A into C
- Otherwise, compare first element of A and B and move the smaller of the two into C
- Repeat until all elements in A and B have been moved



# Merging

```
function Merge(A,m,B,n,C)
// Merge A[0..m-1], B[0..n-1] into C[0..m+n-1]

i = 0; j = 0; k = 0;
// Current positions in A,B,C respectively

while (k < m+n)
// Case 0: One of the two lists is empty
    if (i==m) {j++; k++;}
    if (j==n) {i++; k++;}
// Case 1: Move head of A into C
    if (A[i] <= B[j]) { C[k] = A[i]; i++; k++;}
// Case 2: Move head of B into C
    if (A[i] > B[j]) { C[k] = B[j]; j++; k++;}
```



# Merge Sort

- To sort  $A[0..n-1]$  into  $B[0..n-1]$
- If  $n$  is 1, nothing to be done
- Otherwise
  - Sort  $A[0..n/2-1]$  into  $L$  (left)
  - Sort  $A[n/2..n-1]$  into  $R$  (right)
  - Merge  $L$  and  $R$  into  $B$



# Merge Sort

```
function MergeSort(A, left, right, B)
    // Sort the segment A[left..right-1] into B

    if (right - left == 1) // Base case
        B[0] = A[left]

    if (right - left > 1) // Recursive call

        mid = (left+right)/2

        MergeSort(A, left, mid, L)
        MergeSort(A, mid, right, R)

        Merge(L, mid-left, R, right-mid, B)
```



# Python Code | Merge Sort



```
# merge sort
```

```
def merge_sort(a):
```

```
    if len(a) >= 2:
```

```
        # split list into two halves
```

```
        left = a[0: len(a)//2]
```

```
        right = a[len(a)//2: len(a)]
```

```
    # sort the two halves
```

```
    merge_sort(left)
```

```
    merge_sort(right)
```

```
    # merge the sorted halves into a sorted whole
```

```
    merge(a, left, right)
```



# Python Code | Merge Sort

```
def merge(result, left, right):  
    i1 = 0 # index into left list  
    i2 = 0 # index into right list  
  
    for i in range(0, len(result)):  
        if i2 >= len(right) or (i1 < len(left) and left[i1] <= right[i2]):  
            result[i] = left[i1] # take from left  
            i1 += 1  
        else:  
            result[i] = right[i2] # take from right  
            i2 += 1  
  
d = [22, 18, 12, -4, 27, 30, 36, 50, -9, 90, 10, -1]  
merge_sort(d)  
print(d)
```





# Analysis of Merge

- How much time does Merge take?
- Merge A of size m, B of size n into C
  - In each iteration, we add one element to C
  - At most 7 basic operations per iteration
  - Size of C is m+n
    - $m+n \lesssim 2 \max(m,n)$
- Hence  $O(\max(m,n)) = O(n)$  if  $m \approx n$



# Merge Sort



- To sort  $A[0..n-1]$  into  $B[0..n-1]$
- If  $n$  is 1, nothing to be done Otherwise
  - Sort  $A[0..n/2-1]$  into  $L$  (left)
  - Sort  $A[n/2..n-1]$  into  $R$  (right)
  - Merge  $L$  and  $R$  into  $B$



# Analysis of Merge Sort ...



- $t(n)$ : time taken by Merge Sort on input of size  $n$ 
  - Assume, for simplicity, that  $n = 2^k$
- $t(n) = 2t(n/2) + n$ 
  - Two subproblems of size  $n/2$
  - Merging solutions requires time  $O(n/2 + n/2) = O(n)$
- Solve the recurrence by unwinding



# Analysis of Merge Sort ...

- $t(1) = 1$
- $t(n) = 2t(n/2) + n$ 
  - $= 2 [ 2t(n/4) + n/2 ] + n = 2^2 t(n/2^2) + 2n$
  - $= 2^2 [ 2t(n/2^3) + n/2^2 ] + 2n = 2^3 t(n/2^3) + 3n$
  - ...
  - $= 2^j t(n/2^j) + jn$
- When  $j = \log n$ ,  $n/2^j = 1$ , so  $t(n/2^j) = 1$ 
  - $\log n$  means  $\log_2 n$  unless otherwise specified!
- $t(n) = 2^j t(n/2^j) + jn = 2^{\log n} + (\log n) n = n + n \log n = O(n \log n)$



# $O(n \log n)$ sorting

- Recall that  $O(n \log n)$  is much more efficient than  $O(n^2)$
- Assuming  $10^8$  operations per second, feasible input size goes from 10,000 to 10,000,000.



# Variations on merge

- Union of two sorted lists (discard duplicates)
  - If  $A[i] == B[j]$ , copy  $A[i]$  to  $C[k]$  and increment  $i, j, k$
- Intersection of two sorted lists
  - If  $A[i] < B[j]$ , increment  $i$
  - If  $B[j] < A[i]$ , increment  $j$
  - If  $A[i] == B[j]$ , copy  $A[i]$  to  $C[k]$  and increment  $i, j, k$

**Exercise:** List difference: elements in  $A$  but not in  $B$



# Merge Sort: Shortcomings



- Merging A and B creates a new array C
  - No obvious way to efficiently merge in place
- Extra storage can be costly
- Inherently recursive
  - Recursive call and return are expensive



# Alternative approach



- Extra space is required to merge
- Merging happens because elements in left half must move right and vice versa
- Can we divide so that everything to the left is smaller than everything to the right?
  - No need to merge!





# Divide and conquer without merging



- Suppose the median value in A is m
- Move all values  $\leq m$  to left half of A
  - Right half has values  $> m$
  - This shifting can be done in place, in time  $O(n)$
- Recursively sort left and right halves
- A is now sorted! No need to merge

$$t(n) = 2t(n/2) + n = O(n \log n)$$



# Divide and conquer without merging



- How do we find the median?
  - Sort and pick up middle element
  - But our aim is to sort!
- Instead, pick up some value in A — **pivot**
- Split A with respect to this pivot element



# Quicksort



- Choose a pivot element
  - Typically the first value in the array
- Partition  $A$  into lower and upper parts with respect to pivot
- Move pivot between lower and upper partition
- Recursively sort the two partitions



# Quicksort

- High level view

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----



# Quicksort

- High level view

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----



# Quicksort

- High level view

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----



# Quicksort

- High level view

13	32	22	43	63	57	91	78
----	----	----	----	----	----	----	----



# Quicksort

- High level view

13	22	32	43	57	63	78	91
----	----	----	----	----	----	----	----






# Quicksort: Partitioning

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----




# Quicksort: Partitioning

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----




# Quicksort: Partitioning

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----




# Quicksort: Partitioning

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----




# Quicksort: Partitioning

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----



# Quicksort: Partitioning

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----



A diagram illustrating the partitioning step of the Quicksort algorithm. It shows an array of eight numbers: 43, 32, 22, 78, 63, 57, 91, and 13. The number 22 is highlighted in gold, and the number 78 is highlighted in green. Below the array, two vertical arrows point upwards towards the positions of 22 and 78. The left arrow is gold and points to the position of 22. The right arrow is green and points to the position of 78.



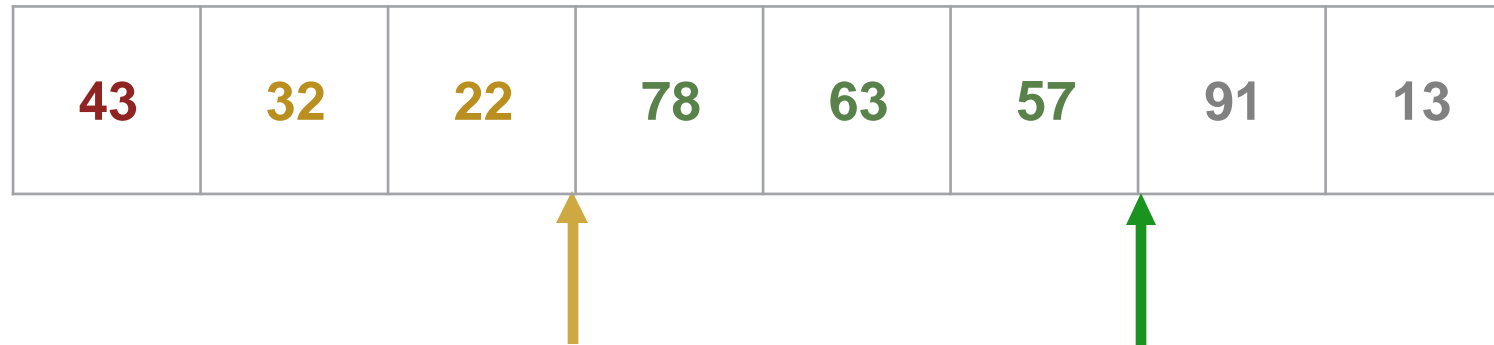
# Quicksort: Partitioning

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

The diagram illustrates the partitioning step of the Quicksort algorithm. It shows an array of eight numbers: 43, 32, 22, 78, 63, 57, 91, and 13. The numbers 22 and 63 are highlighted in yellow and green, respectively, indicating they are the current pivot elements. A yellow arrow points to the position of 22, and a green arrow points to the position of 63, showing the pointers moving towards each other to partition the array.

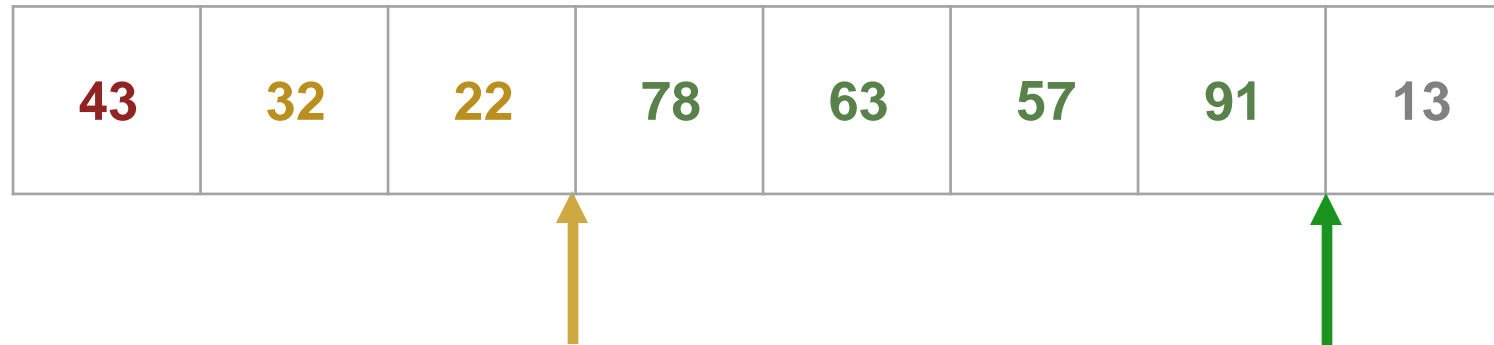


# Quicksort: Partitioning

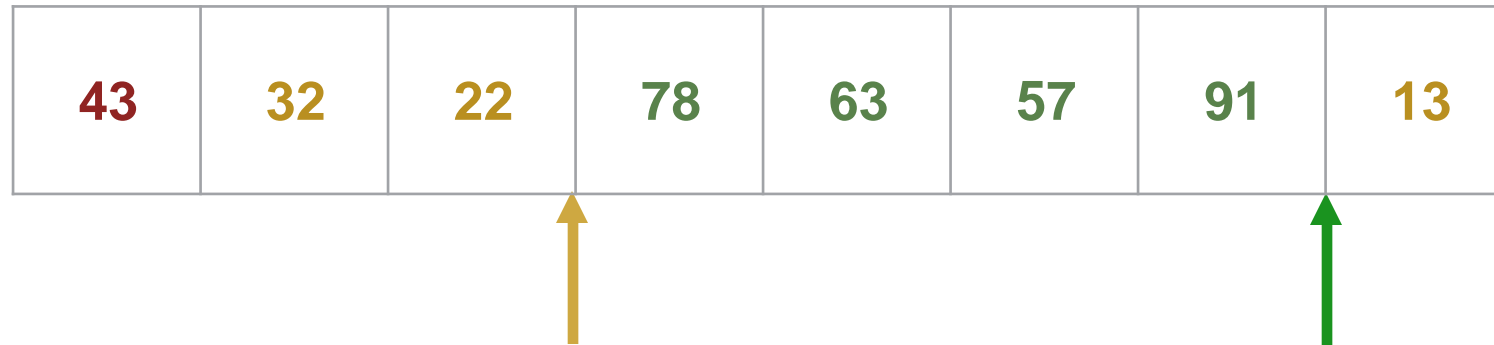




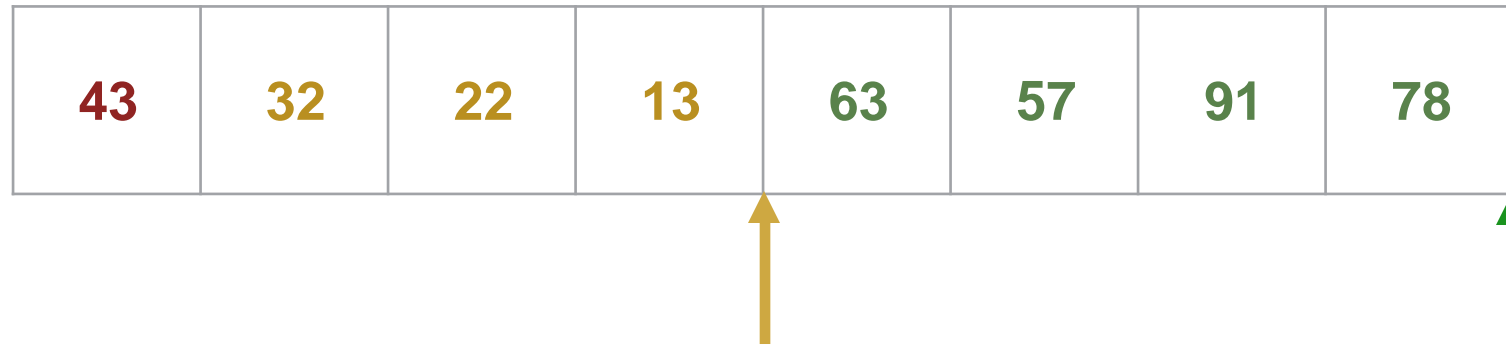
# Quicksort: Partitioning



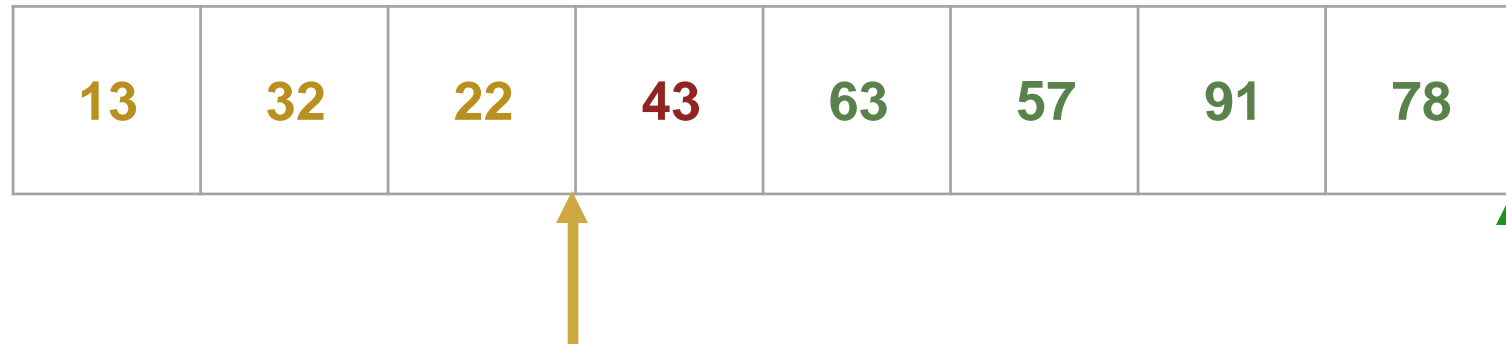
# Quicksort: Partitioning



# Quicksort: Partitioning



# Quicksort: Partitioning



# Quicksort: Implementation

```
Quicksort(A, l, r) // Sort A[l..r-1]

if (r - l <= 1) return; // Base case

// Partition with respect to pivot, a[l]
yellow = l+1;
for (green = l+1; green < r; green++) if
    (A[green] <= A[l])
        swap(A, yellow, green);
        yellow++;

swap(A, l, yellow-1); // Move pivot into place
Quicksort(A, l, yellow); // Recursive calls
Quicksort(A, yellow+1, r);
```



# Quicksort: Another Partitioning Strategy

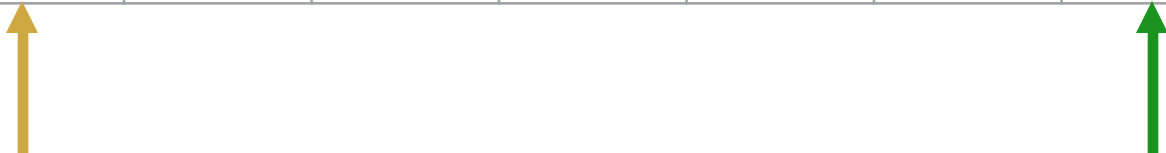


43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----



# Quicksort: Another Partitioning Strategy

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----




A diagram illustrating a partitioning strategy in the Quicksort algorithm. It features a horizontal array of eight numbers: 43, 32, 22, 78, 63, 57, 91, and 13. The number 43 is highlighted in red. Below the array, a yellow arrow points upwards to the number 32, and a green arrow points upwards to the number 13. These arrows likely represent the 'low' and 'high' pointers used in the partitioning process.



# Quicksort: Another Partitioning Strategy

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----





The diagram illustrates a partitioning strategy in the Quicksort algorithm. It shows an array of eight numbers: 43, 32, 22, 78, 63, 57, 91, and 13. An orange arrow points upwards to the number 22, which is the pivot element. A green arrow points upwards to the number 13, which is the element being compared to the pivot. The numbers 43 and 32 are colored red and yellow respectively, while the others are grey.





# Quicksort: Another Partitioning Strategy

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----



# Quicksort: Another Partitioning Strategy



43	32	22	13	63	57	91	78
----	----	----	----	----	----	----	----

↑                      ↑  
63                    91



# Quicksort: Another Partitioning Strategy


43	32	22	13	63	57	91	78
----	----	----	----	----	----	----	----

↑      ↑



# Quicksort: Another Partitioning Strategy

43	32	22	13	63	57	91	78
----	----	----	----	----	----	----	----



# Quicksort: Another Partitioning Strategy

43	32	22	13	63	57	91	78
----	----	----	----	----	----	----	----

↑      ↑



# Quicksort: Another Partitioning Strategy

13	32	22	43	63	57	91	78
----	----	----	----	----	----	----	----

A diagram illustrating a partitioning step in the Quicksort algorithm. It shows an array of eight numbers: 13, 32, 22, 43, 63, 57, 91, and 78. The number 22 is highlighted in gold, and a green arrow points up to it from below. The number 63 is highlighted in green, and a gold arrow points up to it from below. The number 43 is highlighted in red.



# Quicksort



- Choose a pivot element
  - Typically the first value in the array
- Partition  $A$  into lower and upper parts with respect to pivot
- Move pivot between lower and upper partition
- Recursively sort the two partitions



# Analysis of Quicksort

- Partitioning with respect to pivot takes  $O(n)$
- If pivot is median
  - Each partition is of size  $n/2$
  - $t(n) = 2t(n/2) + n = O(n \log n)$
- Worst case?





# Analysis of Quicksort



- Worst case
  - Pivot is maximum or minimum
    - One partition is empty Other is size  $n-1$
    - $t(n) = t(n-1) + n = t(n-2) + (n-1) + n$   
 $= \dots = 1 + 2 + \dots + n = O(n^2)$
  - Already sorted array is worst case input!



# Analysis of Quicksort



- But ...
- Average case is  $O(n \log n)$ 
  - Sorting is a rare example where average case can be computed
- What does average case mean?



# Quicksort: Average case



- Assume input is a permutation of  $\{1, 2, \dots, n\}$ 
  - Actual values not important
  - Only relative order matters
  - Each input is equally likely (uniform probability)
- Calculate running time across all inputs
- **Expected running time** can be shown  $O(n \log n)$



# Quicksort: randomization

- Worst case arises because of fixed choice of pivot
  - We chose the first element
  - For any fixed strategy (last element, midpoint), can work backwards to construct  $O(n^2)$  worst case
- Instead, choose pivot **randomly**
  - Pick any index in  $[0..n-1]$  with uniform probability
  - Expected running time is again  $O(n \log n)$
- Expected running time is again  $O(n \log n)$



# Iterative Quicksort



- Recursive calls work on disjoint segments of array
  - No recombination of results required
- Can use an explicit stack to simulate recursion
  - Stack only needs to store left and right endpoints of interval to be sorted



# Quicksort in practice



- In practice, Quicksort is very fast
  - Typically, the default algorithm for in-built sort functions
    - Spreadsheets
    - Built in sort functions in programming languages



# Stable sorting



- Sorting on multiple criteria
- Assume students are listed in alphabetical order
- Now sort students by marks
  - After sorting, are students with equal marks still in alphabetical order?
- Stability is crucial in applications like spreadsheets
  - Sorting column B should not disturb previous sort on column A



# Stable sorting ...



- Quicksort, as shown, is not stable
  - Swap operation during partitioning disturbs original order
- Merge sort is stable if we merge carefully
  - Do not allow elements from right to overtake elements from left
  - Favour left list when breaking ties





# Other criteria



- Minimize data movement
  - Imagine values are heavy cartons
  - Want to reduce effort of moving values around



# Which is the best?



- Typically, Quicksort
  - Be careful to avoid worst-case
  - Randomize choice of pivot element
- Merge sort is used for “external” sorting
  - Database tables do not fit in memory
  - Need to sort on disk



# Which is the best?



- Other  $O(n \log n)$  algorithms exist
  - Heap sort
- Naive  $O(n^2)$  not used except when data is small
- Hybrid algorithms
  - Use divide and conquer for large  $n$
  - Switch to insertion sort when  $n$  small (e.g.  $n < 16$ )

