Sequences of values

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

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
- Inserting or deleting an element is easy
  - "Plumbing"

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

Operations

- Exchange A[i] and A[j]
  - Constant time in array, linear time in lists
- Delete A[i] or Insert v 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

Worst case

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

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;
```

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 < midpoint, search left half of A
  - If K > 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 …

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

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 …

- Works only for arrays
  - Need to be look 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

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

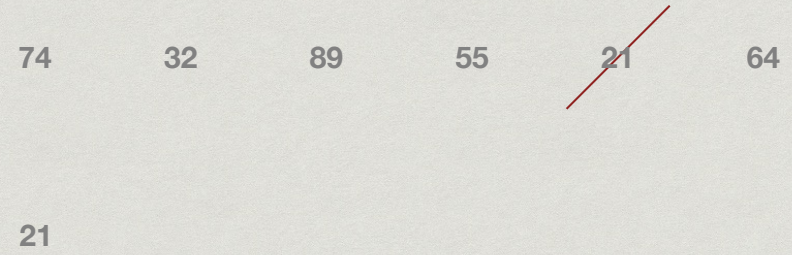
How to sort?

- 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 descending order

Strategy 1 …

74 32 89 55 21 64

Strategy 1 …



Strategy 1 …

Strategy 1 …

Strategy 1 …

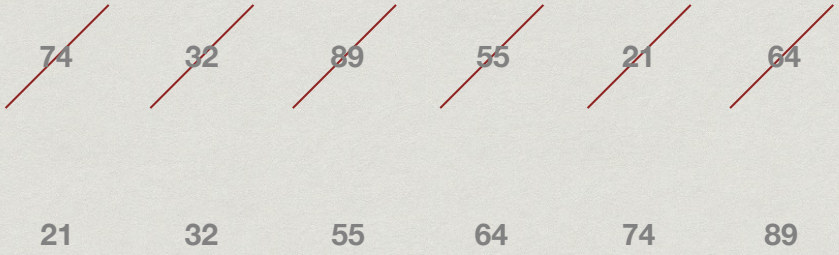
Strategy 1 …

Strategy 1 …

Selection Sort

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

Strategy 1 …



Selection Sort

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

Selection Sort

74 32 89 55 21 64

Selection Sort

21 32 89 55 74 64

Selection Sort

74 32 89 55 21 64

Selection Sort

21 32 89 55 74 64

Selection Sort

21 32 89 55 74 64

Selection Sort

21 32 55 89 74 64

Selection Sort

21 32 89 55 74 64

Selection Sort

21 32 55 89 74 64

Selection Sort

21 32 55 64 74 89

Selection Sort

21 32 55 64 74 89

Selection Sort

21 32 55 64 74 89

Selection Sort

21 32 55 64 74 89

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)
```

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

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
- $t(n) = n + (n-1) + (n-2) + \ldots + 1 = n(n+1)/2 = O(n^2)$

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)
```

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
  - $t(n) = n + t(n-1)$
    $t(1) = 1$
  - $t(n) = n + t(n-1) = n + ((n-1) + t(n-2)) = \ldots =$
    $n + (n-1) + (n-2) + \ldots + 1 = n(n+1)/2 = O(n^2)$

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 descending order

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

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

32 74

Strategy 2 …

~~74~~ 32 89 55 21 64

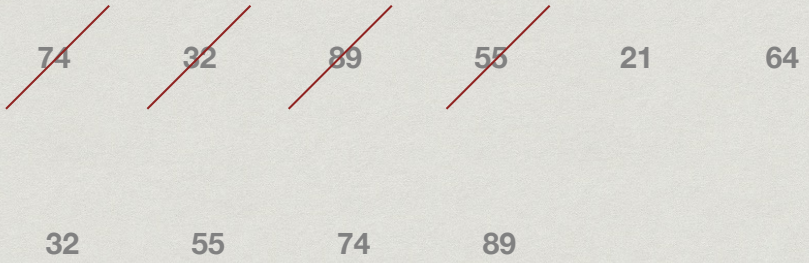
74

Strategy 2 …

~~74~~ ~~32~~ ~~89~~ 55 21 64

32 74 89

Strategy 2 …



Strategy 2 …

Strategy 2 …

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
```

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 55 74 89 21 64

Insertion Sort

32 74 55 89 21 64

Insertion Sort

32 55 74 21 89 64

Insertion Sort

32 55 21 74 89 64

Insertion Sort

21 32 55 74 89 64

Insertion Sort

32 21 55 74 89 64

Insertion Sort

21 32 55 74 64 89

Insertion Sort

21 **32** **55** **64** **74** **89**

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

Analysis of Insertion Sort

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

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;
```

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
  - $t(n) = n-1 + t(n-1)$
    $t(1) = 1$
  - $t(n) = n-1 + t(n-1) = n-1 + ((n-2) + t(n-2)) = \ldots = (n-1) + (n-2) + \ldots + 1 = n(n-1)/2 = O(n^2)$

$O(n^2)$ sorting algorithms

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

$O(n^2)$ 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?

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

21

Merging two sorted lists

32 74 89

21 55 64

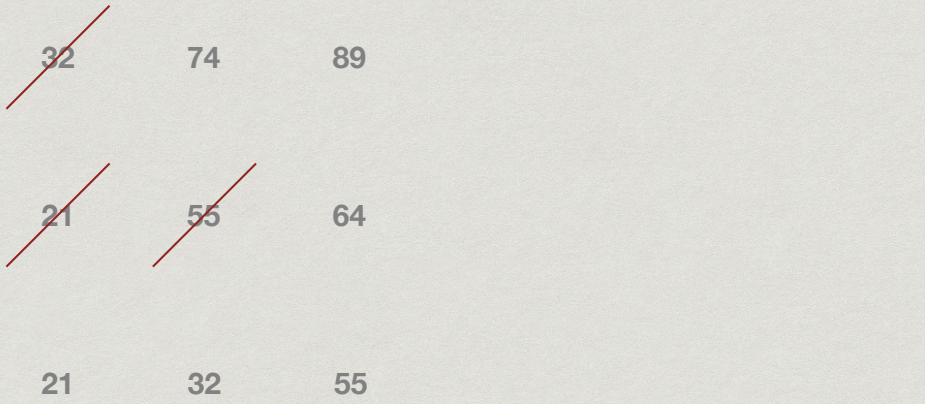
Merging two sorted lists

~~32~~ 74 89

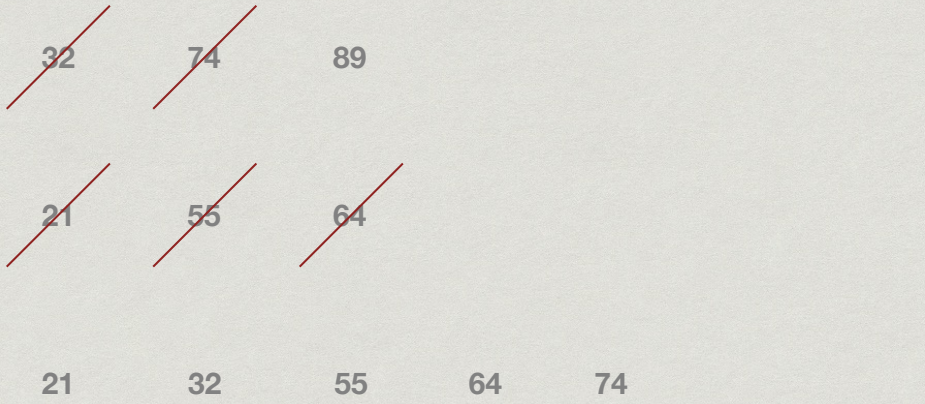
~~21~~ 55 64

21 32

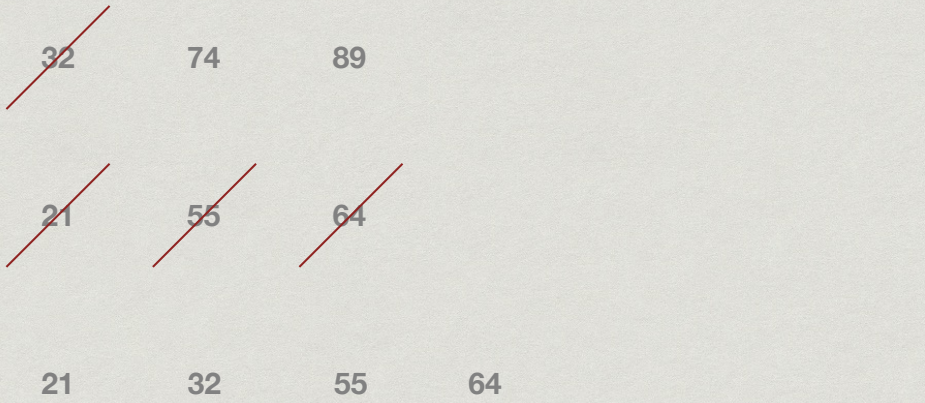
Merging two sorted lists



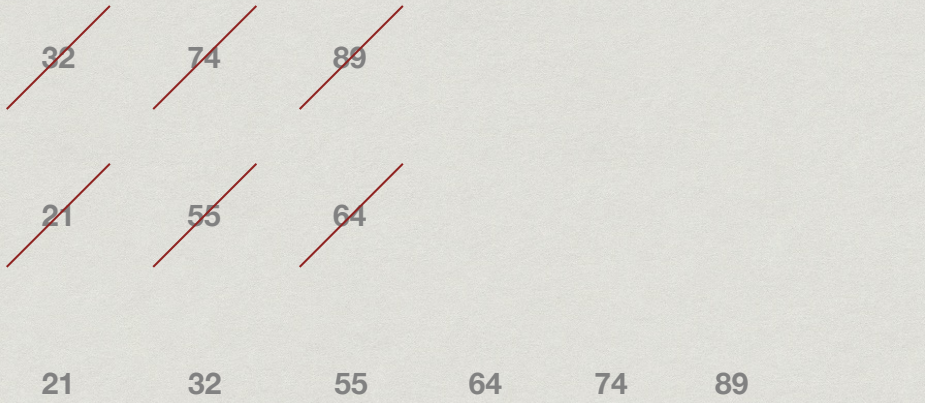
Merging two sorted lists



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

43	32	22	78

Merge Sort

43	32	22	78	63	57	91	13

Merge Sort

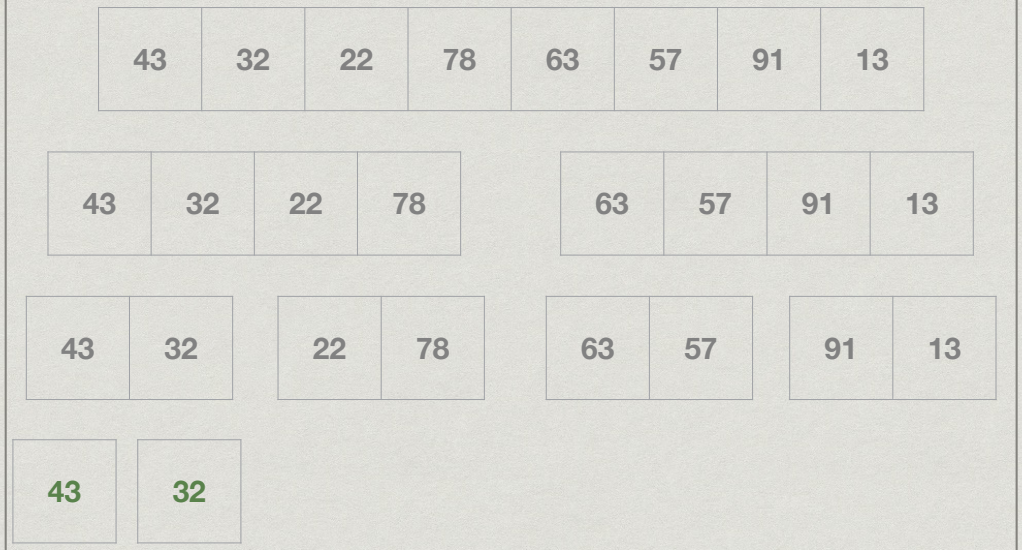
43	32	22	78	63	57	91	13

43	32	22	78

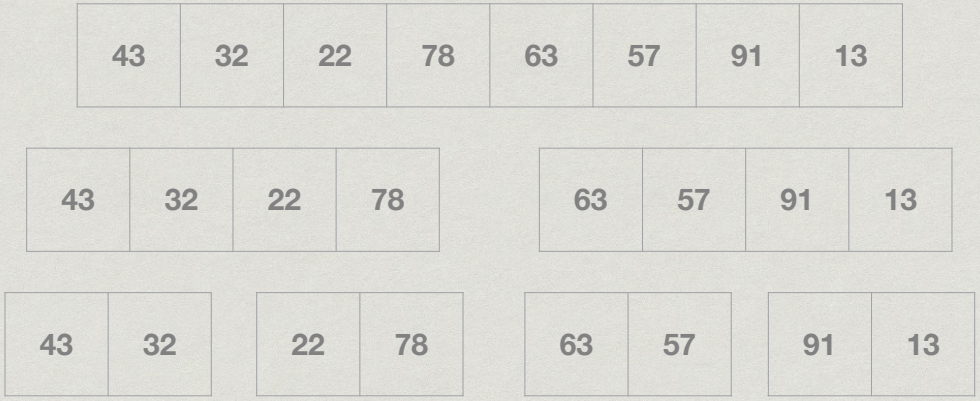
63	57	91	13

Merge Sort

Merge Sort

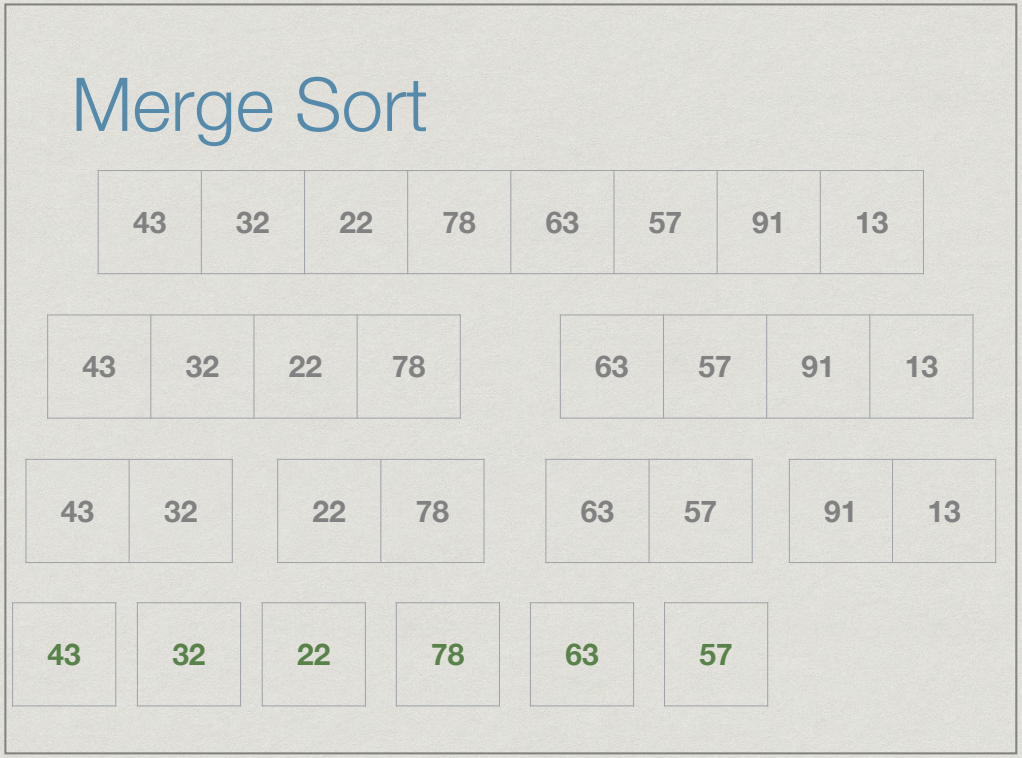


Merge Sort

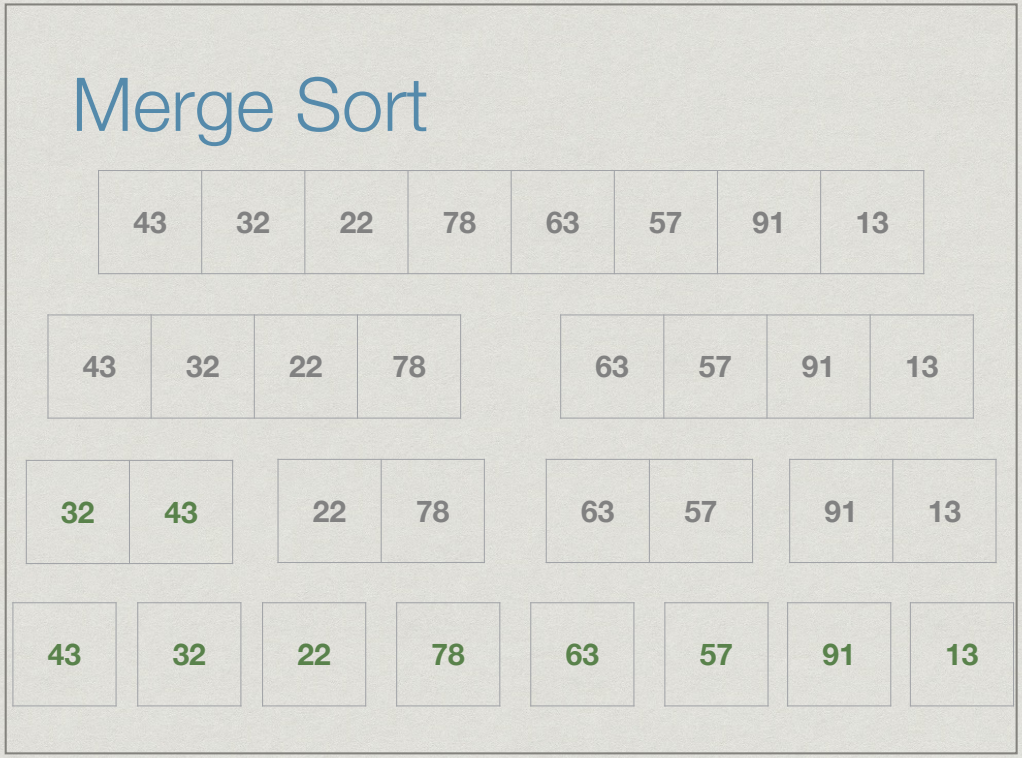


Merge Sort

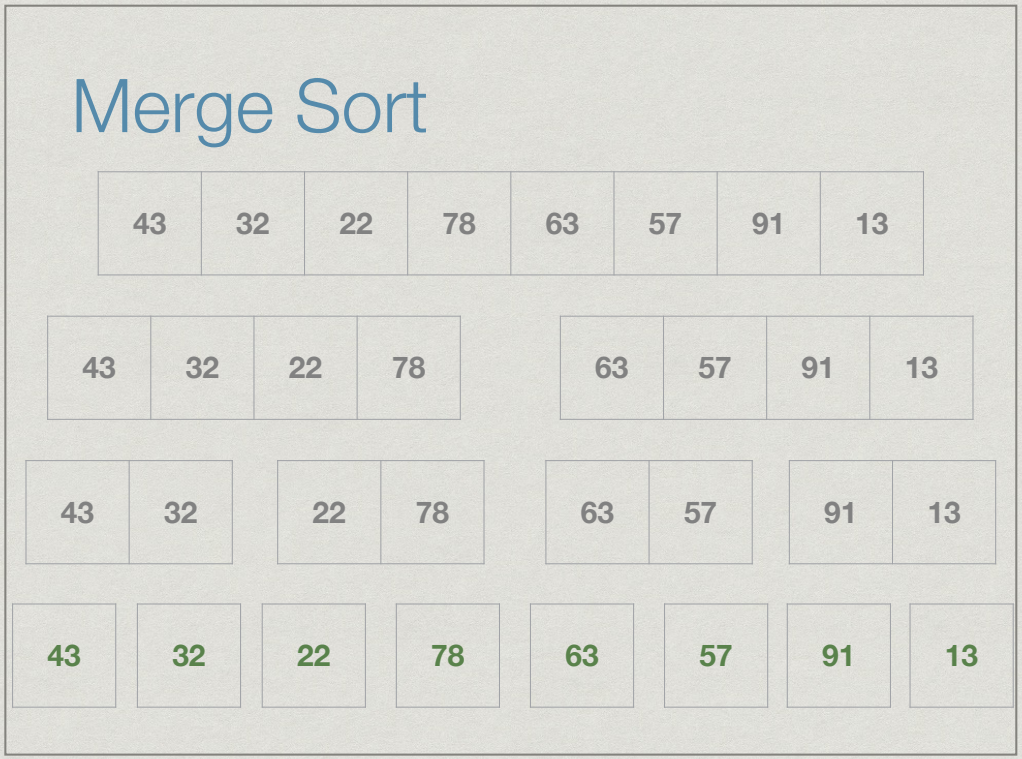
Merge Sort



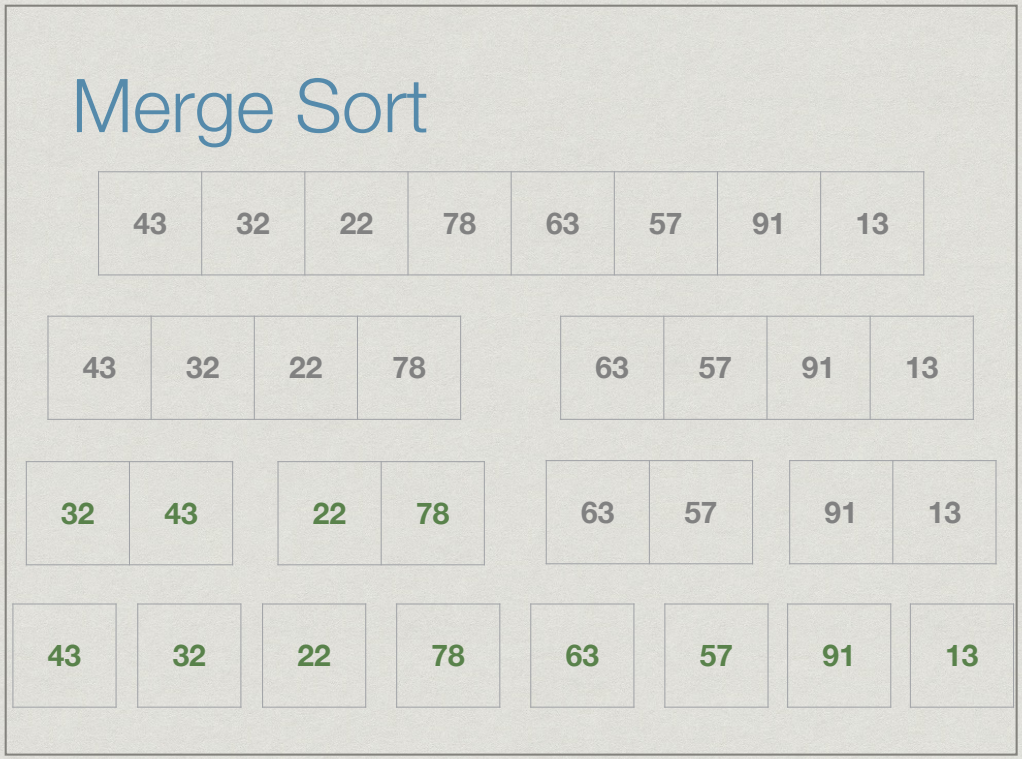
Merge Sort



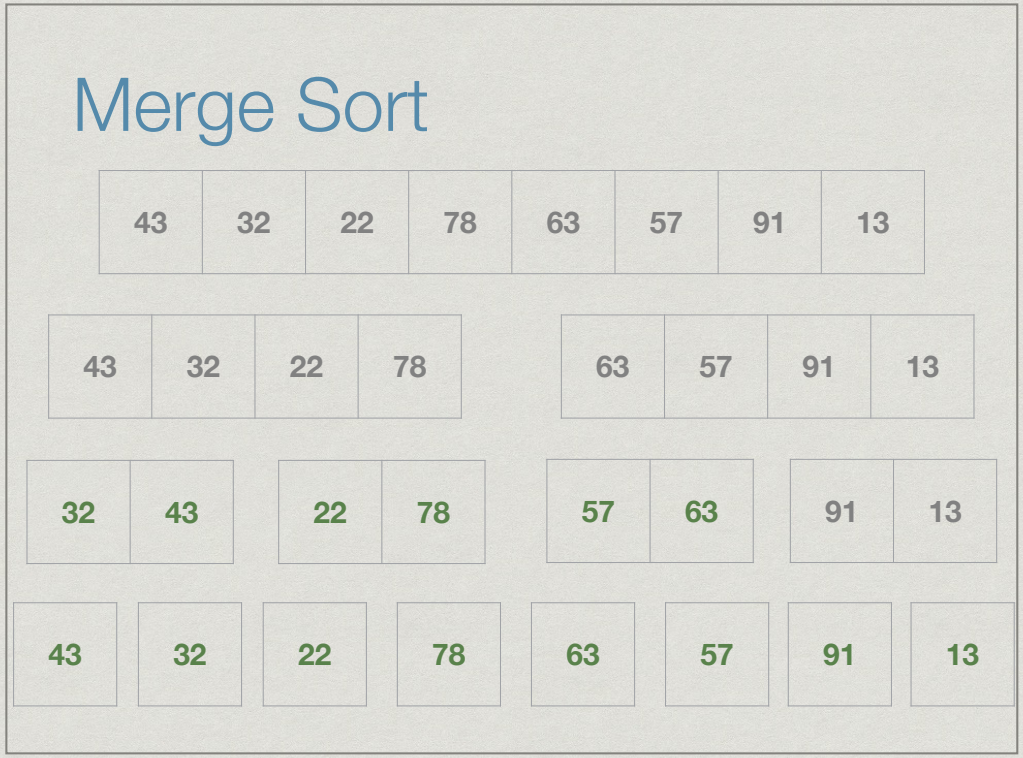
Merge Sort



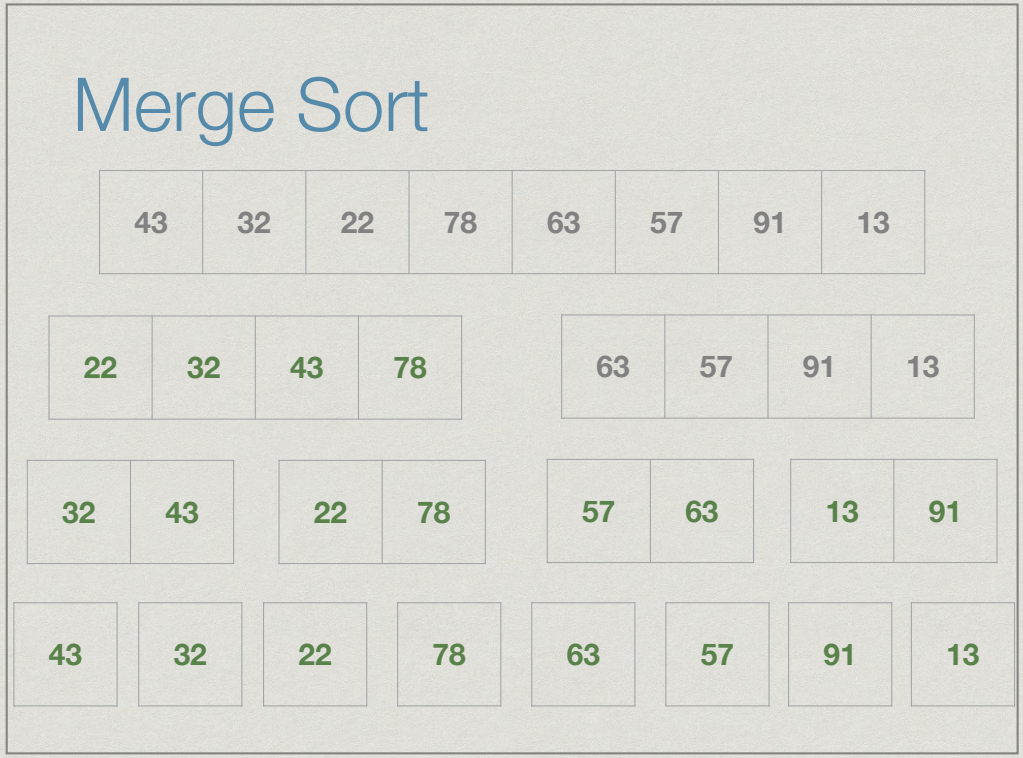
Merge Sort



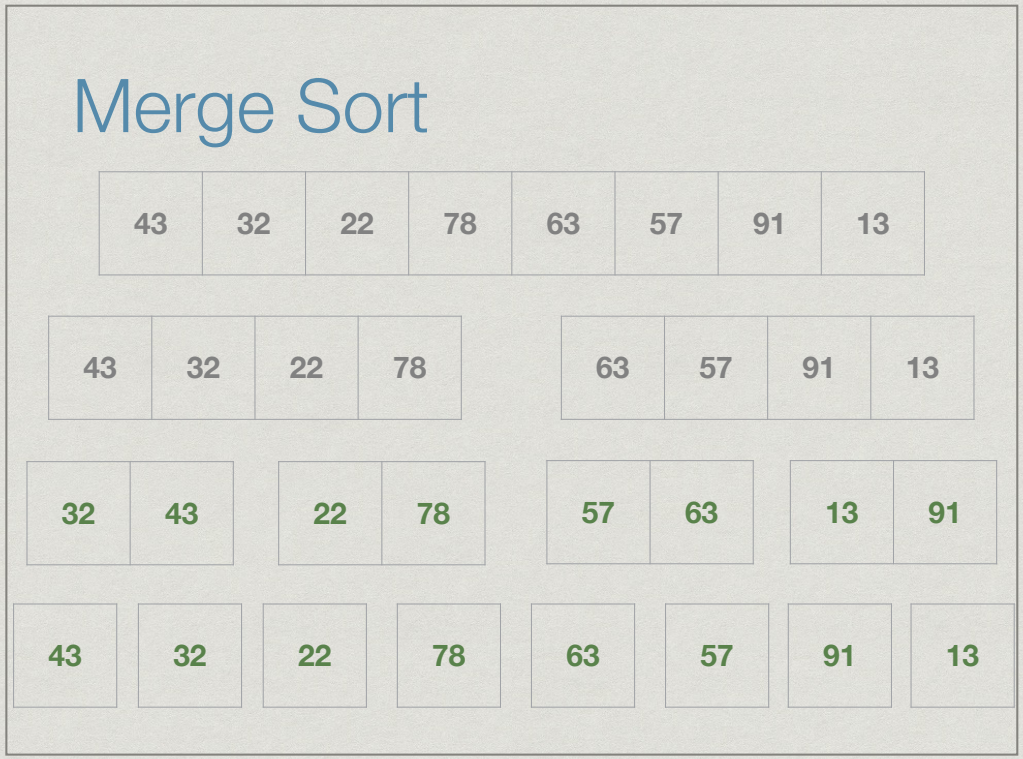
Merge Sort



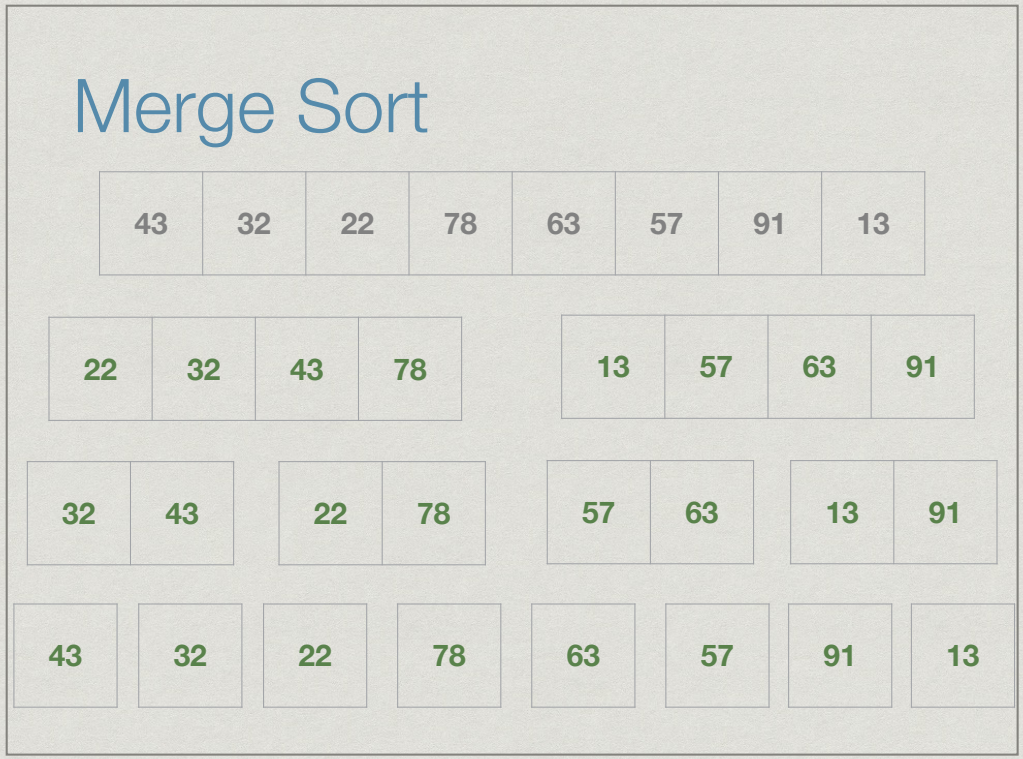
Merge Sort



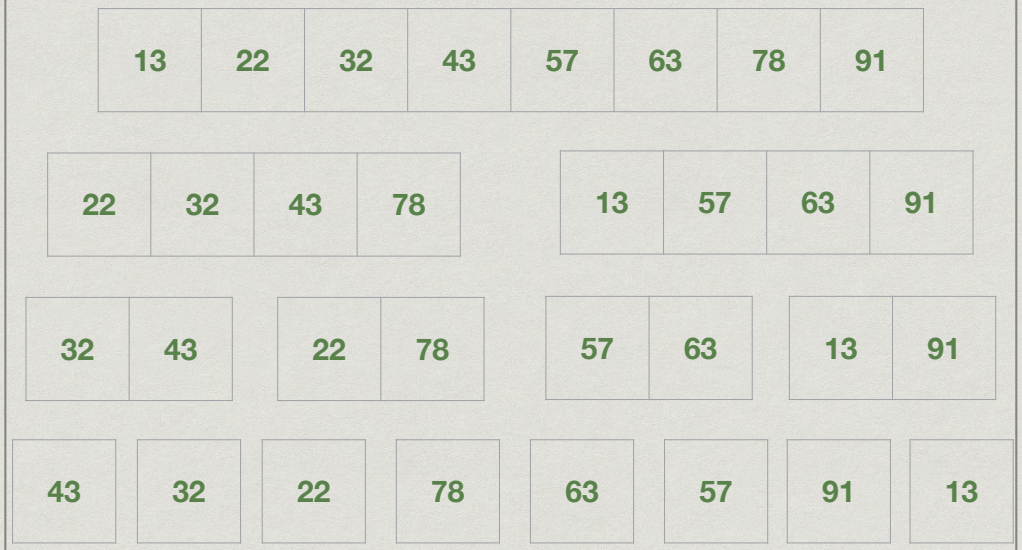
Merge Sort



Merge Sort



Merge Sort



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

Divide and conquer

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

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] = B[j]; j++; 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

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

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)
```

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 1: Move head of A into C
      if (j==n or A[i] <= B[j])
         C[k] = A[i]; i++; k++

      // Case 2: Move head of B into C
      if (i==m or A[i] > B[j])
         C[k] = B[j]; j++; k++
```

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 $m \approx n$

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

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 …

Analysis of Merge Sort …

- $t(1) = 1$

Analysis of Merge Sort …

- $t(1) = 1$
- $t(n) = 2t(n/2) + n$

  $= 2 [2t(n/4) + n/2] + n = 2^2 t(n/2^2) + 2n$

Analysis of Merge Sort …

- $t(1) = 1$
- $t(n) = 2t(n/2) + n$

Analysis of Merge Sort …

- $t(1) = 1$
- $t(n) = 2t(n/2) + n$

  $= 2 [2t(n/4) + n/2] + n = 2^2 t(n/2^2) + 2n$

  $= 2^2 [2t(n/2^3) + n/2^2] + 2n = 2^3 t(n/2^3) + 3n$

  …

Analysis of Merge Sort …

- $t(1) = 1$
- $t(n) = 2t(n/2) + n$

  $= 2\,[\,2t(n/4) + n/2\,] + n = 2^2\,t(n/2^2) + 2n$

  $= 2^2\,[\,2t(n/2^3) + n/2^2\,] + 2n = 2^3\,t(n/2^3) + 3n$

  …

  $= 2^j\,t(n/2^j) + jn$

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!

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 $t(n/2^j) = 1$

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 (10 million or 1 crore)

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

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

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!