Sorting

CS1371
Introduction to Computing
for Engineers

Sorting

Learning Objectives

Sorting Techniques

Cost of Computing

Sorting Algorithms

- Insertion Sort
- Bubble Sort
- Merge Sort
- Quick Sort

First a Motivating Example

Search for a Joe's Crab Shack in the phone book (white pages) and tell me the phone number.
Now search for 404-224-2242 and tell me name associated with that number

Linear Search

If we have an unordered list containing N elements, and we are searching for a particular value that isn't even contained in the list, how many elements do we have to search through?

Linear Search

If we have an unordered list containing N elements, and we are searching for a particular value that isn't even contained in the list, how many elements do we have to search through?

All N of them!

Binary Search

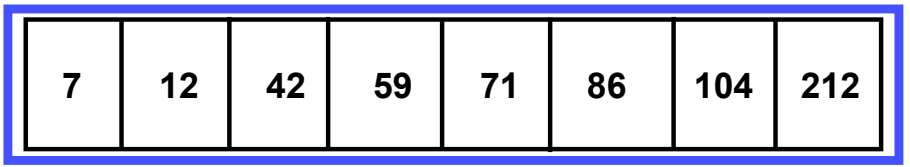
Now consider a sorted list containing N elements, and we are searching for a particular value that isn't even contained in the list, how many elements do we have to search through?

Binary Search

Now consider a sorted list containing N elements, and we are searching for a particular value that isn't even contained in the list, how many elements do we have to search through?

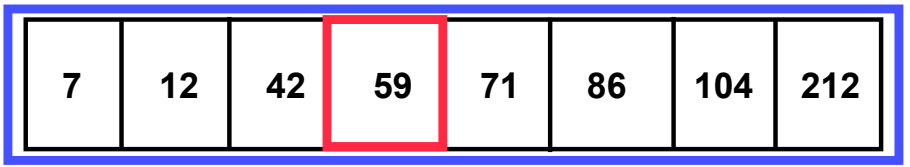
If we pick an element roughly in the middle, we can throw out half the elements after one comparison!

Searching a Sorted Array



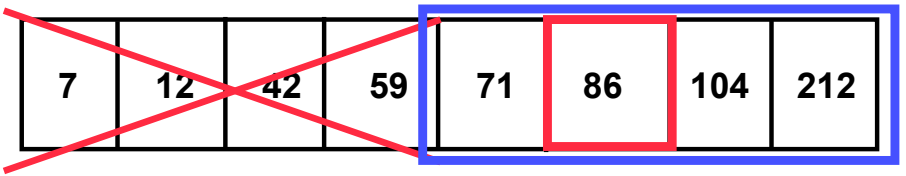
Looking for 89

Searching a Sorted Array



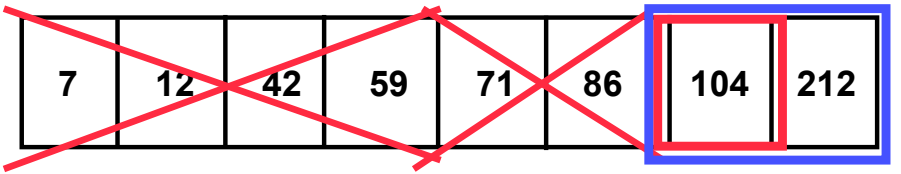
Looking for 89

Binary Search Example



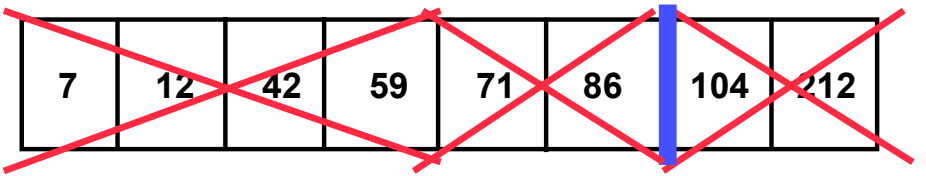
Looking for 89

Binary Search Example



Looking for 89

Binary Search Example



89 not found – 3 comparisons

3 = Log(8)

Binary Search

- An element can be found by comparing and cutting the work in half.
  - We cut work in ½ each time
  - How many times can we cut in half?

Binary Search

- An element can be found by comparing and cutting the work in half.
  - We cut work in ½ each time
  - How many times can we cut in half? Log_2N

Binary Search

- An element can be found by comparing and cutting the work in half.
  - We cut work in ½ each time
  - How many times can we cut in half? Log_2N

- Thus binary search requires Log_2N comparisons in the worst case

“Just how good is my algorithm?”

How many times this semester have you asked yourself this question?

Probably not very often, if ever. After all, we’ve been creating relatively simple programs that work on a small finite set of data. Our functions execute and return an answer with a second or two.

So why worry?

“Just how good is my algorithm?”

In the real programming world, we usually work with much much larger data sets

- **inventory of Wal-Mart™**
- **number of transactions handled by Wachovia™ on a day**

Then it becomes vitally important that we consider the relative “goodness” of the algorithm we’re considering.

In other words, as the amount of data increases, we must care how much our implementation costs us in terms of time, memory & effort!

Big O

Big O is a mathematical concept that attempts to express just how "good" or "bad" a particular thought process or algorithm is.

Please note that the concept as we explain it in this class is a greatly simplified version of explaining the complexity of an algorithm. In later courses, you may be introduced to a more formal definition of Big-O, along with its various relatives: little-O, Big-Ω, little-ω, and Big-Θ.

The Story of Big O

Big O, as noted in this class, is a measure of the "worst-case" scenario for a given algorithm taking in input of size N.

Mathematically:

> **Given an input of size N *(where N is really really really big)*, what is the asymptotic upper bounds of the algorithm in terms of time/space (how long will it take to execute/how much memory will it require)?**

O(log N): Binary Search

This holds true for any particular algorithm that eliminates a large portion of the data set each time it repeats. It's generalized into O(*log N)*.

O(N): Linear Search

Which would you choose?

O(?)

There are a few common notations for Big-O. These are categories into which almost all algorithms fit into (once we throw away all the garbage and reduce the algorithm to its purest form). These are denoted as follows:

- **O(1)**
- **O(log N)**
- **O(N)**
- **O(N^M)**
- **O(M^N)**
- **Or some combination of these.**

O(1): What Computer Scientists Dream About

O(1) doesn't mean we finish everything in one second or one CPU cycle or that our algorithm only uses one layer of the stack rather than building and building and building. We just wish that it did.

O(1) means *constant time or space*. No matter what we through at a particular algorithm, we know EXACTLY how long/how much it will take to execute the program:

> **Given an array of size N (where N is unknown), access a particular value.**
>
> **Given a list of size P (where P is unknown), retrieve the first element of the list.**

$O(N^M)$: Polynomial Order

Examples:

Traversing an N x N matrix

Performing insertion sort on a list of numbers

N times. Which means that this function performs a grand total of N^2 (more or less) calculations before completing.

O(M ^{N}): Our worst Nightmare

Exponential time. This is bad bad stuff. This means that every time we recur, we multiply the amount of work we have to do by a factor of a M. For example, the Fibonacci algorithm specifies that the fib(N) is the sum of the fib(N-1) + fib(N-2).

Already we've doubled the amount of work we have to do.

We have to calculate each of those values, don't we? So we substitute those 2 calculations with 4. And so on.

Eventually, for a value of N, we perform 2^{N} calculations.

Back to Sorting: Ordered Collections

Consider a collection values in random order.

Our goal is to sort them in ascending order.

For now we'll simplify and only talk about linear collection of numbers.

Constraints:

- **Can only compare two things at a time**
- **Don't use built-in sort()**

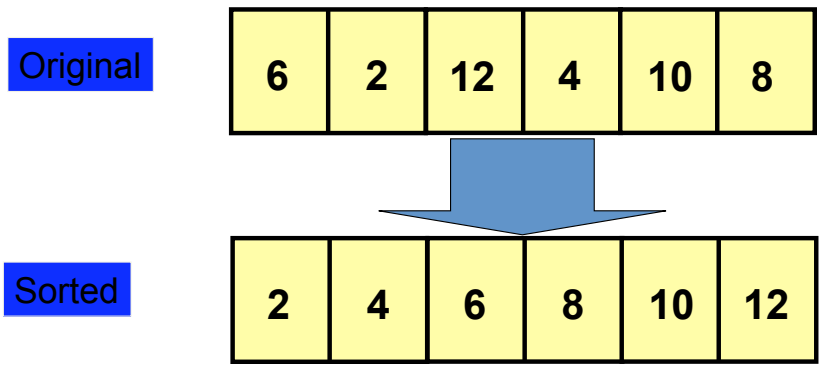
Constraint: Compare only two things at a time

I place a collection of disposable coffee cups, all the same size and covered with a lid, in front of you on a table. I ask you to sort these cups by weight without taking off the lid and looking inside, and without using any tools.

Given that you only have two hands, you can at most pick two cups up at once and compare them.

Our Task…

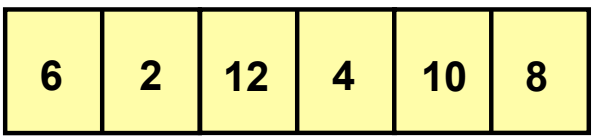
It will take in an collection of numbers and return a new collection with those numbers in numerical order.



Insertion Sort

Why don't we start with an empty result array, iterate through the original array, inserting each number one at a time into the new array?

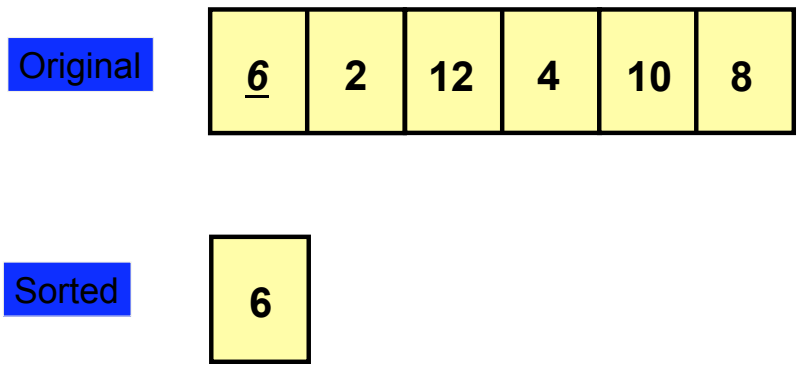
Original



Sorted

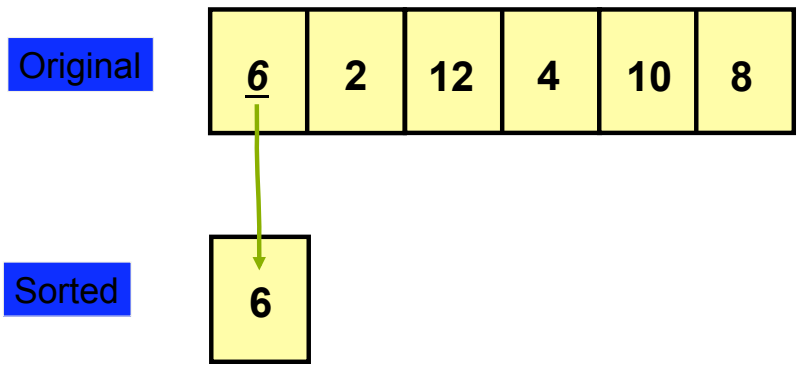
Insertion Sort

Why don't we start with an empty result array, iterate through the original array, inserting each number one at a time into the new array?



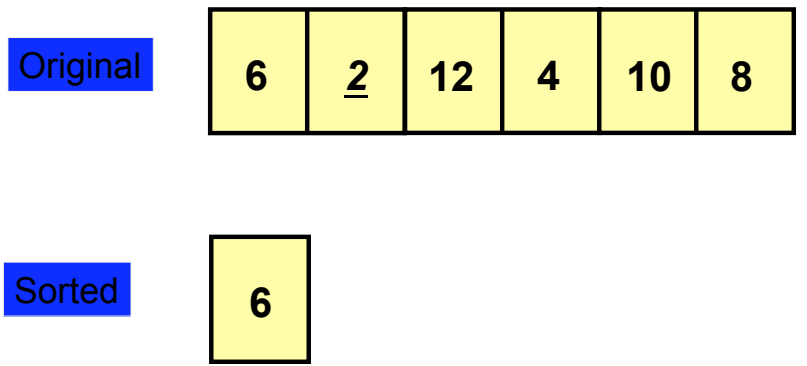
Insertion Sort

Why don't we start with an empty result array, iterate through the original array, inserting each number one at a time into the new array?



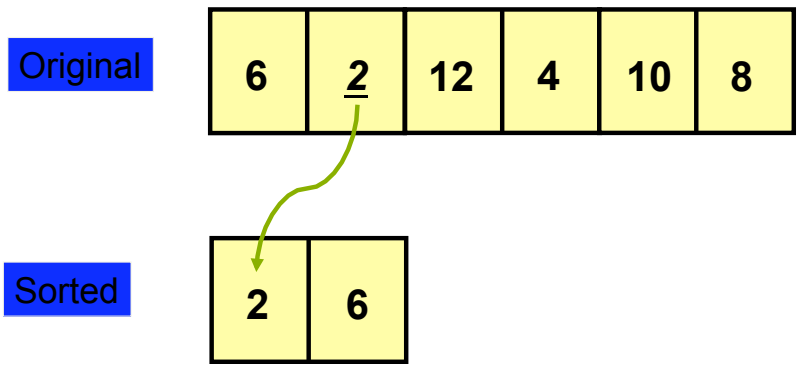
Insertion Sort

Why don't we start with an empty result array, iterate through the original array, inserting each number one at a time into the new array?



Insertion Sort

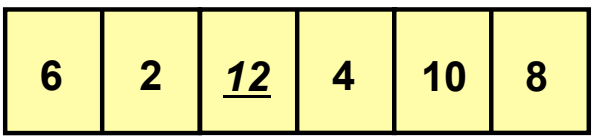
Why don't we start with an empty result array, iterate through the original array, inserting each number one at a time into the new array?



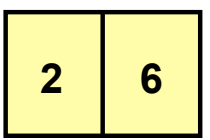
Insertion Sort

Why don't we start with an empty result array, iterate through the original array, inserting each number one at a time into the new array?

Original

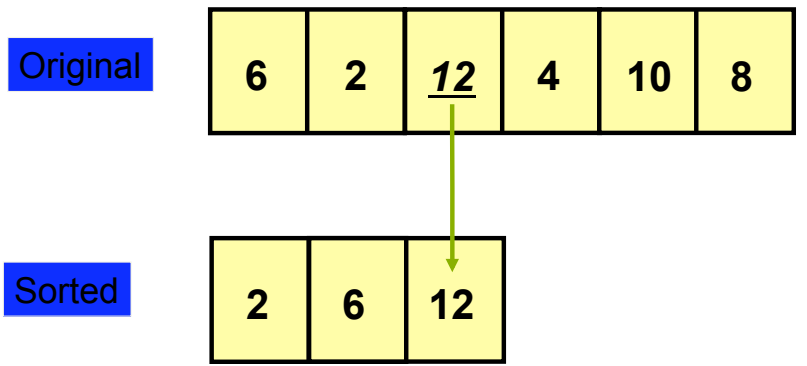


Sorted



Insertion Sort

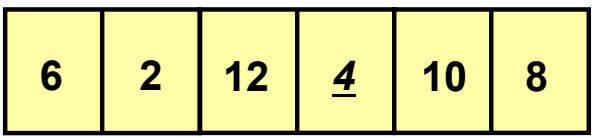
Why don't we start with an empty result array, iterate through the original array, inserting each number one at a time into the new array?



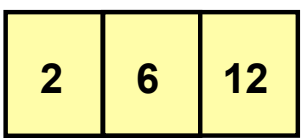
Insertion Sort

Why don't we start with an empty result array, iterate through the original array, inserting each number one at a time into the new array?

Original

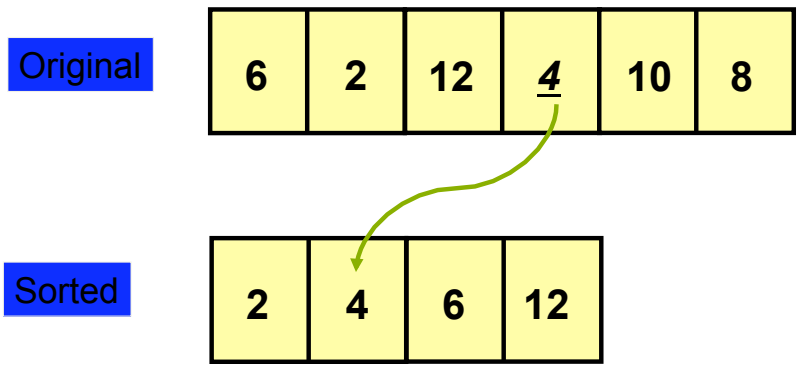


Sorted



Insertion Sort

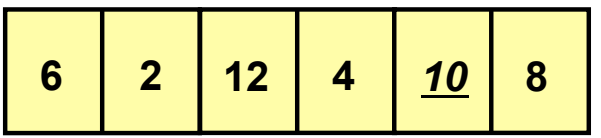
Why don't we start with an empty result array, iterate through the original array, inserting each number one at a time into the new array?



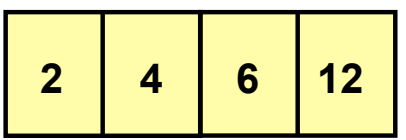
Insertion Sort

Why don't we start with an empty result array, iterate through the original array, inserting each number one at a time into the new array?

Original

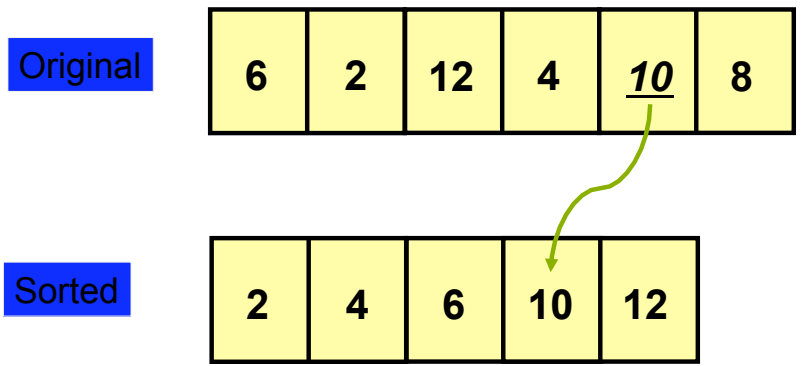


Sorted



Insertion Sort

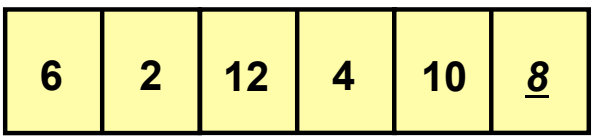
Why don't we start with an empty result array, iterate through the original array, inserting each number one at a time into the new array?



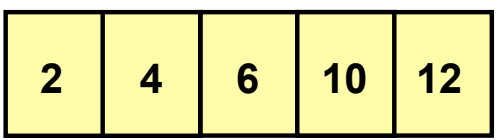
Insertion Sort

Why don't we start with an empty result array, iterate through the original array, inserting each number one at a time into the new array?

Original

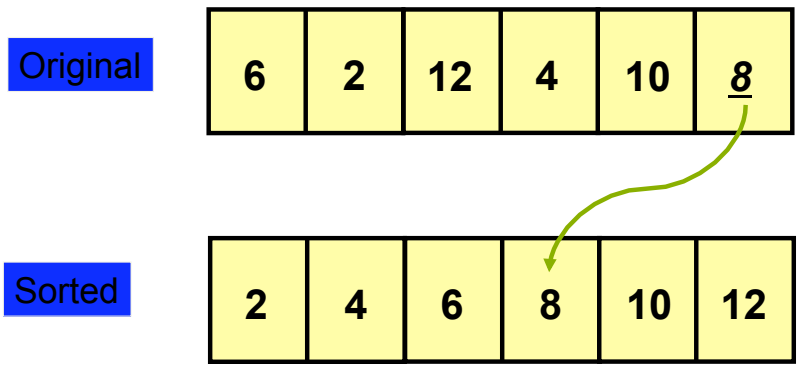


Sorted



Insertion Sort

Why don't we start with an empty result array, iterate through the original array, inserting each number one at a time into the new array?



Algorithm

For each value in original collection:

- **Compare current original value to each item in the new sorted list until…**
  - **We find a value in the new list greater than current original value OR**
  - **We reach the end of the list**

Insertion Sort Algorithm

For each value in original collection:

- **Compare current original value to each item in the new sorted list until…**
  - **If we find value in new list greater than current original value**
    - **insert original value into new list at current location**
  - **Else if we reach the end of the list**
    - **append current original value to end of new list**
  - **Else (original value is greater than current value in new list)**
    - **move on and check next item in new list**

A Matlab Implementation

```
Function sortedList = insertionsort(unsortedList)
%
%   This function sorts a column array, using
%     the insertion sort algorithm
%
sortedList = [];
i = 1;
sz = length(unsortedList);
while i <= sz % for each element to insert
    sortedList = ...
      insert(sortedList, unsortedList(i));
      % a "helper function"
    i = i + 1;
end
```

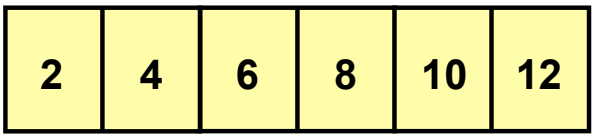
A Matlab Implementation

```
Function sortedList = insertionsort(unsortedList)
%
%    This function sorts a column array, using
%      the insertion sort algorithm
%
sortedList = [];
i = 1;
sz = length(unsortedList);
while i <= sz % for each element to insert
    sortedList = ...
      insert(sortedList, unsortedList(i));
      % a "helper function"
    i = i + 1;
end
```

Could we have used a for loop instead?

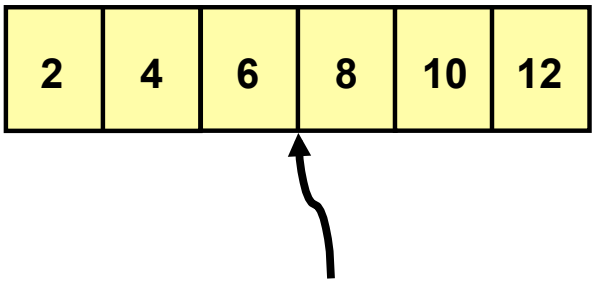
Insertion

Given the following array:



We want to think about how to insert a new number, say 7, into this array.

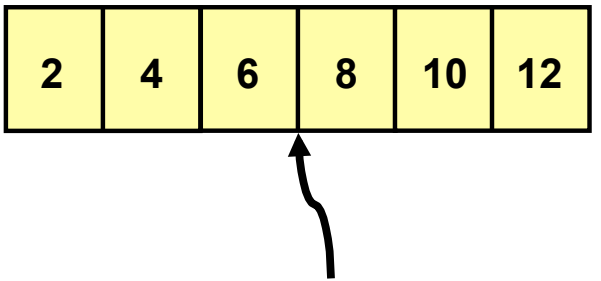
Inserting in the Middle



Say we want to insert 7 here – how would you do it?

1. **Make space for it**
2. **Insert it.**

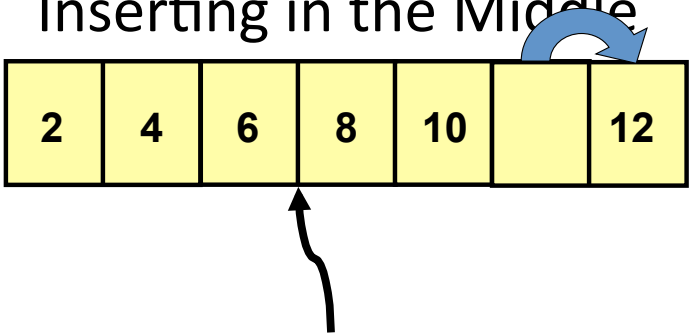
Inserting in the Middle



Say we want to insert 7 here – how would you do it?

1. **Make space for it**
2. **Insert it.**

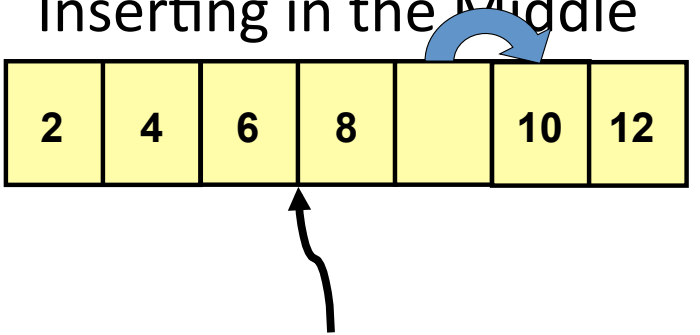
Inserting in the Middle



Say we want to insert 7 here – how would you do it?

1. **Make space for it**
2. **Insert it.**

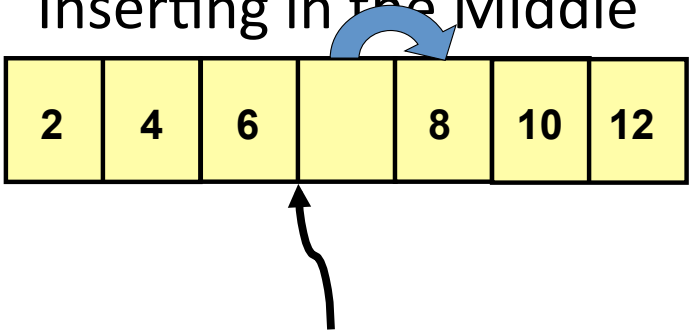
Inserting in the Middle



Say we want to insert 7 here – how would you do it?

1. **Make space for it**
2. **Insert it.**

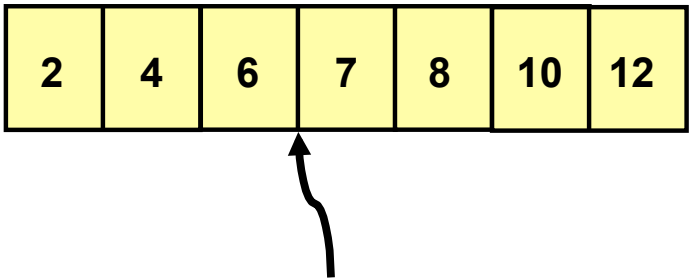
Inserting in the Middle



Say we want to insert 7 here – how would you do it?

1. **Make space for it**
2. **Insert it.**

Inserting in the Middle



Say we want to insert 7 here – how would you do it?

1. **Make space for it**
2. **Insert it.**

The Insertion Function

```
Function sortedList = insert(sortedList,v)
% this function inserts the value v
%    into the sortedList array
i = 1;
sz = length(sortedList);
done = 0;
while i <= sz
%%  code to insert in the middle
end

if done == 0
    sortedList(sz+1) = v;
end
```

Extends the length of `sortedList` by 1

Guts of the Insertion Function

```
while i <= sz
%%  code to insert in the middle
    if v < sortedList(i)
        done = 1;
        j = sz + 1;
        while j > i
          sortedList(j) = sortedList(j-1);
          j = j - 1;
        end
        sortedList(i) = v;
        i = sz + 1;
    else
        i = i + 1;
    end
end
```

Count backwards from the end of the array

Move each item forward one spot

Insert v before the number at index i

Guts of the Insertion Function

```
while i <= sz
%%  code to insert in the middle
    if v < sortedList(i)
        done = 1;
        j = sz +  ;
        while j     i
          sort      st(j) = sortedList(j-1);
          j =       1;
        end
        s           (i) = v;
```

Note: insert could also be implemented using slicing and concatenating!

```
[sortedList(1:i-1), v, sortedList(i:end)]
```

```
end
```

Congratulations!

We've just written insertion-sort!

(We could also use insertion-sort on an array that contains complex objects)

Question: how would you decide the ordering?

Analysis of Insertion Sort

- How many comparisons in the inner loop?
- How many "passes" of the outer loop?

First let's learn some more about big O analysis

Big O for Sequential Operations

O(N + M)

We add Big O's when we are performing a series of sequential and unrelated operations.

```
function dummy()
   for i = 1:N
      disp(i)
   end
   for k = 1:M
      disp(k)
   end
end
```

Big O for Nested Operations

O(M * N)

We multiply Big O's when we are "nesting" tasks.

```
function dummy2()
   for i = 1:N
      disp(i)
      for k = 1:M
           disp(k)
      end
   end
end
```

Big O…the rules

1. **Constant multipliers don't matter. There's firm mathematical basis for this, but it would take an entire lecture to explain it:**
   - **O(2 * N) is the same as O(N).**
   - **O(8 * 3^N) is the same as O(3^N)**
   - **O(13000) is the same as O(1)**

Big O…the rules

2. When combining Big O's that are using the same unknown number, only keep the Big O that grows faster:

- **O(N + 1) reduces to O(N)**
- **O(N^2 + log N) reduces to O(N^2)**
- **O(N * log N + 2^N + 370000) reduces to O(2^N)**

Big O…the rules

3. **If our calculations use two or more different unknown numbers, we cannot immediately remove them without careful consideration:**
   - **$O(N + P + Q)$ reduces to $O(N + P + Q)$**
   - **$O(N^2 + 2 * N + M * P + Q)$ reduces to $O(N^2 + M * P + Q)$**
   - **$O(R^2 \log V + R)$ reduces to $O(R^2 \log V)$**
   - **$O(M * N + N^2)$ reduces to $O(M * N + N^2)$**

Analysis of Insertion Sort

Given we N=length(unsortedList), how many "passes" of the outer loop that's responsible for keeping track of the next element to insert into the newly sorted list?

How many comparisons in the inner loop that's responsible for actually inserting the element?

Insertion Sort Complexity

Look at the relationship between the two loops:

- Inner is nested inside outer
- Number of times inner loop executes, in relation to number of times outer loop has gone around
  - 1st time around outer loop, insert into empty list
  - 2nd time around outer loop, insert into list of length 1
  - 3rd time around outer loop, insert into list of length 2
  - …Nth time around out loop, insert into list of length N-1

Insertion Sort Complexity

Can be thought of as N sequential loops…

Amount of work:

1 + 2 + 3 + 4 + 5 + …+ n-1 + n

Recall:

$$T_n = 1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2} = \frac{n^2 + n}{2}$$

Insertion Sort Complexity

Can be thought of as N sequential loops…

Amount of work:

1 + 2 + 3 + 4 + 5 + …+ n-1 + n

Recall:

$$T_n = 1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2} = \frac{n^2 + n}{2}$$

Therefore the complexity is:

$O((N/2)+(N^2/2)) = O(N^2/2) = O(N^2)$

Insertion Sort Complexity

Can be thought of as N sequential loops…

Amount of work:

1 + 2 + 3 + 4 + 5 + …+ n-1 + n

Recall:

$$T_n = 1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2} = \frac{n^2 + n}{2}$$

Therefore the complexity is:

$O((N/2)+(N^2/2)) = O(N^2/2) = O(N^2)$

$O(N^2)$ Runtime Example

Assume you are sorting 250,000,000 items

$N = 250{,}000{,}000$

$N^2 = 6.25 \times 10^{16}$

If you can do one operation per nanosecond (10^{-9} sec) *which is fast!*

It will take 6.25×10^7 seconds

So $\frac{6.25 \times 10^7}{60 \times 60 \times 24 \times 365}$

$= 1.98$ years

Questions?

Bubble Sort Algorithm

- Repeatedly step through unordered list N-1 times
  - Each time, compare each pair of adjacent items and swap them if they are in the wrong order

Slight Optimization

- The algorithm gets its name from the way largest elements "bubble" to the correct position first
  - After the first pass, the largest element in the array will be in the very last position
  - After the second pass, the $n - 1^{th}$ element will be in its final place, and so on…
- Therefore, each pass can be one comparison shorter than the previous pass

Another Possible Optimization

- Once we complete a pass in which no swaps are needed, the list is sorted

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

Bubble sort compares the numbers in pairs from left to right exchanging when necessary. Here the first number is compared to the second and as it is larger they are exchanged.

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

Now the next pair of numbers are compared. Again the 9 is the larger and so this pair is also exchanged.

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

In the third comparison, the 9 is not larger than the 12 so no exchange is made. We move on to compare the next pair without any change to the list.

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

The 12 is larger than the 11 so they are exchanged.

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 11, 12, 9, 3, 7

The twelve is greater than the 9 so they are exchanged

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 11, 12, 9, 3, 7

6, 2, 9, 11, 9, 12, 3, 7

The 12 is greater than the 3 so they are exchanged.

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 11, 12, 9, 3, 7

6, 2, 9, 11, 9, 12, 3, 7

6, 2, 9, 11, 9, 3, 12, 7

The 12 is greater than the 7 so they are exchanged.

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 11, 12, 9, 3, 7

6, 2, 9, 11, 9, 12, 3, 7

6, 2, 9, 11, 9, 3, 7, **12**

The end of the list has been reached so this is the end of the first pass. The twelve at the end of the list must be largest number in the list and so is now in the correct position. We now start a new pass from left to right.

Bubble Sort Example

First Pass

6, 2, 9, 11, 9, 3, 7, **12**

Second Pass

6, 2, 9, 11, 9, 3, 7, **12**

Bubble Sort Example

First Pass

6, 2, 9, 11, 9, 3, 7, **12**

Second Pass

6, 2, 9, 11, 9, 3, 7, **12**

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 11, 9, 3, 7, **12**

Bubble Sort Example

First Pass

6, 2, 9, 11, 9, 3, 7, **12**

Second Pass

2, 6, 9, 11, 9, 3, 7, **12**

Bubble Sort Example

First Pass

6, 2, 9, 11, 9, 3, 7, **12**

Second Pass

2, 6, 9, 11, 9, 3, 7, **12**

Bubble Sort Example

First Pass

6, 2, 9, 11, 9, 3, 7, **12**

Second Pass

2, 6, 9, 11, 9, 3, 7, **12**

Bubble Sort Example

First Pass

6, 2, 9, 11, 9, 3, 7, **12**

Second Pass

2, 6, 9, 9, 11, 3, 7, **12**

Bubble Sort Example

First Pass

6, 2, 9, 11, 9, 3, 7, **12**

Second Pass

2, 6, 9, 9, 11, 3, 7, **12**

Bubble Sort Example

First Pass

6, 2, 9, 11, 9, 3, 7, **12**

Second Pass

2, 6, 9, 9, 3, 11, 7, **12**

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 11, 7, **12**

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, 12

Second Pass 2, 6, 9, 9, 3, 7, 11, **12**

Bubble Sort Example

First Pass

6, 2, 9, 11, 9, 3, 7, **12**

Second Pass

2, 6, 9, 9, 3, 7, **11**, **12**

Notice that this time we do not have to compare the last two numbers as we know the 12 is in position. This pass therefore only required 6 comparisons.

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Bubble Sort Example

First Pass: 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass: 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass: 2, 6, 9, 9, 3, 7, **11**, **12**

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 3, 9, 7, **11**, **12**

Bubble Sort Example

First Pass: 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass: 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass: 2, 6, 9, 3, 9, 7, **11**, **12**

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 3, 7, 9, **11**, **12**

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Notice that this time we do not have to compare the last three numbers. This pass therefore only required 5 comparisons.

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Fourth Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Each pass requires fewer comparisons. This time only 4 needed.

Bubble Sort Example

First Pass: 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass: 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass: 2, 6, 9, 3, 7, **9**, **11**, **12**

Fourth Pass: 2, 6, 9, 3, 7, **9**, **11**, **12**

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Fourth Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Fourth Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Fourth Pass 2, 6, 3, 9, 7, **9**, **11**, **12**

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Fourth Pass 2, 6, 3, 9, 7, **9**, **11**, **12**

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Fourth Pass 2, 6, 3, 7, 9, **9**, **11**, **12**

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Fourth Pass 2, 6, 3, 7, **9**, **9**, **11**, **12**

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Fourth Pass 2, 6, 3, 7, **9**, **9**, **11**, **12**

Fifth Pass 2, 6, 3, 7, **9**, **9**, **11**, 12

How many comparisons do we need to do this time around?

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Fourth Pass 2, 6, 3, 7, **9**, **9**, **11**, **12**

Fifth Pass 2, 6, 3, 7, **9**, **9**, **11**, 12

Bubble Sort Example

First Pass: 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass: 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass: 2, 6, 9, 3, 7, **9**, **11**, **12**

Fourth Pass: 2, 6, 3, 7, **9**, **9**, **11**, **12**

Fifth Pass: 2, 6, 3, 7, **9**, **9**, **11**, 12

Bubble Sort Example

First Pass: 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass: 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass: 2, 6, 9, 3, 7, **9**, **11**, **12**

Fourth Pass: 2, 6, 3, 7, **9**, **9**, **11**, **12**

Fifth Pass: 2, 3, 6, 7, **9**, **9**, **11**, 12

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Fourth Pass 2, 6, 3, 7, **9**, **9**, **11**, **12**

Fifth Pass 2, 3, 6, 7, **9**, **9**, **11**, 12

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Fourth Pass 2, 6, 3, 7, **9**, **9**, **11**, **12**

Fifth Pass 2, 3, 6, **7**, **9**, **9**, **11**, 12

The list is now sorted but the algorithm does not know this until it completes a pass with no exchanges.

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Fourth Pass 2, 6, 3, 7, **9**, **9**, **11**, **12**

Fifth Pass 2, 3, 6, **7**, **9**, **9**, **11**, 12

Sixth Pass 2, 3, 6, **7**, **9**, **9**, **11**, 12

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Fourth Pass 2, 6, 3, 7, **9**, **9**, **11**, **12**

Fifth Pass 2, 3, 6, **7**, **9**, **9**, **11**, 12

Sixth Pass 2, 3, 6, **7**, **9**, **9**, **11**, 12

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, **12**

Second Pass 2, 6, 9, 9, 3, 7, **11**, **12**

Third Pass 2, 6, 9, 3, 7, **9**, **11**, **12**

Fourth Pass 2, 6, 3, 7, **9**, **9**, **11**, **12**

Fifth Pass

This pass no exchanges are made so the algorithm knows the list is sorted. It can therefore save time by not doing the final pass. With other lists this check could save much more work.

Sixth Pass 2, 3, **6**, **7**, **9**, **9**, **11**, **12**

Bubble Sort Questions

1. Which number is definitely in its correct position at the end of the first pass?

Answer: The last number must be the largest.

2. How does the number of comparisons required change as the pass number increases?

Answer: Each pass requires one fewer comparison than the last.

A Matlab Implementation

Simplified version (doesn't check for pass with no exchanges)

```
function bubblesort()
   global b % unsorted list

   N = length(b);
   right = N-1;
   for in = 1:(N-1)
       for jn = 1:right
           if b(jn) > b(jn+1)
               tmp = b(jn);
               b(jn) = b(jn+1);
               b(jn+1) = tmp;
           end
       end
       right = right - 1;
   end
```

A Matlab Implementation

Simplified version (doesn't check for pass with no exchanges)

```
function bubblesort()
    global b

    N = length(b);
    right = N-1;
    for in = 1:(N-1)
        for jn = 1:right
            if b(jn) > b(jn+1)
                tmp = b(jn);
                b(jn) = b(jn+1);
                b(jn+1) = tmp;
            end
        end
        right = right - 1;
    end
```

Use global variable instead of parameter so we can sort "in-place"

A Matlab Implementation

Simplified version (doesn't check for pass with no exchanges)

```
function bubblesort()
    global b

    N = length(b);
    right = N-1;
    for in = 1:(N-1)
        for jn = 1:right
            if b(jn) > b(jn+1)
                tmp = b(jn);
                b(jn) = b(jn+1);
                b(jn+1) = tmp;
            end
        end
        right = right - 1;
    end
```

Outer loop executes N-1 times (which guarantees the result will be sorted); keeps track of "pass number"

A Matlab Implementation

Simplified version (doesn't check for pass with no exchanges)

```
function bubblesort()
   global b

   N = length(b);
   right = N-1;
   for in = 1:(N-1)
       for jn = 1:right
           if b(jn) > b(jn+1)
               tmp = b(jn);
               b(jn) = b(jn+1);
               b(jn+1) = tmp;
           end
       end
       right = right - 1;
   end
```

Number of time inner loop executes depends on which iteration of the outer loop we are on

A Matlab Implementation

Simplified version (doesn't check for pass with no exchanges)

```
function bubblesort()
   global b

   N = length(b);
   right = N-1;
   for in = 1:(N-1)
       for jn = 1:right
           if b(jn) > b(jn+1)
               tmp = b(jn);
               b(jn) = b(jn+1);
               b(jn+1) = tmp;
           end
       end
       right = right - 1;
   end
```

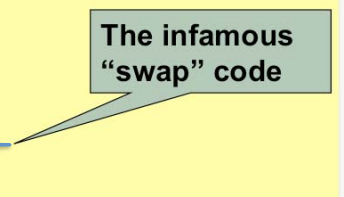
Since each pass of outer loop puts largest element in place, we reduce the number of passes the inner loop has to complete by one

A Matlab Implementation

Simplified version (doesn't check for pass with no exchanges)

```
function bubblesort()
   global b

   N = length(b);
   right = N-1;
   for in = 1:(N-1)
       for jn = 1:right
           if b(jn) > b(jn+1)
               tmp = b(jn);
               b(jn) = b(jn+1);
               b(jn+1) = tmp;
           end
       end
       right = right - 1;
   end
```



Analysis of Bubble Sort

- How many comparisons in the inner loop?
- How many “passes” of the outer loop?

Analysis of Bubble Sort

- How many comparisons in the inner loop?
  - 1st time around outer loop, N-1 comparisons required
  - 2nd time around outer loop, N-2 comparisons required
  - 3rd time around outer loop, N-3 comparisons required
  - …
  - (N-1)th time around out loop, N-(N-1) = 1 comparison required
- How many "passes" of the outer loop? N-1

Bubble Sort Complexity

Can be thought of as N-1 sequential loops…

Amount of work:

(N-1) + (N-2) + (N-3) + … + 1

Recall:

$$T_n = 1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2} = \frac{n^2 + n}{2}$$

Therefore the complexity is:

O((N-1)(N-1+1)/2) = O((N-1)(N/2)

=O((N^2/2) – (N/2)) = O(N^2)

Questions?

Better Approach: Divide and Conquer

Divide and Conquer cuts the problem in half each time, but uses the result of both halves:

- **cut the problem in half until the problem is trivial**
- **solve for both halves**
- **combine the solutions**

Merge Sort

Merge sort works on the premise that combining two sorted arrays is fairly fast.

Consider merging:

[1 2 5 19 27] **[3 4 6 20 30]**

Merging

Merge sort works on the premise that combining two sorted lists is fairly fast.

Consider merging:

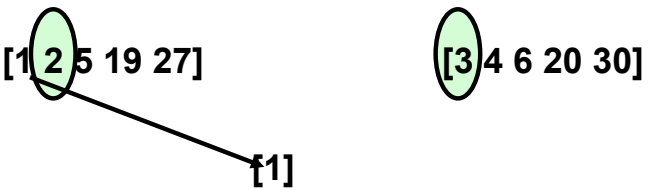
[1 2 5 19 27] **[3 4 6 20 30]**

We can compare the first item in each array, which we know to be the smallest from each. If we compare the two, we now know which is the smallest of BOTH lists, and we can save that in the result array.

Merging

Merge sort works on the premise that combining two sorted lists is fairly fast.

Consider merging:

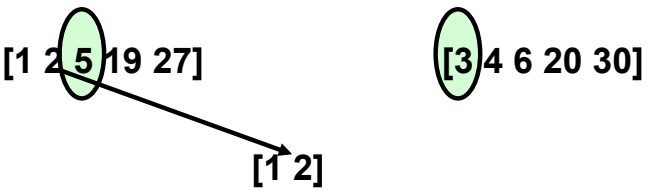


And continue across each array until one is empty.

Merging

Merge sort works on the premise that combining two sorted lists is fairly fast.

Consider merging:

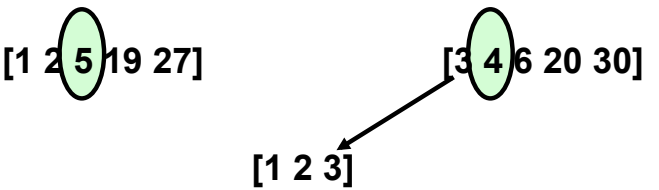


And continue across each array until one is empty.

Merging

Merge sort works on the premise that combining two sorted lists is fairly fast.

Consider merging:

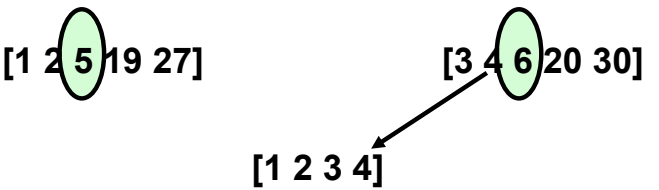


And continue across each array until one is empty.

Merging

Merge sort works on the premise that combining two sorted lists is fairly fast.

Consider merging:

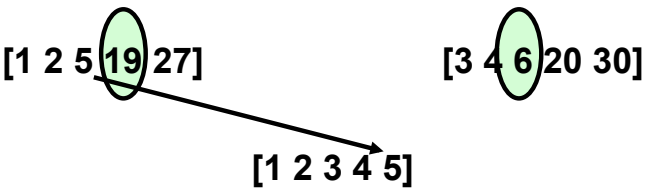


And continue across each array until one is empty.

Merging

Merge sort works on the premise that combining two sorted lists is fairly fast.

Consider merging:

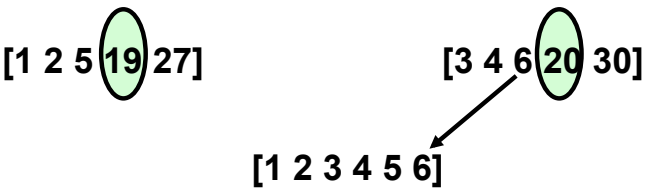


And continue across each array until one is empty.

Merging

Merge sort works on the premise that combining two sorted lists is fairly fast.

Consider merging:

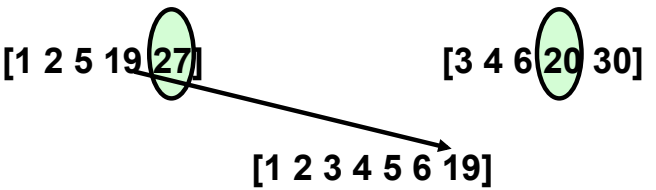


And continue across each array until one is empty.

Merging

Merge sort works on the premise that combining two sorted lists is fairly fast.

Consider merging:

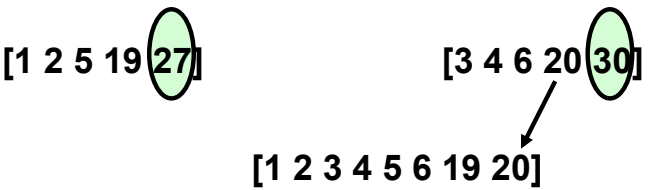


And continue across each array until one is empty.

Merging

Merge sort works on the premise that combining two sorted lists is fairly fast.

Consider merging:

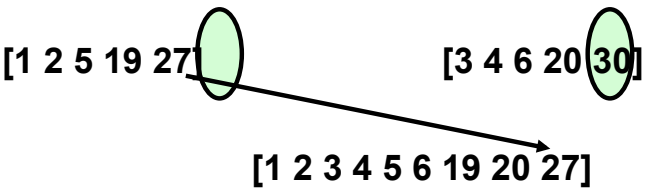


And continue across each array until one is empty.

Merging

Merge sort works on the premise that combining two sorted lists is fairly fast.

Consider merging:

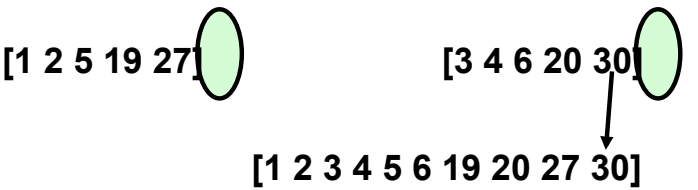


And continue across each array until one is empty.

Merging

Merge sort works on the premise that combining two sorted lists is fairly fast.

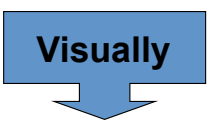
Consider merging:



Then, just copy the rest of the surviving array.

mergesort algorithm

1. **The function receives as input an array of numbers**
2. **If the array contains more than one element, divide the array into two arrays of equal size (plus or minus an element).**
3. **Perform mergesort on the first array**
4. **Perform mergesort on the second array**
5. ***Merge* the results of the two recursive calls together**

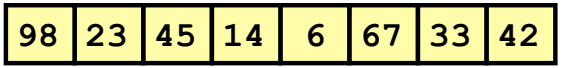


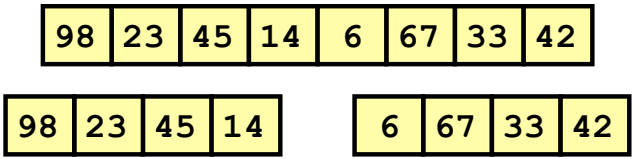
The code

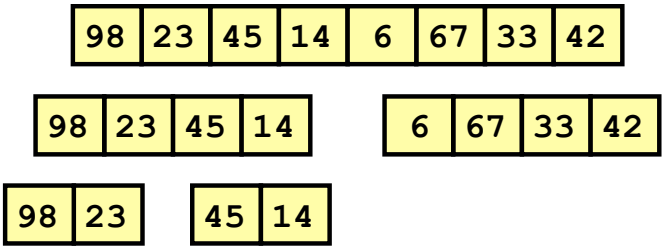
```
function a = mergesort(a)
%
%    This function sorts a column array, using the
%           merge sort algorithm
%
     sz = length(a);
     if sz > 1
          szb2 = floor(sz/2);
          first = a(1:szb2);
          second = a(szb2+1:sz);
          first = mergesort(first);
          second = mergesort(second);
          b = domerge(first, second);
     end
```

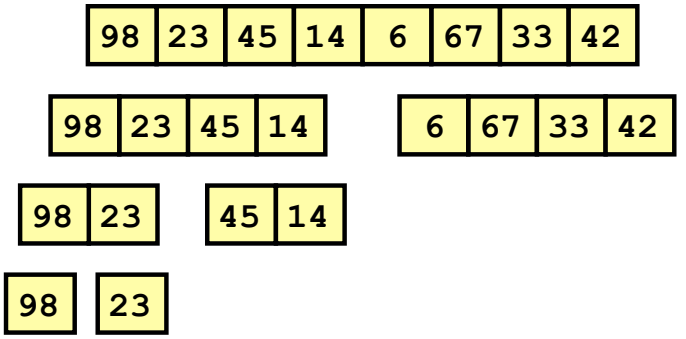
Arbitrarily divides collection in half (down the middle)

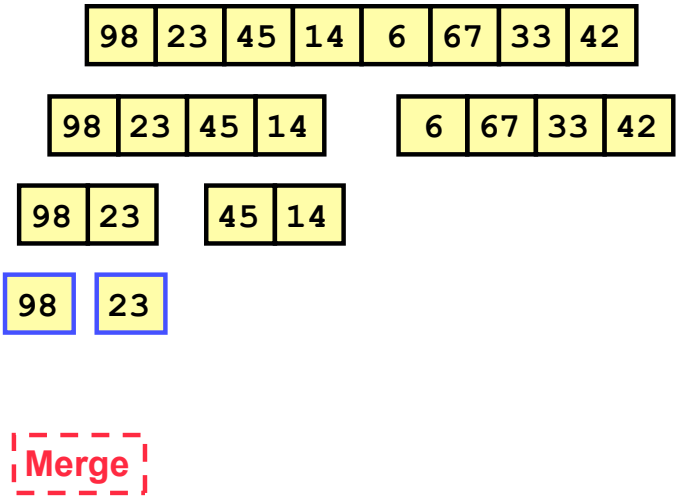
The "hard work" occurs at the end of each recursive call; sorted sub-lists are merged

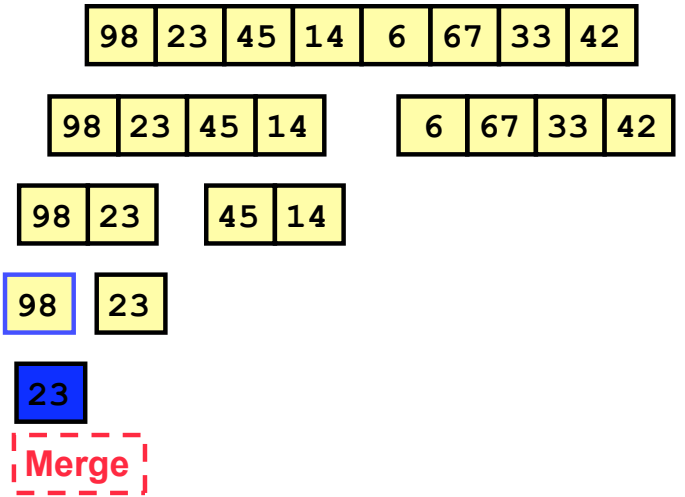

98
23
45
14
6
67
33
42

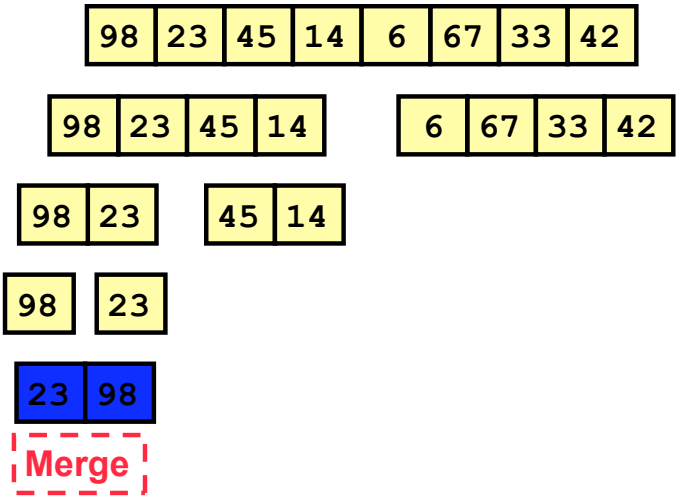

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42

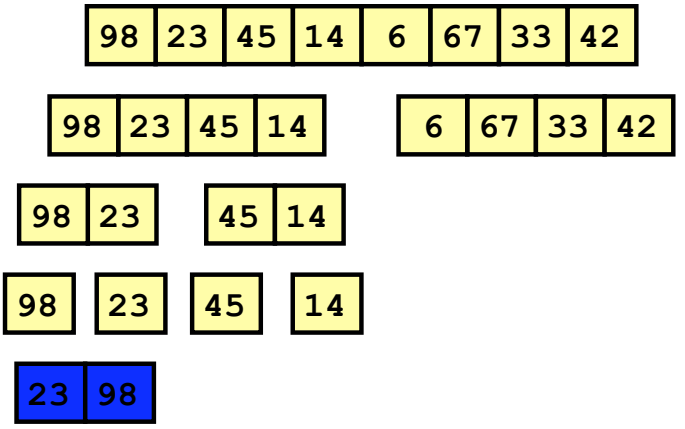

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14

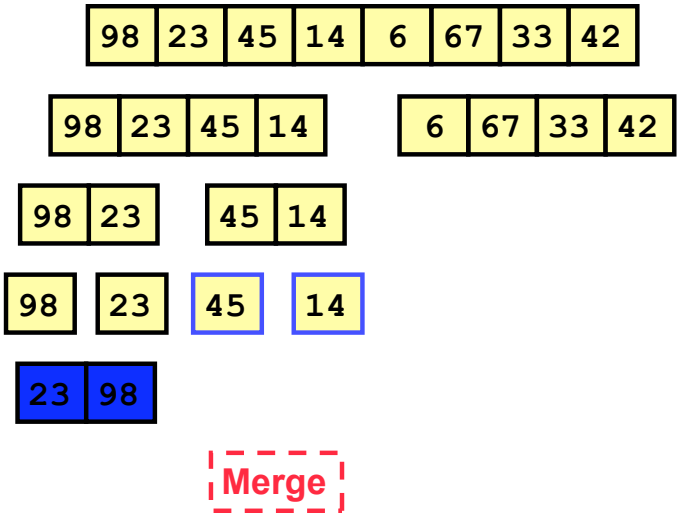

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23

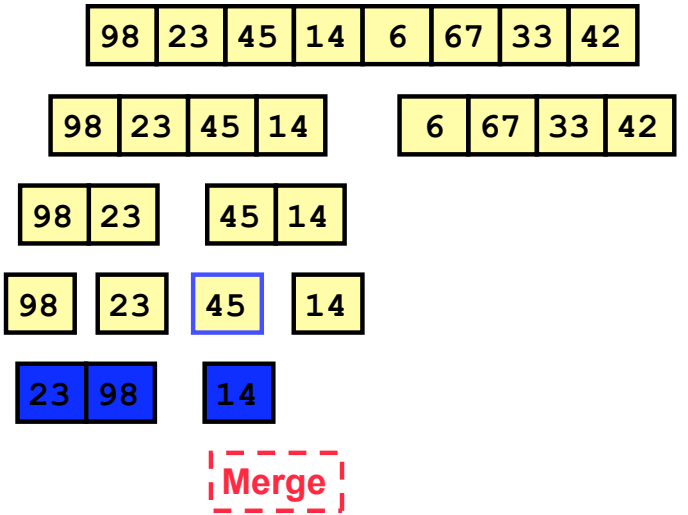

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
Merge

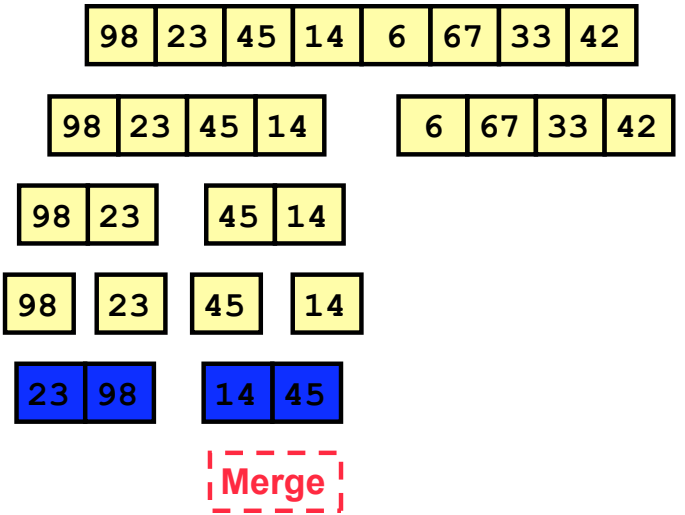

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23
Merge

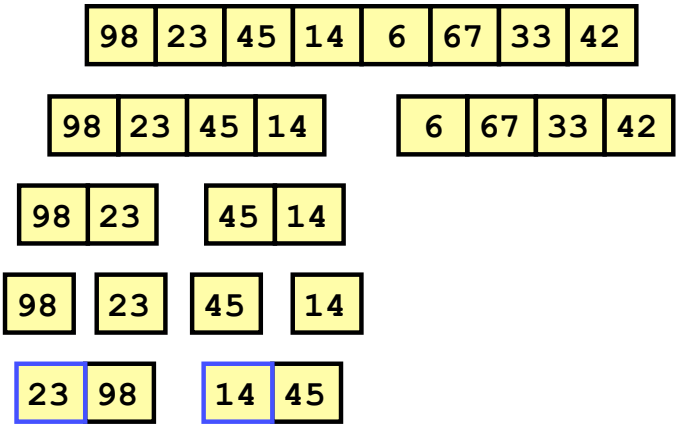

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23 98
Merge

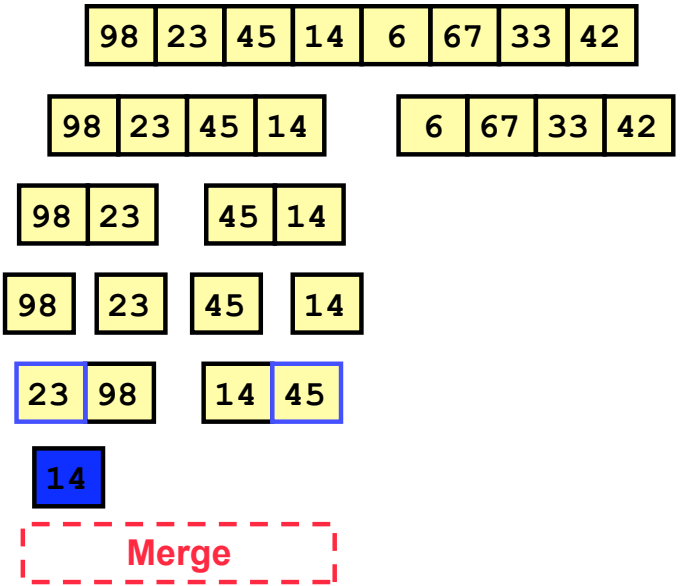

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98

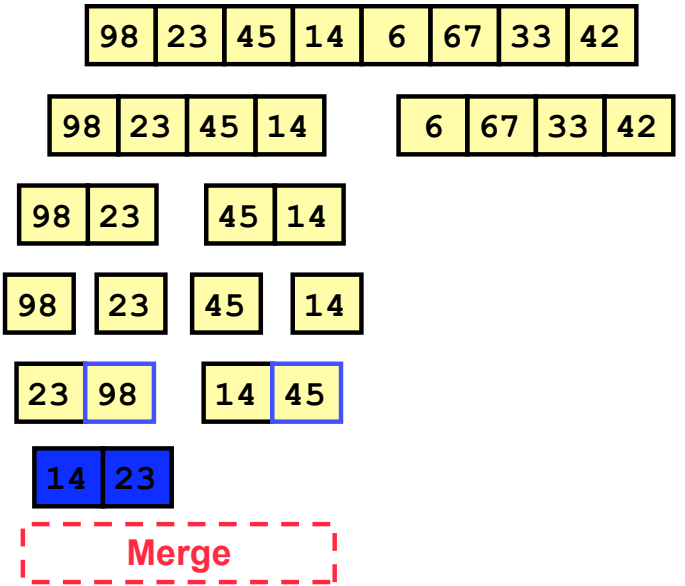

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
Merge

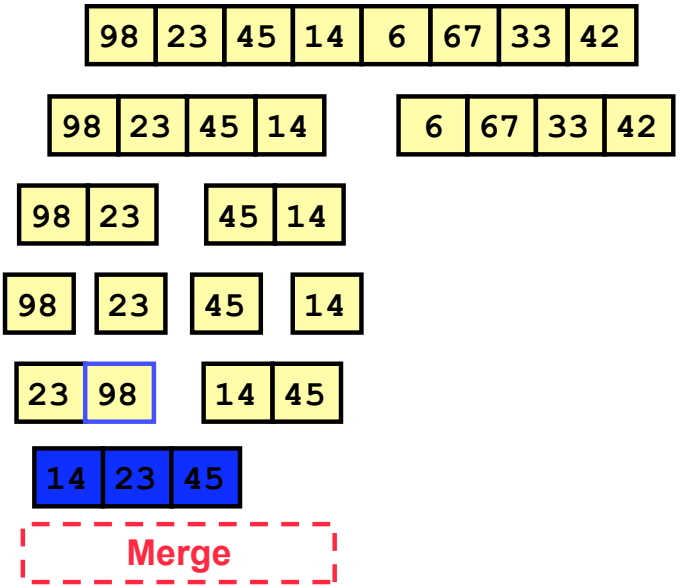

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14
Merge

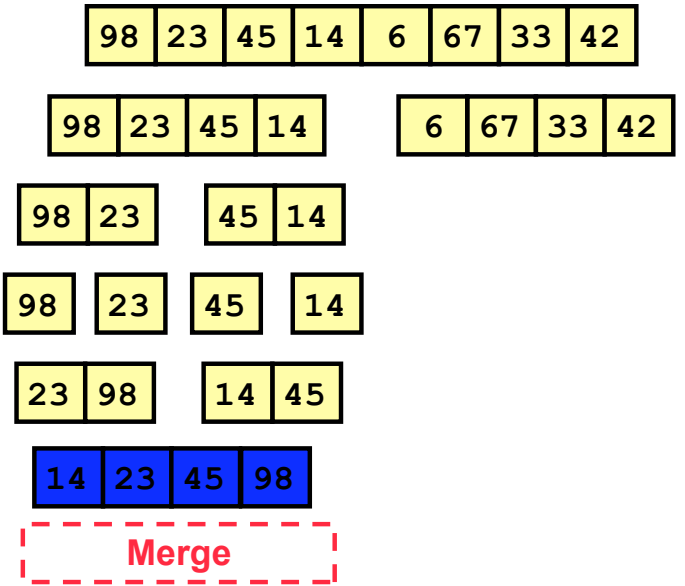

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

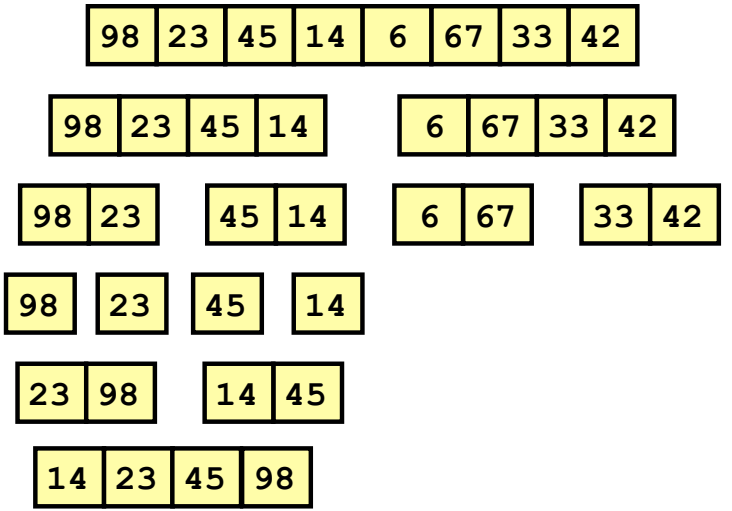

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
Merge

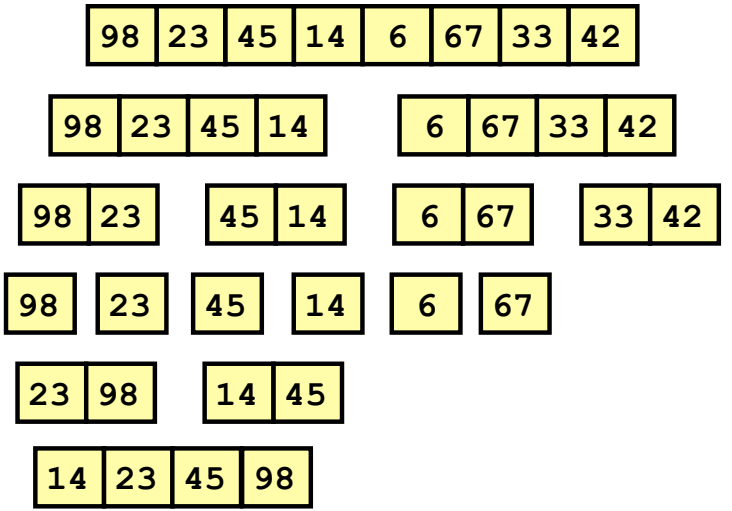

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14
Merge

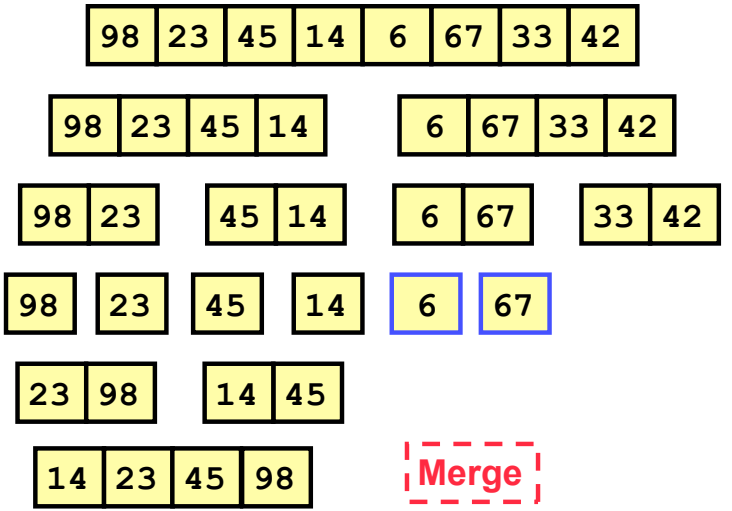

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23
Merge

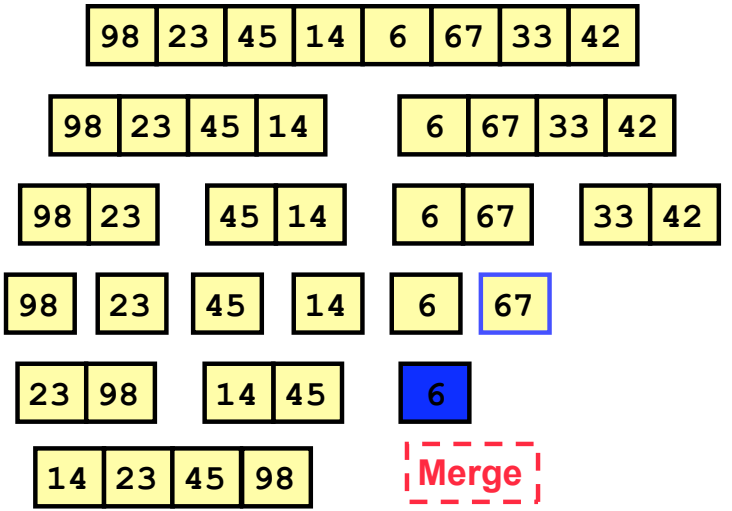

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45
Merge

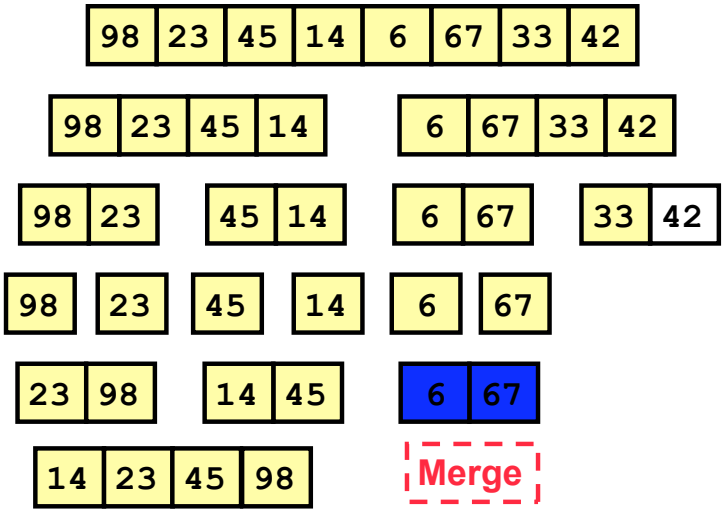

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45 98
Merge

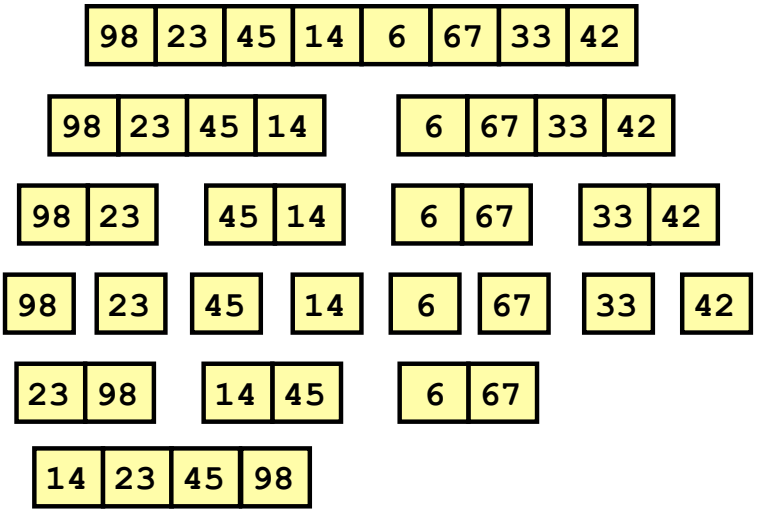

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
23 98
14 45
14 23 45 98

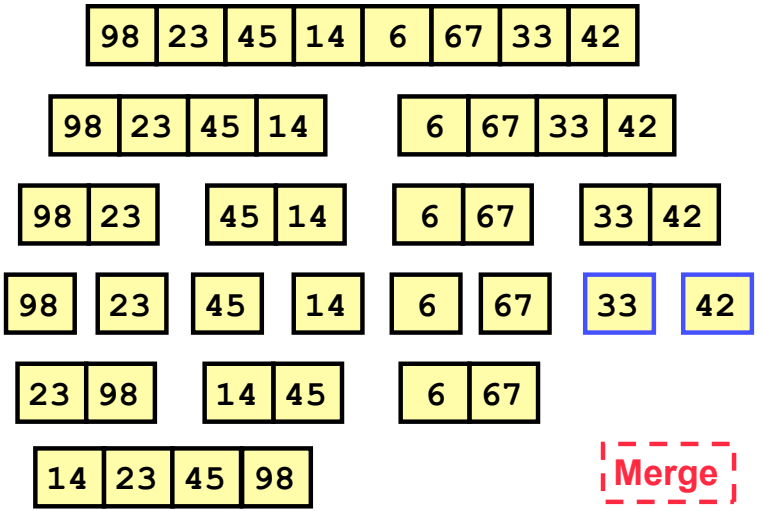

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98

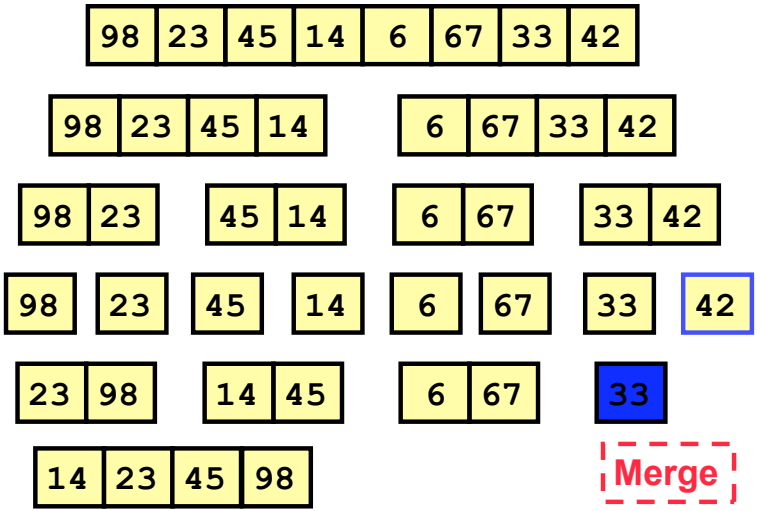

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98
Merge

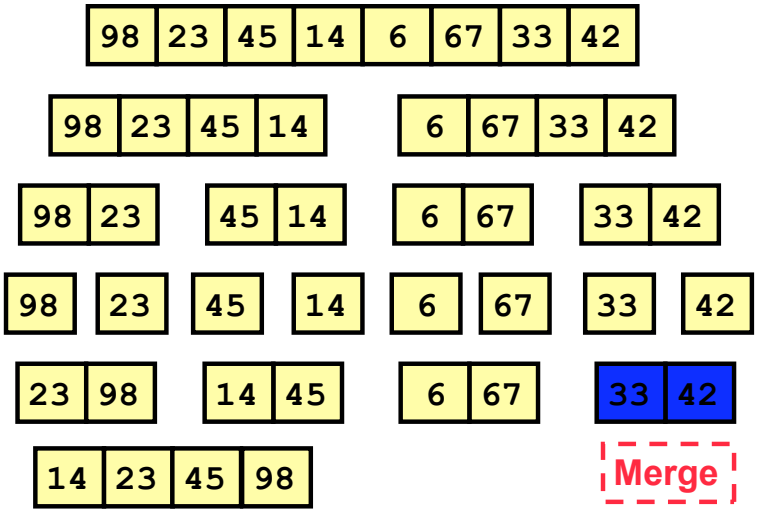

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6
14 23 45 98
Merge

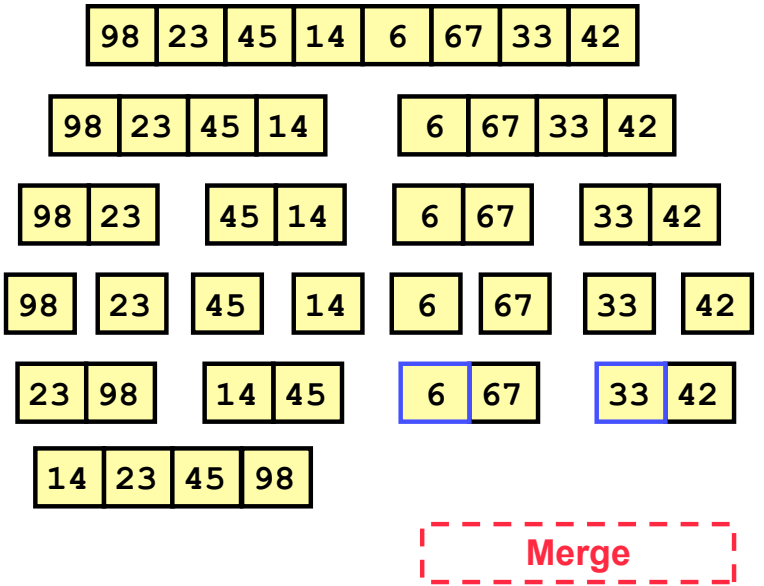

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
6 67
14 23 45 98
Merge

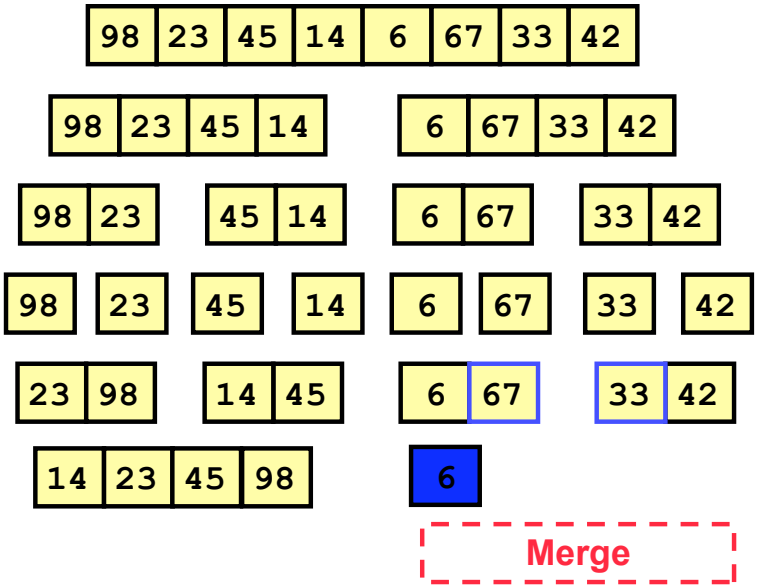

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98

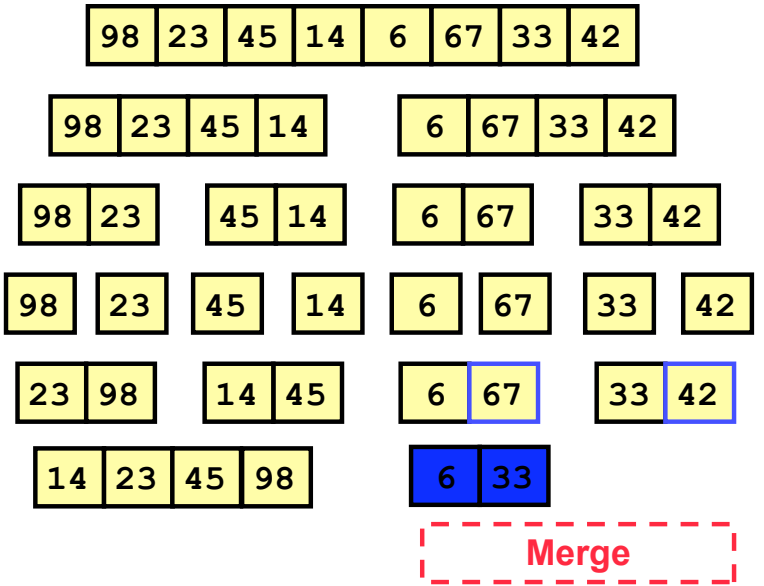

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
14 23 45 98
Merge

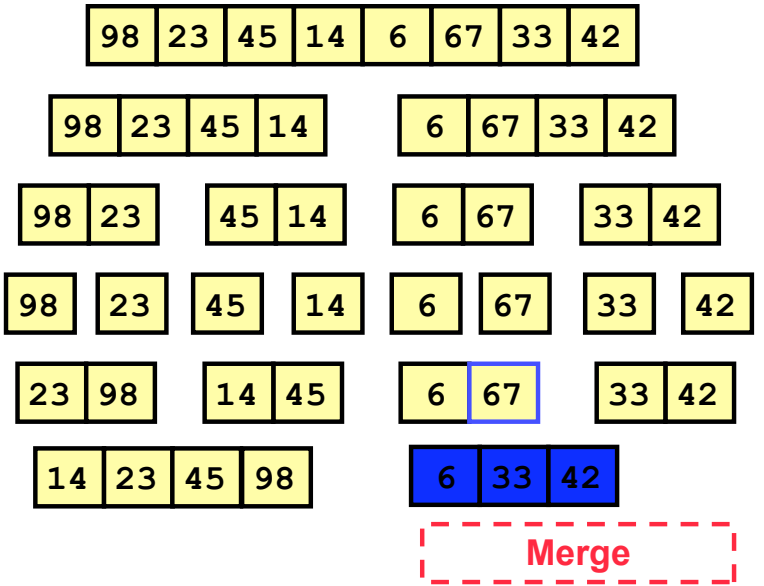

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33
14 23 45 98
Merge

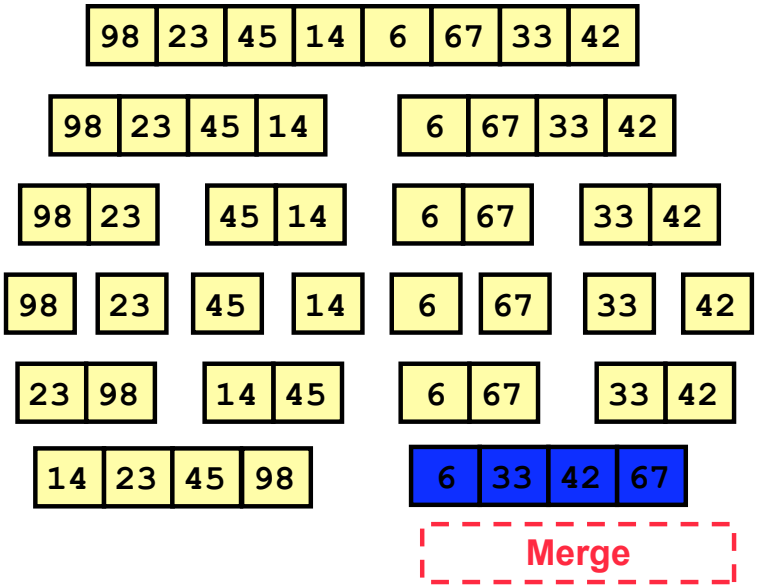

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge

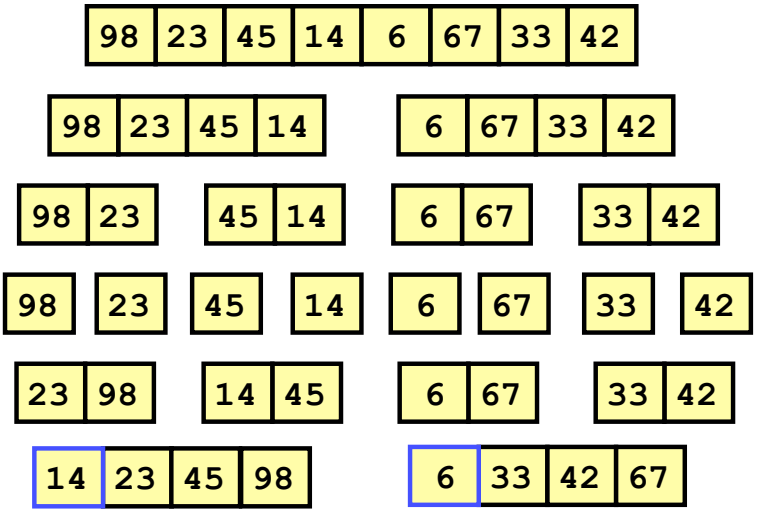

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
Merge

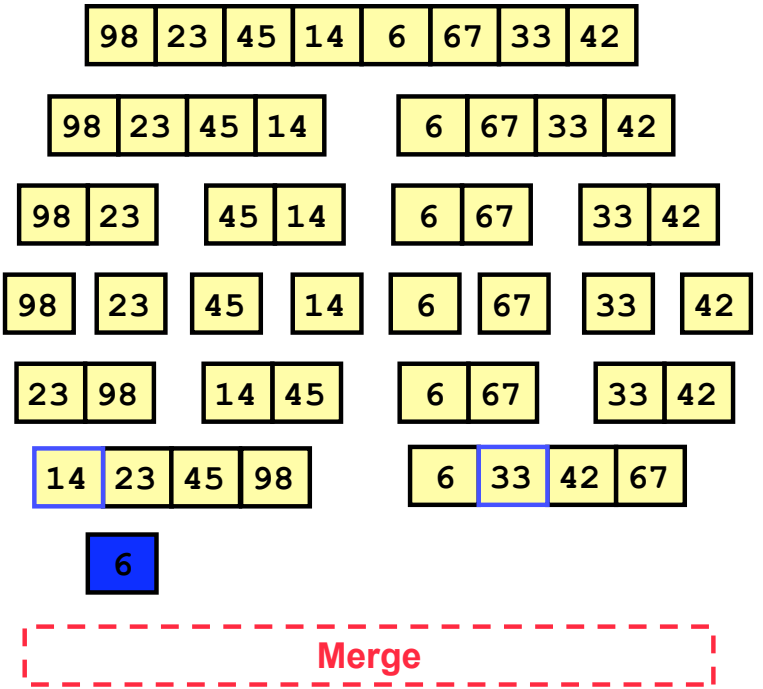

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6
Merge

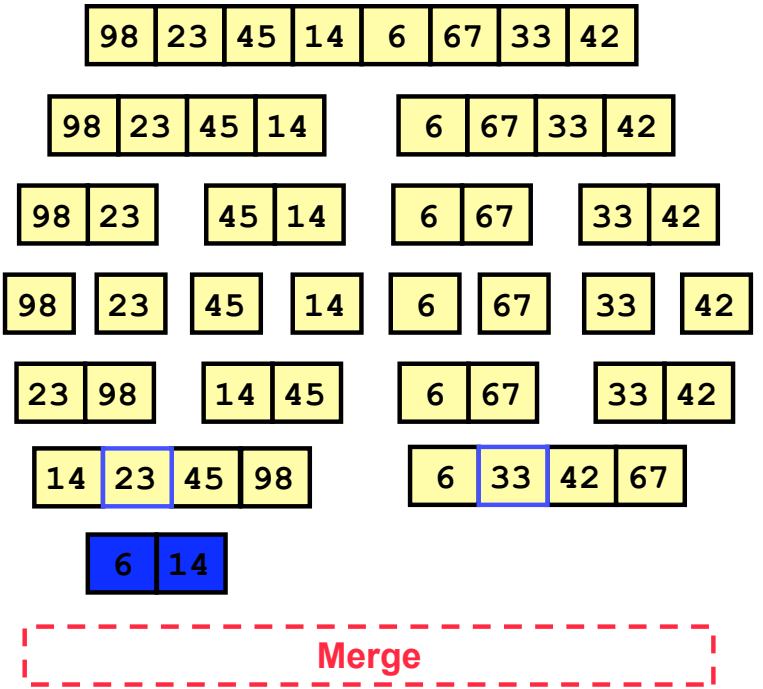

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33
Merge

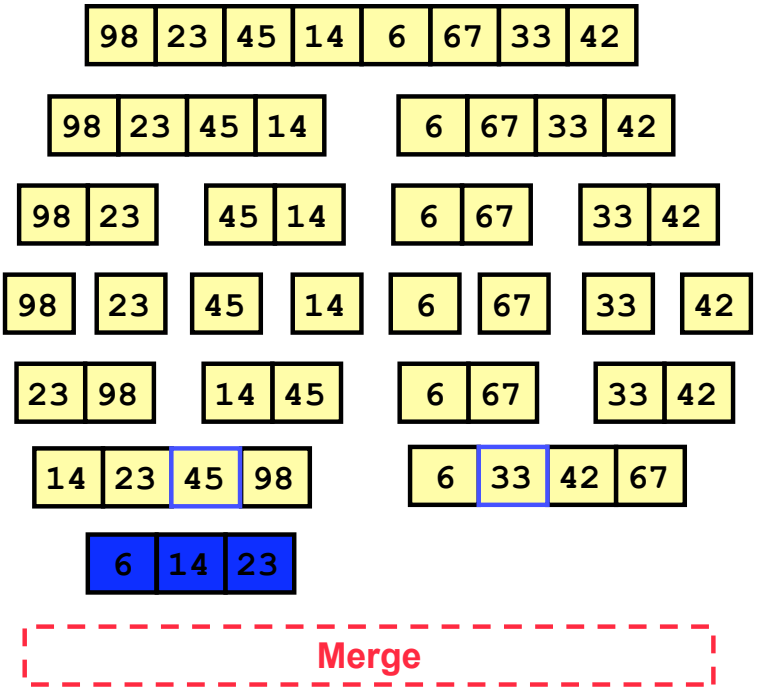

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42
Merge

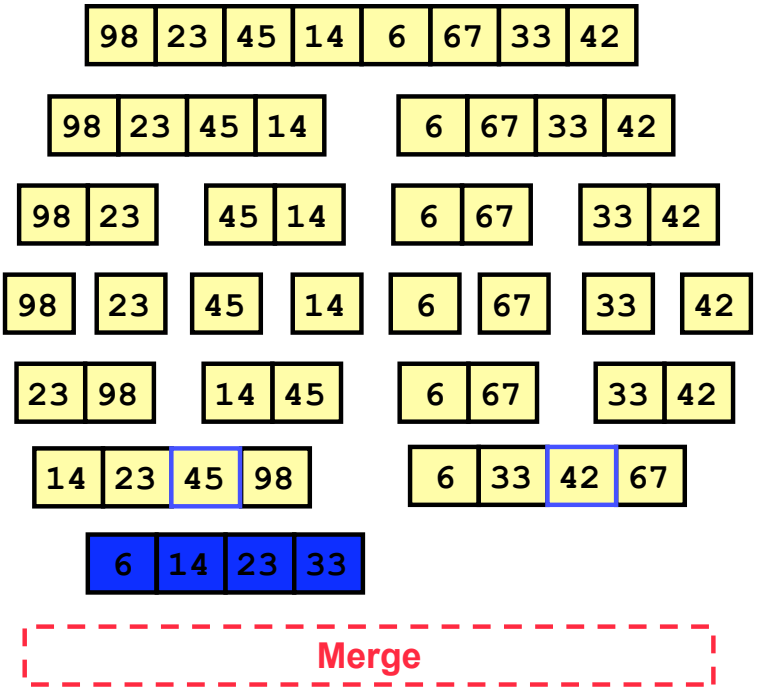

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

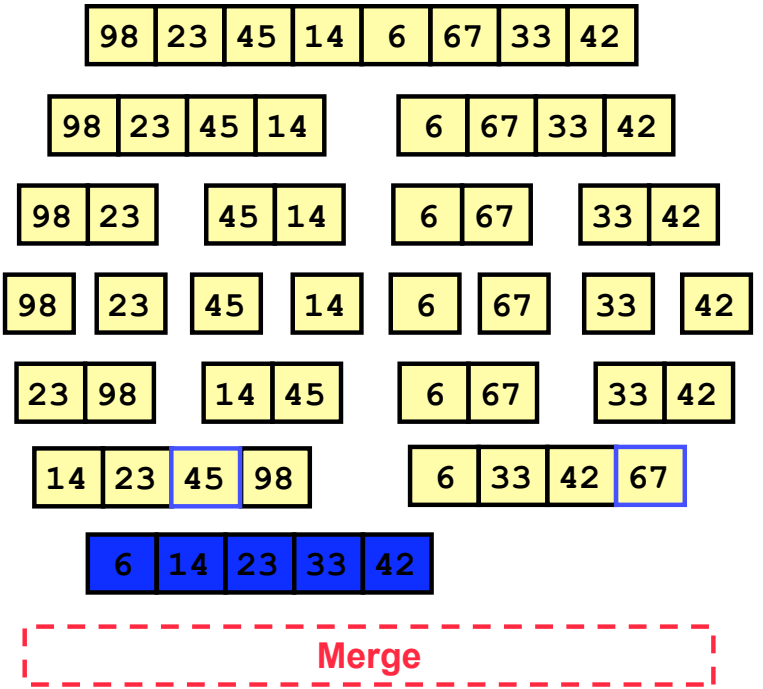

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
Merge

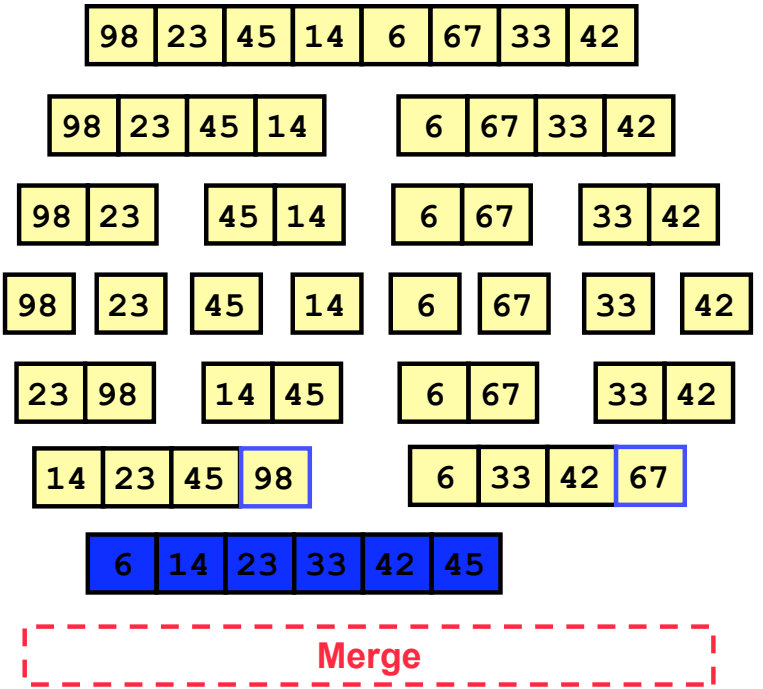

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6
Merge

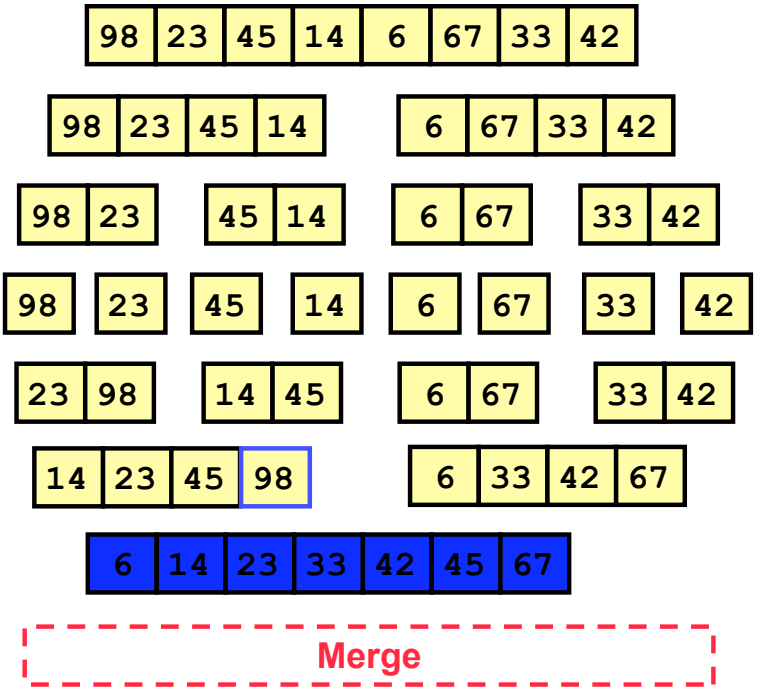

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14
Merge

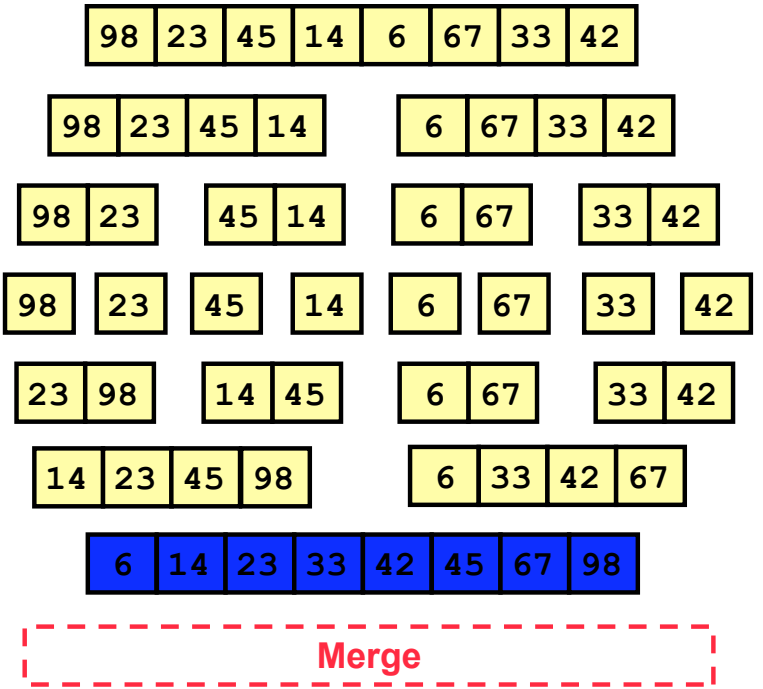

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23
Merge

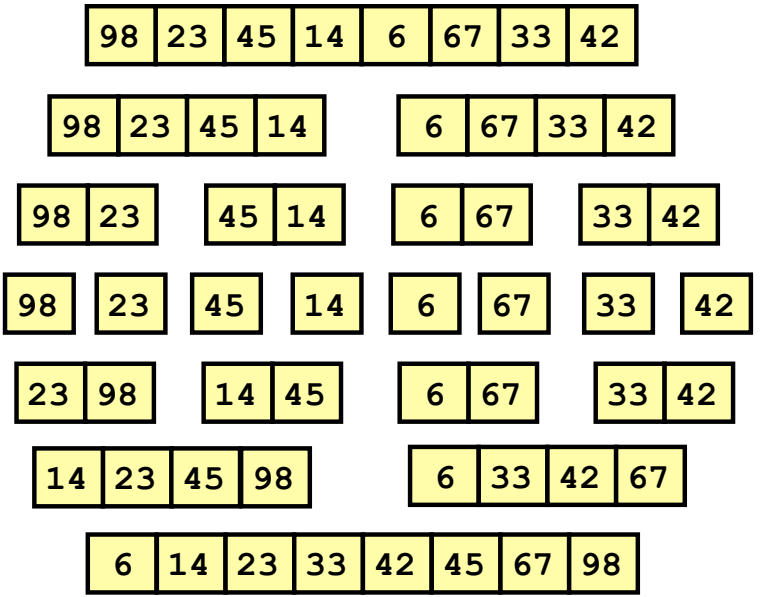

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33
Merge

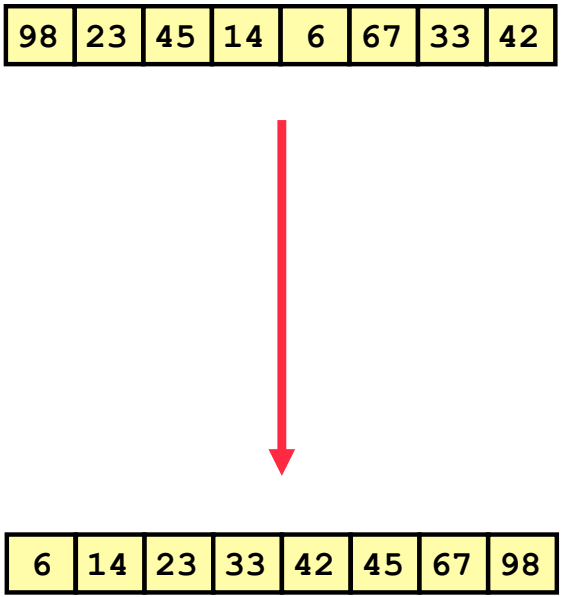

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42
Merge


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42 45
Merge


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42 45 67
Merge


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42 45 67 98
Merge


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42 45 67 98


98 23 45 14 6 67 33 42
6 14 23 33 42 45 67 98

The code

```
function a = mergesort(a)
%
%    This function sorts a column array, using the
%           merge sort algorithm
%
     sz = length(a);
     if sz > 1
          szb2 = floor(sz/2);
          first = a(1:szb2);
          second = a(szb2+1:sz);
          first = mergesort(first);
          second = mergesort(second);
          a = domerge(first, second);
     end
```

Arbitrarily divides collection in half (down the middle)

The "hard work" occurs at the end of each recursive call; sorted sub-lists are merged

```
function b = domerge(first, second)
%
%      Merges two sorted arrays into the array to be
%      sorted by this merge sorter.
%
      iFirst = 1;  % first array index
      iSecond = 1; % second array index
      out = 1;     % out array index
      sf = size(first);
      ss = size(second);
      while (iFirst <= sf) & (iSecond <= ss)
         if first(iFirst) < second(iSecond)
            b(out) = first(iFirst);
            iFirst = iFirst + 1;
         else
            b(out) = second(iSecond);
            iSecond = iSecond + 1;
         end
         out = out + 1;
      end
% continued …
```

```
% copy the remaining items

      % note that only one of the two while loops
      % below is executed

      % copy any remaining entries of the first array
      while iFirst <= sf(1)
          b(out) = first(iFirst);
          out = out + 1;
          iFirst = iFirst + 1;
      end
      % copy any remaining entries of the second array
      while iSecond <= ss(1)
          b(out) = second(iSecond);
          out = out + 1;
          iSecond = iSecond + 1;
      end
```

Analysis of Merge Sort

- Outer loop: How many times must we divide sub-halves in half to get to lists of length 1?

- How much work does each call to domerge cost?

Analysis of Merge Sort

- Outer loop: How many times must we divide sub-halves in half to get to lists of length 1? logN
- How much work does each call to domerge cost?
  - Completion of last recursive call: merge N pairs of lists of length 1
  - Completion of 2nd to last recursive call: merge N/2 pairs of lists of length 2 each
  - Completion of 3rd to last recursive call: merge N/4 pairs of lists of length 4 each
  - …
  - Completion of 1st recursive call: merge 2 lists of length N/2 each

Complexity of Merge Sort

- Merge Sort is O(NlogN)
- How does this compare to $O(N^2)$?

N	$\log_2 N$	N^2	$N \log_2 N$
32	5	1,024	160
64	6	4.096	384
128	7	16,384	896
256	8	65,536	2,048
512	9	262,144	4,608
1024	10	1,048,576	10,240
2048	11	4,194,304	22,528
4096	12	16,777,216	49,152

O(NlogN) Runtime Example

N = 250,000,000

$N*\log_2 N = 6.97 \times 10^9$

If you can do one operation per nanosecond (10^{-9} sec) *which is fast!*

It will take 6.97 seconds to complete N*logN operations…

As opposed to the 1.98 years required by N^2 algorithm

Questions?

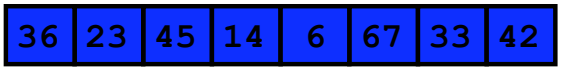
Quick Sort

Quick sort is another generative "divide and conquer" algorithm that involves splitting our sequence of data into parts and sorting each component recursively. Unlike merge sort, however, quick sort performs the actual sorting *as the sequence is being split.* In terms of performance, quick sort is considered to be a "better" algorithm than merge sort unless the original array was already sorted!

Quick Sort: The basic premise

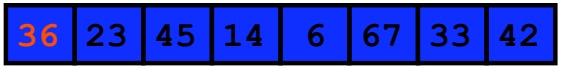
The basic idea of quick sort involves splitting up our array around a *pivot item*.

1. **We arbitrarily choose an element of our array to be a pivot item.**
2. **We then partition our array into two parts:**
   - **those elements that are *smaller* than our pivot**
   - **those elements that are *larger* than our pivot**
3. **We recursively sort the two parts, and join the results together with our pivot item to form a larger, sorted array.**



Here we start off with an array of unsorted elements. We arbitrarily choose a pivot point about which to organize our list. For the sake of expediency, let's just choose the first element* of our list to be our pivot.

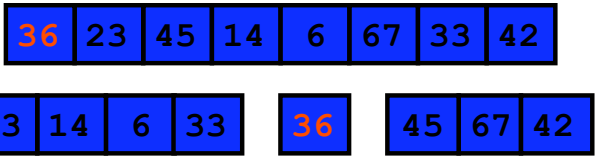
*** In many theory books, there are whole chapters dedicated to the process of choosing which element to use as the optimal pivot point. We're just keeping things simple.**



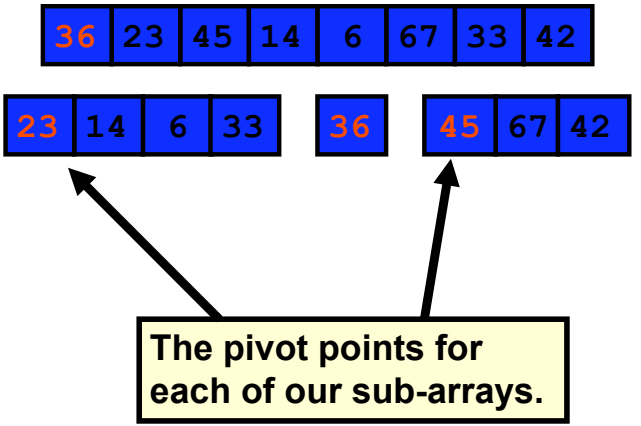
Now, we have to organize our array about our pivot point.

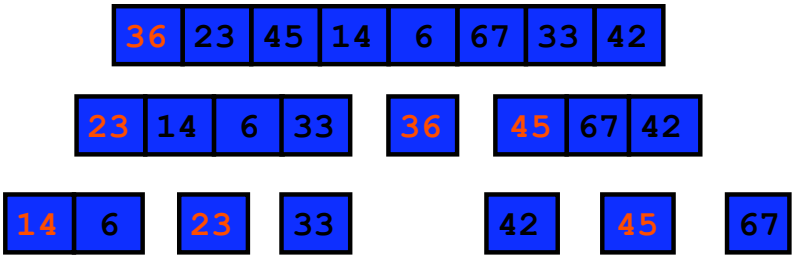
We separate the list into three parts:

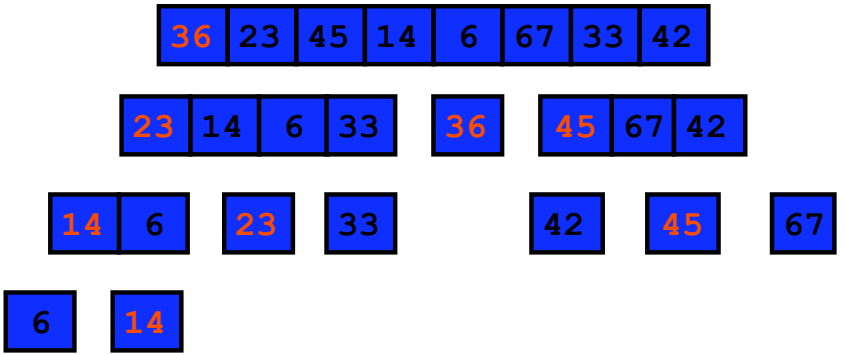
- The pivot value
- An array of items smaller than the pivot value
- An array of items larger than the pivot value:

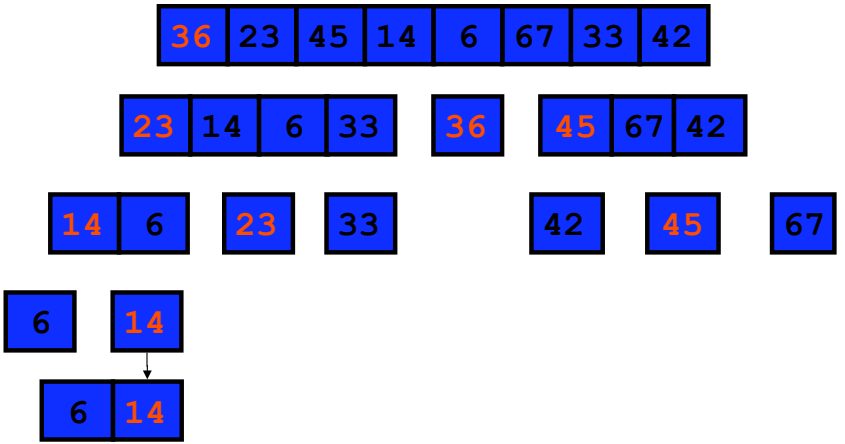


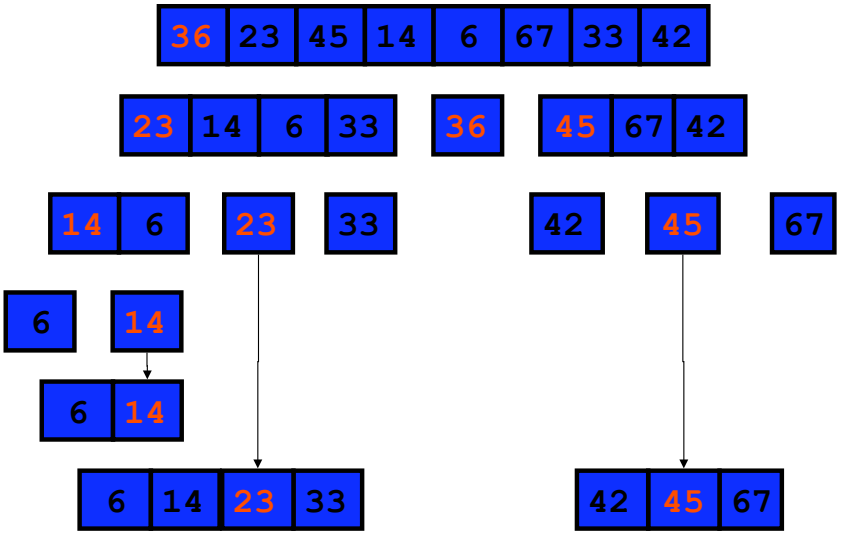
We start recurring on our two arrays.

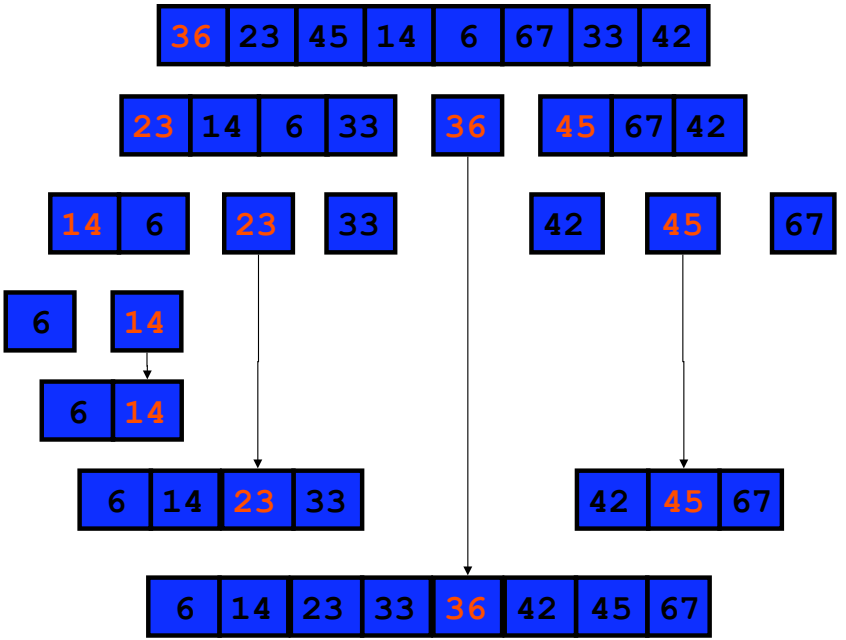

36 23 45 14 6 67 33 42
23 14 6 33
36
45 67 42
The pivot points for each of our sub-arrays.

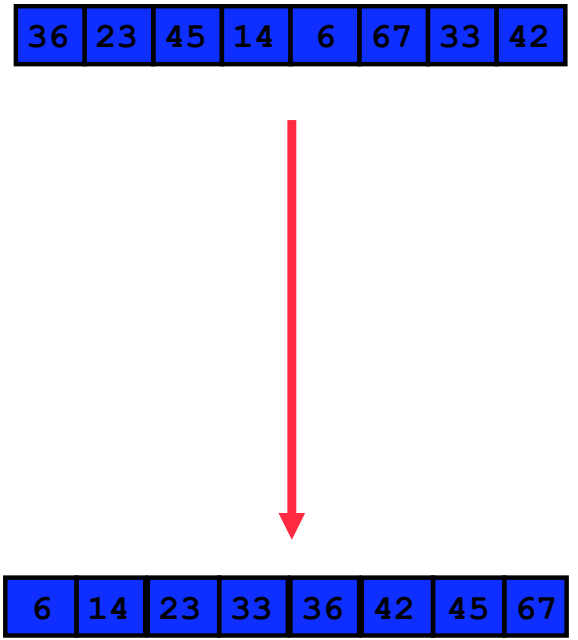

36 23 45 14 6 67 33 42
23 14 6 33
36
45 67 42
14 6
23
33
42
45
67


36 23 45 14 6 67 33 42
23 14 6 33 36 45 67 42
14 6 23 33 42 45 67
6 14


36 23 45 14 6 67 33 42
23 14 6 33
36
45 67 42
14 6
23
33
42
45
67
6
14
6 14


36 23 45 14 6 67 33 42
23 14 6 33
36
45 67 42
14 6
23
33
42
45
67
6
14
6 14
6 14 23 33
42 45 67


36 23 45 14 6 67 33 42
23 14 6 33
36
45 67 42
14 6
23
33
42
45
67
6
14
6 14
6 14 23 33
42 45 67
6 14 23 33 36 42 45 67


36 23 45 14 6 67 33 42
6 14 23 33 36 42 45 67

Our Quicksort Algorithm

- **The particular algorithm we use will actually sort the array "in place":**
  1. **Partition the array by swapping anything smaller than the pivot with the pivot**
  2. **Identifying the parts with *from* and *to* indices**
  3. **Checking for parts that are empty**
  4. **Recursively sorting the rest**

Quicksort: the code

```
function a = quicksort(a, from, to)
%
%   This function sorts a column array, using
%   the quick sort algorithm
%
    if (from < to)
        p = partition(a, from, to);
        a = quicksort(a, from, p);
        a = quicksort(a, p + 1, to);
    end
```

Quicksort: the code

```
function lower = partition(a, from, to)
%   This function partitions a column array for the
%      quick sort algorithm
  pivot = a(from);
  i = from - 1;
  j = to + 1;
  while (i < j) % push i forward over smaller items
      i = i + 1;
      while (a(i) < pivot)
          i = i + 1;
      end
      j = j - 1; % pull j back over larger items
      while (a(j) > pivot)
          j = j - 1;
      end
      if (i < j) % swap the elements at i and j
          temp = a(i);
          a(i) = a(j);
          a(j) = temp;
      end
  end
  lower = j;
```

Complexity of Quick Sort

- Quick Sort is O(NlogN)
- The “hard work” is done when partitioning the sub-lists, rather than merging (as in merge sort)

Questions?

Comparative Performance

- **What we really want to know is which algorithm is "better?"**
- **We actually don't care about performance sorting small amounts of data.**
- **The real question is: how does this algorithm perform as the amount of data gets large?**
- **You could do the math.**
  - ***Insertion Sort and other similar sorts like Bubble Sort and Selection Sort grow as the square of the number of items sorted (N^2).***
  - ***Divide and conquer algorithms grow as N log N***
- **Let's calculate and plot the performance to see if our theoretical big Os match…**

the sort test code

```
%   This program tests the selection, insertion, merge and quick
%   sort algorithms by sorting an array that is filled with
%   random numbers.
disp('-----------------------------------------------');
disp('Start sort test');
disp('-----------------------------------------------');
maxSize = 32768
iterations = 64
a = rand(maxSize,1);
size = 4;
loop = 1;
while size <= maxSize
%
% calculate the loop overhead
%
    tic;
    for iter = 1 : iterations*100
          b = a(1:size);
    end
    overhead = toc / (iterations*100);

% continued …
```

the sort test code

```
%
% test selection sort over 10 iterations
%
    tic;
    slow = iterations / 10;
    if slow < 1
        slow = 1;
    end
    for iter = 1 : slow
           b = a(1:size);
           b = selectionsort(b);
    end
    select = (toc / slow) - overhead;
%
% test insertion sort over 10 iterations
%
    tic;
    for iter = 1 : slow
           b = a(1:size);
           b = insertionsort(b);
    end
    insert = (toc / slow) - overhead;

% continued …
```

the sort test code

```
%
% test merge sort over 100 iterations
%
    tic;
    for iter = 1 : iterations
            b = a(1:size);
            b = mergesort(b);
    end
    merged = (toc / iterations) - overhead;
%
% test quick sort over 100 iterations
%
    tic;
    for iter = 1 : iterations
        b = a(1:size);
        b = quicksort(b, 1, size);
    end
    quick = (toc / iterations) - overhead;

% continued …
```

the sort test code

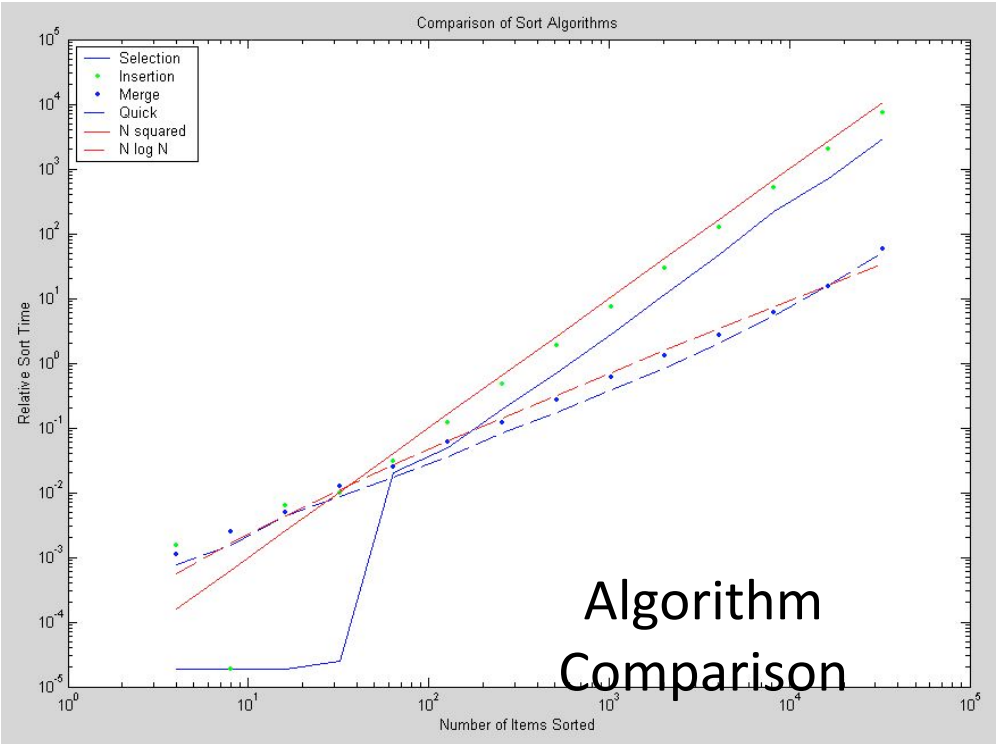
```
%
% show results
%
    fprintf('Size: %f; Sel: %f; Ins: %f; Merge: %f; Quick: %f;
Matlab: %f / 1000', ...
        size, select, insert, merged, quick, matlab);
    sz(loop) = size;
    sl(loop) = select;
    in(loop) = insert;
    mg(loop) = merged;
    qk(loop) = quick;

    loop = loop + 1;
    size = size * 2;
    iterations = iterations / 2;
    if iterations < 2
        iterations = 2;
    end
end
save sortdata
```

Hmmm … wonder how I would do that? Why not just plot the data?

the plotting code

```
load sortdata;
loglog(sz,sl);
title('Comparison of Sort Algorithms');
xlabel('Number of Items Sorted');
ylabel('Relative Sort Time');
hold on;
loglog(sz,in,'g.');
loglog(sz,mg,'.');
loglog(sz,qk,'--');
n = 1 : 15
nn = 2.^n
nsq = nn.*nn/100000
nlogn = nn.*log(nn) / 10000;
loglog(n, nsq, 'r');
loglog(n, nlogn, 'r--');
legend('Selection','Insertion','Merge','Quick', 'N squared', ...
    'N log N', 2);
```


Comparison of Sort Algorithms
Selection
Insertion
Merge
Quick
N squared
N log N
Relative Sort Time
Number of Items Sorted
Algorithm Comparison

Thoughts…

1. **This is a log-log plot. Sorting 32,000 items takes the N^2 algorithms *300 times* longer than the NlogN algorithms**

2. **In defense of Quick Sort, the performance is affected by our implementation. We return new arrays each recursive cycle, rather than editing a global variable in place. This really slowed down Quick Sort which is usually 2 – 3 times faster than Merge Sort.**

3. **Merge Sort had the same problem to a lesser extent.**

Thoughts…

1. **Matlab's built-in sort – going to be fastest option in most cases (so long as it can be applied to the data you are working with)**

When to use each algorithm

1. **Insertion sort: inserting a small number of items into an already sorted collection**

2. **Bubble sort: fine for small collections of data**

3. **Quick sort: quickest, so long as the data has a fairly high level of randomness**

4. **Merge Sort: use instead of quick sort when original data may already be partially sorted**

Questions?

Some Remarks

- Insertion-sort is a good choice for small input size (say, less than 50) and for sequences that are already "almost" sorted.
- Merge-sort is difficult to run in-place, is an excellent algorithm for situations where the input can not fit into main memory, but must be stored in blocks on an external memory device, e.g., disks.
- Quick sort is an excellent choice for general-purpose, in-memory sorting. In spite of its slow worst-case running time. The constant factors hidden in O(nlgn) for average case are quite small.