Sorting

Textbook Ch 2, 7, 8

Outline

- Introduction
- Inversions
- Insertion sort
- Bubble sort
- Heap sort
- Merge sort
- Quicksort
- Bucket sort
- Radix sort

Definition

Sorting is the process of:

– Taking a list of objects which could be stored in a linear order

$$(a_0, a_1, ..., a_{n-1})$$

e.g., numbers, and returning an reordering

$$(a'_0, a'_1, ..., a'_{n-1})$$

such that

$$a'_0 \le a'_1 \le \cdot \cdot \cdot \le a'_{n-1}$$

The conversion of an Abstract List into an Abstract Sorted List

Definition

Seldom will we sort isolated values

– Usually we will sort a number of records containing a number of fields based on a *key*:

19991532	Stevenson	Monica	3 Glendridge Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19985832	Kilji	Islam	37 Masterson Ave.
20003541	Groskurth	Ken	12 Marsdale Ave.
19981932	Carol	Ann	81 Oakridge Ave.
20003287	Redpath	David	5 Glendale Ave.

Numerically by ID Number

19981932	Carol	Ann	81 Oakridge Ave.
19985832	Khilji	Islam	37 Masterson Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19991532	Stevenson	Monica	3 Glendridge Ave.
20003287	Redpath	David	5 Glendale Ave.
20003541	Groskurth	Ken	12 Marsdale Ave.

Lexicographically by surname, then given name

19981932	Carol	Ann	81 Oakridge Ave.
20003541	Groskurth	Ken	12 Marsdale Ave.
19985832	Kilji	Islam	37 Masterson Ave.
20003287	Redpath	David	5 Glendale Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19991532	Stevenson	Monica	3 Glendridge Ave.

Definition

In these topics, we will assume that:

- We are sorting integers
  - The algorithms can also be applied to other types of objects as long as we can compare any two objects
- Arrays are to be used for both input and output

In-place Sorting

Sorting algorithms may be performed *in-place*, that is, with the allocation of at most $\Theta(1)$ additional memory (*e.g.*, fixed number of local variables)

- Some definitions of *in place* as using $\mathbf{o}(n)$ memory

Other sorting algorithms require the allocation of second array of equal size

- Requires $\Theta(n)$ additional memory

We will prefer in-place sorting algorithms

Run-time

The run time of the sorting algorithms we will look at fall into one of three categories:

$$\mathbf{\Theta}(n) \qquad \mathbf{\Theta}(n \ln(n)) \qquad \mathbf{O}(n^2)$$

We will examine average- and worst-case scenarios for each algorithm

The run-time may change significantly based on the scenario

Run-time

We will review the more traditional $\mathbf{O}(n^2)$ sorting algorithms:

- Insertion sort, Bubble sort

Some of the faster $\boldsymbol{\Theta}(n \ln(n))$ sorting algorithms:

- Heap sort, Quicksort, and Merge sort

And linear-time sorting algorithms

- Bucket sort and Radix sort
- We must make assumptions about the data

Lower-bound Run-time

Any sorting algorithm must examine each entry in the array at least once

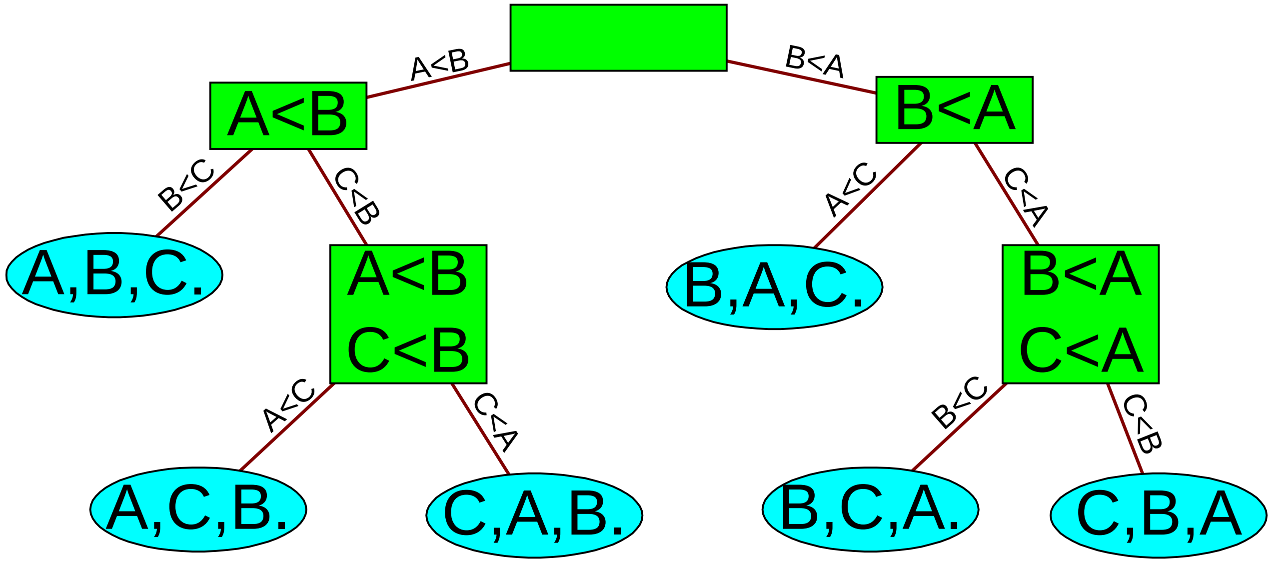
– Consequently, all sorting algorithms must be $\Omega(n)$

We will not be able to achieve $\Theta(n)$ behaviour without additional assumptions

Lower-bound Run-time

The general run time is $\Omega(n \ln(n))$

The proof is based on the idea of a *comparison tree*



Legend

facts — Internal node, with facts known so far

A,B,C — Leaf node, with ordering of A,B,C

C<A — Edge, with result of one comparison

Lower-bound Run-time

The general run time is $\mathbf{\Omega}(n \ln(n))$

The proof:

- Any comparison-based sorting algorithm can be represented by a comparison tree
- Worst-case running time cannot be less than the height of the tree
- How many leaves does the tree have?
  - The number of permutations of n objects, which is $n!$
- What's the shallowest tree with $n!$ leaves?
  - A complete tree, whose height is $\lg(n!)$
  - It can be shown that $\lg(n!) = \mathbf{\Theta}(n \ln(n))$

Sub-optimal Sorting Algorithms

Before we look at other algorithms, we will consider the Bogosort algorithm:

1. Randomly order the objects, and
2. Check if they're sorted, if not, go back to Step 1.

Run time analysis:

- best case: $\Theta(n)$
- worst: unbounded
- average: $\Theta(n \cdot n!)$

Sub-optimal Sorting Algorithms

There is also the Bozosort algorithm:

1. Check if the entries are sorted,
2. If they are not, randomly swap two entries and go to Step 1.

Run time analysis:

- More difficult than bogosort...
- O(n!) is the expected average case

Outline

- Introduction
- Inversions
- Insertion sort
- Bubble sort
- Heap sort
- Merge sort
- Quicksort
- Bucket sort
- Radix sort

Inversions

Consider the following three lists:

```
 1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

 1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99

22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80  1 31 16 92
```

To what degree are these three lists unsorted?

Inversions

The first list requires only a few exchanges to make it sorted

1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

1 12 16 25 26 33 35 42 45 56 58 67 74 75 81 83 86 88 95 99

Inversions

The second list has two entries significantly out of order

1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99

1 17 21 23 24 27 32 35 42 45 47 57 66 69 70 76 85 87 95 99

however, most entries (13) are in place

Inversions

The third list would, by any reasonable definition, be significantly unsorted

22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92

1 10 12 16 20 22 26 31 38 44 48 79 80 81 84 87 92 95 96 99

Inversions

Given any list of n numbers, there are

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

pairs of numbers

For example, the list (1, 3, 5, 4, 2, 6) contains the following 15 pairs:

(1, 3)	(1, 5)	(1, 4)	(1, 2)	(1, 6)
	(3, 5)	(3, 4)	(3, 2)	(3, 6)
		(5, 4)	(5, 2)	(5, 6)
			(4, 2)	(4, 6)
				(2, 6)

Inversions

You may note that 11 of these pairs of numbers are in order:

(1, 3)	**(1, 5)**	**(1, 4)**	**(1, 2)**	**(1, 6)**
	(3, 5)	**(3, 4)**	(3, 2)	**(3, 6)**
		(5, 4)	(5, 2)	**(5, 6)**
			(4, 2)	**(4, 6)**
				(2, 6)

Inversions

The remaining four pairs are *reversed*, or *inverted*

(1, 3)	(1, 5)	(1, 4)	(1, 2)	(1, 6)
	(3, 5)	(3, 4)	**(3, 2)**	(3, 6)
		(5, 4)	**(5, 2)**	(5, 6)
			(4, 2)	(4, 6)
				(2, 6)

Inversions

Given a permutation of n elements

$$a_0, a_1, ..., a_{n-1}$$

an inversion is defined as a pair of entries which are reversed

That is, (a_j, a_k) forms an inversion if

$$j < k \text{ but } a_j > a_k$$

Inversions

Therefore, the permutation

1, 3, 5, 4, 2, 6

contains four inversions:

(3, 2) (5, 4) (5, 2) (4, 2)

Inversions

Exchanging (or swapping) two adjacent entries either:

– removes an inversion, *e*.*g*.,

1 3 5 4 2 6

1 3 5 2 4 6

removes the inversion (4, 2)

– or introduces a new inversion, *e*.*g*., (5, 3) with

1 3 5 4 2 6

1 5 3 4 2 6

Number of Inversions

There are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of numbers in any set of n objects

Consequently, each pair contributes to

- the set of ordered pairs, or
- the set of inversions

For a random ordering, we would expect approximately half of all pairs are inversions:

$$\frac{1}{2}\binom{n}{2} = \frac{n(n-1)}{4} = \mathbf{O}(n^2)$$

Number of Inversions

Let us consider the number of inversions in our first three lists:

1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99

22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92

Each list has 20 entries, and therefore:

- There are $\binom{20}{2} = \frac{20(20-1)}{2} = 190$ pairs
- On average, $190/2 = 95$ pairs would form inversions

Number of Inversions

The first list

1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

has 13 inversions:

(16, 12) (26, 25) (35, 33) (58, 45) (58, 42) (58, 56) (45, 42)
(83, 75) (83, 74) (83, 81) (75, 74) (86, 81) (99, 95)

This is well below 95, the expected number of inversions

– Therefore, this is likely not to be a *random* list

Number of Inversions

The second list

1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99

also has 13 inversions:

(42, 24) (42, 27) (42, 32) (42, 35) (42, 23) (24, 23) (27, 23)
(32, 23) (35, 23) (45, 23) (47, 23) (57, 23) (87, 85)

This, too, is not a random list

Number of Inversions

The third list

22 20 81 38 95 84 **99** 12 79 44 26 87 96 10 48 80 **1** 31 16 92

has 100 inversions:

(22, 20)	(22, 12)	(22, 10)	(22, **1**)	(22, 16)	(20, 12)	(20, 10)	(20, **1**)	(20, 16)	(81, 38)
(81, 12)	(81, 79)	(81, 44)	(81, 26)	(81, 10)	(81, 48)	(81, 80)	(81, **1**)	(81, 16)	(81, 31)
(38, 12)	(38, 26)	(38, 10)	(38, **1**)	(38, 16)	(38, 31)	(95, 84)	(95, 12)	(95, 79)	(95, 44)
(95, 26)	(95, 87)	(95, 10)	(95, 48)	(95, 80)	(95, **1**)	(95, 16)	(95, 31)	(95, 92)	(84, 12)
(84, 79)	(84, 44)	(84, 26)	(84, 10)	(84, 48)	(84, 80)	(84, **1**)	(84, 16)	(84, 31)	(**99**, 12)
(**99**, 79)	(**99**, 44)	(**99**, 26)	(**99**, 87)	(**99**, 96)	(**99**, 10)	(**99**, 48)	(**99**, 80)	(**99**, **1**)	(**99**, 16)
(**99**, 31)	(**99**, 92)	(12, 10)	(12, **1**)	(79, 44)	(79, 26)	(79, 10)	(79, 48)	(79, **1**)	(79, 16)
(79, 31)	(44, 26)	(44, 10)	(44, **1**)	(44, 16)	(44, 31)	(26, 10)	(26, **1**)	(26, 16)	(87, 10)
(87, 48)	(87, 80)	(87, **1**)	(87, 16)	(87, 31)	(96, 10)	(96, 48)	(96, 80)	(96, **1**)	(96, 16)
(96, 31)	(96, 92)	(10, **1**)	(48, **1**)	(48, 16)	(48, 31)	(80, **1**)	(80, 16)	(80, 31)	(31, 16)

Outline

- Introduction
- Inversions
- Insertion sort
- Bubble sort
- Heap sort
- Merge sort
- Quicksort
- Bucket sort
- Radix sort

Background

Consider the following observations:

- A list with one element is sorted
- In general, if we have a sorted list of k items, we can insert a new item to create a sorted list of size $k + 1$

Background

For example, consider this sorted array containing of eight sorted entries

5	**7**	**12**	**19**	**21**	**26**	**33**	**40**	**14**	9	18	21	2

Suppose we want to insert 14 into this array leaving the resulting array sorted

Background

Starting at the back, if the number is greater than 14, copy it to the right

– Once an entry less than 14 is found, insert 14 into the resulting vacancy

5	7	12	19	21	26	33	40	14	9	18	21	2
5	7	12	19	21	26	33	14	40	9	18	21	2
5	7	12	19	21	26	14	33	40	9	18	21	2
5	7	12	19	21	14	26	33	40	9	18	21	2
5	7	12	19	14	21	26	33	40	9	18	21	2
5	7	12	14	19	21	26	33	40	9	18	21	2

The Algorithm

For any unsorted list:

- Treat the first element as a sorted list of size 1

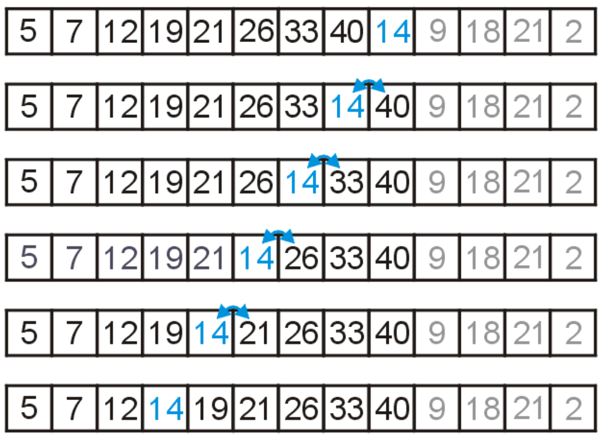
Then, given a sorted list of size $k - 1$

- Insert the k^{th} item into the sorted list
- The sorted list is now of size k

The Algorithm

Code for this would be:

```
for ( int j = k; j > 0; --j ) {
    if ( array[j - 1] > array[j] ) {
        std::swap( array[j - 1], array[j] );
    } else {
        // As soon as we don't need to swap, the (k + 1)st
        // is in the correct location
        break;
    }
}
```



Implementation and Analysis

This would be embedded in a function call such as

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > array[j] ) {
                std::swap( array[j - 1], array[j] );
            } else {
                // As soon as we don't need to swap, the (k + 1)st
                // is in the correct location
                break;
            }
        }
    }
}
```

Implementation and Analysis

Let’s do a run-time analysis of this code

- the outer for-loop will be executed a total of $n - 1$ times
- In the worst case, the inner for-loop is executed k times
- Thus, the worst-case run time is $\mathrm{O}(n^2)$

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > array[j] ) {
                std::swap( array[j - 1], array[j] );
            } else {
                // As soon as we don't need to swap, the (k + 1)st
                // is in the correct location
                break;
            }
        }
    }
}
```

Implementation and Analysis

Problem: we may break out of the inner loop…

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > array[j] ) {
                std::swap( array[j - 1], array[j] );
            } else {
                // As soon as we don't need to swap, the (k + 1)st
               // is in the correct location
               break;
            }
        }
    }
}
```

Implementation and Analysis

Recall: each time we perform a swap, we remove an inversion

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > array[j] ) {
                std::swap( array[j - 1], array[j] );
            } else {
                // As soon as we don't need to swap, the (k + 1)st
               // is in the correct location
               break;
            }
        }
    }
}
```

Implementation and Analysis

Thus, the body is run only as often as there are inversions

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > array[j] ) {
                std::swap( array[j - 1], array[j] );
            } else {
                // As soon as we don't need to swap, the (k + 1)st
               // is in the correct location
               break;
            }
        }
    }
}
```

If the number of inversions is d, the run time is $\Theta(n + d)$

Consequences of Our Analysis

The average random list has $d = \Theta(n^2)$ inversions

Insertion sort, however, will run in $\Theta(n)$ time whenever $d = \mathrm{O}(n)$

Other benefits:

- The algorithm is easy to implement
- Even in the worst case, the algorithm is fast for small problems

Size	**Approximate Time (ns)**
8	175
16	750
32	2700
64	8000

Consequences of Our Analysis

Unfortunately, it is not very useful in general:

- Sorting a random list of size $2^{23} \approx 8\ 000\ 000$ would require approximately one day
- Doubling the size of the list quadruples the required run time
- An optimized quick sort requires less than $4\ \text{s}$ on a list of the above size

Consequences of Our Analysis

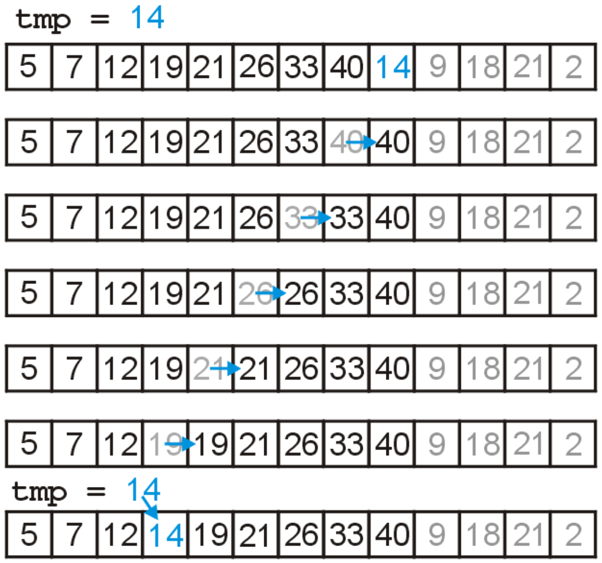
The following table summarizes the run-times of insertion sort

Case	**Run Time**	**Comments**
Worst	$\Theta(n^2)$	Reverse sorted
Average	$\mathrm{O}(d + n)$	Slow if $d = \omega(n)$
Best	$\Theta(n)$	Very few inversions: $d = \mathrm{O}(n)$

A small improvement

Swapping is expensive, so we could just temporarily assign the new entry

- this reduces assignments by a factor of 3



Outline

- Introduction
- Inversions
- Insertion sort
- Bubble sort
- Heap sort
- Merge sort
- Quicksort
- Bucket sort
- Radix sort

Obama on bubble sort

When asked the most efficient way to sort a million 32-bit integers, Obama had an answer:

http://player.youku.com/embed/XMjQyMTk4ODk2

Description

Suppose we have an array of data which is unsorted:

- Starting at the front, traverse the array, find the largest item, and move (or *bubble*) it to the top
- With each subsequent iteration, find the next largest item and *bubble* it up towards the top of the array

The basic algorithm

Starting with the first item, assume that it is the largest

Compare it with the second item:

- If the first is larger, swap the two,
- Otherwise, assume that the second item is the largest

Continue up the array, either swapping or redefining the largest item

The basic algorithm

After one pass, the largest item must be the last in the list

Start at the front again:

- the second pass will bring the second largest element into the second last position

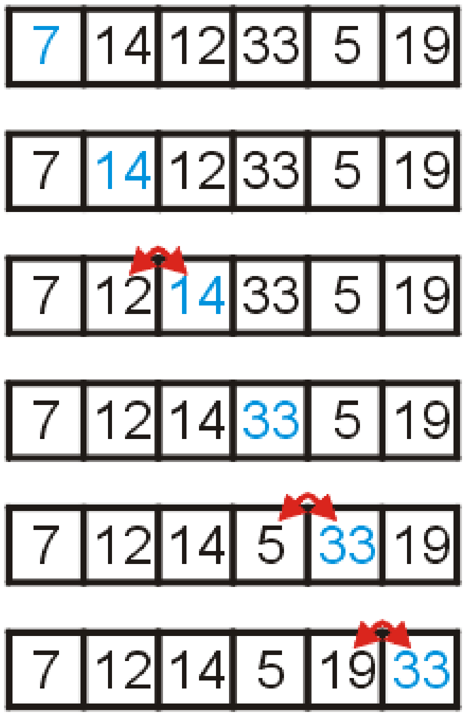
Repeat $n - 1$ times, after which, all entries will be in place

Example

Consider the unsorted array to the right

We start with the element in the first location, and move forward:

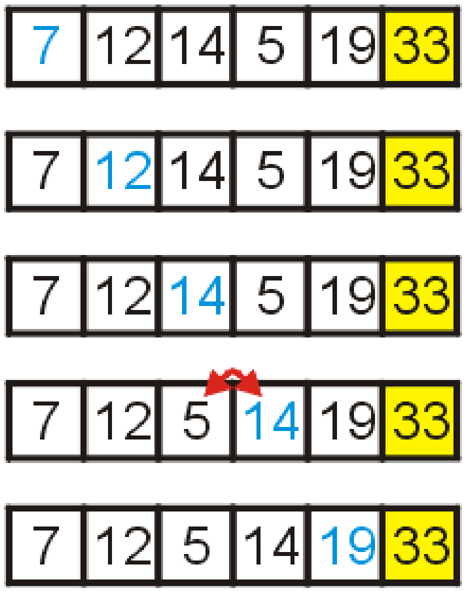
- if the current and next items are in order, continue with the next item, otherwise
- swap the two entries



Example

After one loop, the largest element is in the last location

- Repeat the procedure



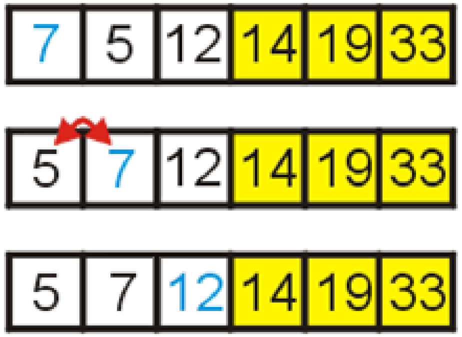
Example

Now the two largest elements are at the end

- Repeat again

Example

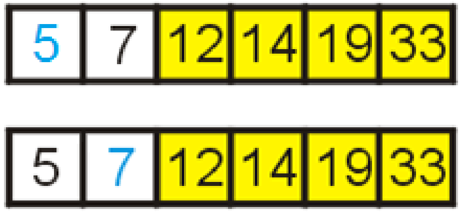
With this loop, 5 and 7 are swapped



Example

Finally, we check the last two entries

– At this point, we have a sorted array



The basic algorithm

The default algorithm:

```
template <typename Type>
void bubble( Type *const array, int const n ) {
        for ( int i = n - 1; i > 0; --i ) {
                for ( int j = 0; j < i; ++j ) {
                        if ( array[j] > array[j + 1] ) {
                                std::swap( array[j], array[j + 1] );
                        }
                }
        }
}
```

Analysis

Here we have two nested loops, and therefore calculating the run time is straight-forward:

$$\sum_{k=1}^{n-1}(n-k) = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2} = \Theta(n^2)$$

Implementations and Improvements

The next few slides show some implementations of bubble sort together with a few improvements:

- reduce the number of swaps,
- halting if the list is sorted,
- limiting the range on which we must bubble
- alternating between bubbling up and sinking down

First Improvement

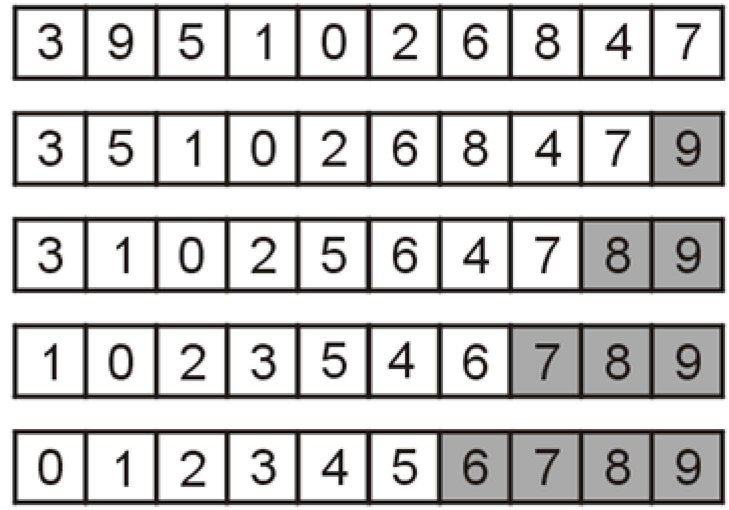
We could avoid so many swaps...

```
template <typename Type>
void bubble( Type *const array, int const n ) {
  for ( int i = n - 1; i > 0; --i ) {
        Type max = array[0];                    // assume a[0] is the max
        for ( int j = 1; j <= i; ++j ) {
              if ( array[j] < max ) {
                    array[j - 1] = array[j];   // move
              } else {
                    array[j – 1] = max;         // store the old max
                    max = array[j];             // get the new max
              }
        }
        array[i] = max;                         // store the max
  }
}
```

Flagged Bubble Sort

One useful modification would be to check if no swaps occur:

- If no swaps occur, the list is sorted
- In this example, no swaps occurred during the 5th pass



Use a Boolean flag to check if no swaps occurred

Flagged Bubble Sort

Check if the list is sorted (no swaps)

```
template <typename Type>
void bubble( Type *const array, int const n ) {
   for ( int i = n - 1; i > 0; --i ) {
         Type max = array[0];
         bool sorted = true;
         for ( int j = 1; j <= i; ++j ) {
               if ( array[j] < max ) {
                     array[j - 1] = array[j];
                     sorted = false;
               } else {
                     array[j – 1] = max;
                     max = array[j];
               }
         }
         array[i] = max;
         if ( sorted ) {
               break;
         }
   }
}
```

Range-limiting Bubble Sort

Intuitively, one may believe that limiting the loops based on the location of the last swap may significantly speed up the algorithm

- For example, after the second pass, we are certain all entries after 4 are sorted

The implementation is easier than that for using a Boolean flag

Unfortunately, in practice, this does little to affect the number of comparisons

Range-limiting Bubble Sort

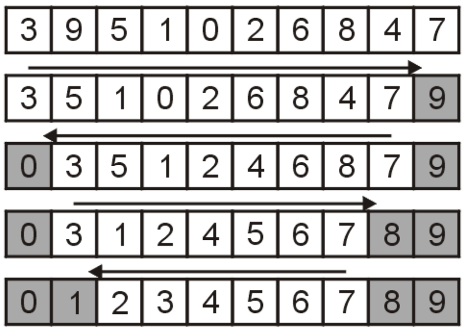
Update **`i`** to at the place of the last swap

```
template <typename Type>
void bubble( Type *const array, int const n ) {
  for ( int i = n - 1; i > 0; ) {
        Type max = array[0];
        int ii = 0;
        for ( int j = 1; j <= i; ++j ) {
              if ( array[j] < max ) {
                    array[j - 1] = array[j];
                    ii = j - 1;
              } else {
                    array[j – 1] = max;
                    max = array[j];
              }
        }
        array[i] = max;
        i = ii;
  }
}
```

Alternating Bubble Sort

One operation which does significantly improve the run time is to alternate between

- bubbling the largest entry to the top, and
- sinking the smallest entry to the bottom



Run-time Analysis

Because the bubble sort simply swaps adjacent entries, it cannot be any better than insertion sort which does $n + d$ comparisons where d is the number of inversions

Unfortunately, run-time analysis isn't that easy:

- There are numerous unnecessary comparisons

Empirical Analysis

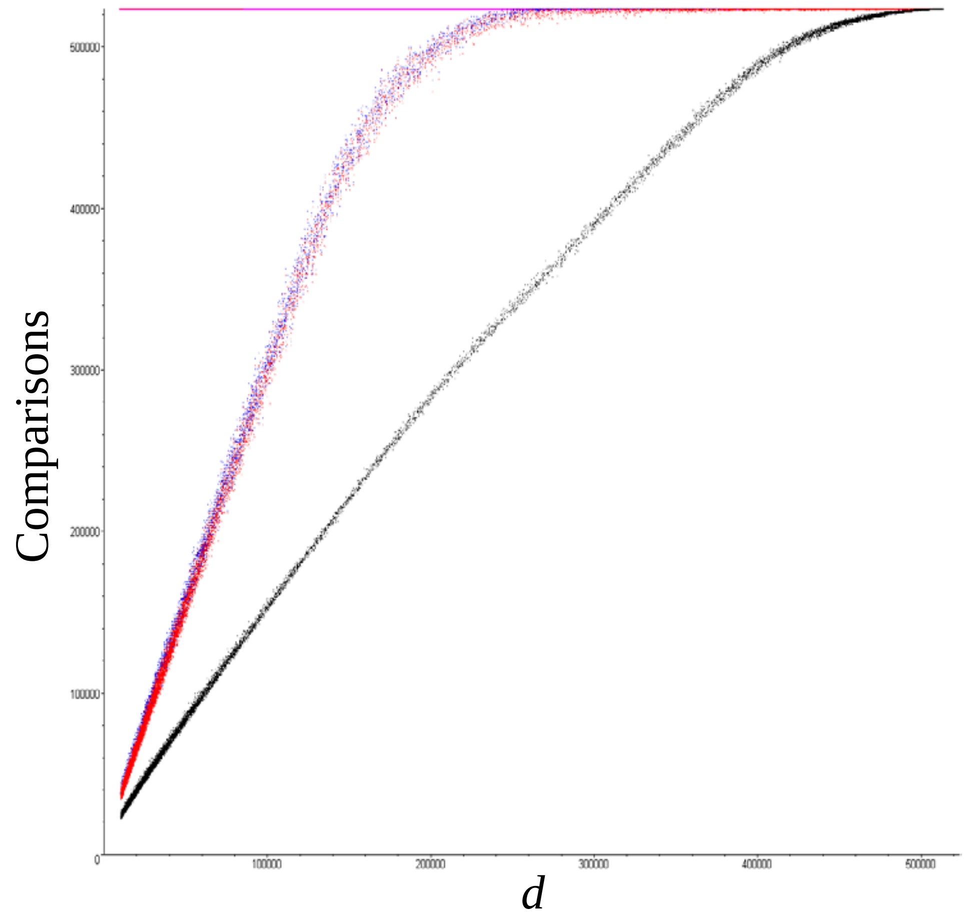
The next slide map the number of required comparisons necessary to sort 32768 arrays of size 1024 where the number of inversions range from 10000 to 523776

- Each point (d, c) is the number of inversions in an unsorted list d and the number of required comparisons c

Empirical Analysis

The following for plots show the required number of comparisons required to sort an array of size 1024

Basic implementation
Flagged
Range limiting
Alternating

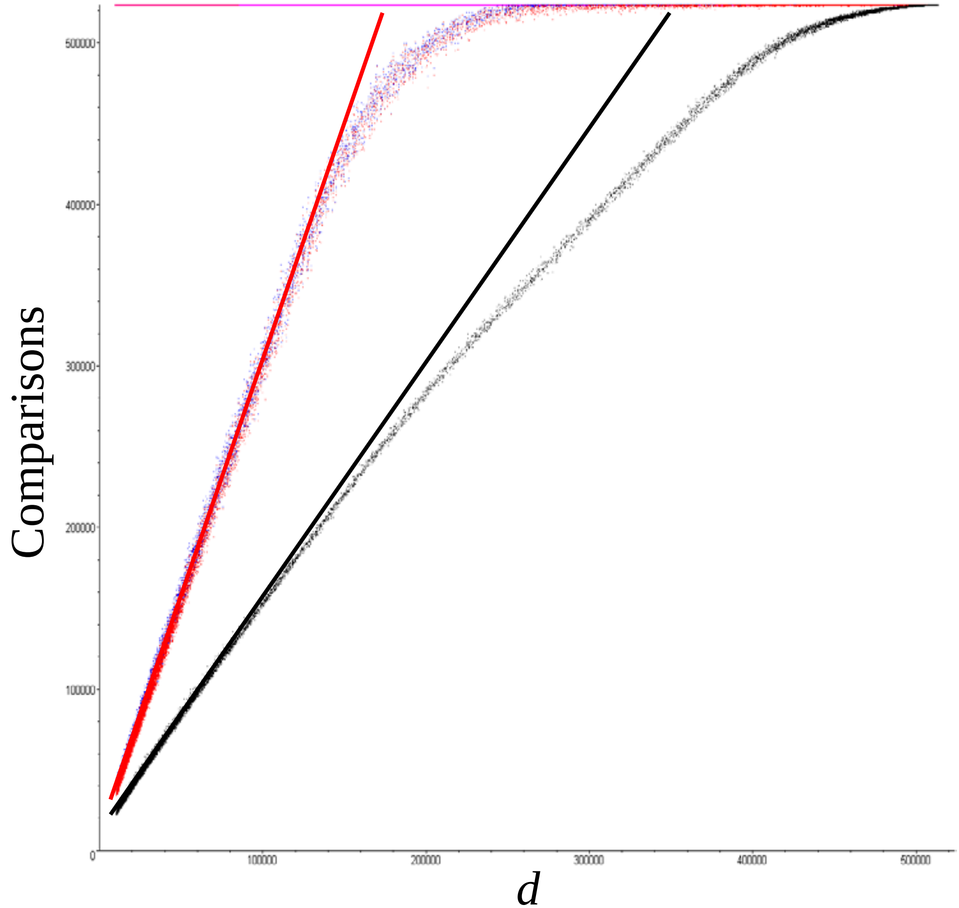


Empirical Analysis

The number of comparisons with the flagged/limiting sort is initially $n + 3d$

For the alternating variation, it is initially $n + 1.5d$

Basic implementation
Flagged
Range limiting
Alternating

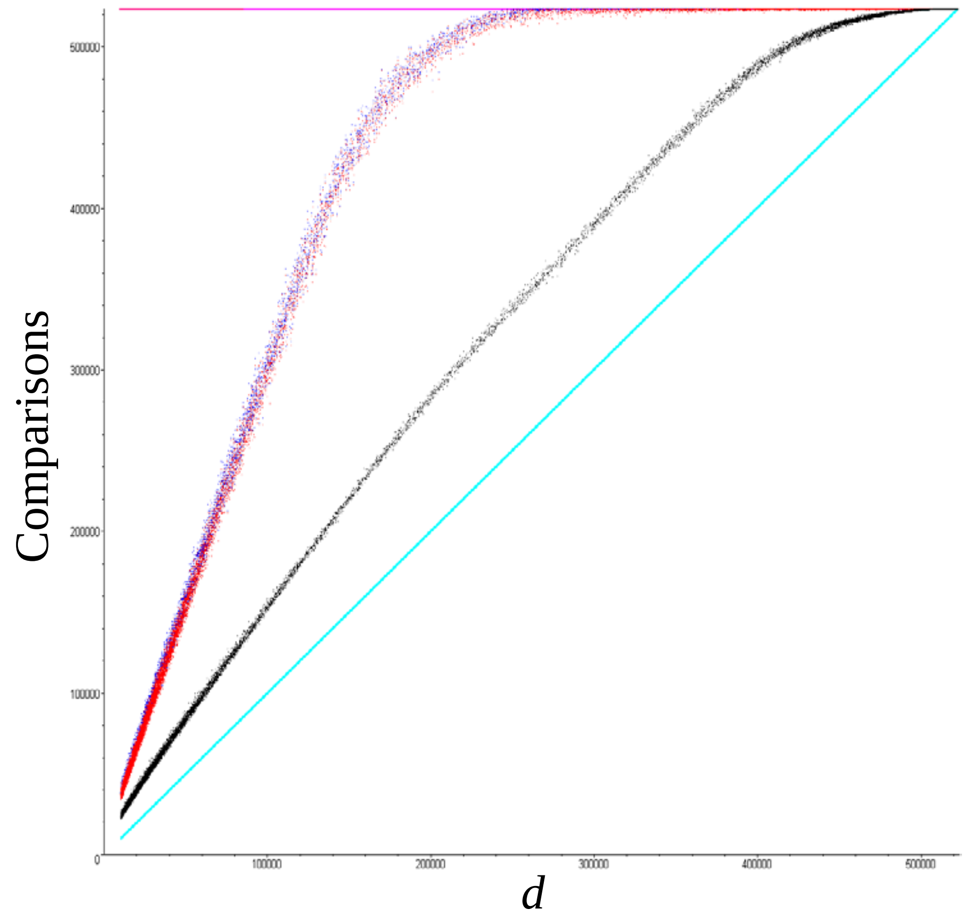


Empirical Analysis

Unfortunately, the comparisons for insertion sort is $n + d$ which is better in all cases except when the list is

- Sorted, or
- Reverse sorted

Basic implementation
Flagged
Range limiting
Alternating
Insertion Sort



Run-Time

The following table summarizes the run-times of our modified bubble sorting algorithms; however, they are all worse than insertion sort in practice

Case	Run Time	Comments
Worst	$\Theta(n^2)$	$\Theta(n^2)$ inversions
Average	$\Theta(n + d)$	Slow if $d = \omega(n)$
Best	$\Theta(n)$	$d = O(n)$ inversions

Outline

- Introduction
- Inversions
- Insertion sort
- Bubble sort
- Heap sort
- Merge sort
- Quicksort
- Bucket sort
- Radix sort

Basic Implementation

- Heapsort
  - Place the objects into a min-heap
    - O(n) time
  - Repeatedly pop the top object until the heap is empty
    - O(n ln(n)) time
  - Time complexity: O(n ln(n))

- Problem
  - This solution requires additional memory: a min-heap of size n
  - This requires $\Theta(n)$ memory

In-place Implementation

Instead of implementing a min-heap, consider a max-heap:

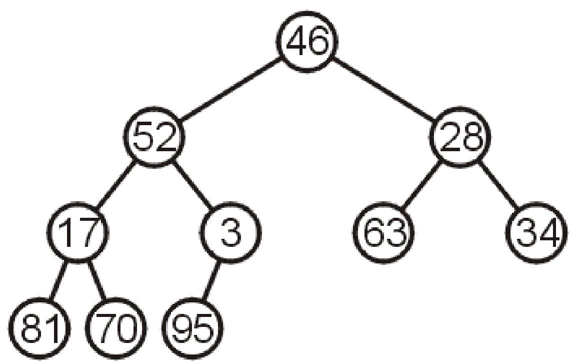
- Place the objects into a max-heap, stored in the input array
- Repeatedly pop the top object and move it to the end of the array, until the heap is empty
- $\Theta(1)$ memory

Example Heap Sort

Now, consider this unsorted array:

46	52	28	17	3	63	34	81	70	95

This array represents the following complete tree:



Children	`2*k + 1`	`2*k + 2`
Parent	`(k + 1)/2 - 1`	

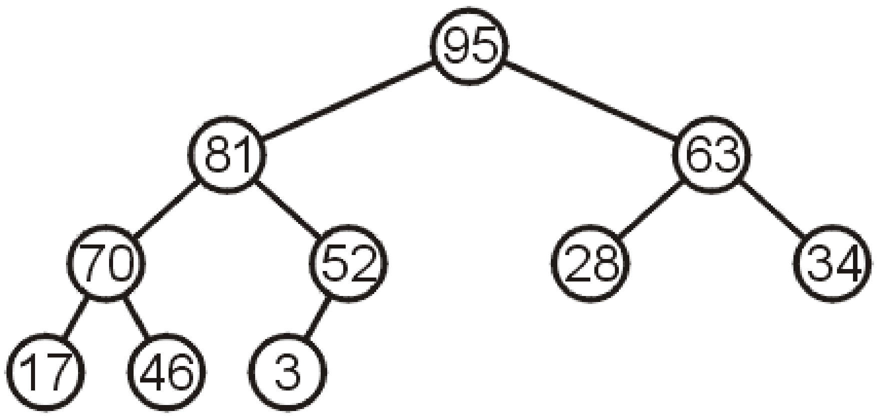
Heap Sort Example

Convert the input array

46	52	28	17	3	63	34	81	70	95

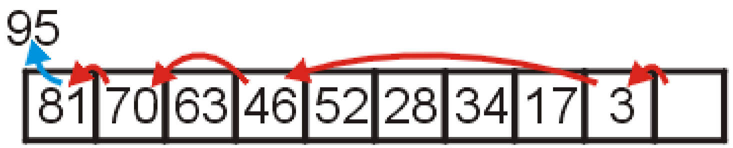
into a max-heap using Floyd's method

95	81	63	70	52	28	34	17	46	3

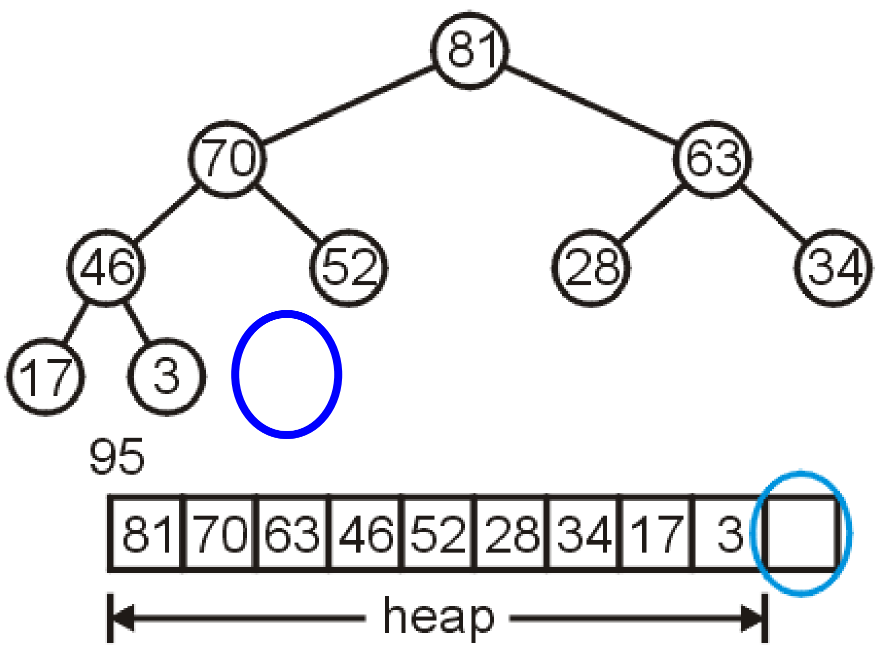


Heap Sort Example

We pop the maximum element of this heap



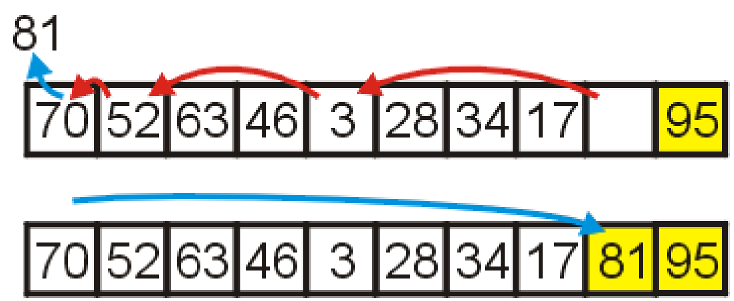
This leaves a gap at the back of the array:



Heap Sort Example

This is the last entry in the array, so we fill it with the largest element

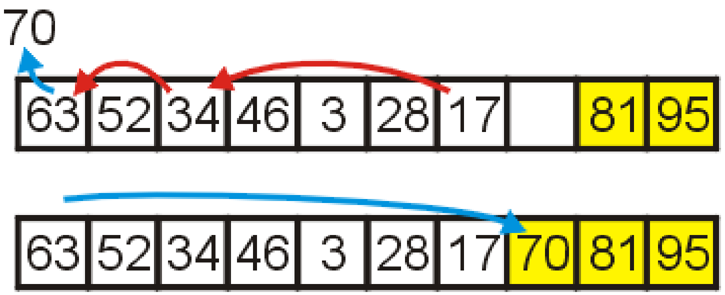
Repeat this process: pop the maximum element, and then insert it at the end of the array:



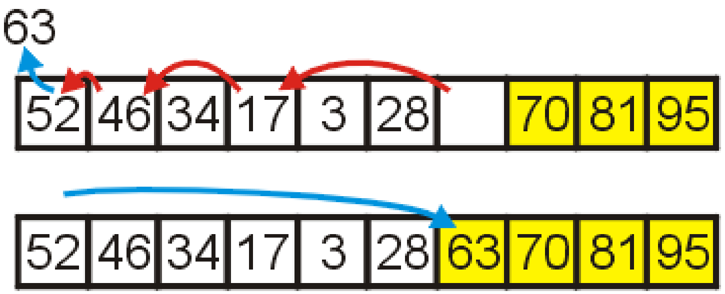
Heap Sort Example

Repeat this process

- Pop and append 70



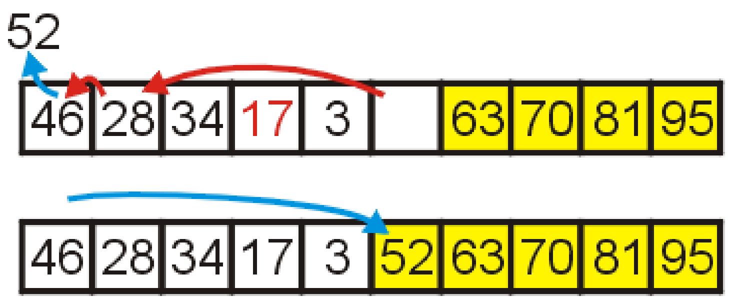
- Pop and append 63



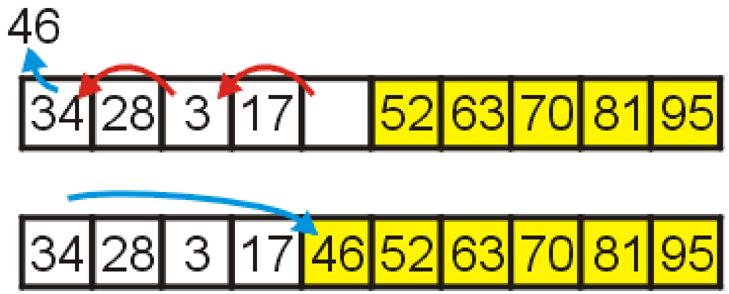
Heap Sort Example

We have the 4 largest elements in order

- Pop and append 52



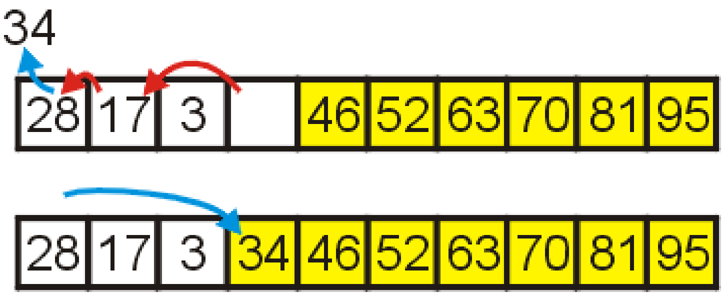
- Pop and append 46



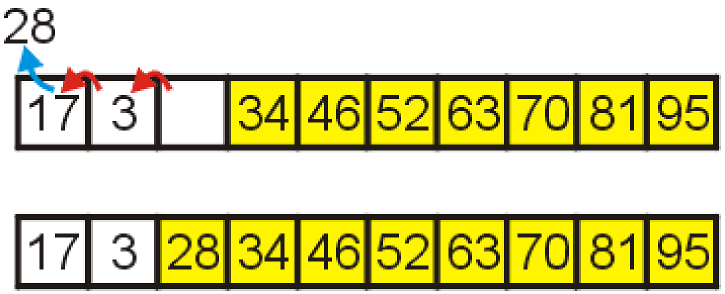
Heap Sort Example

Continuing...

– Pop and append 34

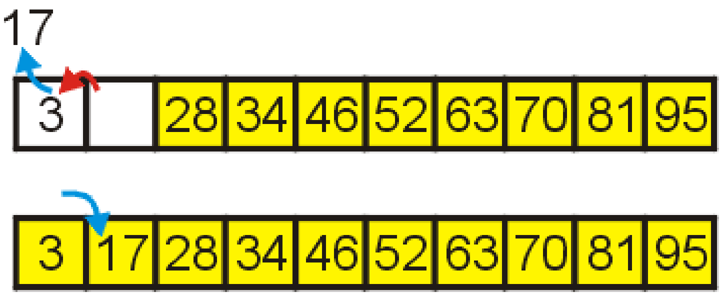


– Pop and append 28



Heap Sort Example

Finally, we can pop 17, insert it into the 2nd location, and the resulting array is sorted



Run-time

There are no worst-case scenarios for heap sort

– Dequeuing from the heap will always require $\mathbf{O}(\ln(n))$ time

Best case: if all or most entries are identical, then the run time is $\mathbf{\Theta}(n)$

– Why?

Run-time Summary

The following table summarizes the run-times of heap sort

Case	Run Time	Comments
Worst	$\mathbf{O}(n \ln(n))$	No worst case
Average	$\mathbf{O}(n \ln(n))$	
Best	$\mathbf{\Theta}(n)$	All or most entries are the same

Outline

- Introduction
- Inversions
- Insertion sort
- Bubble sort
- Heap sort
- Merge sort
- Quicksort
- Bucket sort
- Radix sort

Merge Sort

The merge sort algorithm is defined recursively:

- If the list is of size 1, it is sorted—we are done;
- Otherwise:
  - Divide an unsorted list into two sub-lists,
  - Sort each sub-list recursively using merge sort, and
  - Merge the two sorted sub-lists into a single sorted list

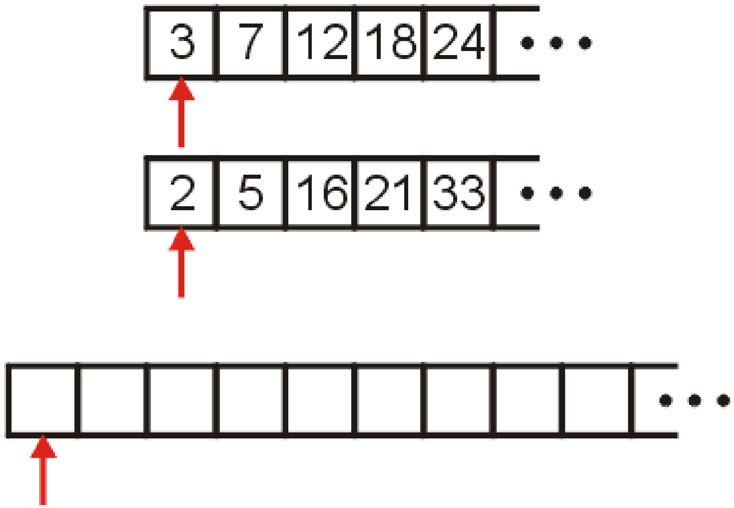
This strategy is called *divide-and-conquer*

Question: How can we merge two sorted sub-lists into a single sorted list?

Merging Example

Consider the two sorted arrays and an empty array

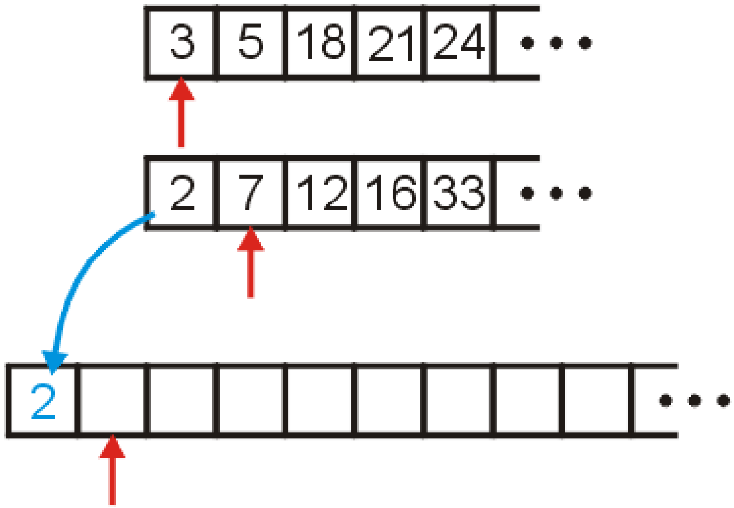
Define three indices at the start of each array



Merging Example

We compare 2 and 3: 2 < 3

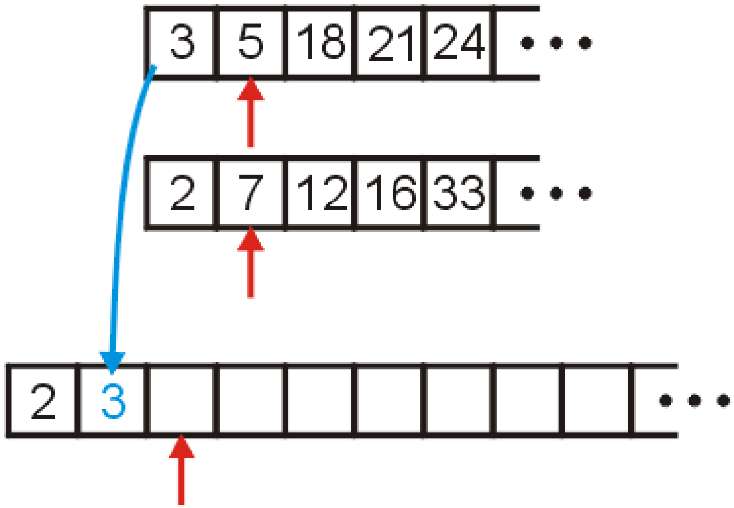
- Copy 2 down
- Increment the corresponding indices



Merging Example

We compare 3 and 7

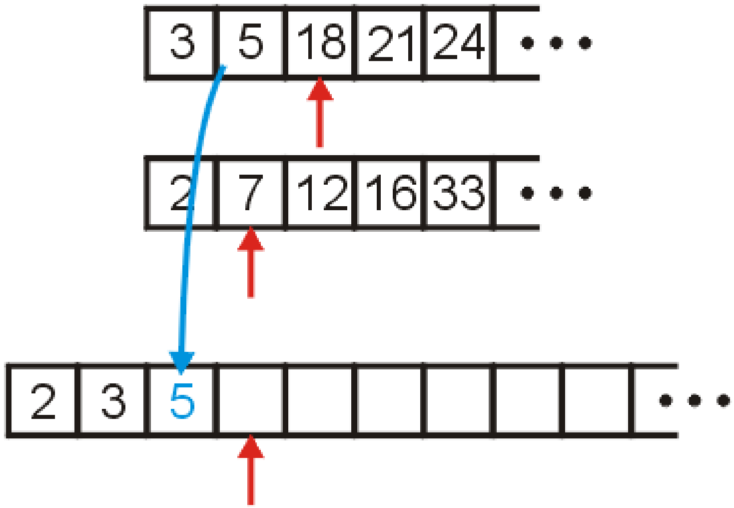
- Copy 3 down
- Increment the corresponding indices



Merging Example

We compare 5 and 7

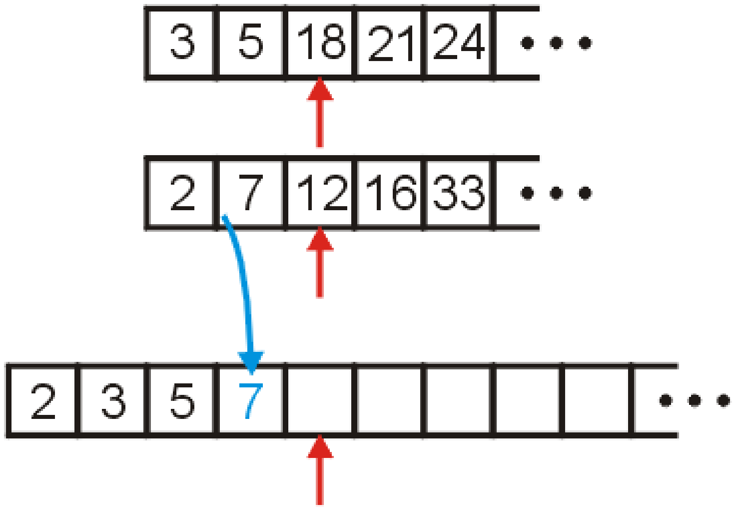
- Copy 5 down
- Increment the appropriate indices



Merging Example

We compare 18 and 7

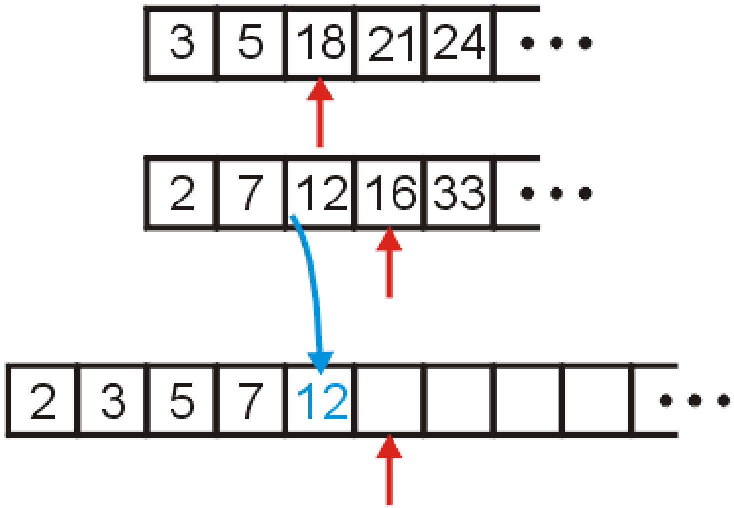
- Copy 7 down
- Increment...



Merging Example

We compare 18 and 12

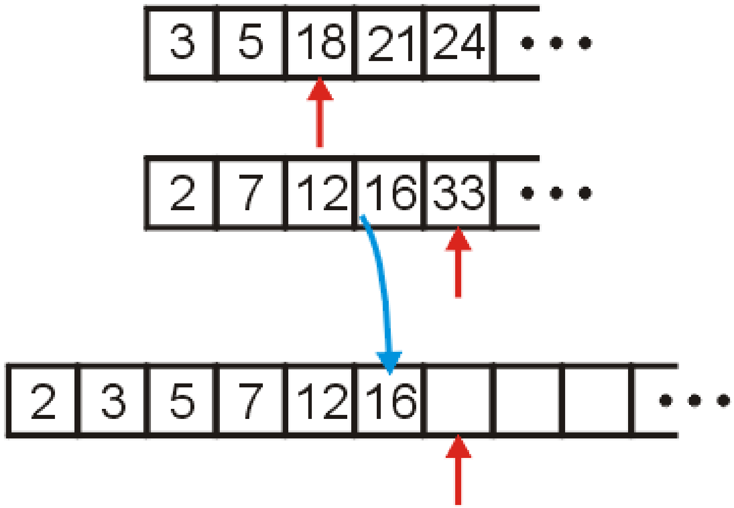
- Copy 12 down
- Increment...



Merging Example

We compare 18 and 16

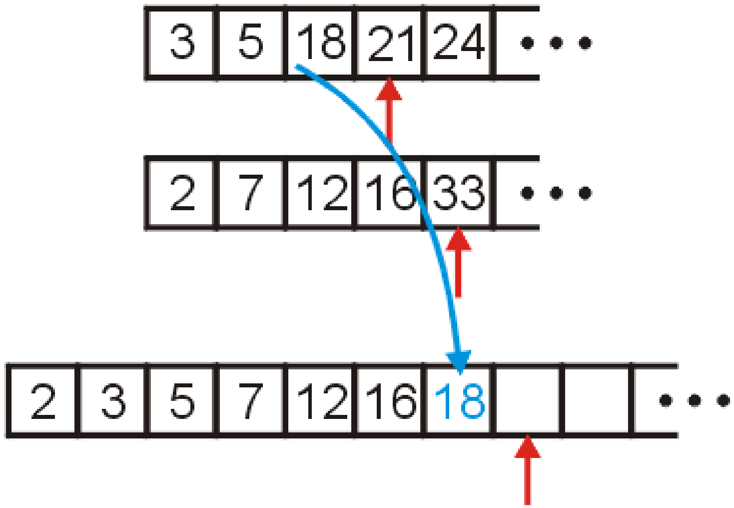
- Copy 16 down
- Increment...



Merging Example

We compare 18 and 33

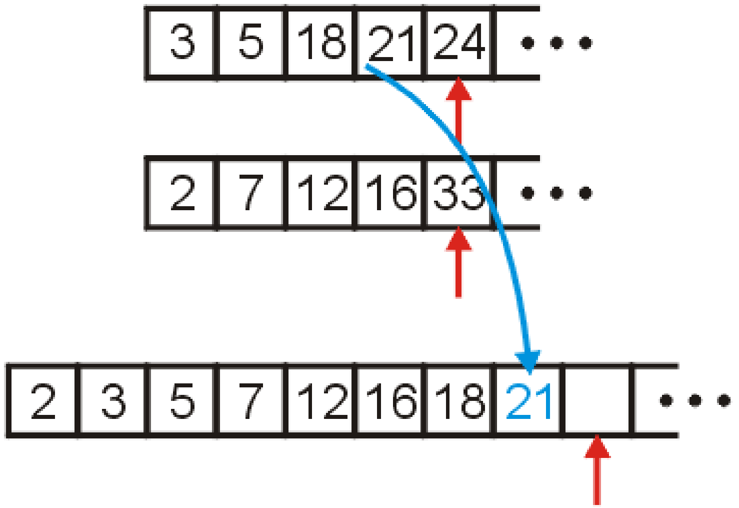
- Copy 18 down
- Increment...



Merging Example

We compare 21 and 33

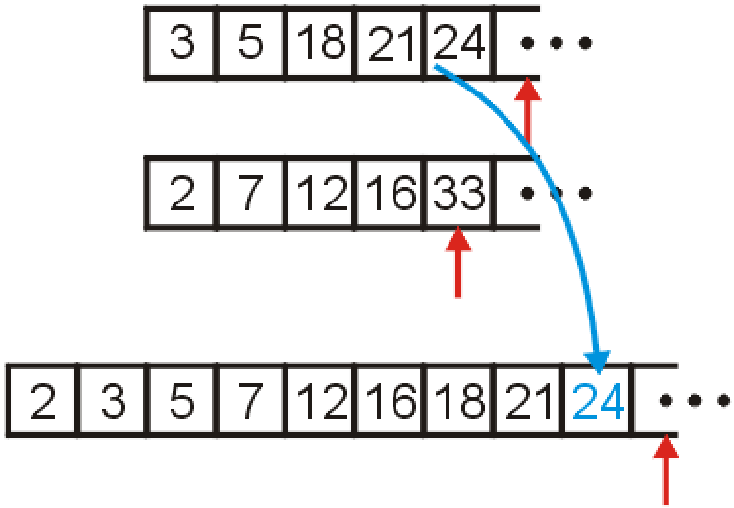
- Copy 21 down
- Increment...



Merging Example

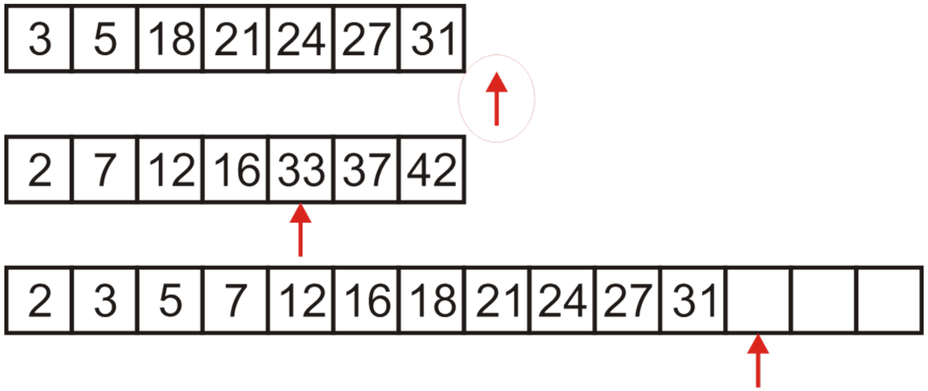
We compare 24 and 33

- Copy 24 down
- Increment...

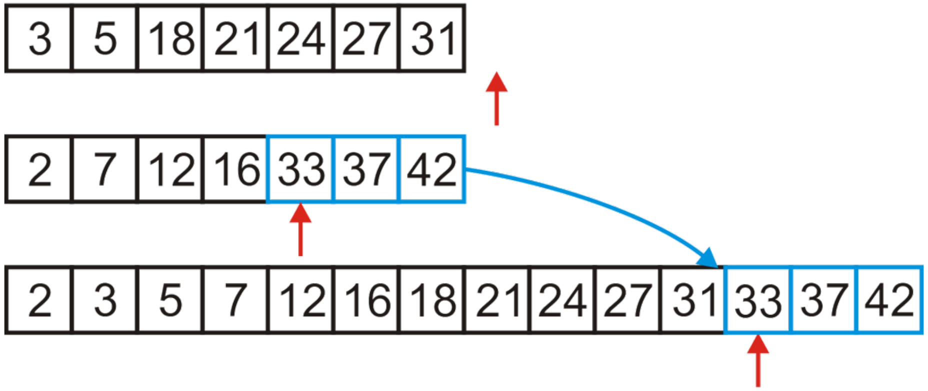


Merging Example

We would continue until we have passed beyond the limit of one of the two arrays



After this, we simply copy over all remaining entries in the non-empty array



Merging Two Lists

Programming a merge is straight-forward:

- the sorted arrays, `array1` and `array2`, are of size `n1` and `n2`, respectively, and
- we have an empty array, `arrayout`, of size `n1 + n2`

Define three variables

```
int i1 = 0, i2 = 0, k = 0;
```

which index into these three arrays

Merging Two Lists

We can then run the following loop:

```
#include <cassert>
//...
int i1 = 0, i2 = 0, k = 0;

while ( i1 < n1 && i2 < n2 ) {
    if ( array1[i1] < array2[i2] ) {
        arrayout[k] = array1[i1];
        ++i1;
    } else {
        assert( array1[i1] >= array2[i2] );
        arrayout[k] = array2[i2];
        ++i2;
    }
    ++k;
}
```

Merging Two Lists

We’re not finished yet, we have to empty out the remaining array

```
for ( ; i1 < n1; ++i1, ++k ) {
    arrayout[k] = array1[i1];
}

for ( ; i2 < n2; ++i2, ++k ) {
    arrayout[k] = array2[i2];
}
```

Analysis of merging

Time: we have to copy $n_1 + n_2$ elements

- Hence, merging may be performed in $\Theta(n_1 + n_2)$ time
- If the arrays are approximately the same size, $n = n_1 \approx n_2$, we can say that the run time is $\Theta(n)$

Space: we cannot merge two arrays in-place

- This algorithm always required the allocation of a new array
- Therefore, the memory requirements are also $\Theta(n)$

The Algorithm

The merge sort algorithm is defined recursively:

- If the list is of size 1, it is sorted—we are done;
- Otherwise:
  - Divide an unsorted list into two sub-lists,
  - Sort each sub-list recursively using merge sort, and
  - Merge the two sorted sub-lists into a single sorted list

In practice:

- If the list size is less than a threshold, use an algorithm like insertion sort
- Otherwise:
  - Divide…

Implementation

Suppose we already have a function

```
template <typename Type>
void merge( Type *array, int a, int b, int c );
```

that assumes that the entries

`array[a]` through `array[b - 1]`, and

`array[b]` through `array[c - 1]`

are sorted and merges these two sub-arrays into a single sorted array from index a through index `c - 1`, inclusive

Implementation

For example, given the array,

9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
13	77	49	35	61	3	23	48	73	89	95	17	32	37	57	94	99	28	15	55	7	51	88	97	62

a call to

```
void merge( array, 14, 20, 26 );
```

merges the two sub-lists

9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
13	77	49	35	61	3	23	48	73	89	95	17	32	37	57	94	99	28	15	55	7	51	88	97	62

forming

9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
13	77	49	35	61	3	17	23	32	37	48	57	73	89	94	95	99	28	15	55	7	51	88	97	62

Implementation

We implement a function

```
template <typename Type>
void merge_sort( Type *array, int first, int last );
```

that will sort the entries in the positions `first <= i` and `i < last`

- If the number of entries is less than N, call insertion sort
- Otherwise:
  - Find the mid-point,
  - Call merge sort recursively on each of the halves, and
  - Merge the results

Implementation

```
template <typename Type>
void merge_sort( Type *array, int first, int last ) {
    if ( last - first <= N ) {
        insertion_sort( array, first, last );
    } else {
        int midpoint = (first + last)/2;

        merge_sort( array, first, midpoint );
        merge_sort( array, midpoint, last );
        merge( array, first, midpoint, last );
    }
}
```

Example

Consider the following is of unsorted array of 25 entries

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

We will call insertion sort if the list being sorted of size $N = 6$ or less

Example

We call `merge_sort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

```
merge_sort( array,  0, 25 )
```

Example

We are calling `merge_sort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

First, $25 - 0 > 6$, so find the midpoint and call `merge_sort` recursively

```
midpoint = (0 + 25)/2; // == 12
merge_sort( array, 0, 12 );
```

`merge_sort( array,  0, 25 )`

Example

We are now executing `merge_sort( array, 0, 12 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

First, $12 - 0 > 6$, so find the midpoint and call `merge_sort` recursively

```
midpoint = (0 + 12)/2; // == 6
merge_sort( array, 0, 6 );
```

```
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We are now executing `merge_sort( array, 0, 6 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

Now, $6 - 0 \le 6$, so find we call insertion sort

```
merge_sort( array,  0,  6 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from `0` to `5`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

```
insertion_sort( array, 0, 6 )
merge_sort( array,  0,  6 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from `0` to `5`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

– This function call completes and so we exit

```
insertion_sort( array, 0, 6 )
merge_sort( array,  0,  6 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

This call to `merge_sort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

```
merge_sort( array,  0,  6 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We return to continue executing `merge_sort( array, 0, 12 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

We continue calling

```
midpoint = (0 + 12)/2; // == 6
merge_sort( array, 0, 6 );
merge_sort( array, 6, 12 );
```

```
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We are now executing `merge_sort( array, 6,  12 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

Now, $12 - 6 \le 6$, so find we call insertion sort

```
merge_sort( array,  6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from 6 to 11

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

```
insertion_sort( array, 6, 12 )
merge_sort( array,  6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from 6 to 11

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	3	23	37	73	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

– This function call completes and so we exit

```
insertion_sort( array, 6, 12 )
merge_sort( array,  6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

This call to `merge_sort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	3	23	37	73	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

```
merge_sort( array,  6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We return to continue executing `merge_sort( array, 0, 12 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	3	23	37	73	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

We continue calling

```
midpoint = (0 + 12)/2; // == 6
merge_sort( array, 0, 6 );
merge_sort( array, 6, 12 );
merge( array, 0, 6, 12 );
```

```
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We are executing `merge( array, 0, 6, 12 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	3	23	37	73	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

These two sub-arrays are merged together

```
merge( array, 0, 6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We are executing `merge( array, 0, 6, 12 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

These two sub-arrays are merged together

- This function call exists

```
merge( array, 0, 6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We return to executing `merge_sort( array, 0, 12 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

We are finished calling this function as well

```
midpoint = (0 + 12)/2; // == 6
merge_sort( array, 0, 6 );
merge_sort( array, 6, 12 );
merge( array, 0, 6, 12 );
```

Consequently, we exit

```
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We return to executing `merge_sort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

We continue calling

```
midpoint = (0 + 25)/2; // == 12
merge_sort( array, 0, 12 );
merge_sort( array, 12, 25 );
```

```
merge_sort( array,  0, 25 )
```

Example

We are now executing `merge_sort( array, 12, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

First, $25 - 12 > 6$, so find the midpoint and call `merge_sort` recursively

```
midpoint = (12 + 25)/2; // == 18
merge_sort( array, 12, 18 );
```

```
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are now executing `merge_sort( array, 12, 18 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

Now, $18 - 12 \le 6$, so find we call insertion sort

```
merge_sort( array, 12, 18 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from 12 to 17

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	57	99	17	32	94	28	15	55	7	51	88	97	62

```
insertion_sort( array, 12, 18 )
merge_sort( array, 12, 18 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from 12 to 17

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	15	55	7	51	88	97	62

– This function call completes and so we exit

```
insertion_sort( array, 12, 18 )
merge_sort( array, 12, 18 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

This call to `merge_sort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	15	55	7	51	88	97	62

```
merge_sort( array, 12, 18 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We return to continue executing `merge_sort( array, 12, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	15	55	7	51	88	97	62

We continue calling

```
midpoint = (12 + 25)/2; // == 18
merge_sort( array, 12, 18 );
merge_sort( array, 18, 25 );
```

```
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are now executing `merge_sort( array, 18, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	15	55	7	51	88	97	62

First, $25 - 18 > 6$, so find the midpoint and call `merge_sort` recursively

```
midpoint = (18 + 25)/2; // == 21
merge_sort( array, 18, 21 );
```

```
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are now executing `merge_sort( array, 18, 21 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	15	55	7	51	88	97	62

Now, $21 - 18 \leq 6$, so find we call insertion sort

```
merge_sort( array, 18, 21 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from `18` to `20`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	15	55	7	51	88	97	62

```
insertion_sort( array, 18, 21 )
merge_sort( array, 18, 21 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from `18` to `20`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	88	97	62

- This function call completes and so we exit

```
insertion_sort( array, 18, 21 )
merge_sort( array, 18, 21 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

This call to `merge_sort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	88	97	62

```
merge_sort( array, 18, 21 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We return to executing `merge_sort( array, 18, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	88	97	62

We continue calling

```
midpoint = (18 + 25)/2; // == 21
merge_sort( array, 18, 21 );
merge_sort( array, 21, 25 );
```

```
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are now executing `merge_sort( array, 21, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	88	97	62

Now, $25 - 21 \leq 6$, so find we call insertion sort

```
merge_sort( array, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from 21 to 24

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	88	97	62

```
insertion_sort( array, 21, 25 )
merge_sort( array, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from 21 to 24

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	62	88	97

– This function call completes and so we exit

```
insertion_sort( array, 21, 25 )
merge_sort( array, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

This call to `merge_sort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	62	88	97

```
merge_sort( array, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We return to continue executing `merge_sort( array, 18, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	62	88	97

We continue calling

```
midpoint = (18 + 25)/2; // == 21
merge_sort( array, 18, 21 );
merge_sort( array, 21, 25 );
merge( array, 18, 21, 25 );
```

```
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are executing `merge( array, 18, 21, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	51	62	88	97

These two sub-arrays are merged together

```
merge( array, 18, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are executing `merge( array, 18, 21, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	51	55	62	88	97

These two sub-arrays are merged together

– This function call exists

```
merge( array, 18, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We return to executing `merge_sort( array, 18, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	51	55	62	88	97

We are finished calling this function as well

```
midpoint = (18 + 25)/2; // == 21
merge_sort( array, 18, 21 );
merge_sort( array, 21, 25 );
merge( array, 18, 21, 25 );
```

Consequently, we exit

```
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We return to continue executing `merge_sort( array, 12, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	51	55	62	88	97

We continue calling

```
midpoint = (12 + 25)/2; // == 18
merge_sort( array, 12, 18 );
merge_sort( array, 18, 25 );
merge( array, 12, 18, 25 );
```

```
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are executing `merge( array, 12, 18, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	51	55	62	88	97

These two sub-arrays are merged together

```
merge( array, 12, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are executing `merge( array, 12, 18, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	7	15	17	28	32	51	55	57	62	88	94	97	99

These two sub-arrays are merged together

– This function call exists

```
merge( array, 12, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We return to executing `merge_sort( array, 12, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	7	15	17	28	32	51	55	57	62	88	94	97	99

We are finished calling this function as well

```
midpoint = (12 + 25)/2; // == 18
merge_sort( array, 12, 18 );
merge_sort( array, 18, 25 );
merge( array, 12, 18, 25 );
```

Consequently, we exit

```
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We return to continue executing `merge_sort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	7	15	17	28	32	51	55	57	62	88	94	97	99

We continue calling

```
midpoint = (0 + 25)/2; // == 12
merge_sort( array, 0, 12 );
merge_sort( array, 12, 25 );
merge( array, 0, 12, 25 );
```

`merge_sort( array,  0, 25 )`

Example

We are executing `merge( array, 0, 12, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	7	15	17	28	32	51	55	57	62	88	94	97	99

These two sub-arrays are merged together

```
merge( array, 0, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are executing `merge( array, 0, 12, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

These two sub-arrays are merged together
- This function call exists

```
merge( array, 0, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We return to executing `merge_sort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

We are finished calling this function as well

```
midpoint = (0 + 25)/2; // == 12
merge_sort( array, 0, 12 );
merge_sort( array, 12, 25 );
merge( array, 0, 12, 25 );
```

Consequently, we exit

`merge_sort( array,  0, 25 )`

Example

The array is now sorted

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

Run-time Analysis of Merge Sort

The time required to sort an array of size $n > 1$ is:

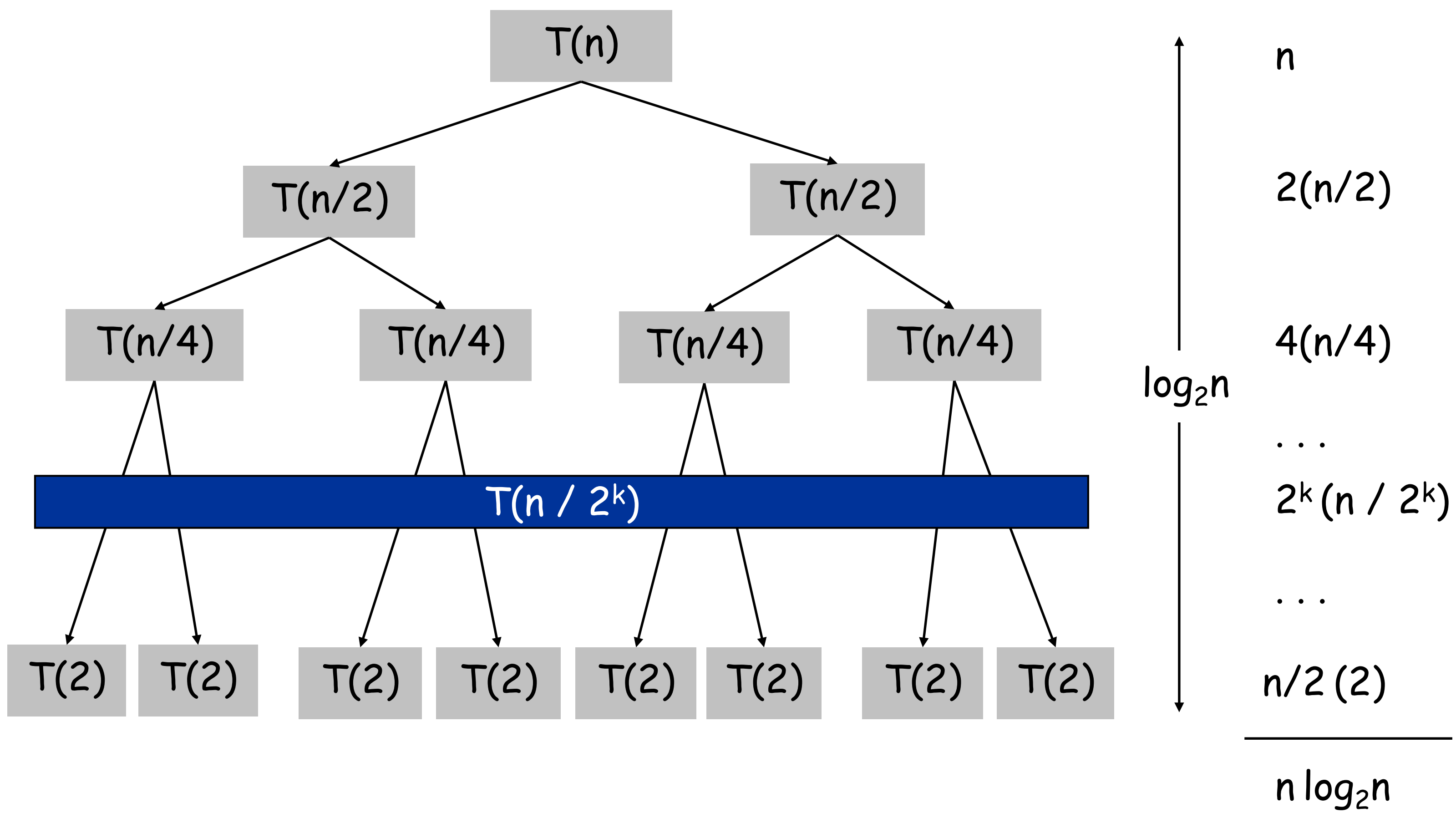
- the time required to sort the first half,
- the time required to sort the second half, and
- the time required to merge the two lists

That is: $$\mathrm{T}(n) = \begin{cases} \boldsymbol{\Theta}(1) & n = 1 \\ 2\,\mathrm{T}\left(\frac{n}{2}\right) + \boldsymbol{\Theta}(n) & n > 1 \end{cases}$$

Solution: $\mathrm{T}(n) = \Theta(n \ln(n))$

Proof by Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



Run-time Summary

The following table summarizes the run-times of merge sort

Case	Run Time	Comments
Worst	$\Theta(n \ln(n))$	No worst case
Average	$\Theta(n \ln(n))$	
Best	$\Theta(n \ln(n))$	No best case

Comments

In practice, merge sort is faster than heap sort, though they both have the same asymptotic run times

Merge sort requires an additional array

- Heap sort does not require

Next we see quick sort

- Faster, on average, than either heap or quick sort
- Requires $\mathbf{o}(n)$ additional memory

Merge Sort

The (likely) first proposal of merge sort was by John von Neumann in 1945

- The creator of the *von Neumann architecture* used by all modern computers:

http://en.wikipedia.org/wiki/Von_Neumann

Outline

- Introduction
- Inversions
- Insertion sort
- Bubble sort
- Heap sort
- Merge sort
- Quicksort
- Bucket sort
- Radix sort

Quicksort

Merge sort splits the array into two sub-lists and sorts them

- It splits the larger problem into two sub-problems based on *location* in the array

Consider the following alternative:

- Chose an object in the array and partition the remaining objects into two groups relative to the chosen entry

Quicksort

For example, given

80	38	95	84	66	10	79	**44**	26	87	96	12	43	81	3

we can select the middle entry, 44, and sort the remaining entries into two groups, those less than 44 and those greater than 44:

38	10	26	12	43	3	**44**	80	95	84	66	79	87	96	81

Notice that 44 is now in the correct location if the list was sorted

- Proceed by recursively applying the algorithm to the first six and last eight entries

Run-time analysis

Like merge sort, we can either:

- Sort the sub-lists using quicksort
- If the size of the sub-list is sufficiently small, apply insertion sort

In the best case, the list will be split into two approximately equal sub-lists, and thus, the run time could be very similar to that of merge sort: $\Theta(n \ln(n))$

What happens if we don't get that lucky?

Worst-case scenario

Suppose we choose the middle element as our pivot and we try ordering a sorted list:

80	38	95	84	66	10	79	**2**	26	87	96	12	43	81	3

Using 2, we partition into

2	80	38	95	84	66	10	79	26	87	96	12	43	81	3

We still have to sort a list of size $n - 1$

The run time is $\mathrm{T}(n) = \mathrm{T}(n - 1) + \boldsymbol{\Theta}(n) = \boldsymbol{\Theta}(n^2)$

– Thus, the run time drops from $n \ln(n)$ to n^2

Worst-case scenario

Our goal is to choose the median element in the list as our pivot:

80	38	95	84	**66**	10	79	2	26	87	96	12	43	81	3

Using the median element 66, we can get two equal-size sub-lists

3	38	43	12	2	10	26	**66**	79	87	96	84	95	81	80

Unfortunately, median is difficult to find

Median-of-three

Consider another strategy:

– Choose the median of the first, middle, and last entries in the list

This will usually give a better approximation of the actual median

80	38	95	84	99	10	79	44	26	87	96	12	43	81	3

Median-of-three

If we choose a random pivot, this will, on average, divide a set of n items into two sets of size $1/4\ n$ and $3/4\ n$

Choosing the median-of-three, this will, on average, divide the n items into two sets of size $5/16\ n$ and $11/16\ n$

- Median-of-three helps speed the algorithm
- This requires order statistics:

$$2\int_0^{\frac{1}{2}} x \cdot \left(6x\left(1-x\right)\right) dx = \frac{5}{16} = 0.3125$$

Implementation

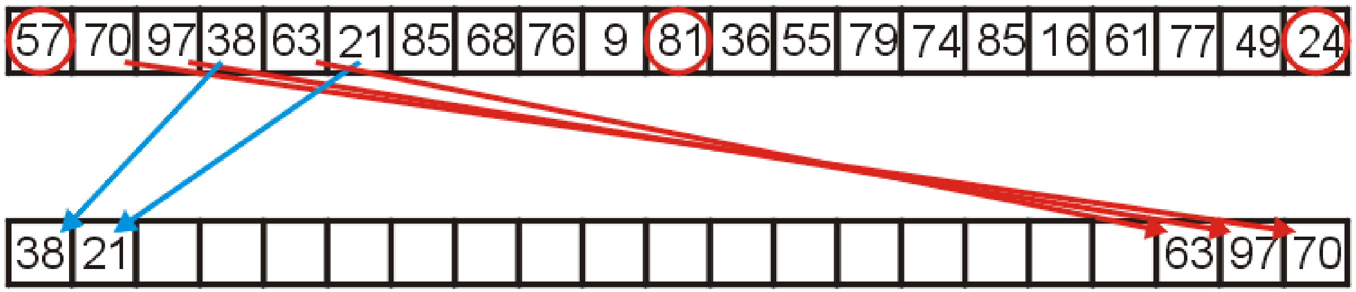
If we choose to allocate memory for an additional array, we can implement the partitioning by

- copying elements either to the front or the back of the additional array
- placing the pivot into the resulting hole

Implementation

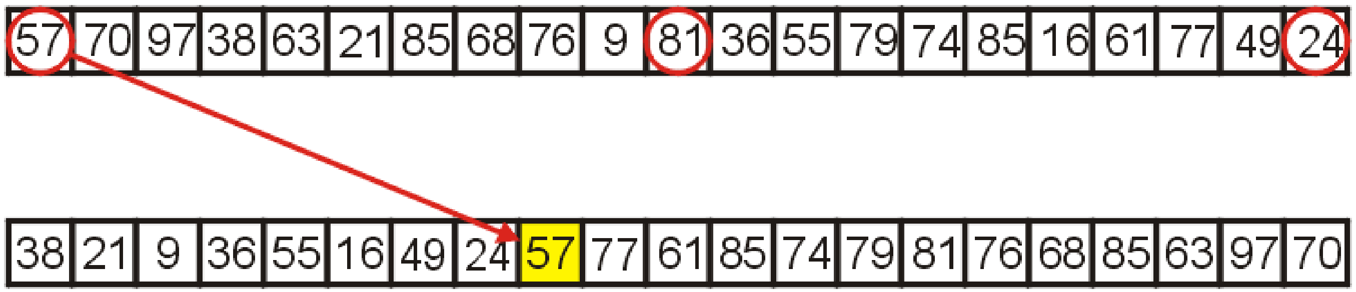
For example, consider the following:

- 57 is the median-of-three
- we go through the remaining elements, assigning them either to the front or the back of the second array



Implementation

Once we are finished, we copy the median-of-three, 57, into the resulting hole



Implementation

Can we implement quicksort in place?

Yes!

- Swap the pivot to the last slot of the list
- We repeatedly try to find two entries:
  - Staring from the front: an entry larger than the pivot
  - Starting from the back: an entry smaller than the pivot
- Such two entries are out of order, so we swap them
- Repeat until all the entries are in order
- Move the leftmost entry larger than the pivot into the last slot of the list and fill the hole with the pivot

Quicksort example

Consider the following unsorted array of 25 entries

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

We will call insertion sort if the list being sorted of size $N = 6$ or less

Quicksort example

We call `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	**57**	99	17	32	94	28	15	55	7	51	88	97	**62**

First, $25 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 25)/2; // == 12
```

`quicksort( array,  0, 25 )`

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	**62**	99	17	32	94	28	15	55	7	51	88	97	

First, $25 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 25)/2; // == 12
pivot = 57;
```

`quicksort( array, 0, 25 )`

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	62	99	17	32	94	28	15	55	7	51	88	97	

Starting from the front and back:

- Find the next element greater than the pivot
- The last element less than the pivot

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	**77**	49	35	61	48	73	23	95	3	89	37	62	99	17	32	94	28	15	55	7	**51**	88	97	

Searching forward and backward:

```
low = 1;
high = 21;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	61	48	73	23	95	3	89	37	62	99	17	32	94	28	15	55	7	77	88	97	

Searching forward and backward:

```
low = 1;
high = 21;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	61	48	73	23	95	3	89	37	62	99	17	32	94	28	15	55	7	77	88	97	

Continue searching

```
low = 4;
high = 20;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	73	23	95	3	89	37	62	99	17	32	94	28	15	55	61	77	88	97	

Continue searching

```
low = 4;
high = 20;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	**73**	23	95	3	89	37	62	99	17	32	94	28	15	**55**	61	77	88	97	

Continue searching

```
low = 6;
high = 19;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	95	3	89	37	62	99	17	32	94	28	15	73	61	77	88	97	

Continue searching

```
low = 6;
high = 19;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	**95**	3	89	37	62	99	17	32	94	28	**15**	73	61	77	88	97	

Continue searching

```
low = 8;
high = 18;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	89	37	62	99	17	32	94	28	95	73	61	77	88	97	

Continue searching

```
low = 8;
high = 18;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	89	37	62	99	17	32	94	28	95	73	61	77	88	97	

Continue searching

```
low = 10;
high = 17;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	62	99	17	32	94	89	95	73	61	77	88	97	

Continue searching

```
low = 10;
high = 17;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	62	99	17	32	94	89	95	73	61	77	88	97	

Continue searching

```
low = 12;
high = 15;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	99	17	62	94	89	95	73	61	77	88	97	

Continue searching

```
low = 12;
high = 15;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	99	17	62	94	89	95	73	61	77	88	97	

Continue searching

```
low = 13;
high = 14;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	17	99	62	94	89	95	73	61	77	88	97	

Continue searching

```
low = 13;
high = 14;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	17	99	62	94	89	95	73	61	77	88	97	

Continue searching

```
low = 14;
high = 13;
```

Now, `low > high`, so we stop

`pivot = 57;`

`quicksort( array,  0, 25 )`

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	17	**57**	62	94	89	95	73	61	77	88	97	99

Continue searching

```
low = 14;
high = 13;
```

Now, `low > high`, so we stop

`pivot = 57;`

`quicksort( array,  0, 25 )`

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	17	**57**	62	94	89	95	73	61	77	88	97	99

We now begin calling quicksort recursively on the first half

```
quicksort( array, 0, 14 );
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	17	57	62	94	89	95	73	61	77	88	97	99

First, $14 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 14)/2; // == 7
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	**23**	15	3	28	37	32	**17**	57	62	94	89	95	73	61	77	88	97	99

First, $14 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 14)/2; // == 7
pivot = 17
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32		57	62	94	89	95	73	61	77	88	97	99

First, $14 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 14)/2; // == 7
```

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Starting from the front and back:

- Find the next element greater than the pivot
- The last element less than the pivot

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 1;
high = 9;
```

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	49	35	7	48	55	23	15	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 1;
high = 9;
```

Swap them

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	49	35	7	48	55	23	15	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 2;
high = 8;
```

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	15	35	7	48	55	23	49	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 2;
high = 8;
```

Swap them

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	15	35	7	48	55	23	49	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 3;
high = 4;
```

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	15	7	35	48	55	23	49	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 3;
high = 4;
```

Swap them

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	15	7	35	48	55	23	49	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 4;
high = 3;
```

Now, `low > high`, so we stop

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	15	7	**17**	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

We continue calling quicksort recursively

```
quicksort( array, 0, 4 );
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 4 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	15	7	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

Now, $4 - 0 \leq 6$, so find we call insertion sort

```
quicksort( array,  0,  4 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from `0` to `3`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	15	7	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

```
insertion_sort( array, 0, 4 )
quicksort( array,  0,  4 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from `0` to `3`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

– This function call completes and so we exit

```
insertion_sort( array, 0, 4 )
quicksort( array,  0,  4 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

This call to `quicksort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

```
quicksort( array,  0,  4 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	**17**	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

We continue calling quicksort recursively on the second half

```
quicksort( array, 0,  4 );
quicksort( array, 5, 14 );
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

First, $14 - 5 > 6$, so find the midpoint and the pivot

```
midpoint = (5 + 14)/2; // == 9
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	**48**	55	23	49	**51**	28	37	32	**35**	57	62	94	89	95	73	61	77	88	97	99

First, $14 - 5 > 6$, so find the midpoint and the pivot

```
midpoint = (5 + 14)/2; // == 9
pivot = 48
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	**35**	55	23	49	**51**	28	37	32		57	62	94	89	95	73	61	77	88	97	99

First, $14 - 5 > 6$, so find the midpoint and the pivot

```
midpoint = (5 + 14)/2; // == 9
pivot = 48
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	35	55	23	49	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Starting from the front and back:

- Find the next element greater than the pivot
- The last element less than the pivot

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	35	55	23	49	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 6;
high = 12;
```

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	35	32	23	49	51	28	37	55		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 6;
high = 12;
```

Swap them

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	35	32	23	**49**	51	28	**37**	55		57	62	94	89	95	73	61	77	88	97	99

Continue searching

```
low = 8;
high = 11;
```

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	35	32	23	37	51	28	49	55		57	62	94	89	95	73	61	77	88	97	99

Continue searching

```
low = 8;
high = 11;
```

Swap them

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	35	32	23	37	**51**	**28**	49	55		57	62	94	89	95	73	61	77	88	97	99

Continue searching

```
low = 8;
high = 11;
```

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	35	32	23	37	28	51	49	55		57	62	94	89	95	73	61	77	88	97	99

Continue searching

```
low = 8;
high = 11;
```

Swap them

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	35	32	23	37	28	51	49	55		57	62	94	89	95	73	61	77	88	97	99

Continue searching

```
low = 8;
high = 11;
```

Now, `low > high`, so we stop

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	35	32	23	37	28	**48**	49	55	51	57	62	94	89	95	73	61	77	88	97	99

Continue searching

```
low = 8;
high = 11;
```

Now, `low > high`, so we stop

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	35	32	23	37	28	**48**	49	55	51	57	62	94	89	95	73	61	77	88	97	99

We now begin calling quicksort recursively on the first half

```
quicksort( array, 5, 10 );
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	35	32	23	37	28	**48**	49	55	51	57	62	94	89	95	73	61	77	88	97	99

We now begin calling quicksort recursively

```
quicksort( array, 5, 10 );
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 5, 10 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	35	32	23	37	28	48	49	55	51	57	62	94	89	95	73	61	77	88	97	99

Now, $10 - 5 \leq 6$, so find we call insertion sort

```
quicksort( array,  5, 10 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 5 to 9

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	35	32	23	37	28	48	49	55	51	57	62	94	89	95	73	61	77	88	97	99

```
insertion_sort( array, 5, 10 )
quicksort( array,  5, 10 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 5 to 9

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	55	51	57	62	94	89	95	73	61	77	88	97	99

– This function call completes and so we exit

```
insertion_sort( array, 5, 10 )
quicksort( array,  5, 10 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

This call to `quicksort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	55	51	57	62	94	89	95	73	61	77	88	97	99

```
quicksort( array,  5, 10 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 5, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	**48**	49	55	51	57	62	94	89	95	73	61	77	88	97	99

We continue calling quicksort recursively on the second half

```
quicksort( array, 5, 10 );
quicksort( array, 6, 14 );
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 11, 15 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	55	51	57	62	94	89	95	73	61	77	88	97	99

Now, $15 - 11 \leq 6$, so find we call insertion sort

```
quicksort( array,  6, 14 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 11 to 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	55	51	57	62	94	89	95	73	61	77	88	97	99

```
insertion_sort( array, 11, 14 )
quicksort( array, 11, 14 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 11 to 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	62	94	89	95	73	61	77	88	97	99

– This function call completes and so we exit

```
insertion_sort( array, 11, 14 )
quicksort( array, 11, 14 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

This call to `quicksort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	62	94	89	95	73	61	77	88	97	99

```
quicksort( array, 11, 14 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

This call to `quicksort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	62	94	89	95	73	61	77	88	97	99

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

This call to `quicksort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	62	94	89	95	73	61	77	88	97	99

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	**57**	62	94	89	95	73	61	77	88	97	99

We continue calling quicksort recursively on the second half

```
quicksort( array, 0, 14 );
quicksort( array, 15, 25 );
```

`quicksort( array,  0, 25 )`

Quicksort example

We are back to executing `quicksort( array, 15, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	62	94	89	95	73	61	77	88	97	99

First, $25 - 15 > 6$, so find the midpoint and the pivot

```
midpoint = (15 + 25)/2; // == 20
```

```
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 15, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	**61**	94	89	95	73	**99**	77	88	97	

First, $25 - 15 > 6$, so find the midpoint and the pivot

```
midpoint = (15 + 25)/2; // == 20
pivot = 62;
```

```
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 15, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	**61**	**94**	89	95	73	99	77	88	97	

Searching forward and backward:

```
low = 16;
high = 15;
```

Now, `low > high`, so we stop

```
pivot = 62;
```

```
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 15, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	**62**	89	95	73	99	77	88	97	94

Searching forward and backward:

```
low = 16;
high = 15;
```

Now, `low > high`, so we stop

- Note, this is the worst-case scenario
- The pivot is the second smallest element

```
pivot = 62;
```

```
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 15, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	89	95	73	99	77	88	97	94

We continue calling quicksort recursively on the first half

```
quicksort( array, 15, 16 );
```

```
quicksort( array, 15, 16 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 15, 16 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	89	95	73	99	77	88	97	94

Now, $16 - 15 \le 6$, so find we call insertion sort

```
quicksort( array, 15, 16 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort immediately returns

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	89	95	73	99	77	88	97	94

```
insertion_sort( array, 15, 16 )
quicksort( array, 15, 16 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

This call to `quicksort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	89	95	73	99	77	88	97	94

```
quicksort( array, 15, 16 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 15, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	**62**	89	95	73	99	77	88	97	94

We continue calling quicksort recursively on the second half

```
quicksort( array, 15, 16 );
quicksort( array, 17, 25 );
```

```
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	89	95	73	99	77	88	97	94

First, $25 - 17 > 6$, so find the midpoint and the pivot

```
midpoint = (17 + 25)/2; // == 21
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	**89**	95	73	99	**77**	88	97	**94**

First, $25 - 17 > 6$, so find the midpoint and the pivot

```
midpoint = (17 + 25)/2; // == 21
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	**89**	95	73	99	**77**	88	97	**94**

First, $25 - 17 > 6$, so find the midpoint and the pivot

```
midpoint = (17 + 25)/2; // == 21
pivot = 89
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	**77**	95	73	99	**94**	88	97	

First, $25 - 17 > 6$, so find the midpoint and the pivot

```
midpoint = (17 + 25)/2; // == 21
pivot = 89
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	77	**95**	73	99	94	**88**	97	

Searching forward and backward:

```
low = 18;
high = 22;
```

```
pivot = 89;
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	77	88	73	99	94	95	97	

Searching forward and backward:

```
low = 18;
high = 22;
```

Swap them

```
pivot = 89;
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	77	88	73	99	94	95	97	

Searching forward and backward:

```
low = 20;
high = 19;
```

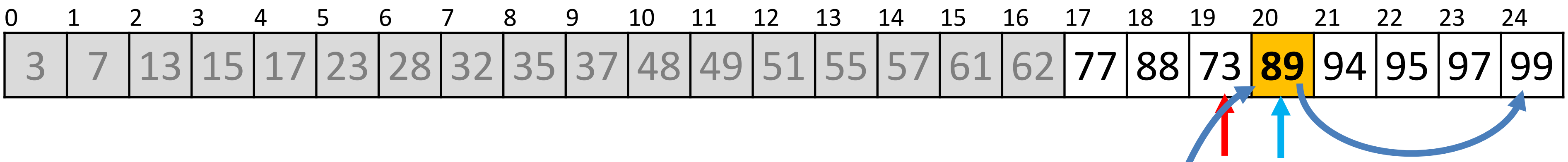
Now, `low > high`, so we stop

```
pivot = 89;
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 25 )`



Searching forward and backward:

```
low = 20;
high = 19;
```

Now, `low > high`, so we stop

```
pivot = 89;
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	77	88	73	**89**	94	95	97	99

We start by calling quicksort recursively on the first half

```
quicksort( array, 17, 20 );
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now executing `quicksort( array, 17, 20 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	77	88	73	89	94	95	97	99

Now, $4 - 0 \leq 6$, so find we call insertion sort

```
quicksort( array, 17, 20 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 17 to 19

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	77	88	73	89	94	95	97	99

```
insertion_sort( array, 17, 20 )
quicksort( array, 17, 20 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 17 to 19

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

– This function call completes and so we exit

```
insertion_sort( array, 17, 20 )
quicksort( array, 17, 20 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

This call to `quicksort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

```
quicksort( array, 17, 20 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 17, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	**89**	94	95	97	99

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 17, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	**89**	94	95	97	99

We continue by calling quicksort on the second half

```
quicksort( array, 17, 20 );
quicksort( array, 21, 25 );
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 21, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

Now, $25 - 21 \leq 6$, so find we call insertion sort

```
quicksort( array, 21, 25 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 21 to 24

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

```
insertion_sort( array, 21, 25 )
quicksort( array, 21, 25 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 21 to 24

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

- In this case, the sub-array was already sorted
- This function call completes and so we exit

```
insertion_sort( array, 21, 25 )
quicksort( array, 21, 25 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

This call to `quicksort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

```
quicksort( array, 21, 25 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

This call to `quicksort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

This call to `quicksort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

```
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

This call to `quicksort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

```
quicksort( array,  0, 25 )
```

Quicksort example

We have now used quicksort to sort this array of 25 entries

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

Memory Requirements

The additional memory?

- Function call stack
  - Each recursive function call places its local variables, parameters, *etc.*, on a stack
- Average case: the depth of the recursion is $\Theta(\ln(n))$
- Worst case: the depth of the recursion is $\Theta(n)$

Run-time Summary

To summarize all three $\Theta(n \ln(n))$ algorithms

	Average Run Time	Worst-case Run Time	Average Memory	Worst-case Memory
Heap Sort	$\mathrm{O}(n \ln(n))$		$\Theta(1)$	
Merge Sort	$\Theta(n \ln(n))$		$\Theta(n)$	
Quicksort	$\Theta(n \ln(n))$	$\Theta(n^2)$	$\Theta(\ln(n))$	$\Theta(n)$

Outline

- Introduction
- Inversions
- Insertion sort
- Bubble sort
- Heap sort
- Merge sort
- Quicksort
- Bucket sort
- Radix sort

Supporting example

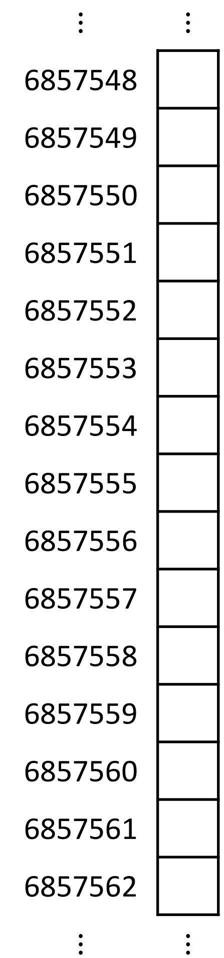
Suppose we are sorting a large number of local phone numbers, say, approximately four million numbers.

Consider the following scheme:

- Create a bit vector with $10\,000\,000$ bits
  - This requires $10^7/1024/1024/8 \approx 1.2$ MiB
- Set each bit to 0 (indicating false)
- For each phone number, set the bit indexed by the phone number to 1 (true)
- Once each phone number has been checked, walk through the array and for each bit which is 1, record that number

Supporting example

For example, consider this section within the bit array



Supporting example

For each phone number, set the corresponding bit

– For example, 685-7550 is a phone number

⋮	⋮
6857548	✓
6857549	✓
6857550	
6857551	
6857552	
6857553	✓
6857554	
6857555	✓
6857556	
6857557	
6857558	✓
6857559	
6857560	
6857561	✓
6857562	✓
⋮	⋮

Supporting example

For each phone number, set the corresponding bit

- For example, 685-7550 is a phone number

⋮	⋮
6857548	✓
6857549	✓
6857550	✓
6857551	
6857552	
6857553	✓
6857554	
6857555	✓
6857556	
6857557	
6857558	✓
6857559	
6857560	
6857561	✓
6857562	✓
⋮	⋮

Supporting example

At the end, we just take all the numbers out that were checked:

…,685-7548, 685-7549, 685-7550,
685-7553, 685-7555, 685-7558,
685-7561, 685-5762, …

⋮	⋮
6857548	✓
6857549	✓
6857550	✓
6857551	
6857552	
6857553	✓
6857554	
6857555	✓
6857556	
6857557	
6857558	✓
6857559	
6857560	
6857561	✓
6857562	✓
⋮	⋮

Supporting example

In this example, the number of phone numbers ($4\ 000\ 000$) is comparable to the size of the array ($10\ 000\ 000$)

The run time of such an algorithm is $\Theta(n)$:

- we make one pass through the data,
- we make one pass through the array and extract the phone numbers which are true

Algorithm

We will term each entry in the bit vector a *bucket*

The algorithm is a simplified version of *bucket sort*, sometimes called *pigeonhole sort*

Example

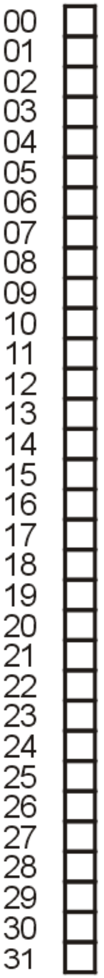
Consider sorting the following set of unique integers in the range 0, ..., 31:

20 1 31 8 29 28 11 14 6 16 15

27 10 4 23 7 19 18 0 26 12 22

Create an bit-vector with 32 buckets

– This requires 4 bytes



Example

For each number, set the corresponding bucket to 1

Now, just traverse the list and record only those numbers for which the bit is 1 (true):

0	1	4	6	7	8	10	11	12	14	15
16	18	19	20	22	23	26	27	28	29	31

Analysis

How is this so fast?

- Recall that an algorithm which can sort arbitrary data must be $\Omega(n \ln(n))$

In this case, we don't have *arbitrary* data

- We have one further constraint: the items being sorted are integers within a small range
- If the size of the range (i.e., number of buckets) is $O(n)$, then we get a $\Theta(n)$ algorithm

Counting sort

Modification: what if there are repetitions in the data

- In this case, a bit vector is insufficient

Two options, each bucket is either:

- a counter, or
- a linked list

The first is better if objects in the bin are the same

Example

Sort the digits

0 3 2 8 5 3 7 5 3 2 8 2 3 5 1 3 2 8 5 3 4 9 2 3 5 1 0 9 3 5 2 3 5 4 2 1 3

We start with an array of 10 counters, each initially set to zero:

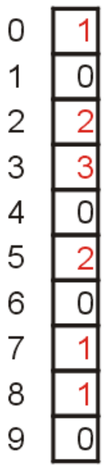
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Example

Moving through the first 10 digits

0 3 2 8 5 3 7 5 3 2 8 2 3 5 1 3 2 8 5 3 4 9 2 3 5 1 0 9 3 5 2 3 5 4 2 1 3

we increment the corresponding buckets



Example

Moving through remaining digits

0 3 2 8 5 3 7 5 3 2 8 2 3 5 1 3 2 8 5 3 4 9 2 3 5 1 0 9 3 5 2 3 5 4 2 1 3

we continue incrementing the corresponding buckets

0	2
1	3
2	7
3	10
4	2
5	7
6	0
7	1
8	3
9	2

Example

We now simply read off the number of each occurrence:

0 0 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4 4 5 5 5 5 5 5 5 7 8 8 8 9 9

For example

- There are seven 2s
- There are two 4s

0	2
1	3
2	7
3	10
4	2
5	7
6	0
7	1
8	3
9	2

Run-time summary

Let m be the number of buckets.

Space

- Bucket sort always requires $\Theta(m)$ memory

Time is $\Theta(n + m)$

- If $m = \Theta(n)$, the run time is $\Theta(n)$ with $\Theta(n)$ memory
- If $m = o(n)$, the run time is $\Theta(n)$ with $o(n)$ memory

Bucket sort

- The general version of Bucket sort
  - We assume that the number of buckets is $O(n)$ and the objects are uniformly distributed into these buckets
  - In each bucket, there might be multiple objects of different values but their number is small, so we apply a sorting algorithm (e.g., insertion sort) on them

- What if the assumption are not true…

Outline

- Introduction
- Inversions
- Insertion sort
- Bubble sort
- Heap sort
- Merge sort
- Quicksort
- Bucket sort
- Radix sort

Radix Sort

Suppose we want to sort 10 digit numbers with repetitions

We could use bucket sort, but this would require the use of 10^{10} buckets

- With one byte per counter, this would require 9 GiB
- This may not be very practical…

Radix Sort

Consider the following scheme

- Given the numbers

  16 31 99 59 27 90 10 26 21 60 18 57 17

- If we first sort the numbers based on their last digit only, we get:

  90 10 60 31 21 16 26 27 57 17 18 99 59

- Now sort according to the first digit:

  10 16 17 18 21 26 27 31 57 59 60 90 99

Radix Sort

The resulting sequence of numbers is a sorted list

Thus, we have the following algorithm:

- Suppose we are sorting decimal numbers
- Create an array of 10 queues
- For each digit, starting with the least significant
  - Place the i-th number into the bin corresponding with the current digit
  - Remove all digits in the order they were placed into the bins in the order of the bins

Correctness

Suppose that two n-digit numbers are equal for the first m digits:

$$a = a_n a_{n-1} a_{n-2} \cdots a_{n-m+1} \boldsymbol{a_{n-m}} \cdots a_1 a_0$$

$$b = a_n a_{n-1} a_{n-2} \cdots a_{n-m+1} \boldsymbol{b_{n-m}} \cdots b_1 b_0$$

where $\boldsymbol{a_{n-m}} < \boldsymbol{b_{n-m}}$

For example, $103574 < 103892$ because $1 = 1$, $0 = 0$, $3 = 3$ but $5 < 8$

Then, on iteration $n - m$, a will be placed in a lower bin than b

When they are taken out, a will precede b in the list

Correctness

For all subsequent iterations, a and b will be placed in the same bin, and will therefore continue to be taken out in the same order

Therefore, in the final list, a must precede b

Example 1

Sort the following decimal numbers:

86 198 466 709 973 981 374 766 473 342

First, interpret 86 as 086

Example 1

Next, create an array of 10 queues:

0				
1				
2				
3				
4				
5				
6				
7				
8				
9				

Example 1

Push according to the 3rd digit:

086 198 466 709 973 981 374 766 473 342

0				
1	98**1**			
2	34**2**			
3	97**3**	47**3**		
4	37**4**			
5				
6	08**6**	46**6**	76**6**	
7				
8	19**8**			
9	70**9**			

and dequeue: 98**1** 34**2** 97**3** 47**3** 37**4** 08**6** 46**6** 76**6** 19**8** 70**9**

Example 1

Enqueue according to the 2nd digit:

981 342 973 473 374 086 466 766 198 709

0	7**0**9			
1				
2				
3				
4	3**4**2			
5				
6	4**6**6	7**6**6		
7	9**7**3	4**7**3	3**7**4	
8	9**8**1	0**8**6		
9	1**9**8			

and dequeue: 7**0**9 3**4**2 4**6**6 7**6**6 9**7**3 4**7**3 3**7**4 9**8**1 0**8**6 1**9**8

Example 1

Enqueue according to the 1st digit:

709 342 466 766 973 473 374 981 086 198

0	086			
1	198			
2				
3	342	374		
4	466	473		
5				
6				
7	709	766		
8				
9	973	981		

and dequeue: **0**86 **1**98 **3**42 **3**74 **4**66 **4**73 **7**09 **7**66 **9**73 **9**81

Example 1

The numbers

086 198 342 374 466 473 709 766 973 981

are now in order

Example 2

Sort the following base 2 numbers:

`1111 11011 11001 10000 11010 101 11100 111 1011 10101`

First, interpret each as a 5-bit number:

`01111  11011  11001  10000  11010  00101  11100  00111  01011  10101`

Next, create an array of two queues:

0								
1								

Example 2

Place the numbers

01111 11011 11001 10000 11010 00101 11100 00111 01011 10101

into the queues based on the 5th bit:

0	10000	11010	11100					
1	01111	11011	11001	00101	00111	01011	10101	

Remove them in order:

10000 11010 11100 01111 11011 11001 00101 00111 01011 10101

Example 2

Place the numbers

10000 11010 11100 01111 11011 11001 00101 00111 01011 10101

into the queues based on the 4th bit:

0	10000	11100	11001	00101	10101			
1	11010	01111	11011	00111	01011			

Remove them in order:

10000 11100 11001 00101 10101 11010 01111 11011 00111 01011

Example 2

Place the numbers

`10000 11100 11001 00101 10101 11010 01111 11011 00111 01011`

into the queues based on the 3rd bit:

0	10000	11001	11010	11011	01011			
1	11100	00101	10101	01111	00111			

Remove them in order:

`10000 11001 11010 11011 01011 11100 00101 10101 01111 00111`

Example 2

Place the numbers

10000 11001 11010 11011 01011 11100 00101 10101 01111 00111

into the queues based on the 2nd bit:

0	10000	00101	10101	00111				
1	11001	11010	11011	01011	11100	01111		

Remove them in order:

10000 00101 10101 00111 11001 11010 11011 01011 11100 01111

Example 2

Place the numbers

10000 00101 10101 00111 11001 11010 11011 01011 11100 01111

into the queues based on the 1st bit:

0	00101	00111	01011	01111				
1	10000	10101	11001	11010	11011	11100		

Remove them in order:

00101 00111 01011 01111 10000 10101 11001 11010 11011 11100

Example 2

The numbers

`00101  00111  01011  01111  10000  10101  11001  11010  11011  11100`

are now in order

This required $5n$ enqueues and dequeues

– In this case, it $n = 10$

Run-time analysis

The number of buckets is 10 or 2, which is $\Theta(1)$

How many times must we iterate to sort numbers on the range of 0, $\ldots$, $m - 1$?

– We require $\lceil \log_{10}(m) \rceil$ digits or $\lceil \log_2(m) \rceil$ bits

Run time is therefore $\Theta(n \ln(m))$

– For this to be more efficient than previous sorting algorithms, it must be true that $\ln(m) << \ln(n)$ or $m << n$

Run-time analysis

The following table summarizes the run-times of radix sort for sorting n numbers on the range $0, \ldots, m-1$

Case	Run Time	Comments
Worst	$\boldsymbol{\Theta}(n \ln(m))$	No worst case
Average	$\boldsymbol{\Theta}(n \ln(m))$	
Best	$\boldsymbol{\Theta}(n \ln(m))$	No best case

It requires $\Theta(n)$ memory for the queues

- It is only useful to use radix sort over quicksort if $n = \omega(m)$

Summary

Simple $\mathbf{O}(n^2)$ sorting algorithms

- Insertion sort, Bubble sort

More sophisticated and faster $\mathbf{\Theta}(n \ln(n))$ sorting algorithms:

- Heap sort, Merge sort, and Quicksort

Linear-time sorting algorithms

- Bucket sort and Radix sort
- Must make assumptions about data