COMP251: DATA STRUCTURES & ALGORITHMS

Instructor: Maryam Siahbani

Computer Information System
University of Fraser Valley

* Some slides from “Algorithms and Data Structures” by Douglas Wilhelm Harder

Sorting Algorithms

Outline

In this topic, we will introduce sorting, including:

- Definitions
- Assumptions
- *In-place* sorting
- Sorting techniques and strategies
- Overview of run times

Lower bound on run times

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 a reordering

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

such that

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

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.

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.

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:

- Arrays are to be used for both input and output,
- We will focus on sorting objects and leave the more general case of sorting records based on one or more fields as an implementation detail

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)

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:

$$\Theta(n) \qquad \Theta(n \ln(n)) \qquad \Theta(n^2)$$

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 $\Theta(n^2)$ sorting algorithms:

- Insertion sort, Bubble sort

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

- Heap sort, Quicksort, and Merge sort

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, the best case of all sorting algorithms must be $\Theta(n)$

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

Optimal Sorting Algorithms

The next slides will cover five common sorting algorithms

- There is no ***optimal*** sorting algorithm which can be used in all places
- Under various circumstances, different sorting algorithms will deliver optimal run-time and memory-allocation requirements

Insertion Sort

Background

Consider the following observations:

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

Background

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

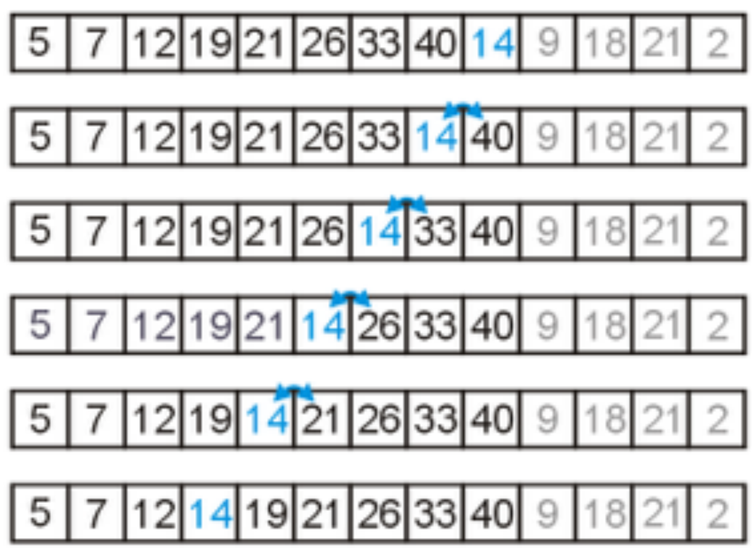
5	**7**	**12**	**19**	**21**	**26**	**33**	**40**	**14**	9	18	21	2

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



The Algorithm

For any unsorted list:

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

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

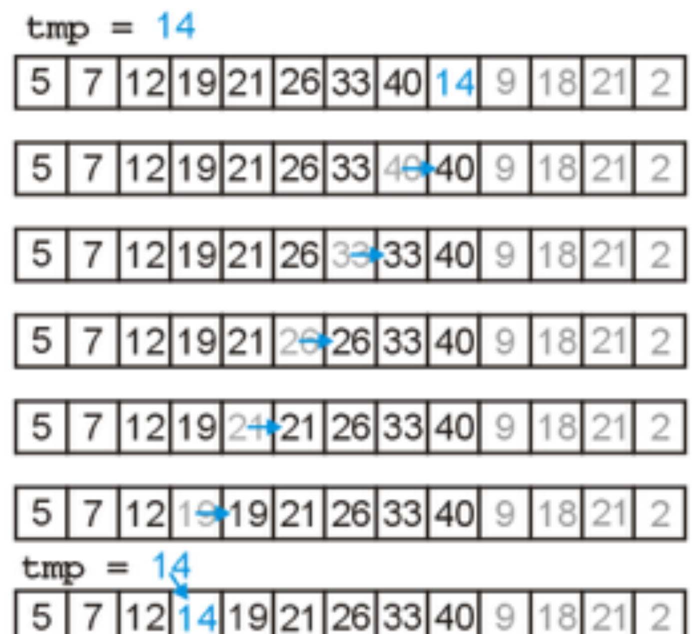
- Insert the k^{th} item in the correct position in the sorted list
- The sorted list is now of size k

The Algorithm

Code for this would be:

```
Comparable tmp = a[k];
for(int j = k; j > 0 && tmp.compareTo( a[j - 1] ) < 0; j--)
    a[j] = a[j - 1];

a[j] = tmp;
```



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

- this reduces assignments by a factor of 3
- speeds up the algorithm by a factor of two

Implementation and Analysis

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

```
public static void insertionSort( Comparable [ ] a )
{
    for( int k = 1; k < a.length; k++ )
    {
        Comparable tmp = a[k];
        for(int j = k; j > 0 && tmp.compareTo( a[j - 1] ) < 0; j--)
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
```

Implementation and Analysis

The $\Theta(1)$-initialization of the outer for-loop is executed once

```
public static void insertionSort( Comparable [ ] a )
{
    for( int k = 1; k < a.length; k++ )
    {
        Comparable tmp = a[k];
        for(int j = k; j > 0 && tmp.compareTo( a[j - 1] ) < 0; j--)
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
```

Implementation and Analysis

This $\Theta(1)$- condition will be tested n (assume there are n items) times at which point it fails

```
public static void insertionSort( Comparable [ ] a )
{
    for( int k = 1; k < a.length; k++ )
    {
        Comparable tmp = a[k];
        for(int j = k; j > 0 && tmp.compareTo( a[j - 1] ) < 0; j--)
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
```

Implementation and Analysis

Thus, the inner for-loop will be executed a total of $n - 1$ times

```
public static void insertionSort( Comparable [ ] a )
{
    for( int k = 1; k < a.length; k++ )
    {
        Comparable tmp = a[k];
        for(int j = k; j > 0 && tmp.compareTo( a[j - 1] ) < 0; j--)
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
```

Implementation and Analysis

In the worst case, the inner for-loop is executed a total of k times

```
public static void insertionSort( Comparable [ ] a )
{
    for( int k = 1; k < a.length; k++ )
    {
        Comparable tmp = a[k];
        for(int j = k; j > 0 && tmp.compareTo( a[j - 1] ) < 0; j--)
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
```

Implementation and Analysis

The body of the inner for-loop runs in $\Theta(1)$

```
public static void insertionSort( Comparable [ ] a )
{
    for( int k = 1; k < a.length; k++ )
    {
        Comparable tmp = a[k];
        for(int j = k; j > 0 && tmp.compareTo( a[j - 1] ) < 0; j--)
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
```

Thus, the worst-case run time is

$$1 + 2 + \ldots + (n\text{-}2) + (n\text{-}1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \mathrm{O}(n^2)$$

Implementation and Analysis

Problem: we may break out of the inner loop…

```
public static void insertionSort( Comparable [ ] a )
{
    for( int k = 1; k < a.length; k++ )
    {
        Comparable tmp = a[k];
        for(int j = k; j > 0 && tmp.compareTo( a[j - 1] ) < 0; j--)
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
```

Implementation and Analysis

As soon as a pair **`a[j-1] <= tmp`**, (the item which should be inserted) we are finished

```
public static void insertionSort( Comparable [ ] a )
{
    for( int k = 1; k < a.length; k++ )
    {
        Comparable tmp = a[k];
        for(int j = k; j > 0 && tmp.compareTo( a[j - 1] ) < 0; j--)
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
```

Implementation and Analysis

In best case: the array is sorted at the beginning and the inner loop takes $\Theta(1)$, which means the sort is $\Theta(n)$

```
public static void insertionSort( Comparable [ ] a )
{
    for( int k = 1; k < a.length; k++ )
    {
        Comparable tmp = a[k];
        for(int j = k; j > 0 && tmp.compareTo( a[j - 1] ) < 0; j--)
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
```

Consequences of Our Analysis

A random list will have $\Theta(n^2)$ complexity on average

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 $4\ \mathrm{s}$ 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	$\Theta(n^2)$	
Best	$\Theta(n)$	Almost sorted

Insertion Sort

– Insert new entries into growing sorted lists

– Run-time analysis

- Actual and average case run time: $\mathbf{O}(n^2)$
- Average for a random array: $\Theta(n^2)$
- Best case (almost sorted list): $\Theta(n)$

- Memory requirements: $\Theta(1)$

Bubble Sort

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

Description

As well as looking at good algorithms, it is often useful too look at sub-optimal algorithms

- Bubble sort is a simple algorithm with:
    - a memorable name, and
    - a simple idea

- It is also significantly worse than insertion sort

Implementation

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

Implementation

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

Implementation

The default algorithm:

```
public static void bubbleSort( Comparable [ ] a ) {
    for( int i = 0; i < a.length; i++ ) {
        for(int j = 1; j < a.length; j++ )
         if (a[j].compareTo( a[j - 1] ) < 0)
             swap(a, j-1, j);
}
```

The Basic Algorithm

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

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

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

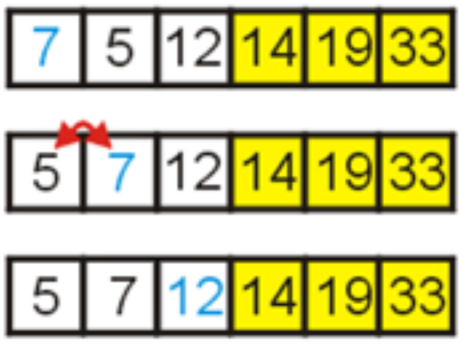
Example

Now the two largest elements are at the end

- Repeat again

Example

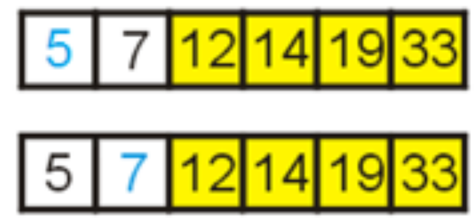
With this loop, 5 and 7 are swapped



Example

Finally, we swap the last two entries to order them

- At this point, we have a sorted array



The Basic Algorithm

The best case?

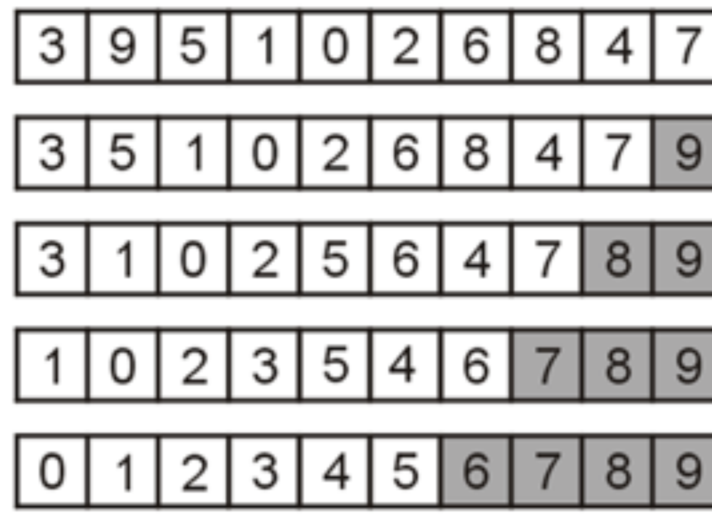
The worst case?

```
public static void bubbleSort( Comparable [ ] a ) {
    for( int i = 0; i < a.length; i++ ) {
        for(int j = 1; j < a.length; j++ )
          if (a[j].compareTo( a[j - 1] ) < 0)
              swap(a, j-1, j);
}
```

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)

```
public static void bubbleSort( Comparable [ ] a ) {
    for( int i = 0; i < a.length; i++ ) {
        boolean changed = false;
        for(int j = 1; j < a.length; j++ )
            if (a[j].compareTo( a[j - 1] ) < 0){
                swapReferences(a, j-1, j);
                changed = true;
            }
        // if j-loop does not make any swaps,
        // the array is now sorted, so stop looping
        if (!changed)
            break;
    }
}
```

Run-Time

The following table summarizes the run-times of our modified bubble sorting algorithm; however, it is worse than insertion sort in practice

Case	Run Time	Comments
Worst	$\Theta(n^2)$	
Average	$\Theta(n^2)$	
Best	$\Theta(n)$	*almost sorted*

heap Sort

Heap Sort

Recall that inserting n objects into a min-heap and then taking n objects will result in them coming out in order

Strategy: given an unsorted list with n objects, place them into a heap, and take them out

Run time Analysis of Heap Sort

Taking an object out of a heap with n items requires $\mathbf{O}(\ln(n))$ time

Therefore, taking n objects out requires

$$\sum_{k=1}^{n} \ln(k) = \ln\left(\prod_{k=1}^{n} k\right) = \ln(n!)$$

Recall that $\ln(a) + \ln(b) = \ln(ab)$

Question: What is the asymptotic growth of $\ln(n!)$?

Run time Analysis of Heap Sort

Using Maple:

```
> asympt( ln( n! ), n );
```

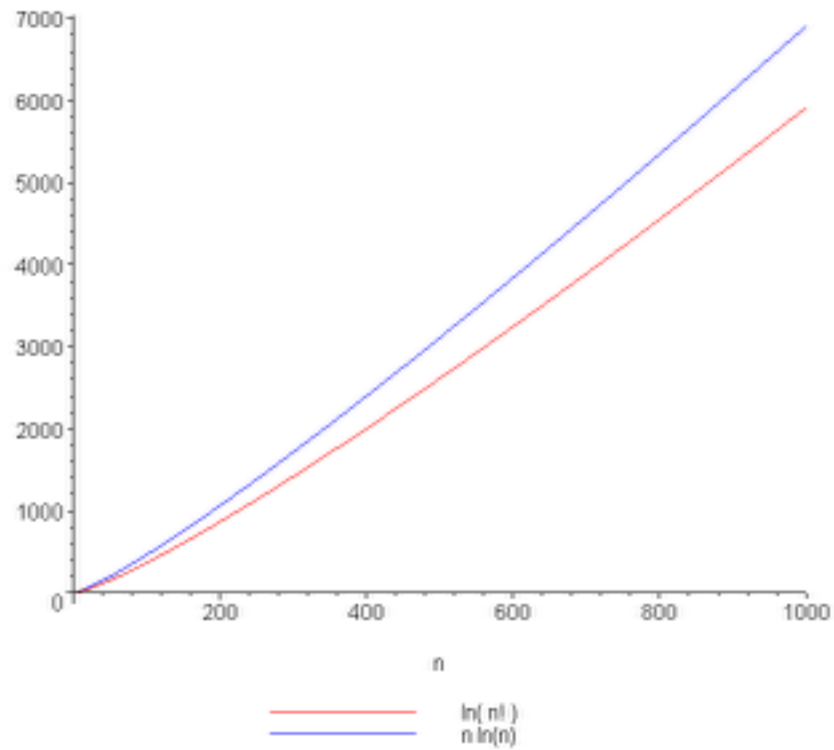
$$(\ln(n) - 1)\,n + \ln(\sqrt{2}\,\sqrt{\pi}) + \frac{1}{2}\ln(n) + \frac{1}{12\,n} - \frac{1}{360\,n^3} + \mathrm{O}\left(\frac{1}{n^5}\right)$$

The leading term is $(\ln(n) - 1)\,n$

Therefore, the run time is $\mathbf{O}(n \ln(n))$

$\ln(n!)$ and $n\ln(n)$

A plot of $\ln(n!)$ and $n \ln(n)$ also suggests that they are asymptotically related:



In-place Implementation

Problem:

- This solution requires additional memory, that is, a min-heap of size n

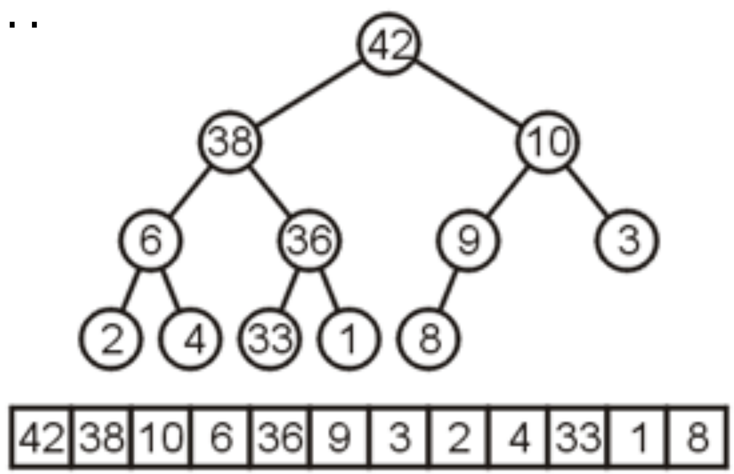
This requires $\Theta(n)$ memory and is therefore not in place

Is it possible to perform a heap sort in place, that is, require at most $\Theta(1)$ memory (a few extra variables)?

In-place Implementation

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

– A heap where the maximum element is at the top of the heap and the next to be popped is one of the children,…

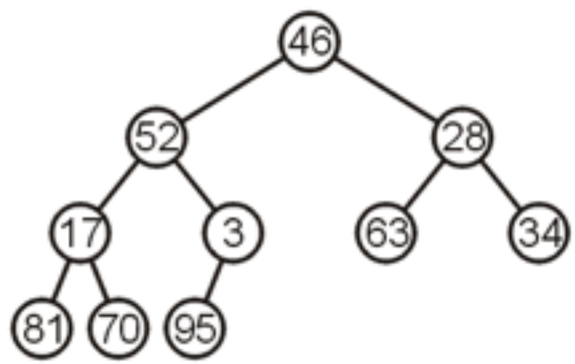


In-place Heapification

Now, consider this unsorted array:

46	52	28	17	3	63	34	81	70	95

This array represents the following complete tree:



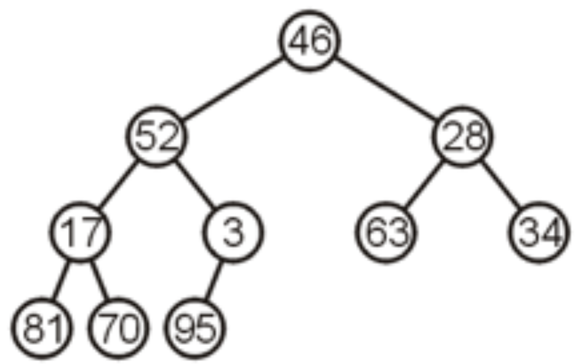
This is neither a min-heap, max-heap, or binary search tree

In-place Heapification

Now, consider this unsorted array:

46	52	28	17	3	63	34	81	70	95

Additionally, because arrays start at 0 (we started at entry 1 for binary heaps) , we need different formulas for the children and parent



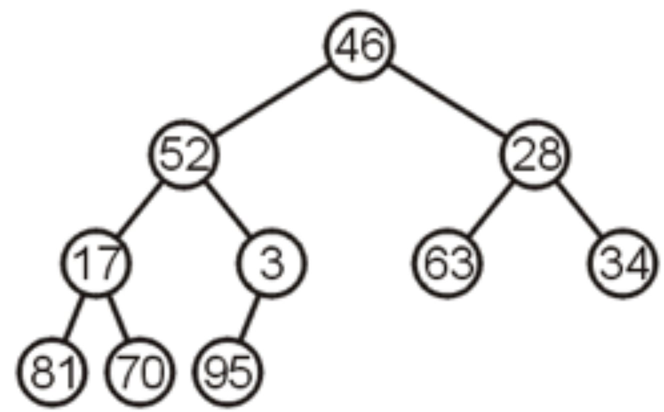
The formulas are now:

Children `2*k + 1`   `2*k + 2`

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

In-place Heapification

Can we convert this complete tree into a max heap?



Restriction:

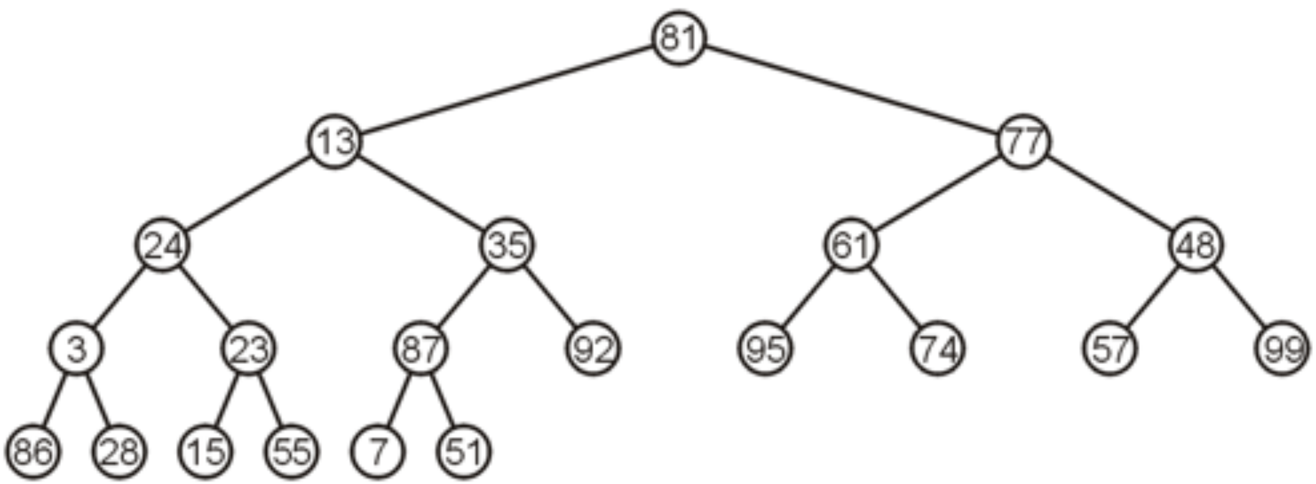
– The operation must be done in-place

In-place Heapification

buildHeap!

each leaf node is a max heap on its own

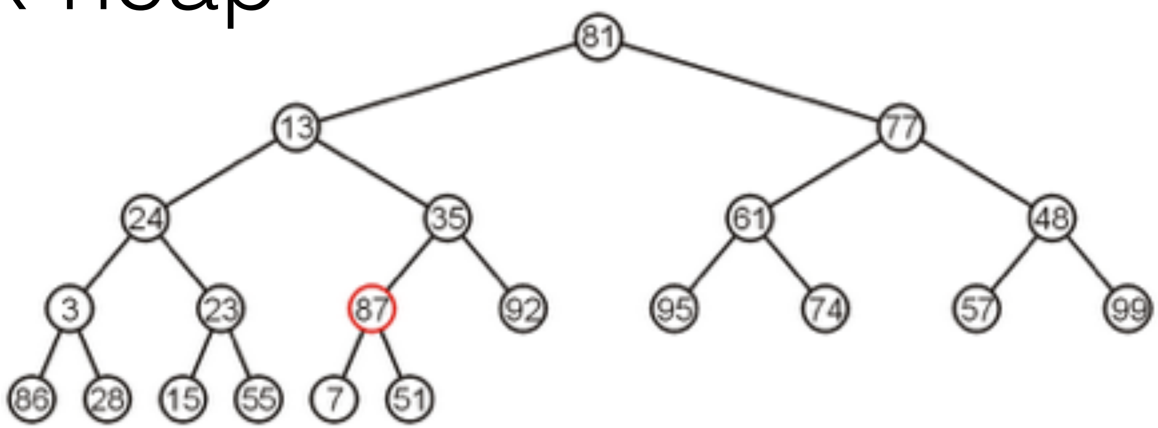
81	13	77	24	35	61	48	3	23	87	92	95	74	57	99	86	28	15	55	7	51



In-place Heapification

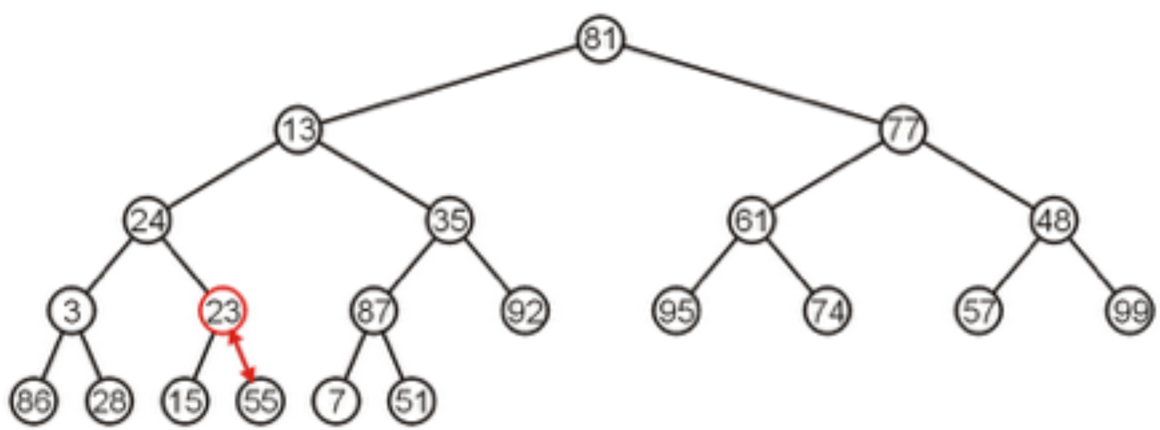
Starting at the back, we note that all leaf nodes are trivial heaps

Also, the subtree with 87 as the root is a max-heap



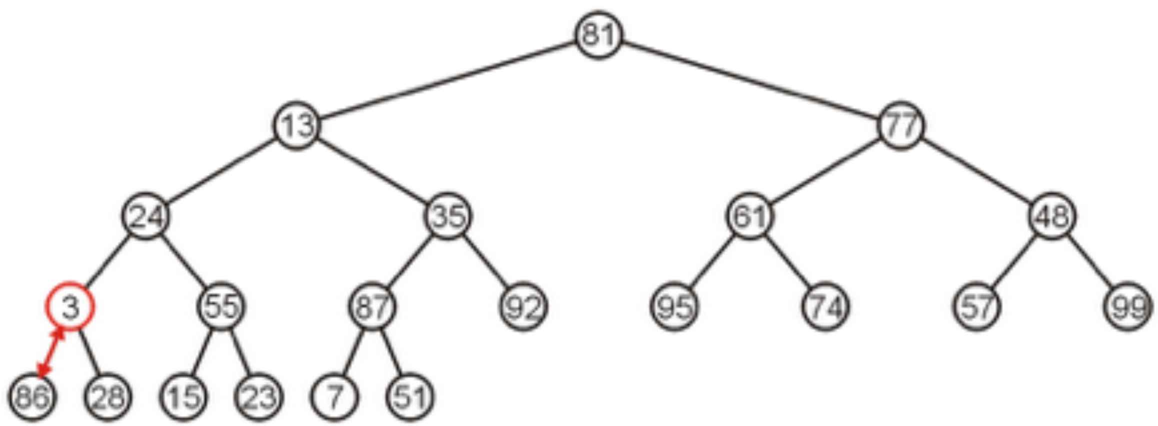
In-place Heapification

The subtree with 23 is not a max-heap, but swapping it with 55 creates a max-heap (*percolating down)*



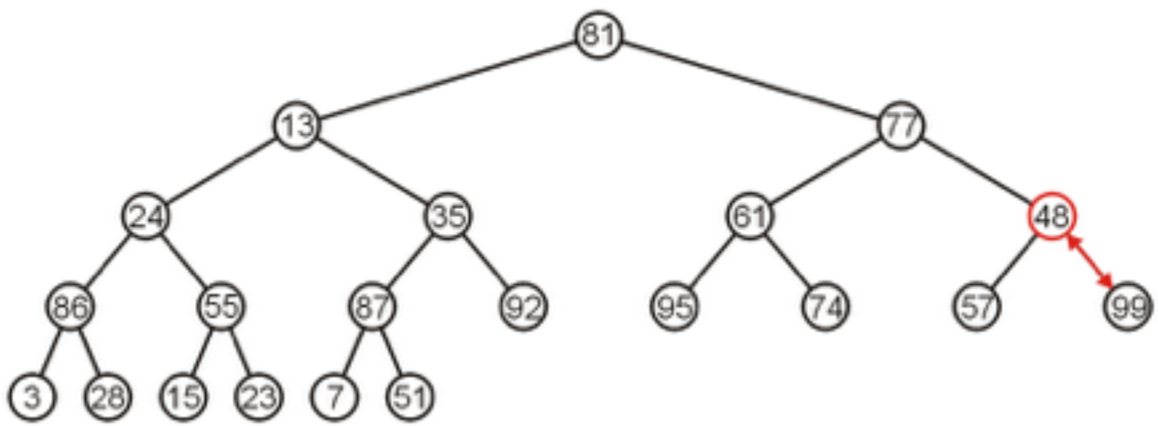
In-place Heapification

The subtree with 3 as the root is not max-heap, but we can swap 3 and the maximum of its children: 86



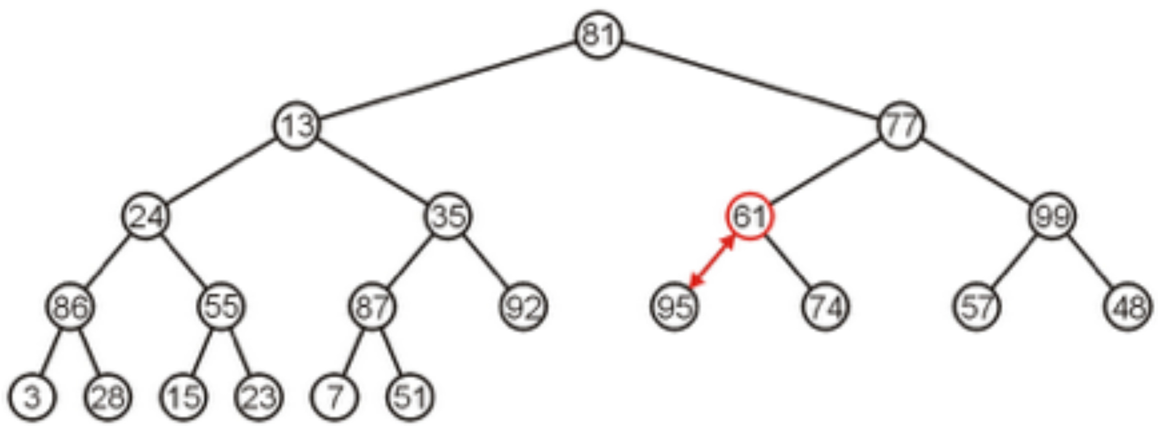
In-place Heapification

Starting with the next higher level, the subtree with root 48 can be turned into a max-heap by swapping 48 and 99



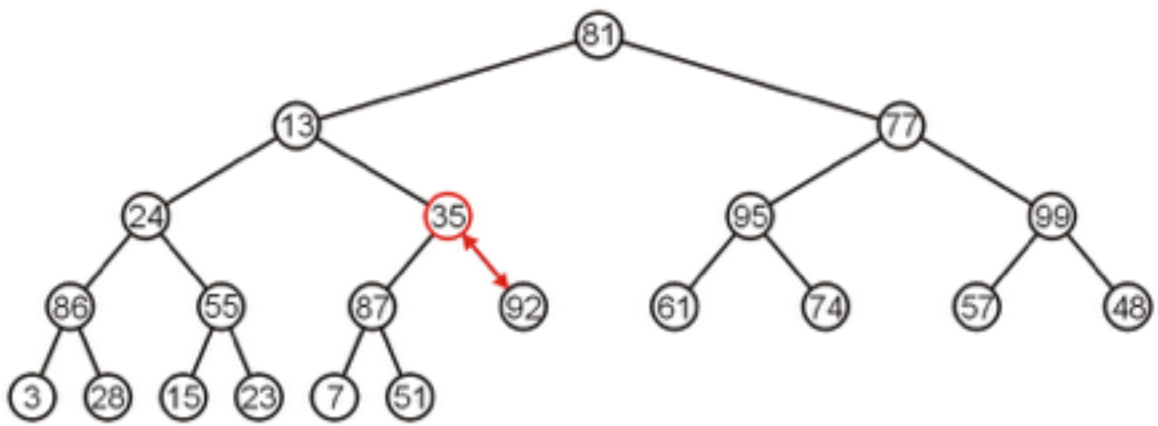
In-place Heapification

Similarly, swapping 61 and 95 creates a max-heap of the next subtree



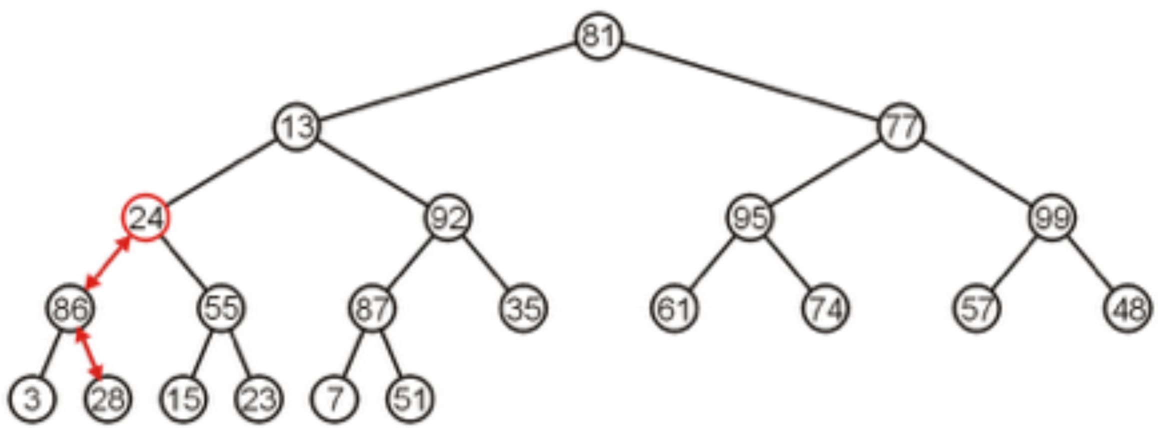
In-place Heapification

As does swapping 35 and 92



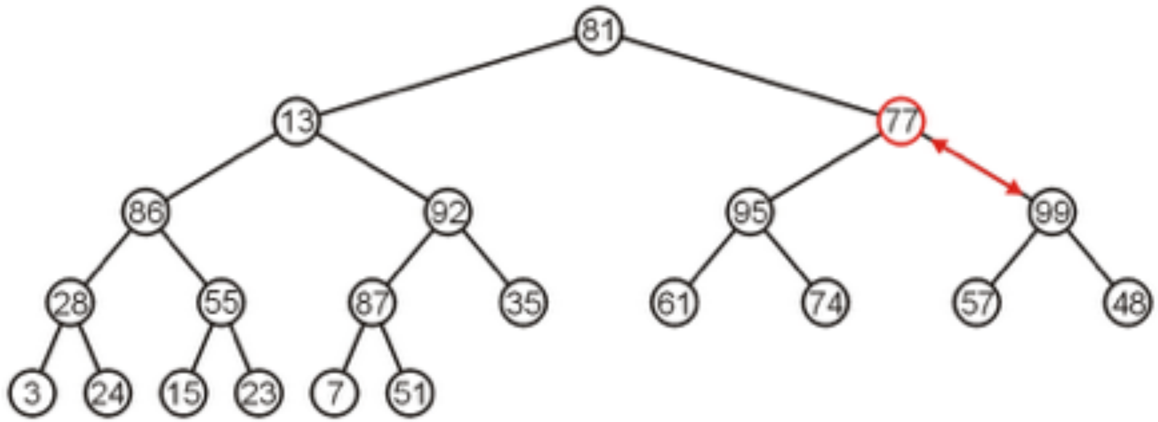
In-place Heapification

The subtree with root 24 may be converted into a max-heap by first swapping 24 and 86 and then swapping 24 and 28



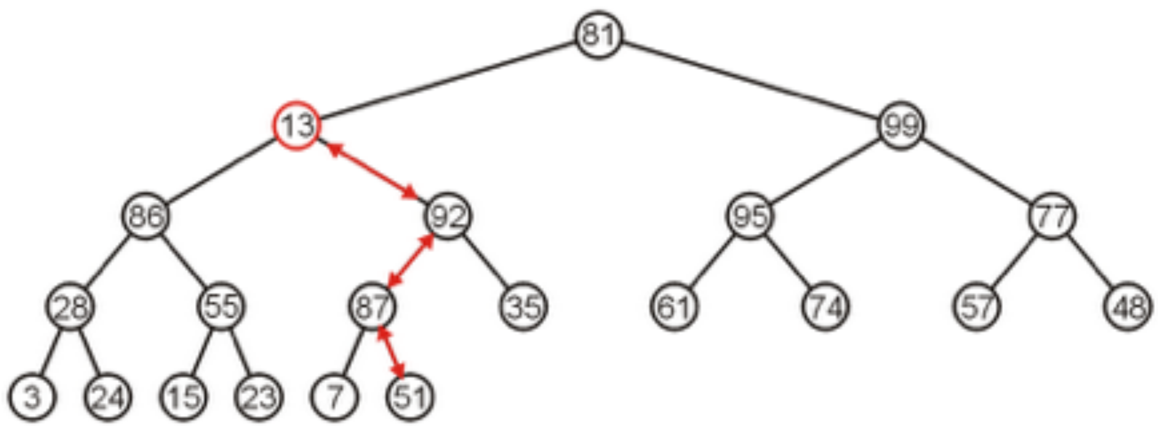
In-place Heapification

The right-most subtree of the next higher level may be turned into a max-heap by swapping 77 and 99



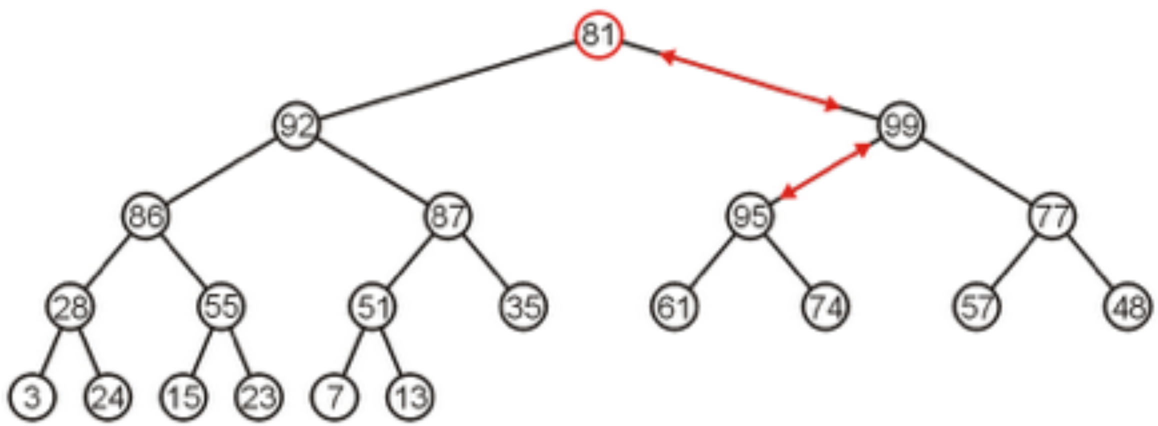
In-place Heapification

13 be percolated down to a leaf node



In-place Heapification

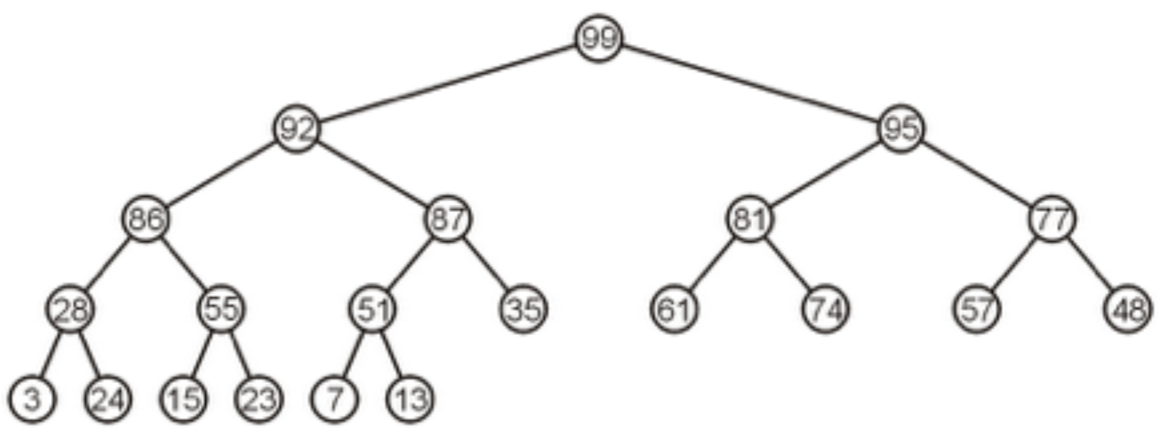
The root need only be percolated down by two levels



In-place Heapification

The final product is a max-heap

This is also called Heapification (method `buildHeap`)



Example Heap Sort

Let us look at this example: we must convert the unordered array with $n = 10$ elements into a max-heap

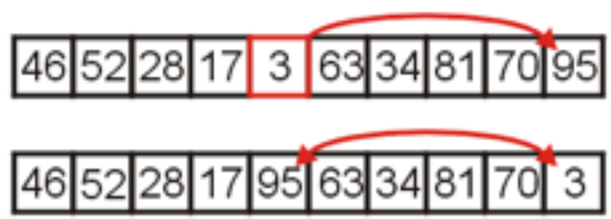
46	52	28	17	3	63	34	81	70	95

None of the leaf nodes need to be percolated down, and the first non-leaf node is in position $n/2$

Thus we start with position $10/2 = 5$

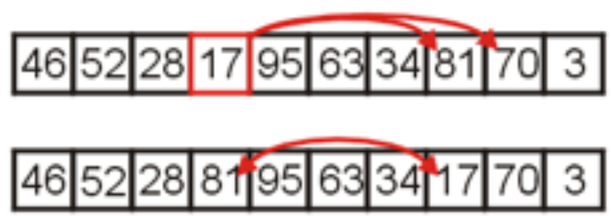
Example Heap Sort

We compare 3 with its child and swap them



Example Heap Sort

We compare 17 with its two children and swap it with the maximum child (70)



Example Heap Sort

We compare 28 with its two children, 63 and 34, and swap it with the largest child

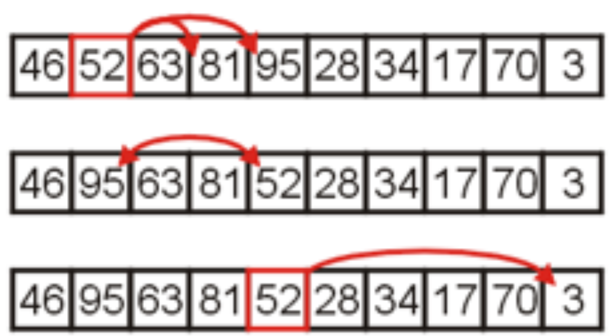
46	52	28	81	95	63	34	17	70	3

46	52	63	81	95	28	34	17	70	3

Example Heap Sort

We compare 52 with its children, swap it with the largest

–Recursing, no further swaps are needed



Example Heap Sort

Finally, we swap the root with its largest child, and recurse, swapping 46 again with 81, and then again with 70

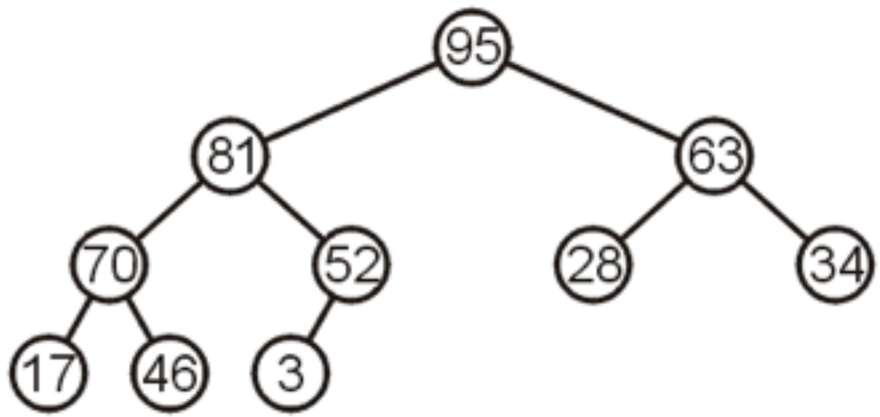
Heap Sort Example

We have now converted the unsorted array

46	52	28	17	3	63	34	81	70	95

into a max-heap:

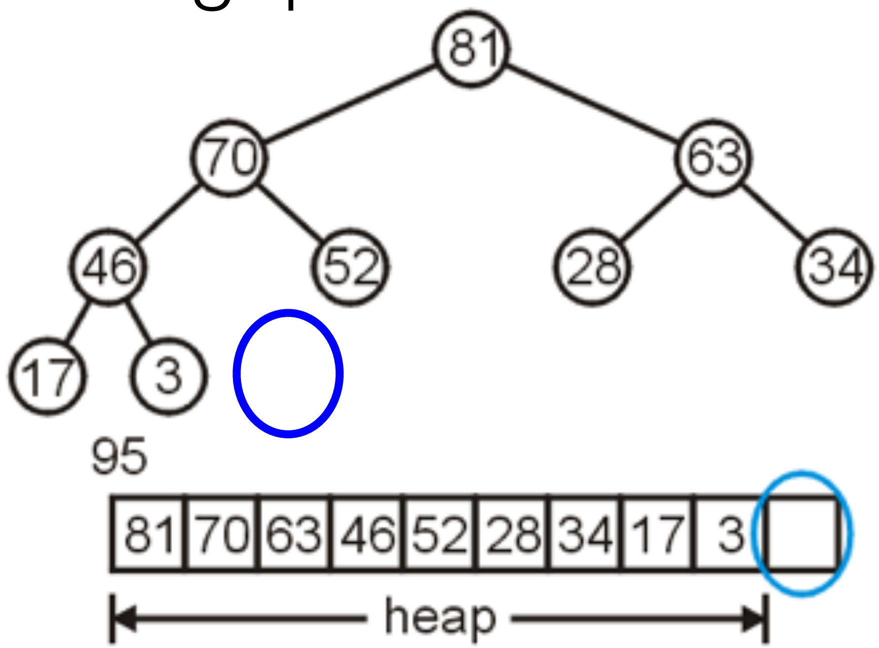
95	81	63	70	52	28	34	17	46	3



Heap Sort Example

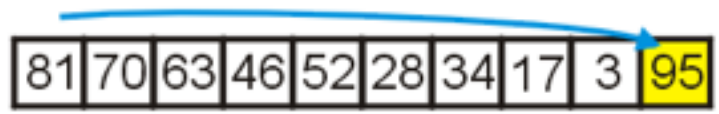
Suppose we pop the maximum element of this heap

This leaves a gap at the back of the array:

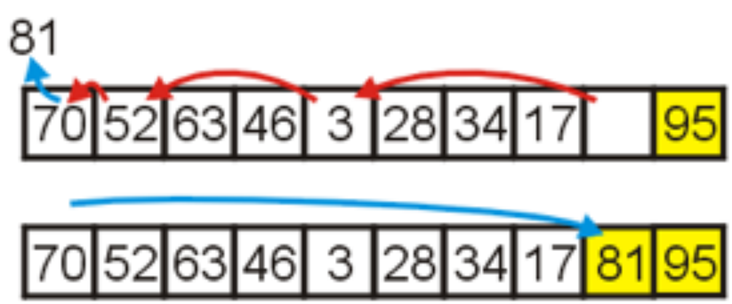


Heap Sort Example

This is the last entry in the array, so why not fill it with the largest element?



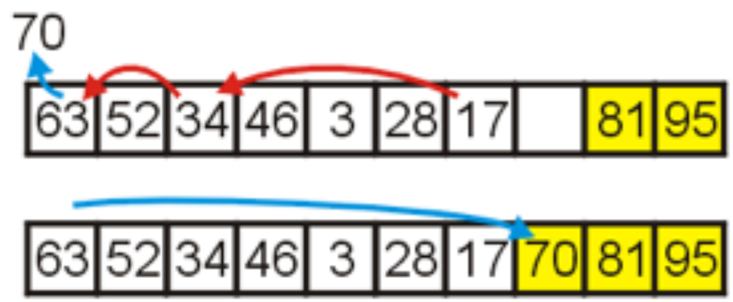
Repeat this process: pop the maximum element, and then insert it at the end of the array:



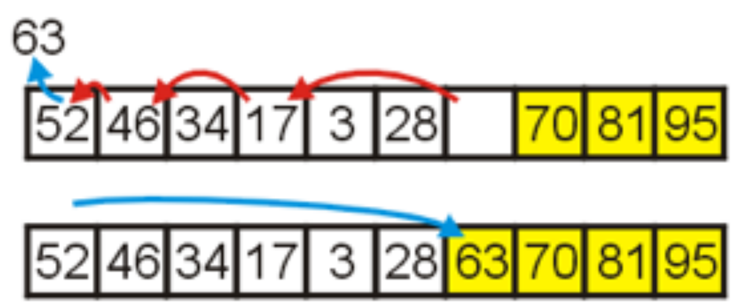
Heap Sort Example

Repeat this process

– Pop and append 70



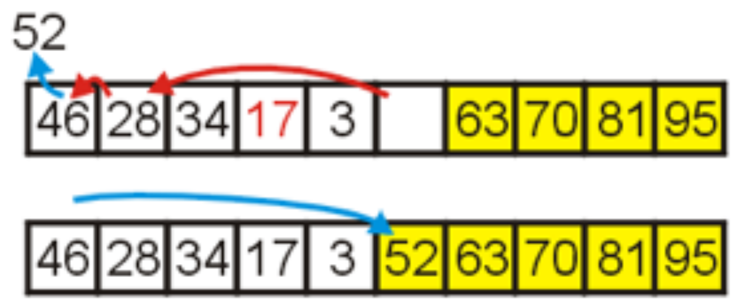
– Pop and append 63



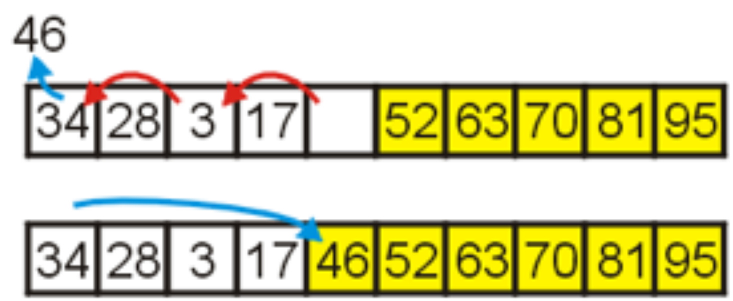
Heap Sort Example

We have the 4 largest elements in order

- Pop and append 52



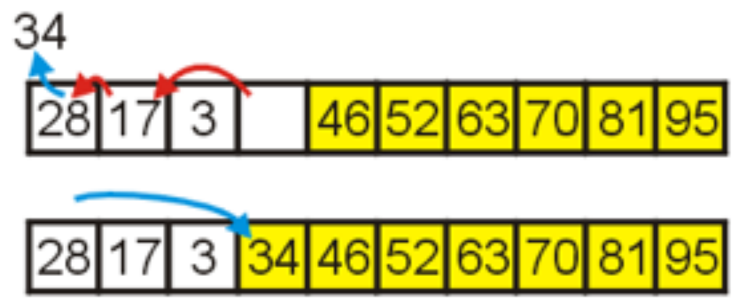
- Pop and append 46



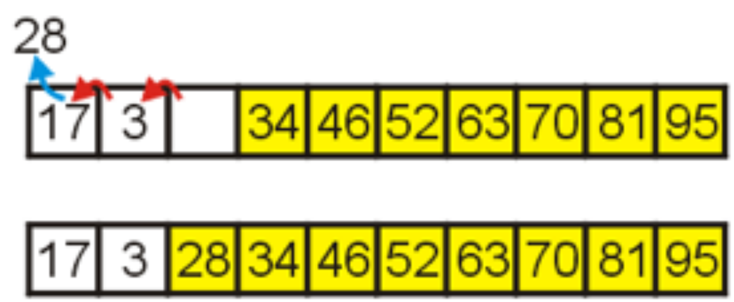
Heap Sort Example

Continuing...

- Pop and append 34

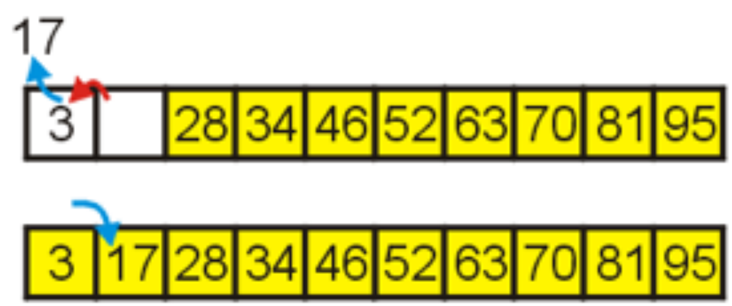


- Pop and append 28



Heap Sort Example

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



Example

Sort the following 12 entries using heap sort

34, 15, 65, 59, 79, 42, 40, 80, 50, 61, 23, 46

Heap Sort

Heapification (buildHeap) runs in $\Theta(n)$

Popping n items from a heap of size n, as we saw, runs in $\Theta(n \ln(n))$ time

- We are only making one additional copy into the blank left at the end of the array

Therefore, the total algorithm will run in $\Theta(n \ln(n))$ time

Heap Sort

There are no worst-case scenarios for heap sort

- Dequeuing from the heap will always require the same number of operations regardless of the distribution of values in the heap

There is one best case: if all the entries are identical, then the run time is $\Theta(n)$

The original order may speed up the *heapification*, however, this would only speed up an $\Theta(n)$ portion of the algorithm

Run-time Summary

The following table summarizes the run-times of heap sort

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

Summary

We have seen our first in-place $\Theta(n \ln(n))$ sorting algorithm:

- Convert the unsorted list into a max-heap as complete array
- Pop the top, n times and place that object into the vacancy at the end
- It requires $\Theta(1)$ additional memory—it is truly in-place

It is a nice algorithm; however, we will see two other faster $n \ln(n)$ algorithms; however:

- Merge sort requires $\Theta(n)$ additional memory
- Quick sort requires $\Theta(\ln(n))$ additional memory

Merge 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

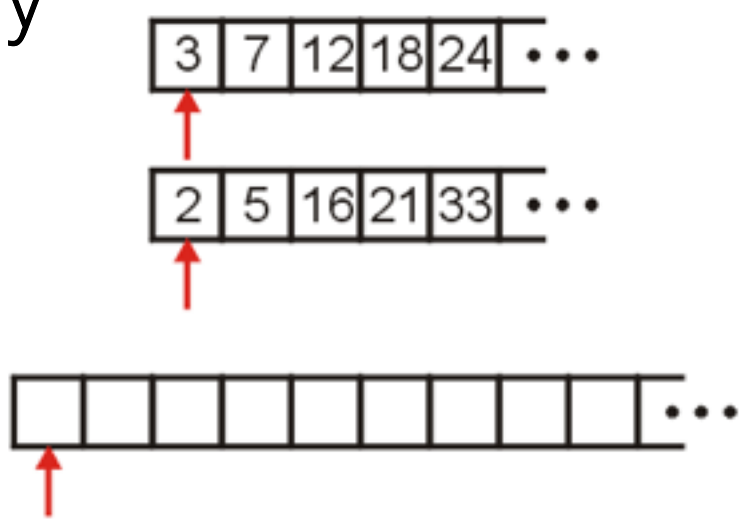
This is the first significant *divide-and-conquer* algorithm we will see

Question: How quickly can we recombine the two sub-lists into a single sorted list?

Merging Example

Consider the two sorted arrays and an empty array

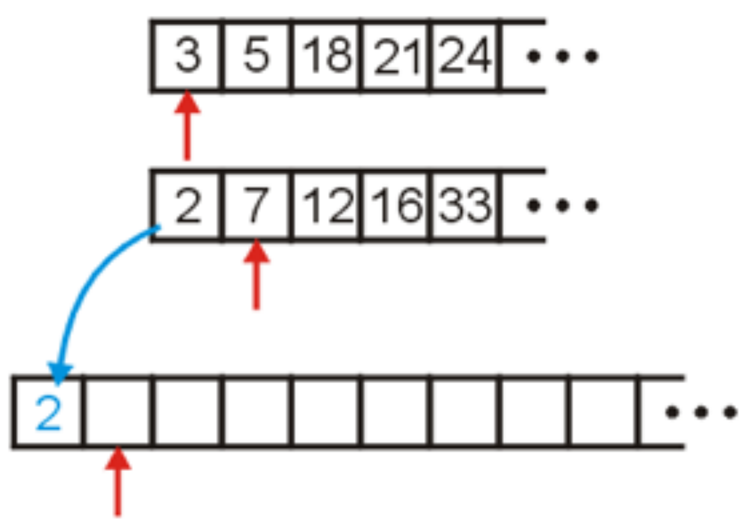
Define three indices at the start of each array



Merging Example

We compare 2 and 3: $2 < 3$

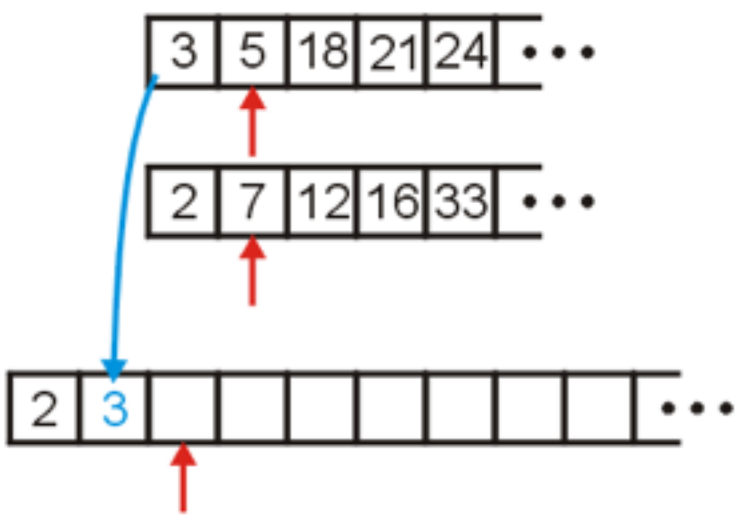
- Copy 2 down
- Increment the corresponding indices



Merging Example

We compare 3 and 7

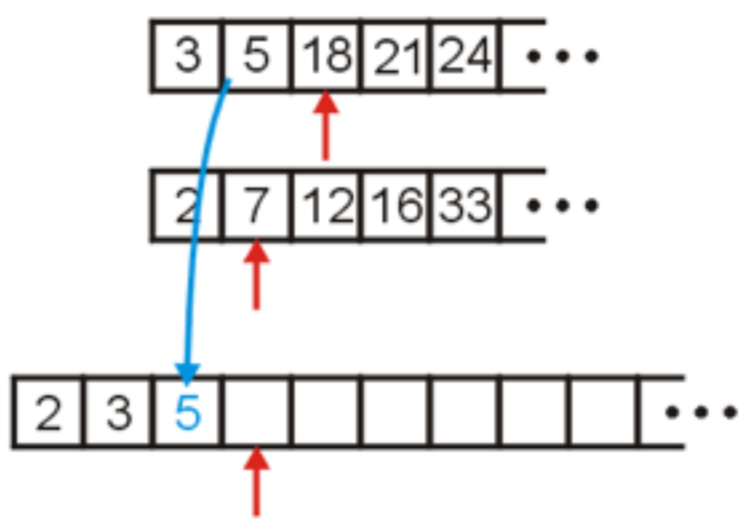
- Copy 3 down
- Increment the corresponding indices



Merging Example

We compare 5 and 7

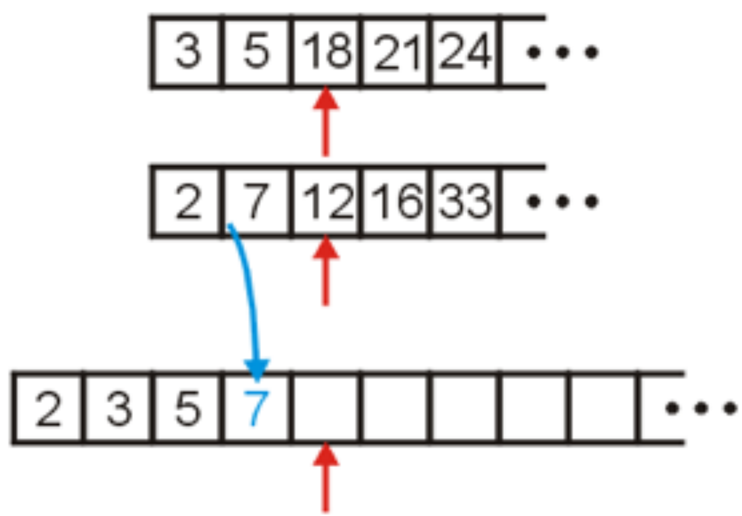
- Copy 5 down
- Increment the appropriate indices



Merging Example

We compare 18 and 7

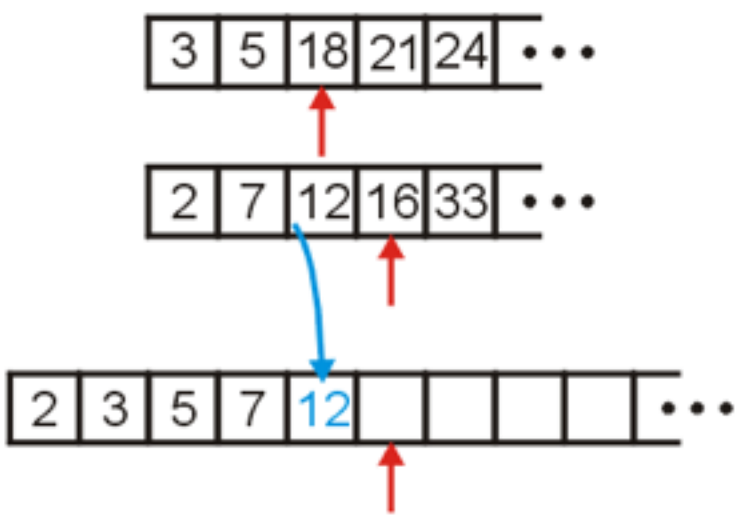
- Copy 7 down
- Increment...



Merging Example

We compare 18 and 12

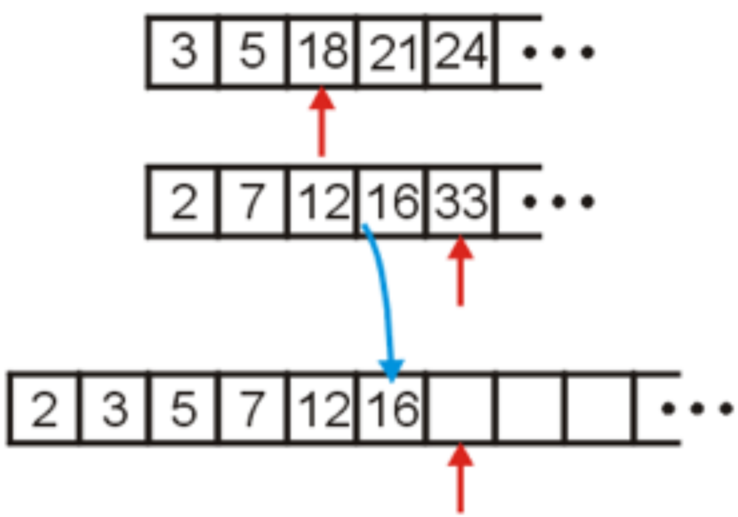
- Copy 12 down
- Increment...



Merging Example

We compare 18 and 16

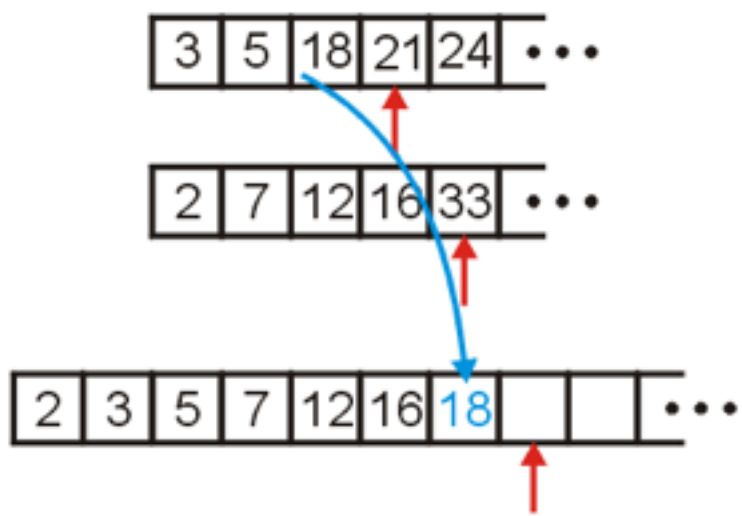
- Copy 16 down
- Increment...



Merging Example

We compare 18 and 33

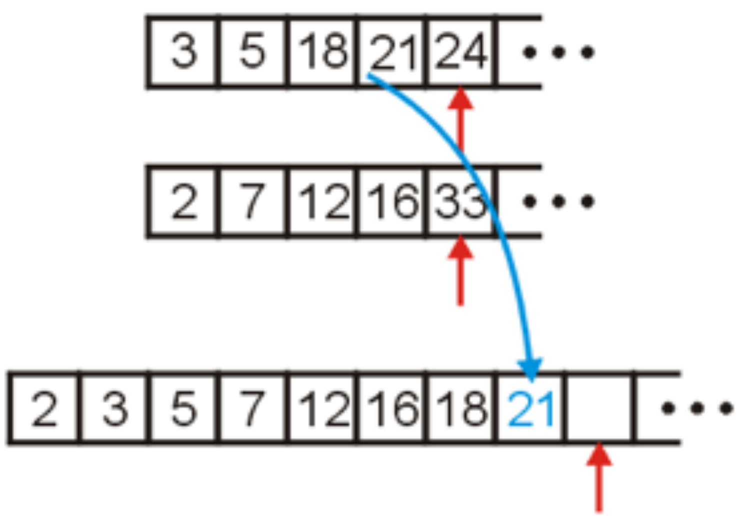
- –Copy 18 down
- –Increment...



Merging Example

We compare 21 and 33

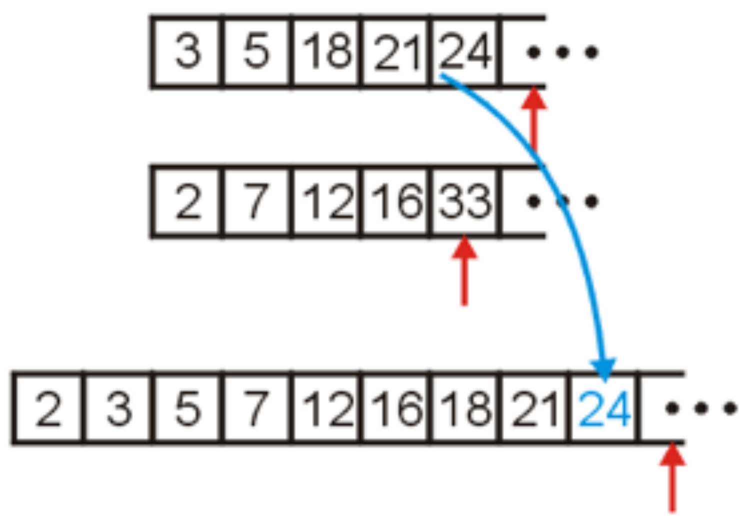
- Copy 21 down
- Increment...



Merging Example

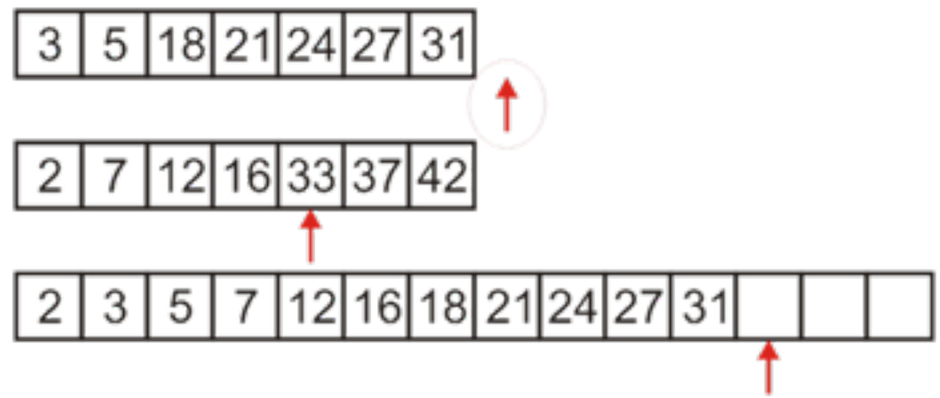
We compare 24 and 33

- Copy 24 down
- Increment...

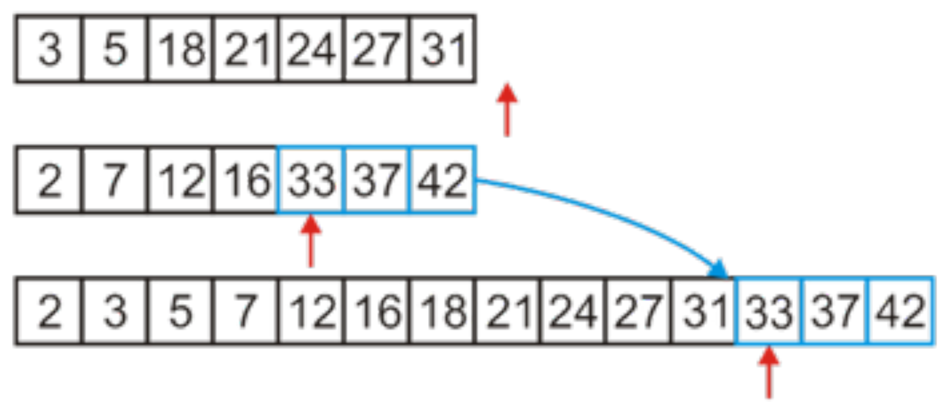


Merging Example

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:

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


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

The statement `++k` will only be run at most n_1+n_2 times

- Therefore, the body of the loops run a total of $n_1 + n_2$ times
- Hence, merging may be performed in $\Theta(n_1 + n_2)$ time

If the arrays are approximately the same size, $n = n_1$ and $n_1 \approx n_2$, we can say that the run time is $\Theta(n)$

Problem: 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 Sort Algorithm

The algorithm:

- Split the list into two approximately equal sub-lists
- Recursively call merge sort on both sub lists
- Merge the resulting sorted lists

The Algorithm

Question:

- we split the list into two sub-lists and sorted them
- how should we sort those lists?

Answer (theoretical):

- if the size of these sub-lists is > 1, use merge sort again
- if the sub-lists are of length 1, do nothing: a list of length one is sorted

Implementation

Suppose we already have a function

```
void merge( Comparable [] array, int a, int b, int c );
```

that assumes that the entries

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

are sorted and merges these two sub-arrays into a single sorted array from index **`a`** through index **`c`**, 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, 25);
```

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 will therefore implement a function

```
void mergeSort(Comparable [ ] a, int first, int last);
```

that will sort the entries in the positions

`first <= i` and `i <= last`

- Find the mid-point,
- Call merge sort recursively on each of the halves, and
- Merge the results

Implementation

The actual body is quite small:

```
private static void mergeSort( Comparable [ ] a, int first,
int last ){

    if( first < last ){
        int center = ( first + last ) / 2;
        mergeSort( a, first, center );
        mergeSort( a, center + 1, last );
        merge( a, first, center + 1, last );
    } // first => last (base case)
}
```

Example

Consider the following unsorted array of 25 entries (the last index is 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
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

Example

We call `mergeSort(a, 0, 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
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

```
mergeSort( a,  0, 24 )
```

Example

We are calling **`mergeSort(a, 0, 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
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

Find the midpoint and call **`mergeSort`** recursively

```
center = (0 + 24)/2; // == 12
mergeSort( a, 0, 12 );
```

```
mergeSort( a,  0, 24 )
```

Example

We are calling **`mergeSort(a, 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

Find the midpoint and call **`mergeSort`** recursively

```
center = (0 + 12)/2; // == 6
mergeSort( a, 0, 6 );
```

```
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

We call `mergeSort(a, 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

Find the midpoint and call `mergeSort` recursively

```
center = (0 + 12)/2; // == 6
mergeSort( a, 0, 6 );
```

```
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

We are calling **`mergeSort(a, 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

Find the midpoint and call **`mergeSort`** recursively

```
center = (0 + 6)/2; // == 3
mergeSort( a, 0, 3 );
```

```
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

We call **`mergeSort(a, 0, 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	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

Find the midpoint and call **`mergeSort`** recursively

```
center = (0 + 6)/2; // == 3
mergeSort( a, 0, 3 );
```

```
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

We are calling **`mergeSort(a, 0, 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	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

Find the midpoint and call **`mergeSort`** recursively

```
center = (0 + 3)/2; // == 1
mergeSort( a, 0, 1 );
```

```
mergeSort( a,  0, 3 )
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

We call **`mergeSort(a, 0, 1 )`**

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

Find the midpoint and call **`mergeSort`** recursively

```
center = (0 + 3)/2; // == 1
mergeSort( a, 0, 1 );
```

```
mergeSort( a,  0, 1 )
mergeSort( a,  0, 3 )
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

We are call **`mergeSort(a, 0, 1 )`**

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

Find the midpoint and call **`mergeSort`** recursively

```
center = (0 + 1)/2; // == 0
mergeSort( a, 0, 0 );
```

```
mergeSort( a,  0, 1 )
mergeSort( a,  0, 3 )
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

We call **`mergeSort(a, 0, 0 )`**

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 should stop

```
first == last
```

```
mergeSort( a,  0, 0 )
mergeSort( a,  0, 1 )
mergeSort( a,  0, 3 )
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

We are back in `mergeSort(a, 0, 1 )`

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 continue calling

```
center = (0 + 1)/2; // == 0
mergeSort( a, 0, 0 );
mergeSort( a, 1, 1 );
merge( a, 0, 1, 1 );
```

```
mergeSort( a,  0, 1 )
mergeSort( a,  0, 3 )
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

We are calling `mergeSort(a, 1, 1 )`

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 should stop

`first == last`

```
mergeSort( a,  1, 1 )
mergeSort( a,  0, 1 )
mergeSort( a,  0, 3 )
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

We are back in `mergeSort(a, 0, 1 )`

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 continue calling

```
center = (0 + 1)/2; // == 0
mergeSort( a, 0, 0 );
mergeSort( a, 1, 1 );
merge( a, 0, 1, 1 );
```

```
mergeSort( a,  0, 1 )
mergeSort( a,  0, 3 )
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

we call `merge(a, 0, 1, 1)`

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( a, 0, 1, 1 )
mergeSort( a,  0, 1 )
mergeSort( a,  0, 3 )
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

terminate `mergeSort(a, 0, 1 )`

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

```
center = (0 + 1)/2; // == 0
mergeSort( a, 0, 0 );
mergeSort( a, 1, 1 );
merge( a, 0, 1, 1 );
```

```
mergeSort( a,  0, 1 )
mergeSort( a,  0, 3 )
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

we are back in **`mergeSort(a, 0, 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	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 continue calling

```
center = (0 + 3)/2; // == 1
mergeSort( a, 0, 1 );
mergeSort( a, 2, 3 );
merge( a, 0, 2, 3);
```

```
mergeSort( a,  0, 3 )
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

we call **`mergeSort(a, 2, 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	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 recursively find the mid point ,….

```
center = (2 + 3)/2; // == 2
mergeSort( a, 2, 2 );
```

```
mergeSort( a,  2, 3 )
mergeSort( a,  0, 3 )
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

we are back in **`mergeSort(a, 0, 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	77	35	49	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

We continue calling

```
center = (0 + 3)/2; // == 1
mergeSort( a, 0, 1 );
mergeSort( a, 2, 3 );
merge( a, 0, 2, 3);
```

```
mergeSort( a,  0, 3 )
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

we call `merge(a, 0, 2, 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	77	35	49	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

```
merge( a, 0, 2, 3 )
mergeSort( a,  0, 3 )
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

we terminate `mergeSort(a, 0, 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	35	49	77	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

done

```
center = (0 + 3)/2; // == 1
mergeSort( a, 0, 1 );
mergeSort( a, 2, 3 );
merge( a, 0, 2, 3 );
```

```
mergeSort( a,  0, 3 )
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

we are back in **`mergeSort(a, 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	35	49	77	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

We continue calling

```
center = (0 + 6)/2; // == 3
mergeSort( a, 0, 3 );
mergeSort( a, 4, 6 );
merge( a, 0, 4, 6 );
```

```
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

we call **`mergeSort(a, 4, 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	35	49	77	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

We find the mid point … (we skip it in this example)

```
center = (4 + 6)/2; // == 5
mergeSort( a, 4, 5 );
mergeSort( a, 6, 6 );
merge( a, 4, 5, 6 );
```

```
mergeSort( a,  4, 6 )
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

we are back in `mergeSort(a, 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	35	49	77	48	61	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

We continue calling

```
center = (0 + 6)/2; // == 3
mergeSort( a, 0, 3 );
mergeSort( a, 4, 6 );
merge( a, 0, 4, 6 );
```

```
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

we call `merge(a, 0, 4, 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	35	49	77	48	61	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

```
merge( a,  0, 4, 6 )
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

we terminate **`mergeSort(a, 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	35	48	49	61	73	77	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

done

```
center = (0 + 6)/2; // == 3
mergeSort( a, 0, 3 );
mergeSort( a, 4, 6 );
merge( a, 0, 4, 6 );
```

```
mergeSort( a,  0, 6 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

we are back in **`mergeSort(a, 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	73	77	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

We continue calling

```
center = (0 + 12)/2; // == 6
mergeSort( a, 0, 6 );
mergeSort( a, 7, 12 );
merge( a, 0, 7, 12 );
```

```
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

we call **`mergeSort(a, 7, 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	73	77	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

We find the mid point … (we skip it in this example)

```
center = (7 + 12)/2; // == 9
mergeSort( a, 7, 9 );
mergeSort( a, 10, 12 );
merge( a, 7, 10, 12 );
```

```
mergeSort( a,  7, 12 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

we are back in **`mergeSort(a, 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	73	77	3	23	37	57	89	95	99	17	32	94	28	15	55	7	51	88	97	62

We continue calling

```
center = (0 + 12)/2; // == 6
mergeSort( a, 0, 6 );
mergeSort( a, 7, 12 );
merge( a, 0, 7, 12 );
```

```
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

we call `merge(a, 0, 7, 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	73	77	3	23	37	57	89	95	99	17	32	94	28	15	55	7	51	88	97	62

```
merge( a,  0, 7, 12 )
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

we terminate **`mergeSort(a, 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	57	61	72	77	89	95	99	17	32	94	28	15	55	7	51	88	97	62

done

```
center = (0 + 12)/2; // == 6
mergeSort( a, 0, 6 );
mergeSort( a, 7, 12 );
merge( a, 0, 7, 12 );
```

```
mergeSort( a,  0, 12 )
mergeSort( a,  0, 24 )
```

Example

we are back in **`mergeSort(a, 0, 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	57	61	72	77	89	95	99	17	32	94	28	15	55	7	51	88	97	62

we continue calling

```
center = (0 + 24)/2; // == 12
mergeSort( a, 0, 12 );
mergeSort( a, 13, 24 );
merge( a, 0, 13, 24 );
```

`mergeSort( a,  0, 24 )`

Example

we call **`mergeSort(a, 13, 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	57	61	72	77	89	95	99	17	32	94	28	15	55	7	51	88	97	62

We find the mid point … (we skip it in this example)

```
center = (13 + 24)/2; // == 18
mergeSort( a, 13, 18 );
mergeSort( a, 19, 24 );
merge( a, 0, 19, 24 );
```

```
mergeSort( a,  13, 24 )
mergeSort( a,  0, 24 )
```

Example

we are back in **`mergeSort(a, 0, 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	57	61	72	77	89	95	7	15	17	28	32	51	55	62	88	94	97	99

we continue calling

```
center = (0 + 24)/2; // == 12
mergeSort( a, 0, 12 );
mergeSort( a, 13, 24 );
merge( a, 0, 13, 24 );
```

`mergeSort( a,  0, 24 )`

Example

we call `merge(a, 0, 13, 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	57	61	72	77	89	95	7	15	17	28	32	51	55	62	88	94	97	99

```
merge( a,  0, 13, 24 )
mergeSort( a,  0, 24 )
```

Example

we terminate **`mergeSort(a, 0, 24)`**

and the array is 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	72	77	88	89	94	95	97	99

done

```
center = (0 + 24)/2; // == 12
mergeSort( a, 0, 12 );
mergeSort( a, 13, 24 );
merge( a, 0, 13, 24 );
```

```
mergeSort( a,  0, 24 )
```

Run-time Analysis of Merge Sort

Thus, 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:

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

Run-time Analysis of Merge Sort

Simplifying this, we have $n + n\,\mathrm{lg}(n)$

– The run time is $\Theta(n\ln(n))$

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

The Algorithm

However, just because an algorithm has excellent asymptotic properties, this does not mean that it is practical at all levels

Answer (practical):

- If the sub-lists are less than some threshold length, use an algorithm like insertion sort to sort the lists
- Otherwise, use merge sort, again

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

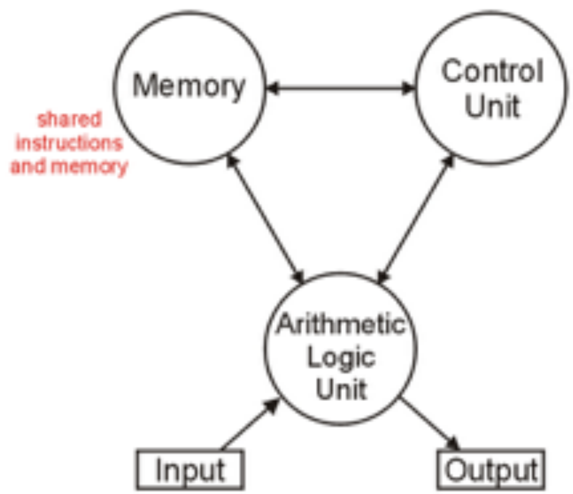
Next we see quick sort

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

Merge Sort

The (likely) first implementation of merge sort was on the ENIAC in 1945 by John von Neumann

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



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