ANALYZING RECURSIVE ALGORITHMS

DISCRETE STRUCTURES II

DARRYL HILL

BASED ON THE TEXTBOOK:

DISCRETE STRUCTURES FOR COMPUTER SCIENCE: COUNTING, RECURSION, AND PROBABILITY

BY MICHIEL SMID

Recursive Algorithms and Recurrences

Analyzing **algorithms** uses a form of counting

- We are counting **significant operations**

We will analyze recursive algorithms and count steps using recurrences

- Recurrences are simply recursive functions

We will analyze the **Mergesort** algorithm by counting the number of **times we copy an element to a new location.**

To find a closed form we will use a new technique called **unfolding.**

Recursive Algorithms and Recurrences

The idea behind Mergesort is, again, to use the power of **recursion**.

We don't know how to sort a list, but...

We can sort a list in the base case...

How do you Sort a List of Length 1 or 0?

4

Luckily it comes presorted

So our base case is satisfied.

Mergesort – Recursive Sorting

Now we want to sort a list of length n by assuming that a recursive call on a shorter list works.

In this case we divide the list in two:

```
sort(item, n):
    if (n ≤ 1):
        return item
    else
        left  ← item[0 : n/2]
        right ← item[n/2 : n]
        sort(left, n/2)
        sort(right, n/2)
```

Now we have two sorted half-lists (by assumption).

We must turn these into one sorted list and then we have proven Mergesort works.

Merging two sorted lists

Assume we have Two Sorted Lists (of lengths x and y)

Could we devise an Algorithm to turn them into a Single Sorted List?

2	5	8	9

1	3	4	6

Compare the front elements of the lists

Merging two sorted lists

Assume we have Two Sorted Lists (of lengths x and y)

Could we devise an Algorithm to turn them into a Single Sorted List?

2	**5**	**8**	**9**

1	**3**	**4**	**6**

1 is smallest, so copy it to the sorted list, and update the pointer

1							

Merging two sorted lists

Assume we have Two Sorted Lists (of lengths x and y)

Could we devise an Algorithm to turn them into a Single Sorted List?

2	5	8	9

1	3	4	6

Repeat the process!

1							

Merging two sorted lists

Assume we have Two Sorted Lists (of lengths x and y)

Could we devise an Algorithm to turn them into a Single Sorted List?

2	5	8	9

1	3	4	6

2 < 3, so append 2 and copy it from original list.

1	2						

Merging two sorted lists

Assume we have Two Sorted Lists (of lengths x and y)

Could we devise an Algorithm to turn them into a Single Sorted List?

2	5	8	9

1	3	4	6

3 < 5, so append 3 and copy it from original list.

1	2	3					

Merging two sorted lists

Assume we have Two Sorted Lists (of lengths x and y)

Could we devise an Algorithm to turn them into a Single Sorted List?

2	5	8	9

1	3	4	6

Process continues…

1	2	3	4				

Merging two sorted lists

Assume we have Two Sorted Lists (of lengths x and y)

Could we devise an Algorithm to turn them into a Single Sorted List?

2	5	8	9

1	3	4	6

Process continues…

1	2	3	4	5			

Merging two sorted lists

Assume we have Two Sorted Lists (of lengths x and y)

Could we devise an Algorithm to turn them into a Single Sorted List?

2	5	8	9

1	3	4	6

When one list is done, we can add the rest to the end at once.

1	2	3	4	5	6		

Merging two sorted lists

Assume we have Two Sorted Lists (of lengths x and y)

Could we devise an Algorithm to turn them into a Single Sorted List?

Final sorted result:

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

Merging two sorted lists

Assume we have Two Sorted Lists (of lengths x and y)

Could we devise an Algorithm to turn them into a Single Sorted List?

Final sorted result:

1	2	3	4	5	6	8	9

We want to count the number of times an element is copied to a new location...

Algorithm: Merge

```
merge(left, right):
    j = 0, k = 0
    for i ∈ [0,len(left)+ len(right)]:
        if left[j] < right[k]:
            item[i] ← left[j]   ; j++
        else:
            item[i] ← right[k] ; k++
    return item
```

Notice the **for-loop** executes

`len(left) + len(right)`

times.

Each iteration copies one element into another list.

Therefore there are $n_1 + n_2$ copies made

(the sum of the lengths of both lists).

Now we have a complete idea for Sorting

Divide my problem in half until it is a manageable size.

Sort(List, n):

Divide the List (of Length n)

into Two Lists (of Length $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ respectively)

Sort the Sublists

Merge them into a Single Sorted List

Divide and Conquer / Recursive Approach

Sort works on a shorter list by Assumption

Merge Sort Demo

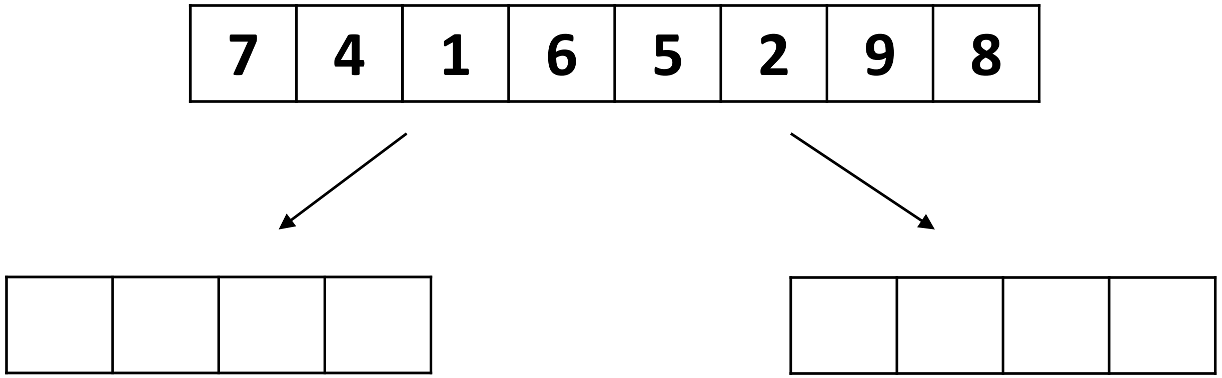
7	**4**	**1**	**6**	**5**	**2**	**9**	**8**

Merge Sort Demo

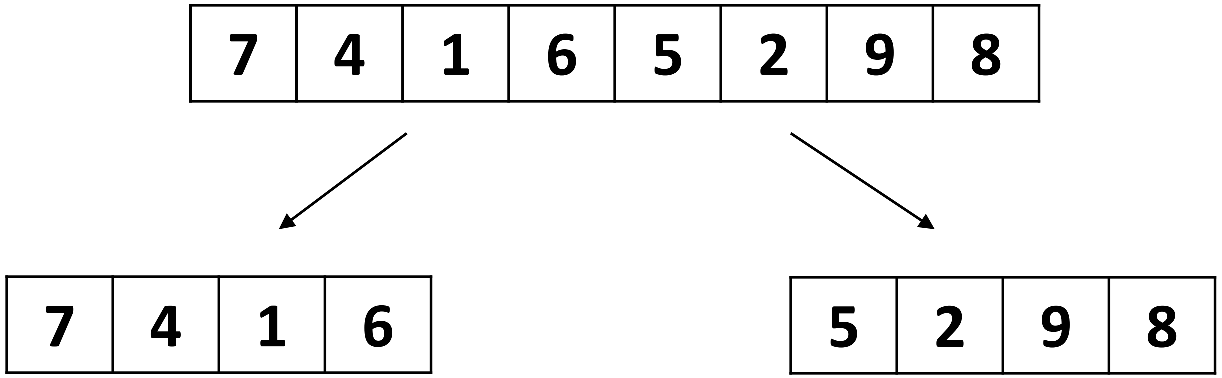
7	**4**	**1**	**6**	**5**	**2**	**9**	**8**

We cut our list in two at each step!

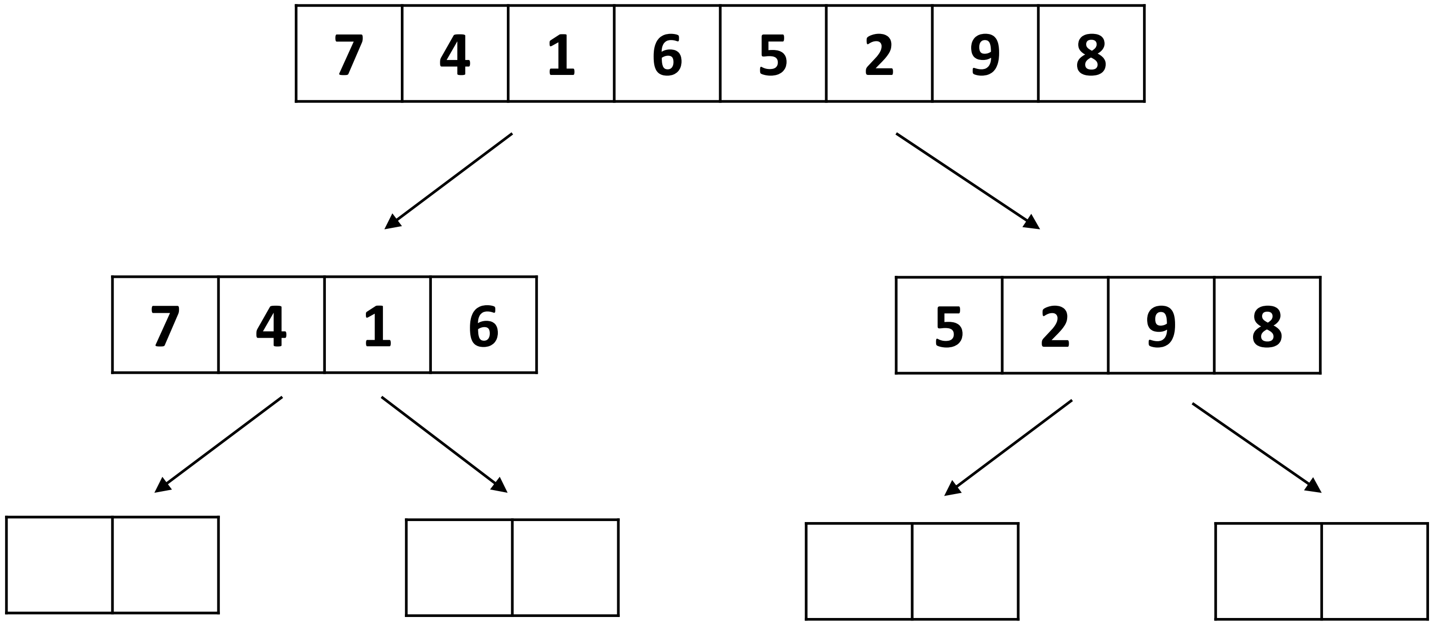
Merge Sort Demo



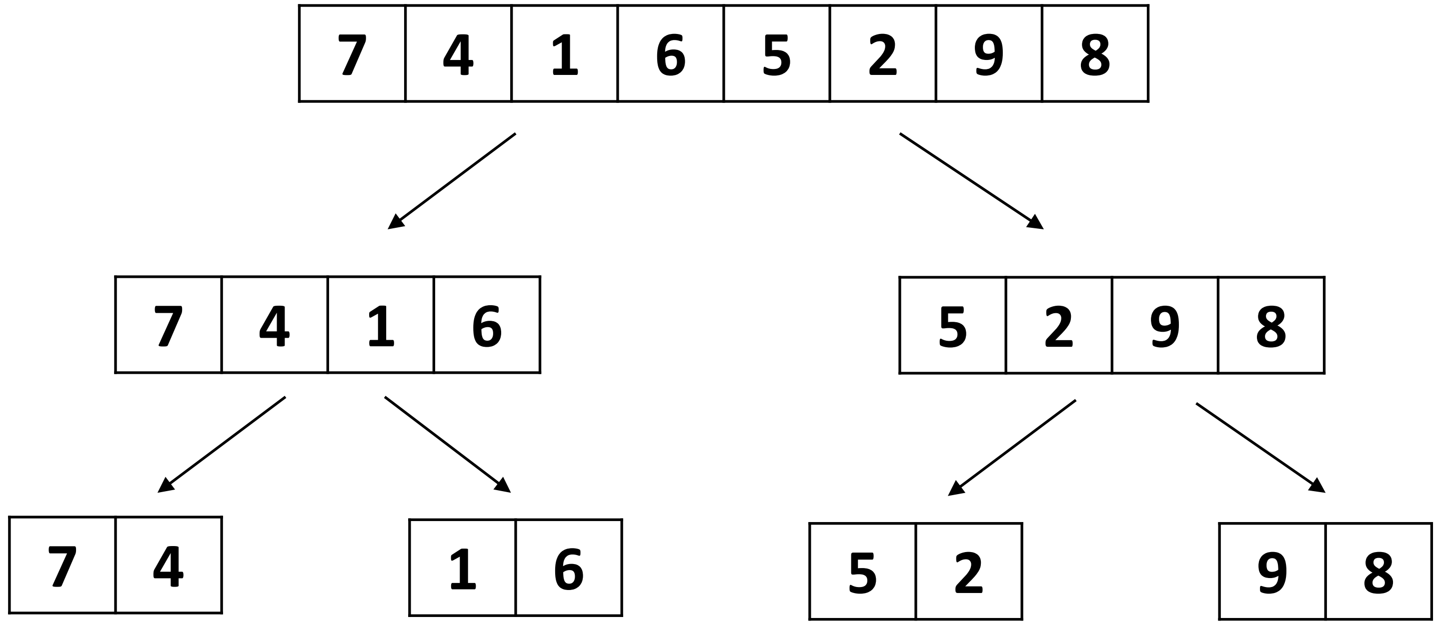
Merge Sort Demo



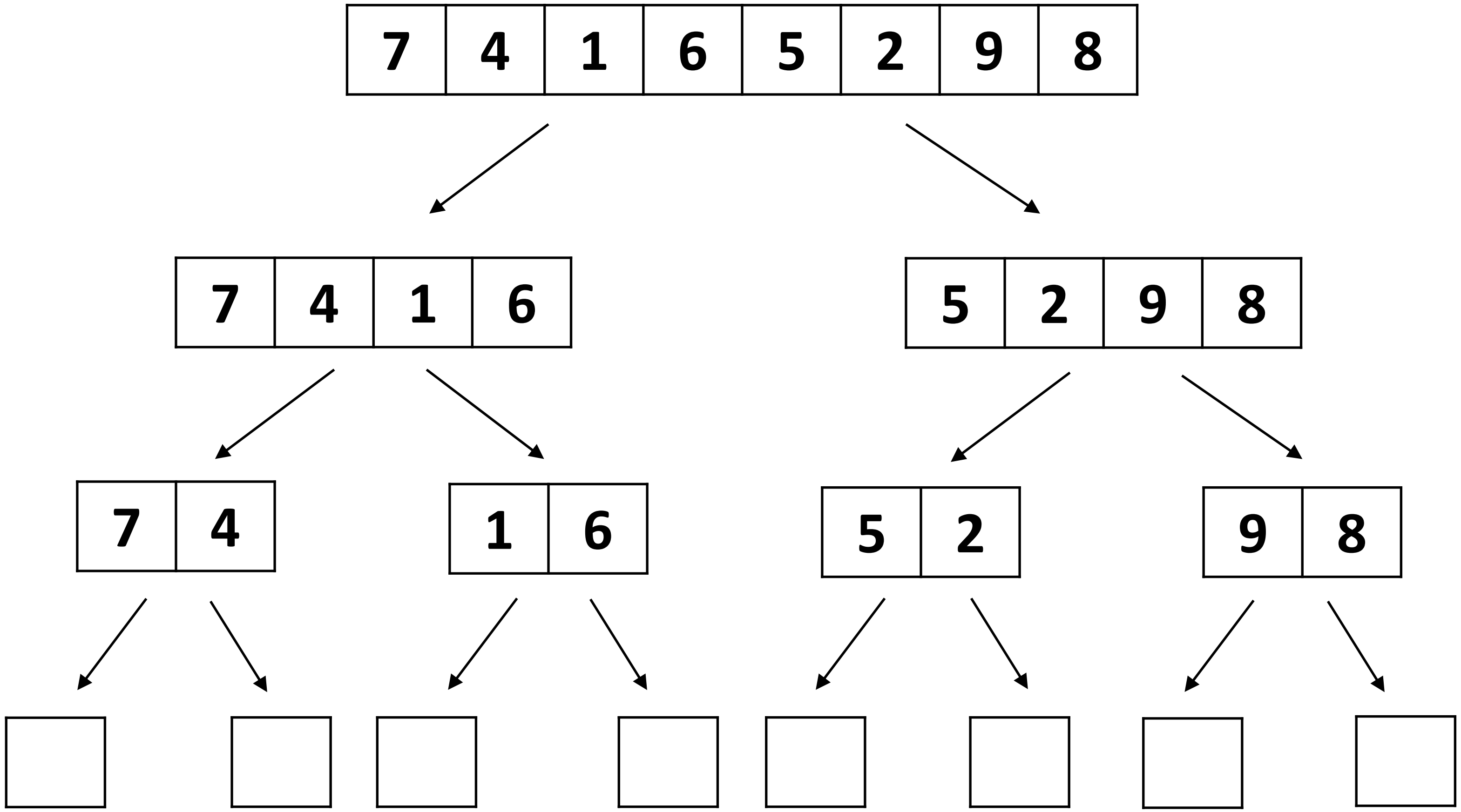
Merge Sort Demo



Merge Sort Demo

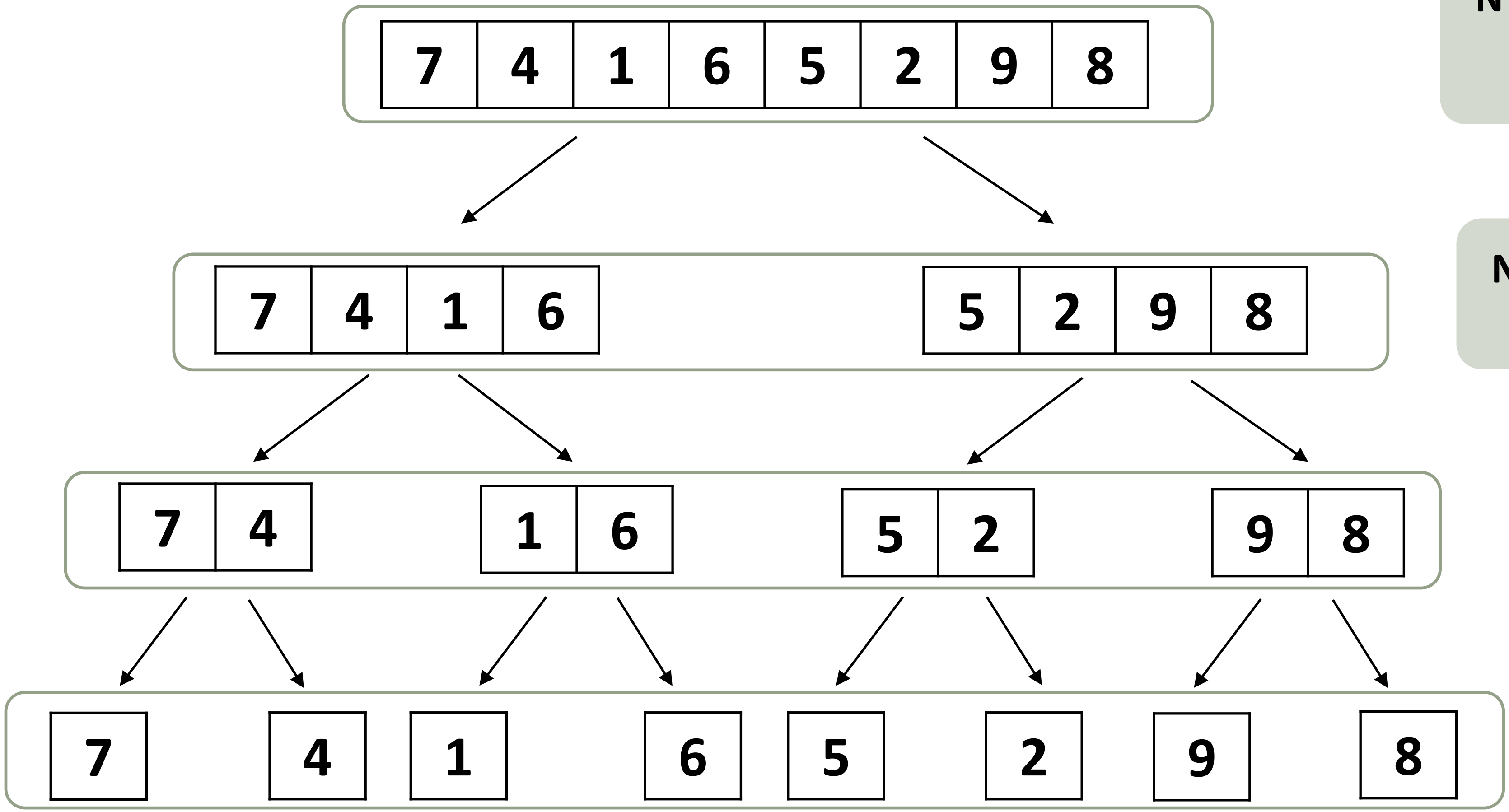


Merge Sort Demo



Merge Sort Demo

How many times do we copy an element? Start with a list of length **N**



N copies are made in total at level 1 (into level 2)

N copies are made in total at level 2

N copies are made in total at level 3

Nothing is copied at level 4

Merge Sort Demo

Now the merging...

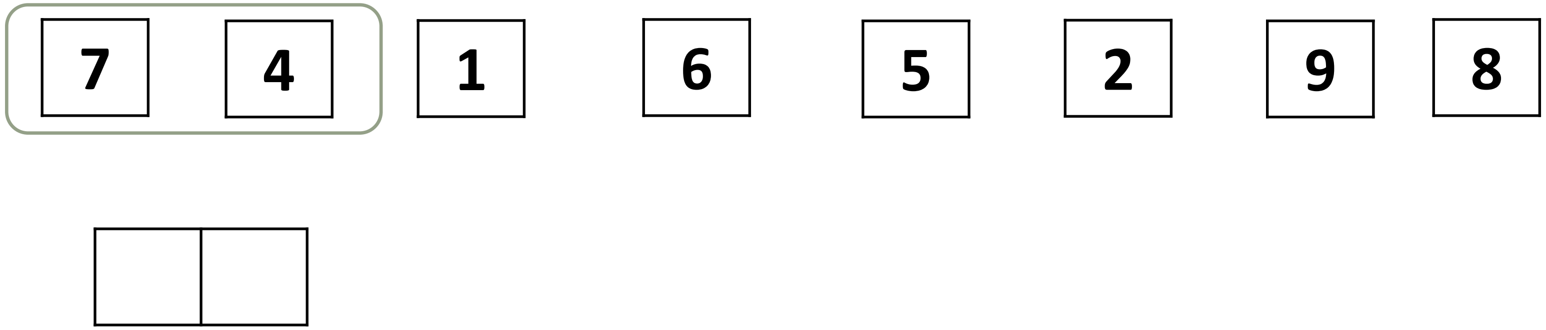
7 4 1 6 5 2 9 8

Merge Sort Demo

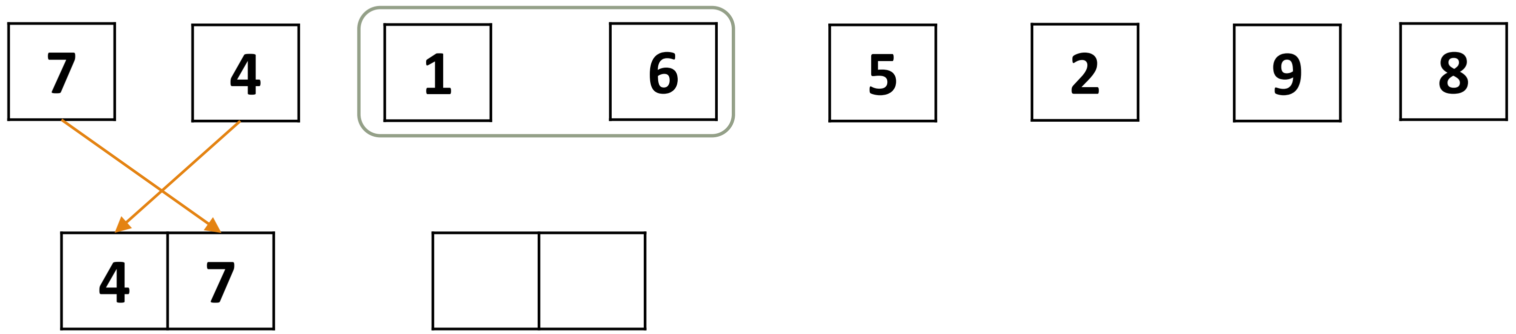
7 4 1 6 5 2 9 8

We employ the same merging technique we saw earlier

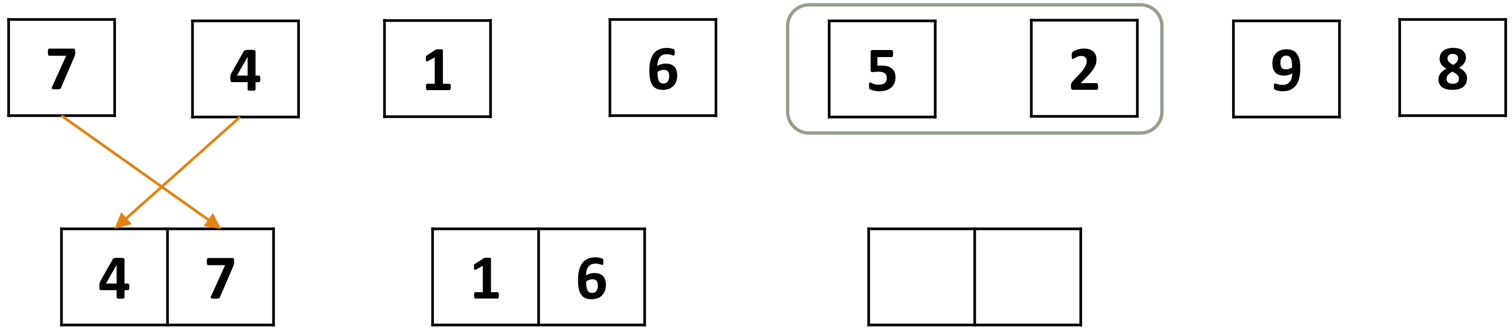
Merge Sort Demo



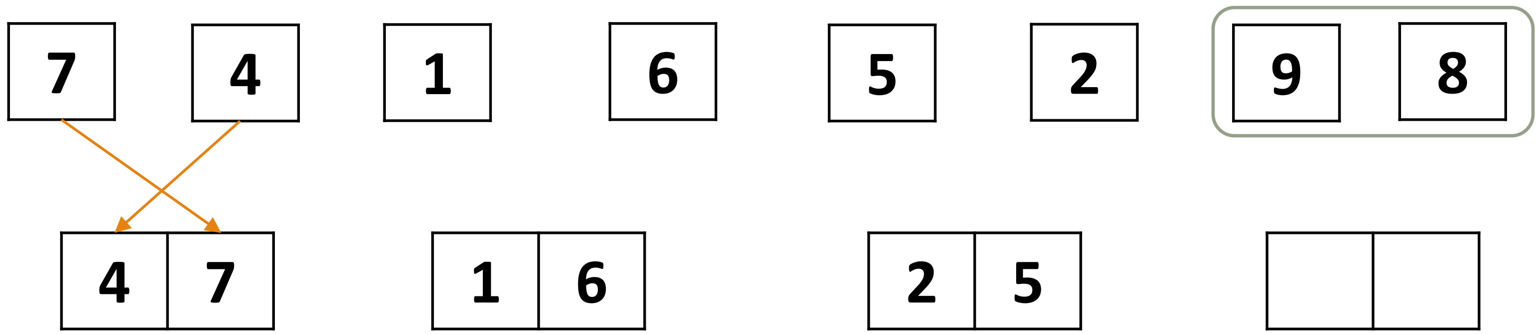
Merge Sort Demo



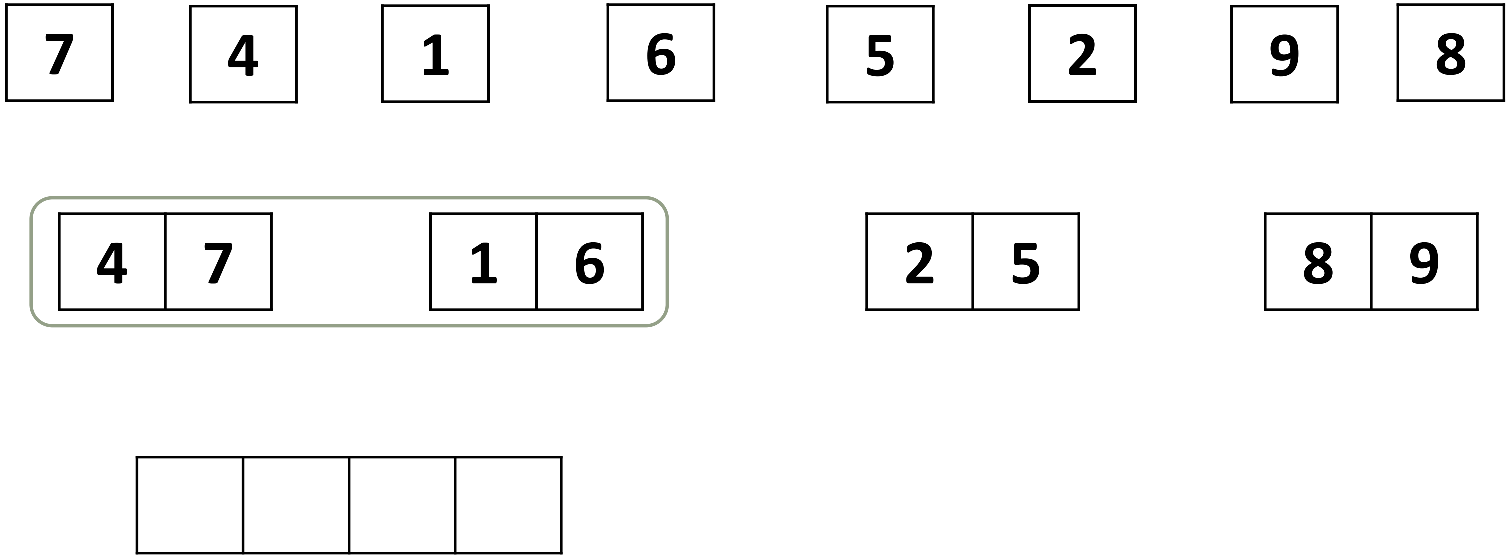
Merge Sort Demo



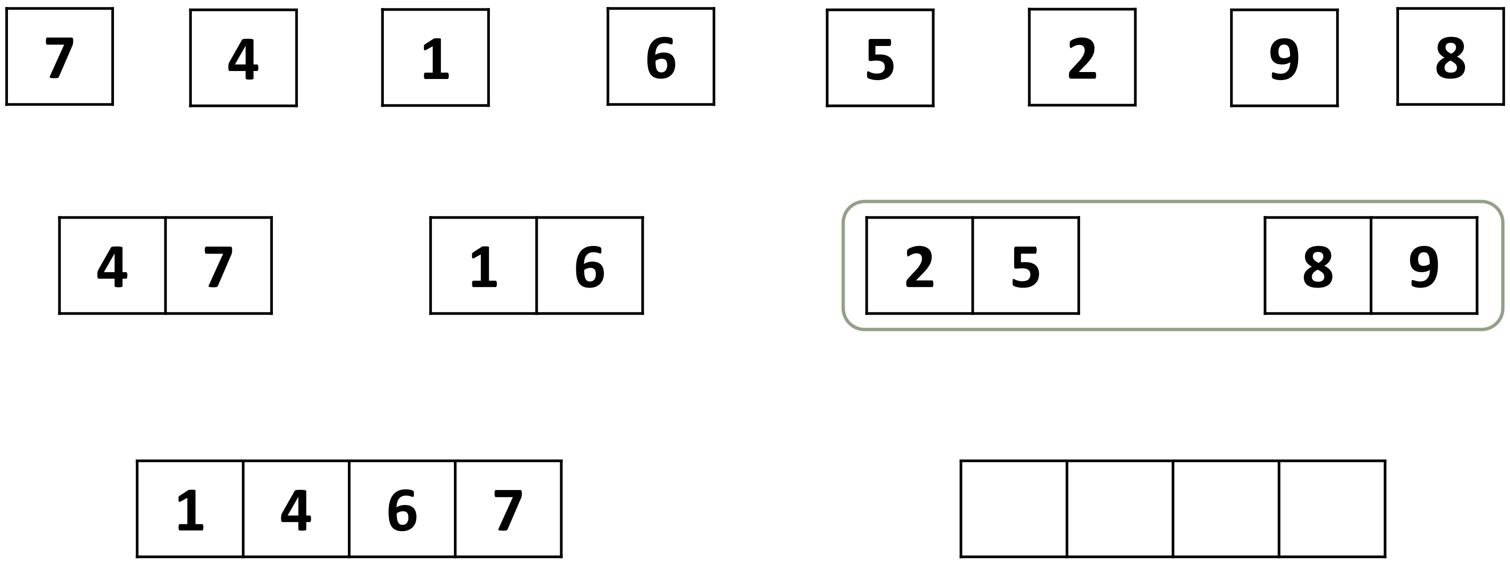
Merge Sort Demo



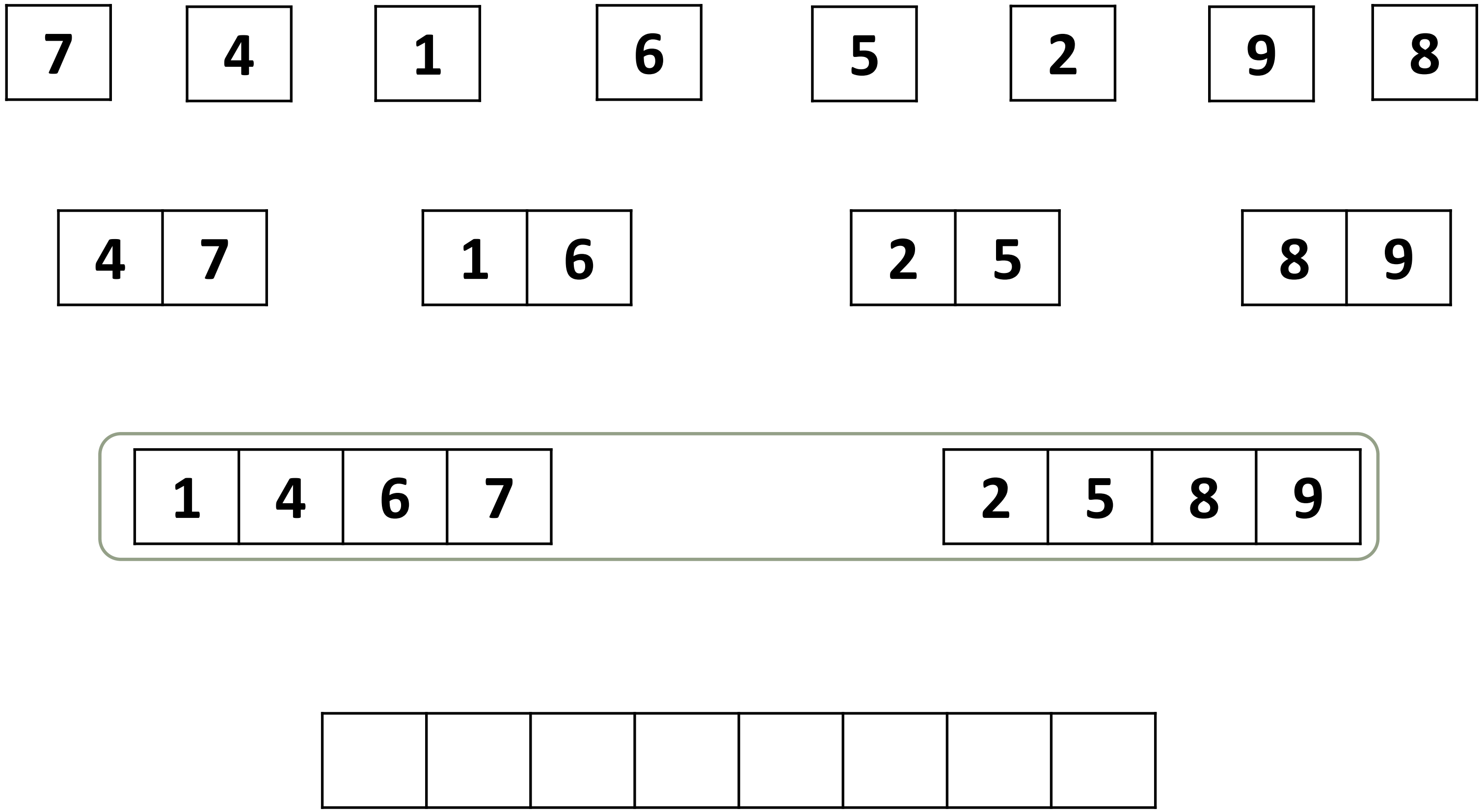
Merge Sort Demo



Merge Sort Demo



Merge Sort Demo



Merge Sort Demo

7	4	1	6	5	2	9	8

4	7

1	6

2	5

8	9

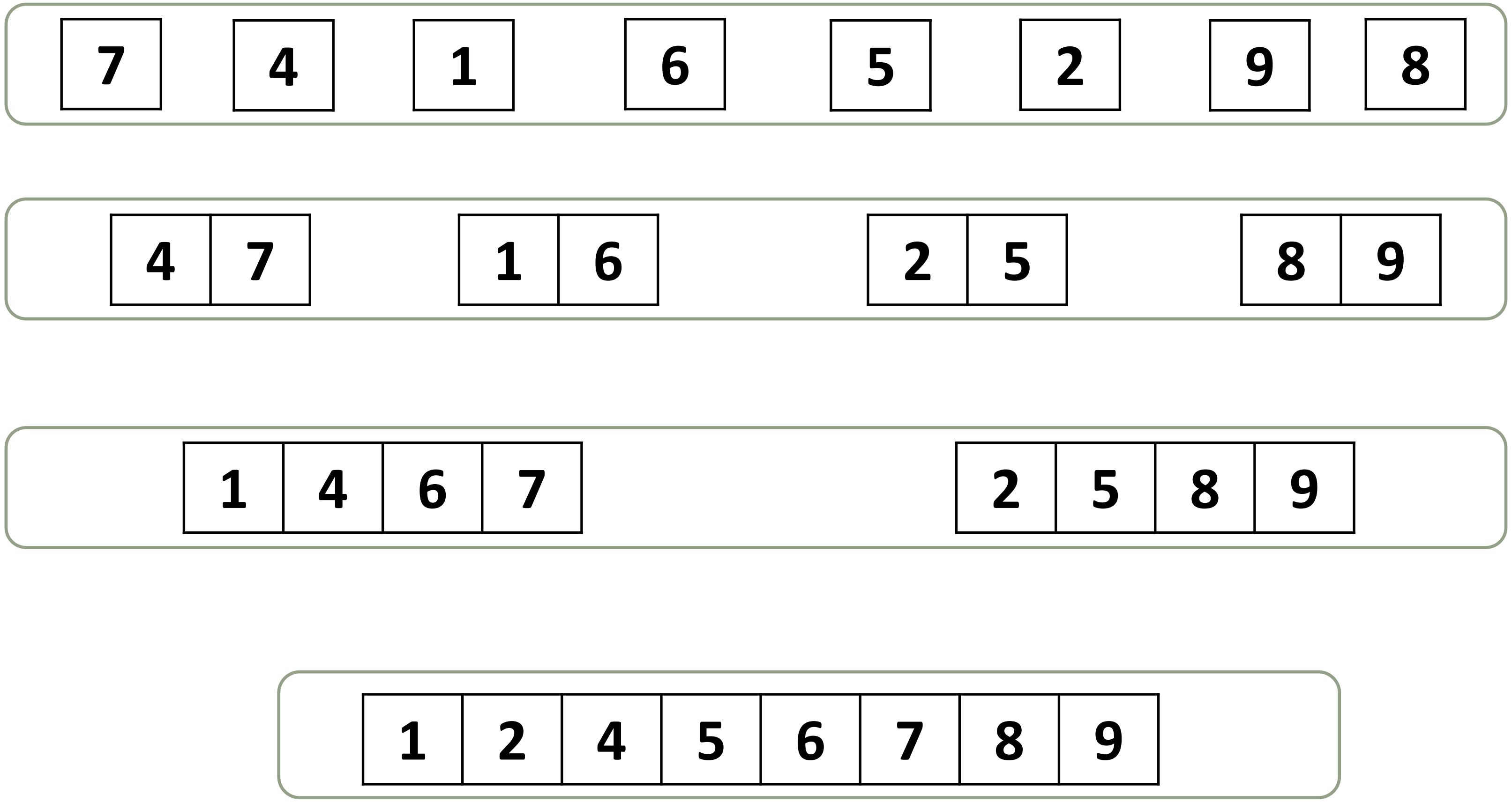
1	4	6	7

2	5	8	9

1	2	4	5	6	7	8	9

Final sorted result:

Merge Sort: Efficiency



Nothing happens at level 4

N copies are made in total at level 3 (to merge the lists returned from level 4)

N copies are made in total at level 2

N copies are made in total at level 1

If the Unsorted List is of Length 1, Return

Otherwise, Divide the list in Half (approximately) into Two Sublists

Recursively Call Mergesort on Each Sublist

Merge the Return Values

```
sort(item, n):
    if (n ≤ 1):
        return item
    else
        left  ← item[0 : n/2]
        right ← item[n/2 : n]
        sort(left, n/2)
        sort(right, n/2)
        merge(left, right)
```

```
merge(left, right):
    j = 0, k = 0
    for i ∊ [0, len(left)+len(right)):
        if left[j] < right[k]:
            item[i] ← left[j]  ; j++
        else:
            item[i] ← right[k] ; k++
    return item
```

Analyzing Sort

for an array of Length n

(to simplify assume $n = 2^m$ for $m \in \mathbb{Z}^+$)

```
sort(item, n):
    if (n ≤ 1):
        return item
    else
        left  ← item[0 : n/2]
        right ← item[n/2 : n]
        sort(left, n/2)
        sort(right, n/2)
        merge(left, right)
```

n/2 copies

n/2 copies

n/2 + n/2
= n copies total

For a call to sort on a list of n items.

Then we call $merge$

Analyzing Merge

for an array of Length n

(to simplify assume $n = 2^m$ for $m \in \mathbb{Z}^+$)

n/2 + n/2

```
merge(left, right):
    j = 0, k = 0
    for i ∊ [0, len(left)+len(right)):
        if left[j] < right[k]:
            item[i] ← left[j]  ; j++
        else:
            item[i] ← right[k] ; k++
    return item
```

Each iteration makes 1 copy

n/2 + n/2

= n copies total

For a call to merge on 2 lists of $n/2$ items each.

Analyzing MergeSort

```
sort(item, n):
    if (n ≤ 1):
        return item
    else
        left  ← item[0 : n/2]
        right ← item[n/2 : n]
        sort(left, n/2)
        sort(right, n/2)
        merge(left, right)
```

```
merge(left, right):
    j = 0, k = 0
    for i ∊ [0, len(left)+len(right)):
        if left[j] < right[k]:
            item[i] ← left[j]  ; j++
        else:
            item[i] ← right[k] ; k++
    return item
```

Let $T(n)$ be the number of copy operations for a call to MergeSort

A call to *Sort* will Split an Array of Length n in two : **n copies**

A call to *Merge* will Merge Two Arrays of Length $^n/_2$: **n copies**

Two recursive calls to *Sort* arrays of Length $^n/_2$: $T(^n/_2) + T(^n/_2)$

$$T(n) = T(^n/_2) + T(^n/_2) + 2n$$

Analyzing Mergesort Performance

let **T(n)** be the Number of Copies Made to Sort array of Length n

```
sort(item, n):
    if (n ≤ 1):
        return item
    else
        left  ← item[0 : n/2]
        right ← item[n/2 : n]
        sort(left, n/2)
        sort(right, n/2)
        merge(left, right)
```

```
merge(left, right):
    j = 0, k = 0
    for i ∈ [0, len(left)+len(right)):
        if left[j] < right[k]:
            item[i] ← left[j]   ; j++
        else:
            item[i] ← right[k]  ; k++
    return item
```

$$T(n) = T(^n/_2) + T(^n/_2) + 2n$$
$$= 2(T(^n/_2)) + 2n$$

Analyze using
Unfolding

Analyzing Mergesort Performance

let **T(n)** be the Number of Copies Made to Sort array of Length n

```
sort(item, n):
    if (n ≤ 1):
        return item
    else
        left  ← item[0 : n/2]
        right ← item[n/2 : n]
        sort(left, n/2)
        sort(right, n/2)
        merge(left, right)
```

```
merge(left, right):
    j = 0, k = 0
    for i ∈ [0, len(left)+len(right)):
        if left[j] < right[k]:
            item[i] ← left[j]  ; j++
        else:
            item[i] ← right[k] ; k++
    return item
```

$$T(n) = T(n/2) + T(n/2) + 2n$$
$$= 2(T(n/2)) + 2n$$

Analyze using **Unfolding**

$$T(n/2) = T(n/4) + T(n/4) + 2(n/2)$$
$$= 2(T(n/4)) + 2(n/2)$$

Analyzing Mergesort Performance

let $T(n)$ be the Number of Copies Made to Sort array of Length n

```
sort(item, n):
    if (n ≤ 1):
        return item
    else
        left  ← item[0 : n/2]
        right ← item[n/2 : n]
        sort(left, n/2)
        sort(right, n/2)
        merge(left, right)
```

```
merge(left, right):
    j = 0, k = 0
    for i ∊ [0, len(left)+len(right)):
        if left[j] < right[k]:
            item[i] ← left[j]   ; j++
        else:
            item[i] ← right[k] ; k++
    return item
```

$$
\begin{aligned}
T(n) &= T(^n/_2) + T(^n/_2) + 2n \\
&= 2(T(^n/_2)) + 2n \\
&= 2(2(T(^n/_4)) + 2(^n/_2)) + 2n
\end{aligned}
$$

Analyze using **Unfolding**

$$
\begin{aligned}
T(^n/_2) &= T(^n/_4) + T(^n/_4) + 2(^n/_2) \\
&= 2(T(^n/_4)) + 2(^n/_2)
\end{aligned}
$$

Analyzing Mergesort Performance

let **T(n)** be the Number of Copies Made to Sort array of Length n

```
sort(item, n):
    if (n ≤ 1):
        return item
    else
        left  ← item[0 : n/2]
        right ← item[n/2 : n]
        sort(left, n/2)
        sort(right, n/2)
        merge(left, right)
```

```
merge(left, right):
    j = 0, k = 0
    for i ∊ [0, len(left)+len(right)):
        if left[j] < right[k]:
            item[i] ← left[j]   ; j++
        else:
            item[i] ← right[k] ; k++
    return item
```

$$
\begin{aligned}
T(n) &= T(n/2) + T(n/2) + 2n \\
&= 2(T(n/2)) + 2n \\
&= 2(2(T(n/4)) + 2(n/2)) + 2n \\
&= 2*2(T(n/4)) + 2n + 2n
\end{aligned}
$$

Analyze using
Unfolding

$$
\begin{aligned}
T(n/2) &= T(n/4) + T(n/4) + 2(n/2) \\
&= 2(T(n/4)) + 2(n/2)
\end{aligned}
$$

Analyzing Mergesort Performance

let **T(n)** be the Number of Copies Made to Sort array of Length n

```
sort(item, n):
    if (n ≤ 1):
        return item
    else
        left  ← item[0 : n/2]
        right ← item[n/2 : n]
        sort(left, n/2)
        sort(right, n/2)
        merge(left, right)
```

```
merge(left, right):
    j = 0, k = 0
    for i ∈ [0, len(left)+len(right)):
        if left[j] < right[k]:
            item[i] ← left[j]   ; j++
        else:
            item[i] ← right[k] ; k++
    return item
```

$$
\begin{aligned}
T(n) &= T(^n/_2) + T(^n/_2) + 2n \\
&= 2(T(^n/_2)) + 2n \\
&= 2(2(T(^n/_4)) + 2(^n/_2)) + 2n \\
&= 2*2(T(^n/_4)) + 2n + 2n
\end{aligned}
$$

Analyze using **Unfolding**

$$
\begin{aligned}
T(^n/_4) &= T(^n/_8) + T(^n/_8) + 2(^n/_4) \\
&= 2(T(^n/_8)) + 2(^n/_4)
\end{aligned}
$$

Analyzing Mergesort Performance

let **T(n)** be the Number of Copies Made to Sort array of Length n

```
sort(item, n):
    if (n ≤ 1):
        return item
    else
        left  ← item[0 : n/2]
        right ← item[n/2 : n]
        sort(left, n/2)
        sort(right, n/2)
        merge(left, right)
```

```
merge(left, right):
    j = 0, k = 0
    for i ∈ [0, len(left)+len(right)):
        if left[j] < right[k]:
            item[i] ← left[j]   ; j++
        else:
            item[i] ← right[k] ; k++
    return item
```

$$
\begin{aligned}
T(n) &= T(^n/_2) + T(^n/_2) + 2n \\
&= 2(T(^n/_2)) + 2n \\
&= 2(2(T(^n/_4)) + 2(^n/_2)) + 2n \\
&= 2*2(T(^n/_4)) + 2n + 2n \\
&= 2*2(2(T(^n/_8)) + 2(^n/_4)) + 2n + 2n
\end{aligned}
$$

Analyze using
Unfolding

$$
\begin{aligned}
T(^n/_4) &= T(^n/_8) + T(^n/_8) + 2(^n/_4) \\
&= 2(T(^n/_8)) + 2(^n/_4)
\end{aligned}
$$

Analyzing Mergesort Performance

let **`T(n)`** be the Number of Copies Made to Sort array of Length n

```
sort(item, n):
    if (n ≤ 1):
        return item
    else
        left  ← item[0 : n/2]
        right ← item[n/2 : n]
        sort(left, n/2)
        sort(right, n/2)
        merge(left, right)
```

```
merge(left, right):
    j = 0, k = 0
    for i ∊ [0, len(left)+len(right)):
        if left[j] < right[k]:
            item[i] ← left[j]   ; j++
        else:
            item[i] ← right[k] ; k++
    return item
```

$$
\begin{aligned}
T(n) &= T(^n/_2) + T(^n/_2) + 2n \\
&= 2(T(^n/_2)) + 2n \\
&= 2(2(T(^n/_4)) + 2(^n/_2)) + 2n \\
&= 2*2(T(^n/_4)) + 2n + 2n \\
&= 2*2(2(T(^n/_8)) + 2(^n/_4)) + 2n + 2n \\
&= 2*2*2(T(^n/_8)) + 2n + 2n + 2n
\end{aligned}
$$

Analyze using **Unfolding**

...after k steps...

$$= 2^k(T(^n/_{2^k})) + 2nk$$

Analyzing Mergesort Performance

let `T(n)` be the Number of Copies Made to Sort array of Length n

```
sort(item, n):
    if (n ≤ 1):
        return item
    else
        left  ← item[0 : n/2]
        right ← item[n/2 : n]
        sort(left, n/2)
        sort(right, n/2)
        merge(left, right)
```

```
merge(left, right):
    j = 0, k = 0
    for i ∊ [0, len(left)+len(right)):
        if left[j] < right[k]:
            item[i] ← left[j]  ; j++
        else:
            item[i] ← right[k] ; k++
    return item
```

$T(n) = 2^k(T(n/2^k)) + 2nk$ *This is the pattern we have unfolded*

...How Many Steps (k) Until we Evaluate `T(1)` ***?***

T(1) occurs when $n = 2^k$

$$= 2^k(T(n/2^k)) + 2nk$$
$$= n(T(1)) + 2nk$$
$$= n(T(1)) + 2n(\log_2(n))$$
$$= 2n(\log_2(n))$$

$n = 2^k$

Take the log of both sides

$\log_2 n = k$

Proving the Closed Form

let $T(n)$ be the Number of Copies Made to Sort array of Length n

$$T(1) = 0 \qquad \text{Base case}$$
$$T(n) = T(n/2) + T(n/2) + 2n \qquad \text{Recursive case}$$

$$T(n) = 2n(\log_2(n))$$

This is our guess for a closed form.

Base Case T(1):

$$\begin{aligned} T(n) &= 2n(\log_2(n)) \\ T(1) &= 2\cdot 1\cdot(\log_2(1)) \\ &= 0 \end{aligned}$$

Proving the Closed Form

let **T(n)** be the Number of Copies Made to Sort array of Length n

$\boxed{T(1) = 0}$ Base case

$T(n) = T(n/2) + T(n/2) + 2n$ Recursive case

$T(n) = 2n(\log_2(n))$ ***This is our guess for a closed form***

Base Case T(1):

$T(n) = 2n(\log_2(n))$

$$\boxed{\begin{aligned} T(1) &= 2 \cdot 1 \cdot (\log_2(1)) \\ &= 0 \end{aligned}}$$

Base case holds

Proving the Closed Form

let $T(n)$ be the Number of Copies Made to Sort array of Length n

$$T(1) = 0 \qquad \text{Base case}$$
$$T(n) = T(n/2) + T(n/2) + 2n \qquad \text{Recursive case}$$

$$\boxed{T(n) = 2n(\log_2(n))}$$

This is our guess for a closed form.

Inductive Hypothesis: Assume that $T(n/2) = 2(n/2)(\log_2(n/2))$

$$\begin{aligned}
T(n) &= 2 \cdot T(n/2) + 2n \\
&= 2 \cdot 2(n/2)(\log_2(n/2)) + 2n \\
&= 2 \cdot n(\log_2 n - \log_2 2) + 2n \\
&= 2 \cdot n(\log_2 n - 1) + 2n \\
&= 2 \cdot n \cdot \log_2 n - 2n + 2n \\
&\boxed{= 2 \cdot n \cdot \log_2 n}
\end{aligned}$$

Thus

$T(n) = n(\log_2(n))$

by induction

Analyzing Mergesort Performance

let `T(n)` be the Number of Copies Made to Sort array of Length n

$$T(1) = 0 \qquad \text{Base case}$$
$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2n \qquad \text{Recursive case}$$

However, that is for $n = 2^m$. The actual recurrence looks like above.

Inductive Hypothesis: **Assume that $T(k) \le 2(k)(\log_2(k))$ for $k < n$**

If $\lfloor n/2 \rfloor = n/2$ then we have the same recurrence as before.
Thus assume that

$$\lfloor n/2 \rfloor = {}^{n-1}/_2 \quad \text{and} \quad \lceil n/2 \rceil = {}^{n+1}/_2$$

Base Case: **$T(0) = 0$ and $T(1) = 0$**

Analyzing Mergesort Performance

let $\mathbf{T(n)}$ be the Number of Copies Made to Sort array of Length n

$$T(1) = 0 \qquad \text{Base case}$$

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2n \qquad \text{Recursive case}$$

Inductive Hypothesis: **Assume that** $T(k) \leq 2(k)(\log_2(k))$ for $k < n$

$$\begin{aligned}
T(n) &= T(^{n-1}/_2) + T(^{n+1}/_2) + 2n \\
&= 2(^{n-1}/_2)(\log_2(^{n-1}/_2)) + 2(^{n+1}/_2)(\log_2(^{n+1}/_2)) + 2n \\
&= (n-1)(\log_2(^{n-1}/_2)) + (n+1)(\log_2(^{n+1}/_2)) + 2n \\
&= (n-1)(\log_2(n-1)-1) + (n+1)(\log_2(n+1)-1) + 2n \\
&= (n-1)(\log_2(n-1)) + (n+1)(\log_2(n+1)) \\
&\leq (n - 1 + n + 1)\log_2 n \\
&\leq 2 \cdot n \cdot \log_2 n
\end{aligned}$$

Kronk

Binary Search

Binary Search Algorithm:

Check the middle item

- If item is what we're looking for:
  - Then Done
- Elif item is > what we're looking for:
  - Search the left half
- Elif item is < what we're looking for:
  - Search the right half

Binary Search

1	**2**	**4**	**6**	**7**	**10**	**14**	**16**	**17**	**21**	**22**	**34**	**41**

Our list is **sorted**.

Searching for: 17

Binary Search

1	**2**	**4**	**6**	**7**	**10**	**14**	**16**	**17**	**21**	**22**	**34**	**41**

We test the **midpoint** first!

Searching for: 17

Binary Search

1	2	4	6	7	10	14	16	17	21	22	34	41

17 is **greater than** 14, so, if it is in the list at all, it **must** be in the second half.

Searching for: 17

Binary Search

1	2	4	6	7	10	14	16	17	21	22	34	41

The first half is **eliminated**. We repeat, testing the midpoint of the remainder.

Searching for: 17

Binary Search

1	2	4	6	7	10	14	16	17	21	22	34	41

17 is **less than** 22, so, if it is in the list at all,
it **must** be in the first half of this sublist.

Searching for: 17

Binary Search

1	2	4	6	7	10	14	16	17	21	22	34	41

We check the midpoint again, and find it is equal to 17.

17 found at index 8, after just *three* comparisons.
(Would be nine for linear search)

Binary Search

```
BinarySearch(item, L, start, end):
   mid = (start + end)/2 ;
   temp = L[mid] ;
   if item == temp: return true;
   if start == end: return false;
   else if item < temp:
      BinarySearch(item, L, start, mid -1)
   else:
      BinarySearch(item, L, mid+1, end)
```

Analysis: count memory accesses

$T(1) = 1, \quad n = 1$

$T(n) \leq 1 + T\left(\frac{n}{2}\right), \quad n > 1$

Using unfolding to find a pattern:

$$T(n) \leq 1 + T\left(\frac{n}{2}\right)$$
$$\leq 1 + 1 + T\left(\frac{n}{4}\right)$$
$$\leq 1 + 1 + 1 + T\left(\frac{n}{2^3}\right)$$

Binary Search

```
BinarySearch(item, L, start, end):
   mid = (start + end)/2 ;
   temp = L[mid] ;
   if item == temp: return true;
   if start == end: return false;
   else if item < temp:
      BinarySearch(item, L, start, mid -1)
   else:
      BinarySearch(item, L, mid+1, end)
```

$$T(n) \le k + T\left(\frac{n}{2^k}\right)$$

$$\le k + T(1) = k + 1$$

$$2^k = n$$

$$k = \log n$$

$$\therefore T(n) \le \log n + 1$$

This is our guess. We must prove using induction.

Binary Search

What we know: $T(1) = 1$

$T(n) \le 1 + T(\frac{n}{2})$

To show: $T(n) \le 1 + \log n$

Base case: $T(1) \le 1 + \log 1$

$= 1$

```
BinarySearch(item, L, start, end):
   mid = (start + end)/2 ;
   temp = L[mid] ;
   if item == temp: return true;
   if start == end: return false;
   else if item < temp:
      BinarySearch(item, L, start, mid -1)
   else:
      BinarySearch(item, L, mid+1, end)
```

Binary Search

```
BinarySearch(item, L, start, end):
    mid = (start + end)/2 ;
    temp = L[mid] ;
    if item == temp: return true;
    if start == end: return false;
    else if item < temp:
        BinarySearch(item, L, start, mid -1)
    else:
        BinarySearch(item, L, mid+1, end)
```

What we know: $T(1) = 1$

$T(n) \le 1 + T(\frac{n}{2})$

To show: $T(n) \le 1 + \log n$

Inductive Hypothesis:

$T(\frac{n}{2}) \le 1 + \log(\frac{n}{2})$

$$
\begin{aligned}
T(n) &\le 1 + T(\tfrac{n}{2}) \\
&\le 1 + 1 + \log \tfrac{n}{2} \\
&\le 1 + 1 + \log n - \log 2 \\
&\le 1 + \log n
\end{aligned}
$$

Things to know about recursion:

1. How to prove a closed form of a recursive function using induction.
2. How to map a problem to the Fibonacci Sequence (also, the Fibonacci sequence).
3. How to analyze a recursive algorithm (by finding a recursive function)
4. Using unfolding on a recurrence to find a closed form.