RECURSION –ANALYZING ALGORITHMS USING RECURRENCES

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

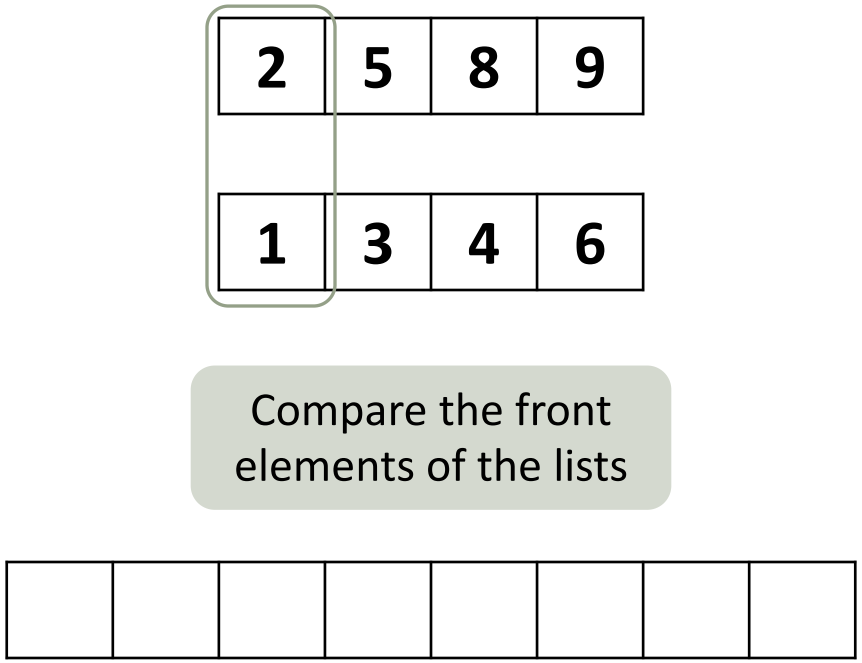
- Recurrences are simply recursive functions
- You have seen this before, however…
- …to be done properly you should prove the closed form using induction

Start by counting the number of **memory accesses** in **Mergesort**

Merging two sorted lists

If you were Provided Two Sorted Lists (of lengths x and y)

Could you Merge them into a Single Sorted List (of length x+y)?



Merging two sorted lists

If you were Provided Two Sorted Lists (of lengths x and y)

Could you Merge them into a Single Sorted List (of length x+y)?

2	5	8	9

	3	4	6

1 is smallest, so append it to the sorted list, and remove it.

1							

Merging two sorted lists

If you were Provided Two Sorted Lists (of lengths x and y)

Could you Merge them into a Single Sorted List (of length x+y)?

2	5	8	9

3	4	6

Repeat the process!

1							

Merging two sorted lists

If you were Provided Two Sorted Lists (of lengths x and y)

Could you Merge them into a Single Sorted List (of length x+y)?

5	8	9

3	4	6

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

1	2						

Merging two sorted lists

If you were Provided Two Sorted Lists (of lengths x and y)

Could you Merge them into a Single Sorted List (of length x+y)?

5	8	9

4	6

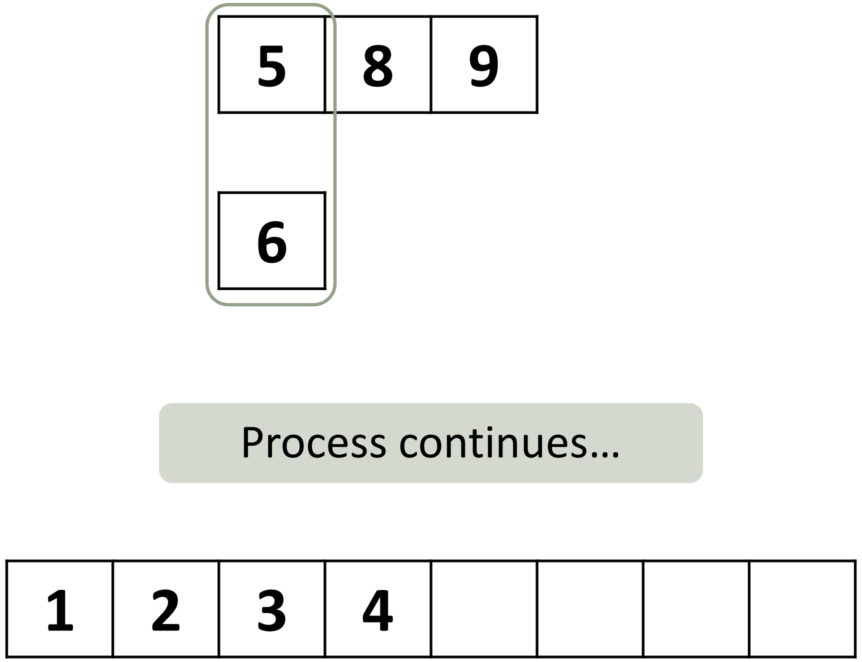
3 < 5, so append 3 and remove it from original list.

1	2	3					

Merging two sorted lists

If you were Provided Two Sorted Lists (of lengths x and y)

Could you Merge them into a Single Sorted List (of length x+y)?



Merging two sorted lists

If you were Provided Two Sorted Lists (of lengths x and y)

Could you Merge them into a Single Sorted List (of length x+y)?

8	9

6

Process continues…

1	2	3	4	5			

Merging two sorted lists

If you were Provided Two Sorted Lists (of lengths x and y)

Could you Merge them into a Single Sorted List (of length x+y)?

8	9

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

1	2	3	4	5	6		

Merging two sorted lists

If you were Provided Two Sorted Lists (of lengths x and y)

Could you Merge them into a Single Sorted List (of length x+y)?

Final sorted result:

1	2	3	4	5	6	8	9

Merging two sorted lists

If you were Provided Two Sorted Lists (of lengths x and y)

Could you Merge them into a Single Sorted List (of length x+y)?

Final sorted result:

1	2	3	4	5	6	8	9

Since we remove an item from a list after each comparison, we require len(list1) + len(list2) memory accesses!

How do you Sort a Singleton List (i.e., a List of Length 1)?

↓

Procedure for Sorting a List:

Divide the Unsorted List (of Length L)

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

Sort the Sublists and then Merge them into a Single Sorted List

How do you Sort a Singleton List (i.e., a List of Length 1)?

Procedure for Sorting a List:

Divide the Unsorted List (of Length L)

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

Sort the Sublists and then Merge them into a Single Sorted List

Can the Procedure Itself be Used to Solve this Subproblem?

Recursive Approach

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 and Merge the Return Values

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

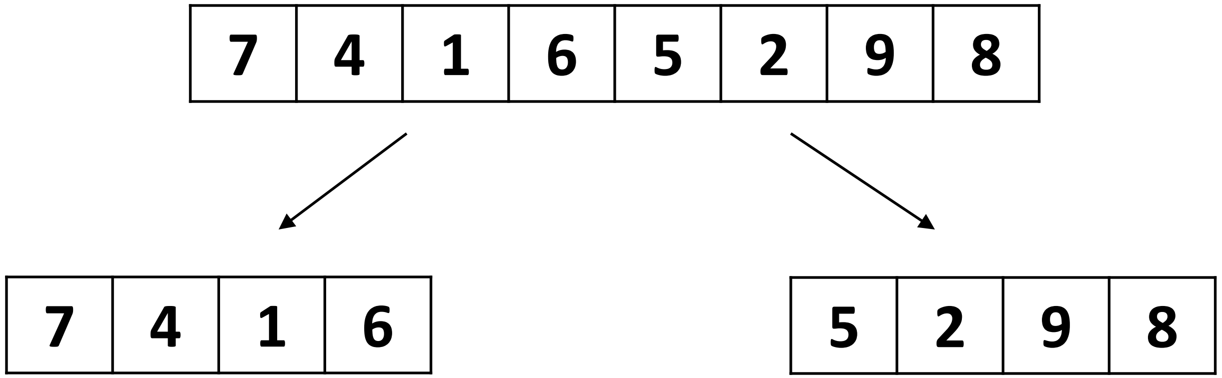
```
merge(left, right):
    j = 0, k = 0
    for i ∊ [0, len(left)+len(right)):
        if left[j] < right[k]:
            item[i] ← left[j]  ; j++
        else:
            item[i] ← right[k] ; k++
    return item
```

Merge Sort Demo

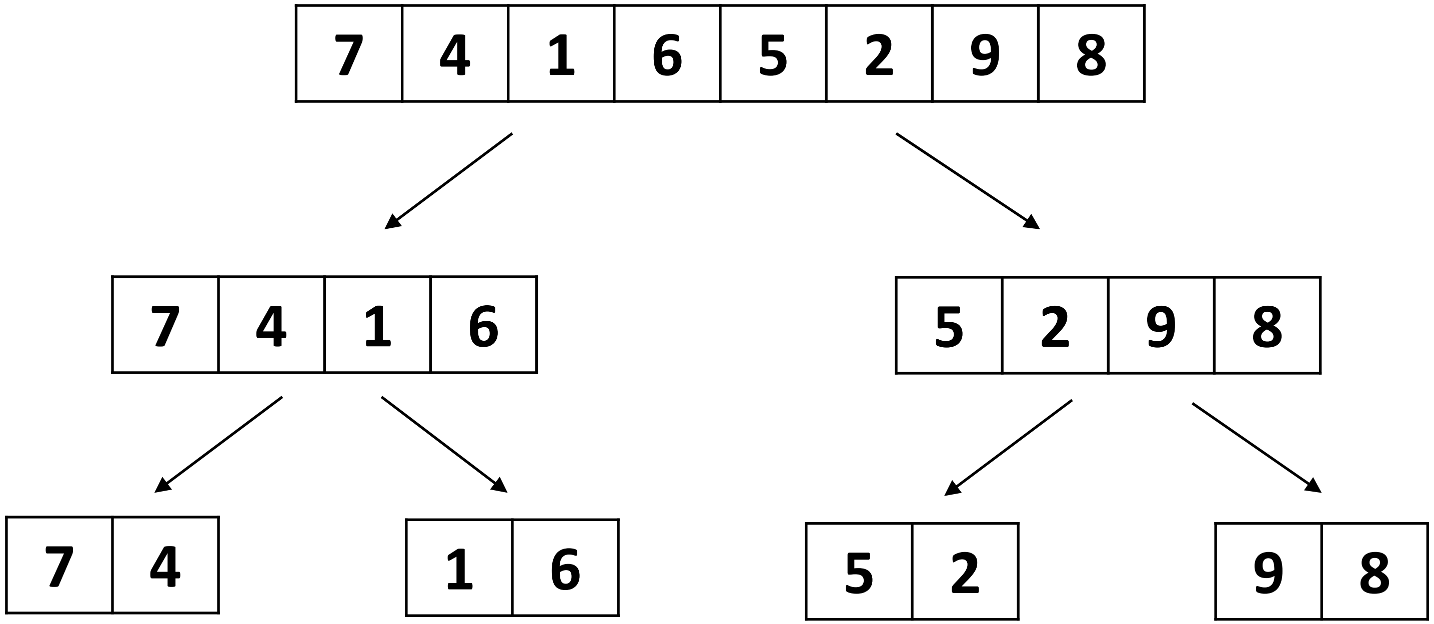
7	**4**	**1**	**6**	**5**	**2**	**9**	**8**

We cut our list in two at each step!

Merge Sort Demo

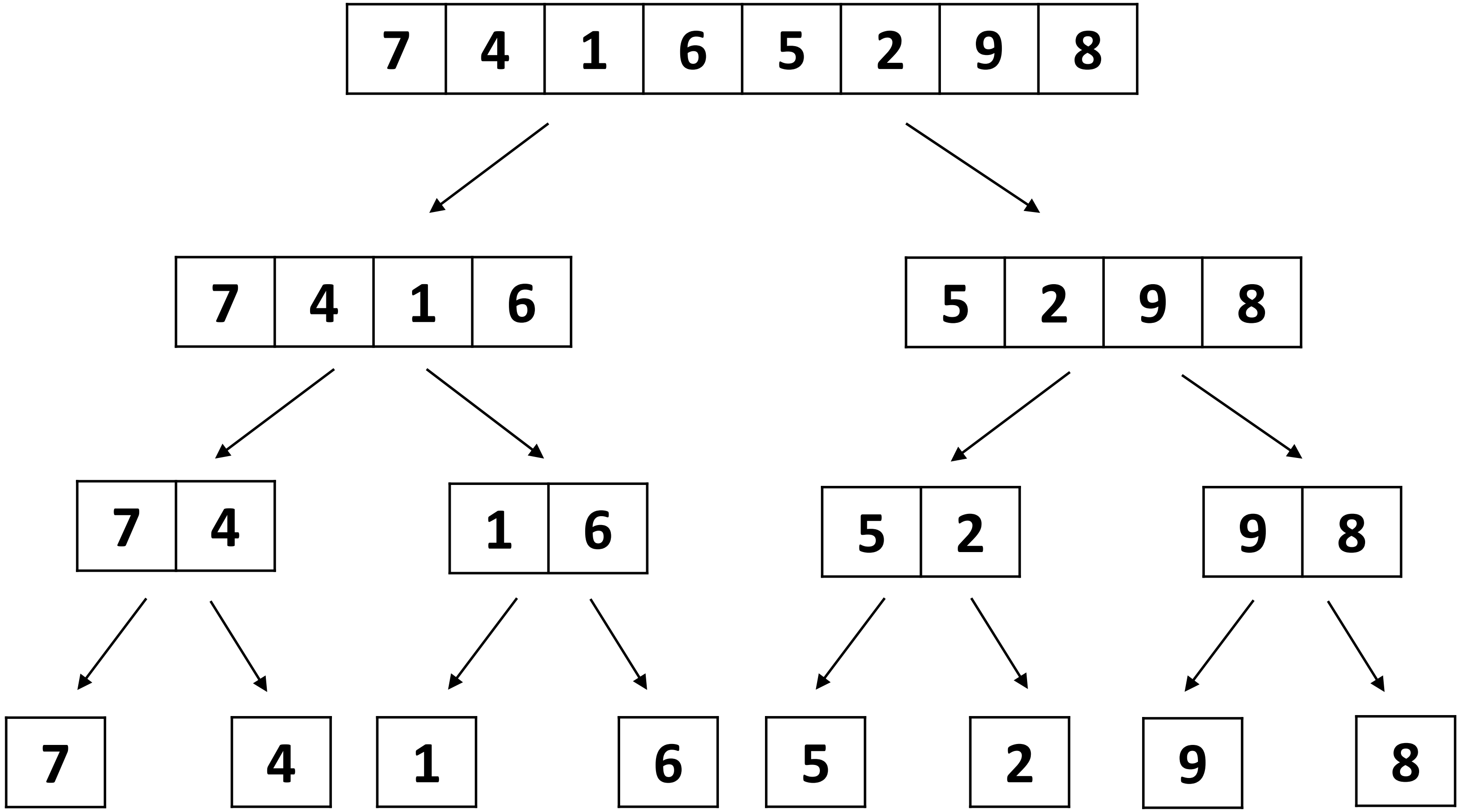


Merge Sort Demo



Merge Sort Demo

N total memory accesses at each level of recursion to split the lists

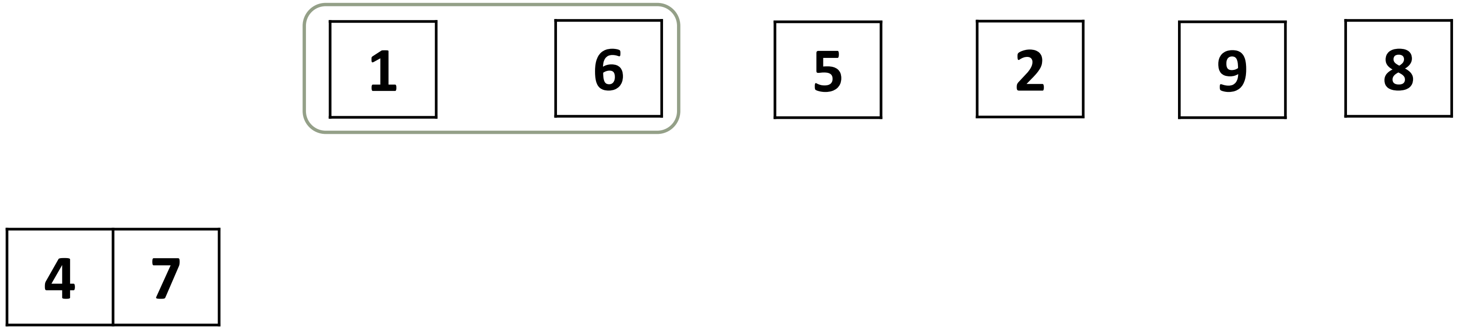


Merge Sort Demo

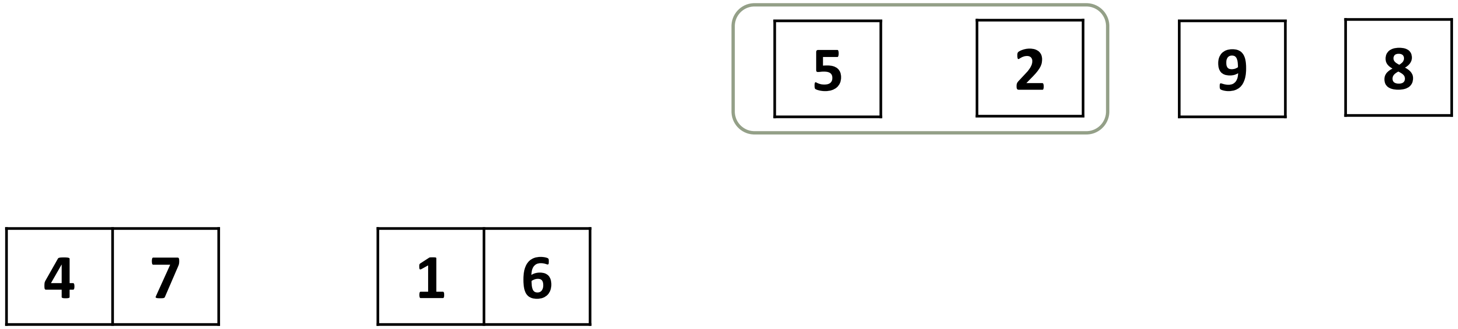
7	4	1	6	5	2	9	8

We employ the same merging technique we saw earlier

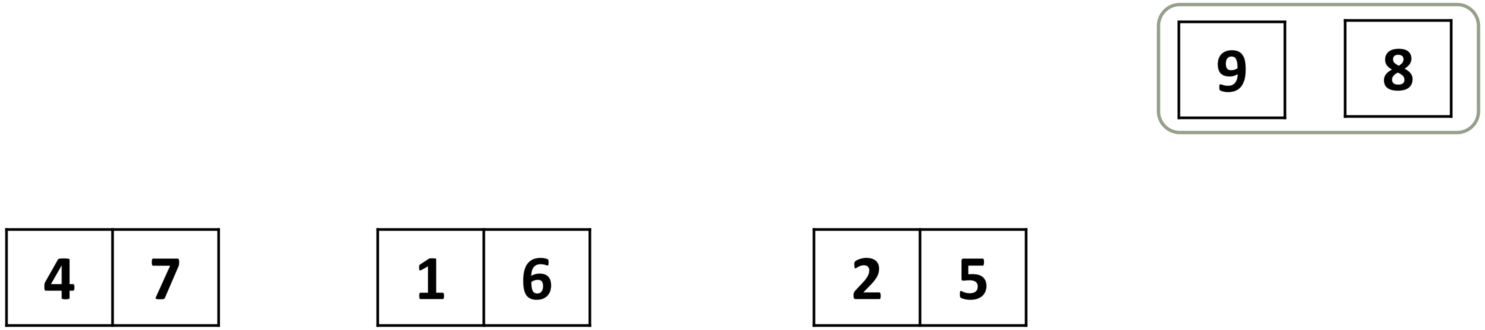
Merge Sort Demo



Merge Sort Demo

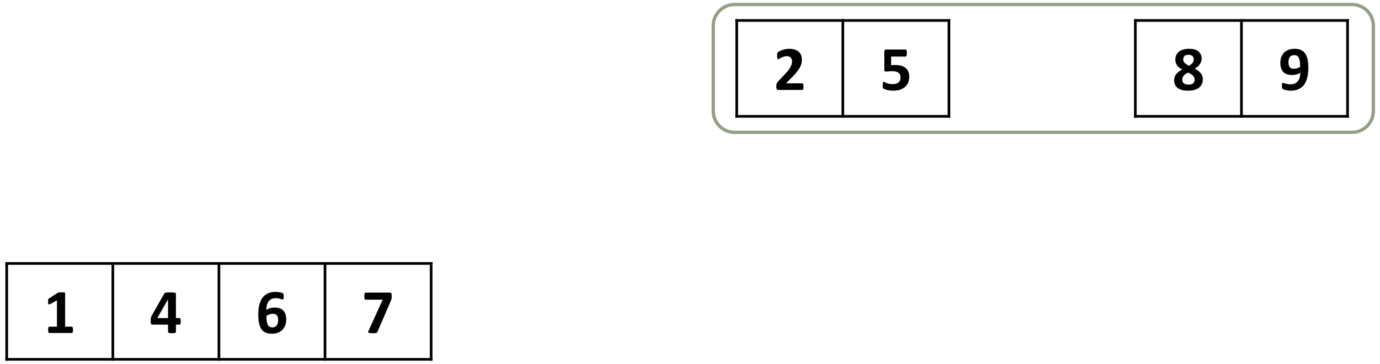


Merge Sort Demo



Merge Sort Demo

Merge Sort Demo



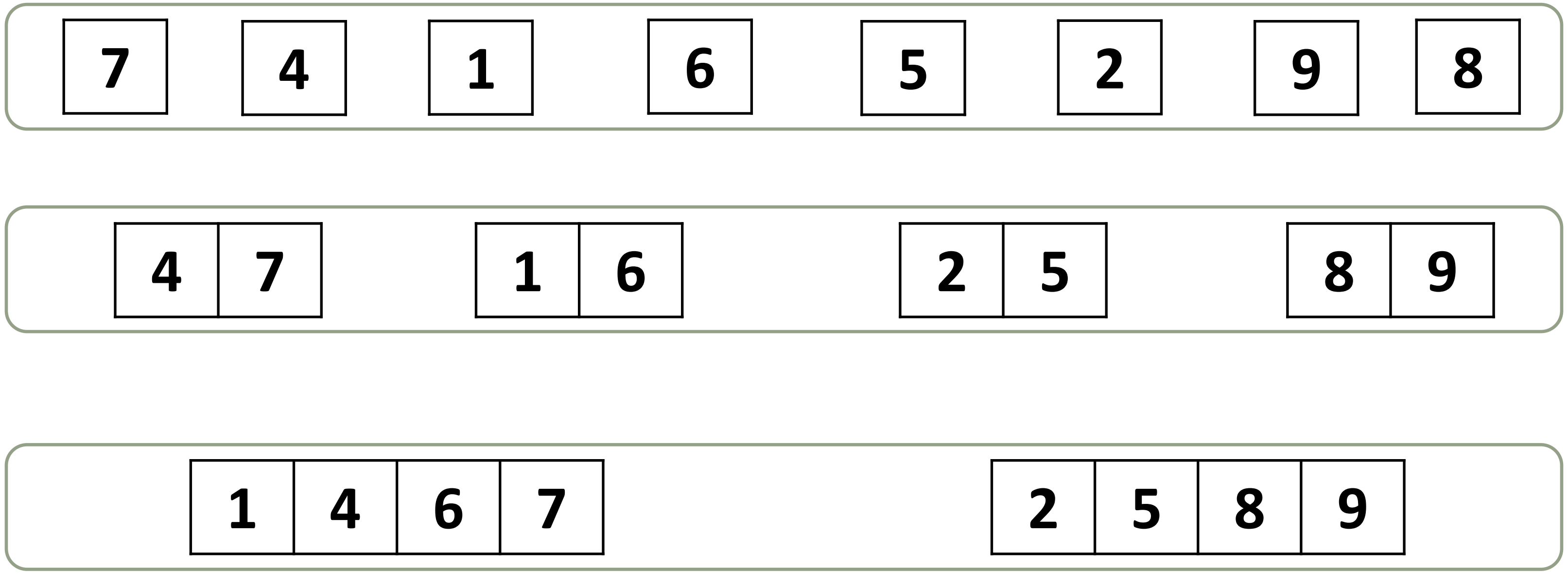
Merge Sort Demo

Merge Sort Demo

Final sorted result:

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

Merge Sort: Efficiency



N total memory accesses at each level of recursion to merge

Analyzing Mergesort Performance

for an array of Length n *(to simplify assume $n = 2^m$ for $m \in \mathbb{Z}^+$)*

```
sort(item):
    if (len(item) ≤ 1):
        return item
    else
        left  ← sort(item[...
        right ← sort(item[...
        return 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
```

How Many Accesses to Split an Array of Length n in two?   n

How Many Accesses to Merge Two Arrays of Length $^n/_2$?   n

Let T(n) be the number of Accesses to sort an array of Length n

How Many Accesses to Mergesort array of Length $^n/_2$?

T(n/2)

Analyzing Mergesort Performance

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

```
sort(item):
    if (len(item) ≤ 1):
        return item
    else
        left  ← sort(item[...
        right ← sort(item[...
        return 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 Accesses to Sort array of Length n

```
sort(item):
    if (len(item) ≤ 1):
        return item
    else
        left  ← sort(item[...
        right ← sort(item[...
        return 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
```

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

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 Accesses to Sort array of Length n

```
sort(item):
    if (len(item) ≤ 1):
        return item
    else
        left  ← sort(item[...
        right ← sort(item[...
        return 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 Accesses to Sort array of Length n

```
sort(item):
    if (len(item) ≤ 1):
        return item
    else
        left  ← sort(item[...
        right ← sort(item[...
        return 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 Accesses to Sort array of Length n

```
sort(item):
    if (len(item) ≤ 1):
        return item
    else
        left  ← sort(item[...
        right ← sort(item[...
        return 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 Accesses to Sort array of Length n

```
sort(item):
    if (len(item) ≤ 1):
        return item
    else
        left  ← sort(item[...
        right ← sort(item[...
        return 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 Accesses to Sort array of Length n

```
sort(item):
    if (len(item) ≤ 1):
        return item
    else
        left  ← sort(item[...
        right ← sort(item[...
        return 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 Accesses to Sort array of Length n

```
sort(item):
    if (len(item) ≤ 1):
        return item
    else
        left  ← sort(item[...
        right ← sort(item[...
        return 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 Accesses to Sort array of Length n

$T(1) = 0$ 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):

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

Proving the Closed Form

let $T(n)$ be the Number of Accesses 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 Accesses 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 Accesses 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 $T(k) \leq 2(k)(\log_2(k))$ 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 T(n) be the Number of Accesses 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 $k < n$

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

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**.

Use pointer!

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

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

Analysis: count memory accesses

$$T(0) = \emptyset$$

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

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

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

$$\le k + 1 + T(0)$$

$$2^k = n$$

$$k = \log n$$

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

This is our guess. We must prove using induction.

Binary Search

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

What we know: $T(0) = \emptyset$, $T(1) = 1$

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

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

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

$= 1$

Binary Search

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

What we know: $T(0) = \emptyset$, $T(1) = 1$

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

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

Inductive Hypothesis:

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

$$T(n) \leq 1 + T(\tfrac{n}{2})$$
$$\leq 1 + 1 + \log \tfrac{n}{2}$$
$$\leq 1 + 1 + \log n - \log 2$$
$$\leq 1 + \log n$$

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.