COMP20003 Workshop Week 8

1	Recurrences: • Building • Solving: using the Master Theorem • Solving: when the Master Theorem not applied
2	Merge Sort: • Top-Down Merge Sort: divide-and-conquer • Bottom-Up Merge Sort • Time & Space Complexity
L **A** **B**	Lab Time: • implementing bottom-up mergesort (P 7.1, 7.2)

How far did you go with ASS2:

A. finished!
B. stage 3 can only work with 4 towers
C. stage 3 can only work with 3 towers
D. stage 3 can only work with 1 towers
E. stage 3 not even worked with 1 tower
F. haven’t finished stage 2

Recurrences= complexity of recursive algorithms

Time Complexity of a recursive algorithm can be expressed as a recurrence: *T(n)* depends on *T(k)* where *k*<*n.* For example:

binary search on sorted arrays	$T(n)= T(n/2) + 1$ if $n>1$ $T(1)= 1$

the worst case of quick sort	$T(n)= T(0) + T(n-1) + Theta(n)$ if $n>$ $T(0)= T(1)= 1$

the best case of quick sort	$T(n)= 2T(n/2) + Theta(n)$ if $n>1$ $T(1)= 1$

Building Recurrences: check your answer

Time Complexity of recursive algorithm can be expressed as a recurrence: $T(n)$ depends on $T(k)$ where $k<n$. For example:

binary search on sorted arrays	$T(n)= T(n/2) + 1$ if $n>1$ $T(1)= 1$

the worst case of quick sort	$T(n)= T(n-1) + 1$ if $n>1$ $T(1)= 1$

the best case of quick sort	$T(n)= 2T(n/2) + n$ if $n>1$ $T(1)= 1$

Solving recurrences: The Master Theorem

When? A task of size n is divided into

- a tasks of size n/b and:
- and if

$$T(n) = aT(n/b) + \Theta(n^d)$$

$$T(1) = \Theta(1)$$

where $a \geq 1$, $b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } d > \log_b a \\ \Theta(n^d \log n) & \text{if } d = \log_b a \\ \Theta(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Master Theorem Examples:

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } d > \log_b a \\ \Theta(n^d \log n) & \text{if } d = \log_b a \\ \Theta(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Example (T(1)=1 for all:

- Binary Search: *T(n)= 1xT(n/2) + n^0 → a=1 b=2 d =0* $log_b a=0$

  T(n)=Theta(n^0 logn)= Theta(logn)

- Quick sort best case: *T(n)= 2 T(n/2) + Theta(n) →*

  a=2, b=2 , d=1 log_b a= 1 -> T(n)= Theta(n logn)

- *T(n)= 5T(n/2) + n^2 + 9nlogn*

  → a= 5 b= 2 d = 2 $log_b a > d$

  T(n)= Theta(n ^ (log-2 5)

Examples: - check your answer

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } d > \log_b a \\ \Theta(n^d \log n) & \text{if } d = \log_b a \\ \Theta(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Examples (T(1)= 1 for all) :

- Binary Search: T(n)= T(n/2) +1 → a=1,b=2,d=0 $d=\log_b a$

  → *Θ(logn)*

- Quick sort best case: T(n)= 2T(n/2) + n

  → *a=2,b=2,d=1* $d=\log_b a$

  → *Θ(nlogn)*

- T(n)= 5T(n/2) + n^2 + 9nlogn

  → a=5,b=2,d=2 $d<\log_b a$

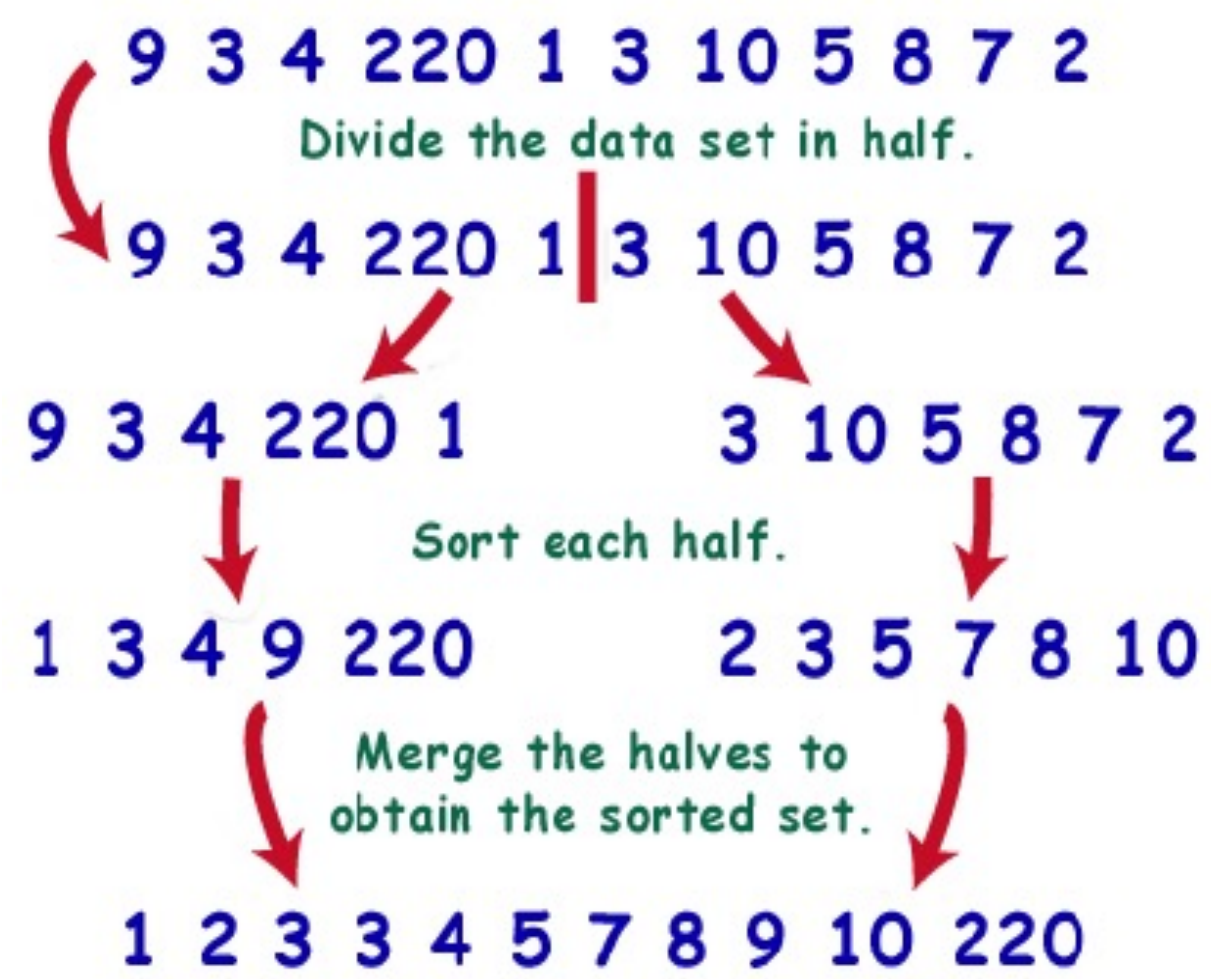
  → $\Theta(n^{\log_2 5})$

Merge Sort

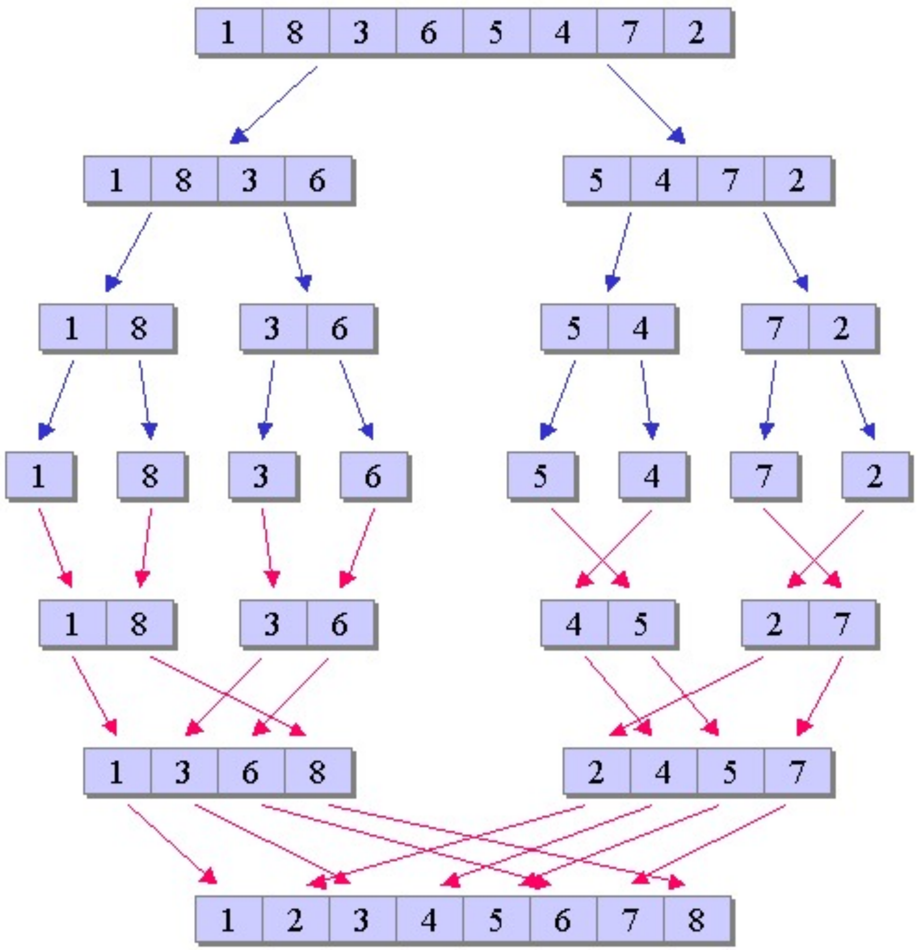
”Merge sort” normally means “top-down merge sort”

We also consider “bottom-up” merge sort

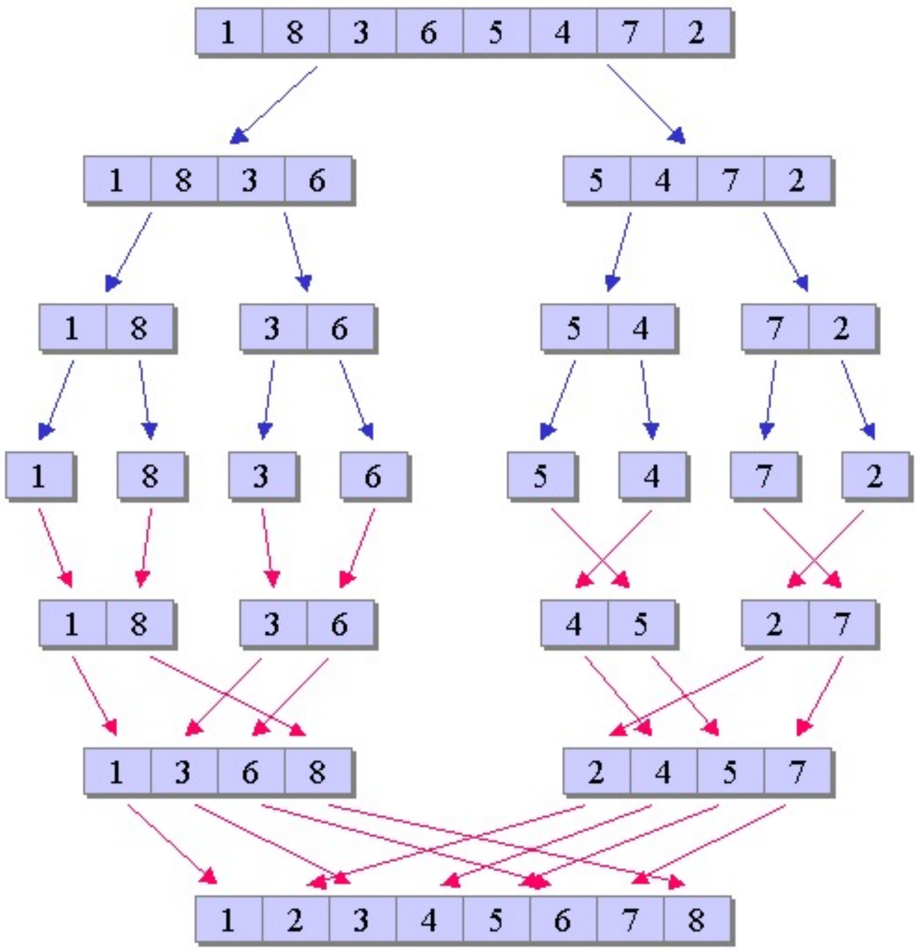
(Top-Down) Merge Sort: Main Idea with Divide & Conq



Top-Down MergeSort: Divide-And-Conquer!



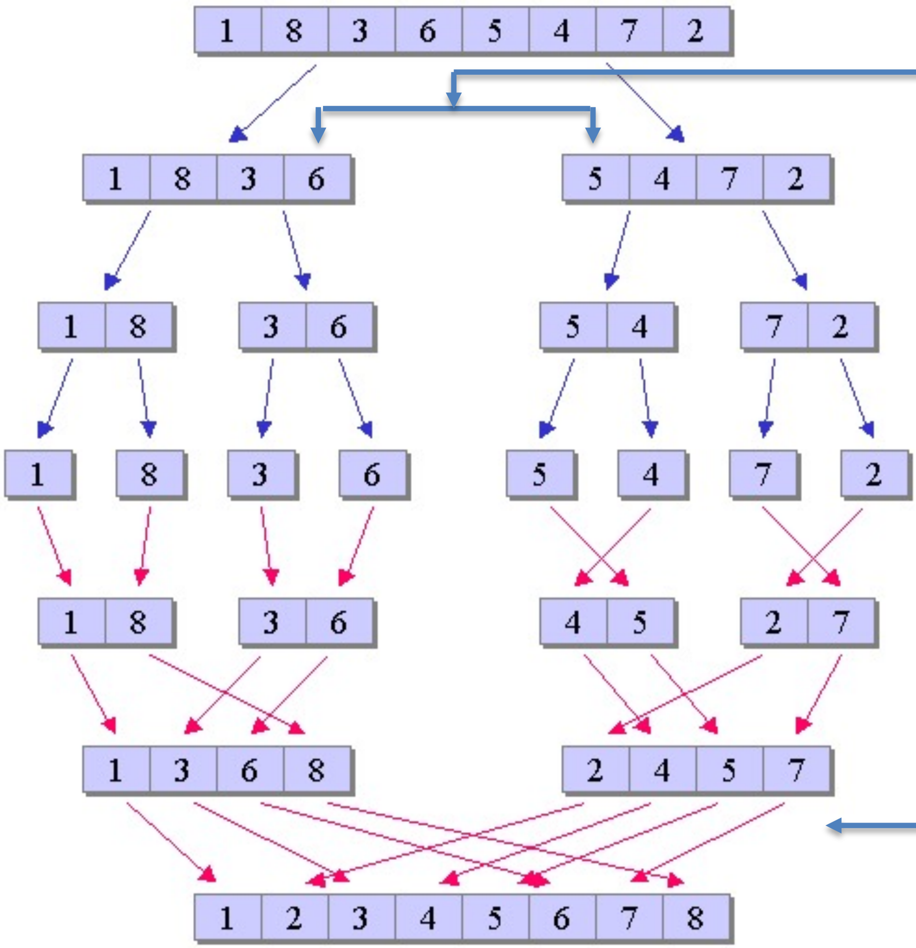
Top-Down MergeSort: Implementation Notes



the sorting algorithm is simple (?)

- "divide" is simple!
- "conquer"= merge 2 sorted arrays into one:
  - more complicated
  - need an additional array for the merging

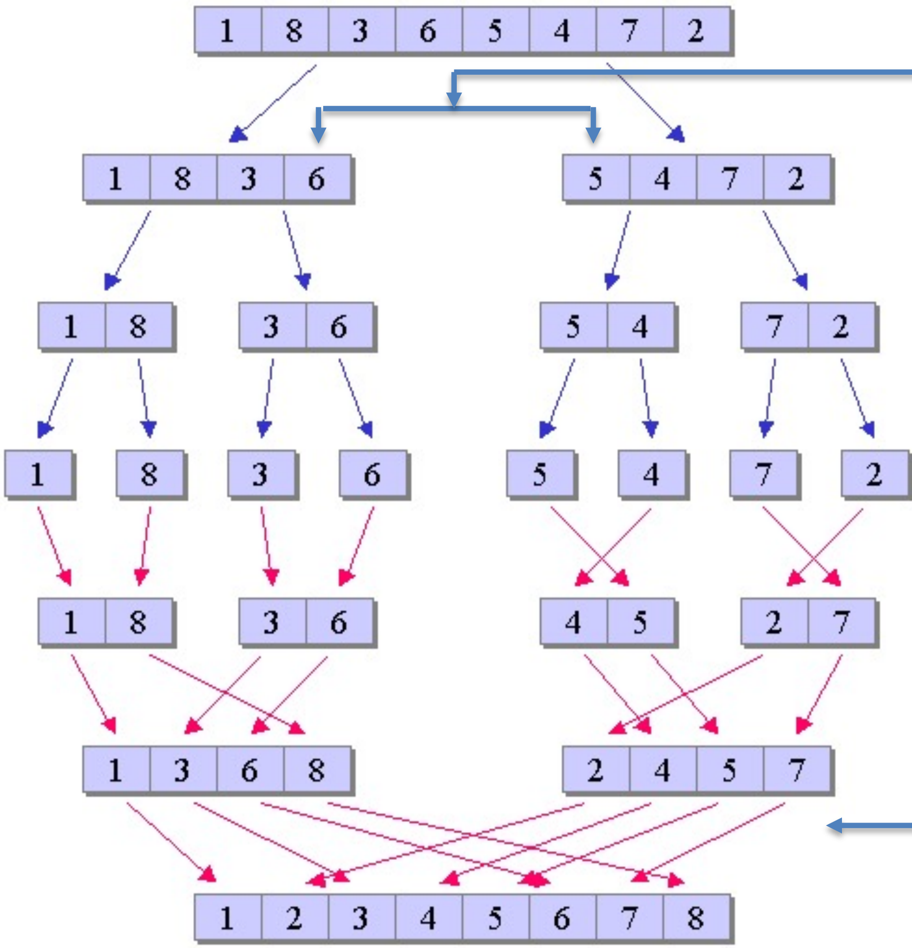
Complexity of mergesort = ?



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

Time Complexity = ?

Complexity of mergesort:



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

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } d > \log_b a \\ \Theta(n^d \log n) & \text{if } d = \log_b a \\ \Theta(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Time Complexity =

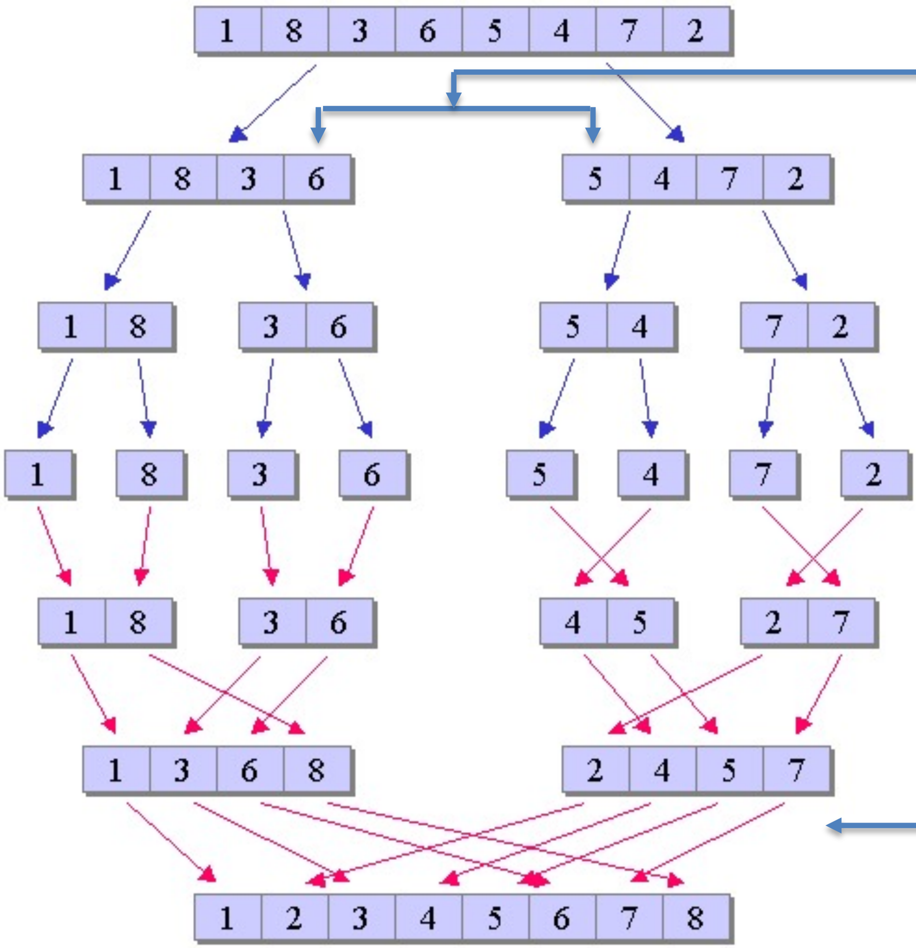
a =

b =

d =

T(n)=

Complexity of mergesort:



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

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } d > \log_b a \\ \Theta(n^d \log n) & \text{if } d = \log_b a \\ \Theta(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Time Complexity =

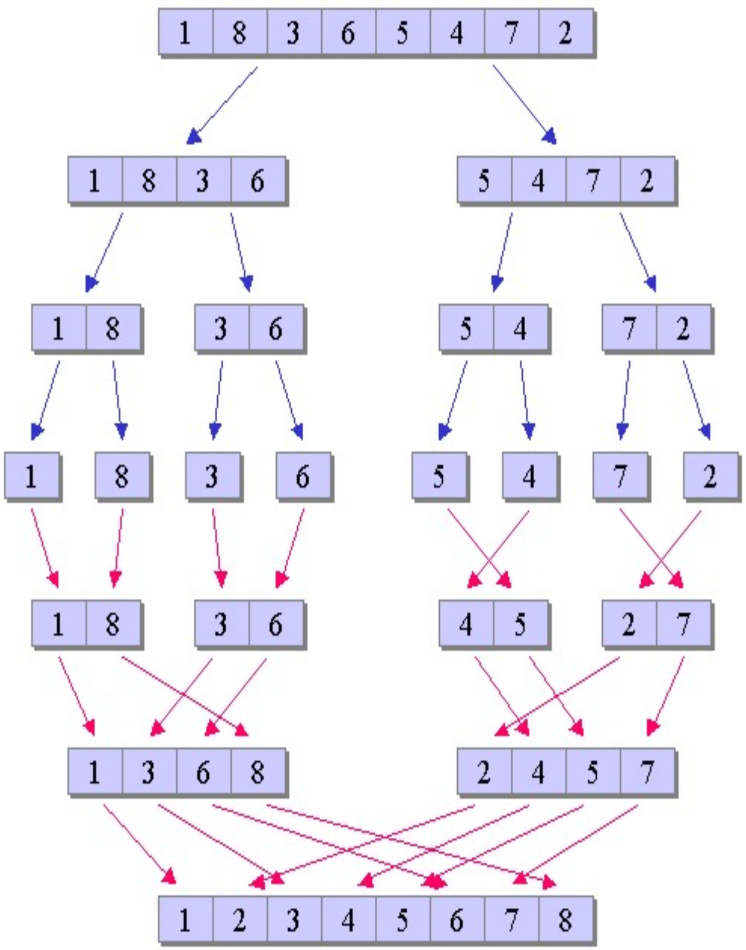
a =

b =

d =

T(n)=

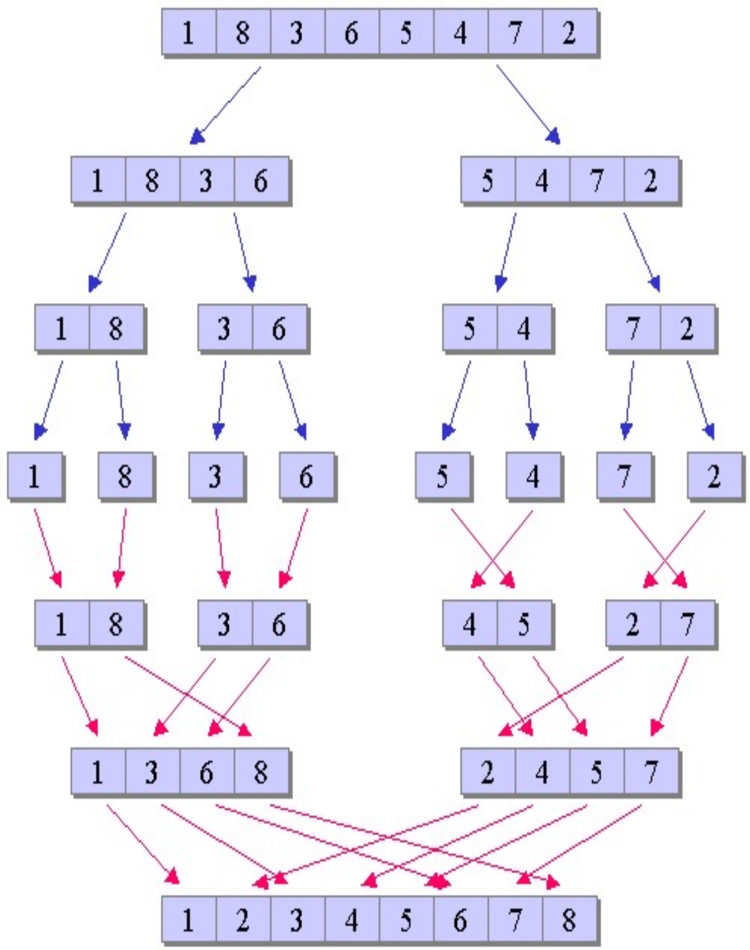
Top-Down MergeSort: Implementation Notes



using recursive calls
that is, using stacks!

```
mergesort(A[]) {
   if (A has > 1 element) {
      B[]= left half of A[]
               // copy to B[]
      C[]= right half of A[]
      mergesort(B[]);
      mergesort(C[]);
      merge B and C to A;
}
```

Top-Down MergeSort: space complexity when using arrays



Additional memory need:

n + log n = $\theta(n)$

for recursive stack (log n)

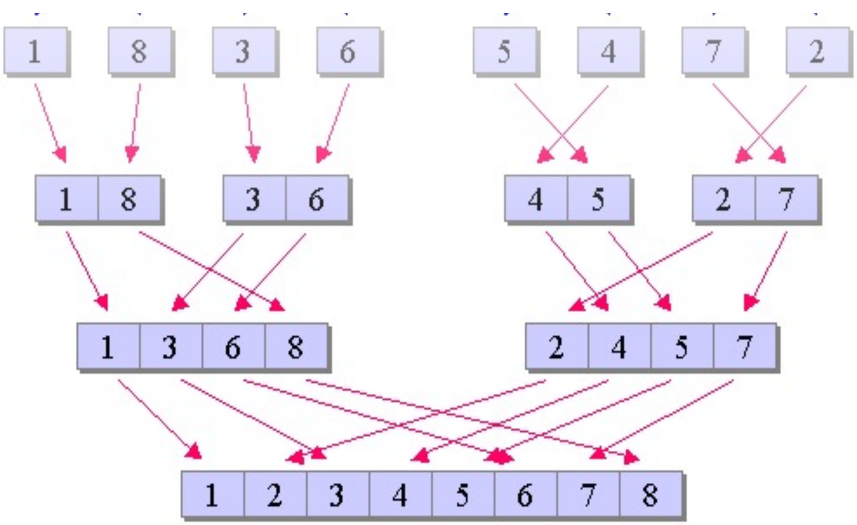
for not-in-place merging (n)

```
mergesort(A[]) {
   if (A has > 1element) {
      B[]= left half of A[]
      C[]= right half of A[]
      mergesort(B[]);
      mergesort(C[]);
      merge B and C to A;
}
```

Note: **we don't** normally **use linked lists for top-down** implementation (why?)

Bottom-Up Merge Sort

Merge Sort: Bottom-Up (shhh… no dividing just conquering)

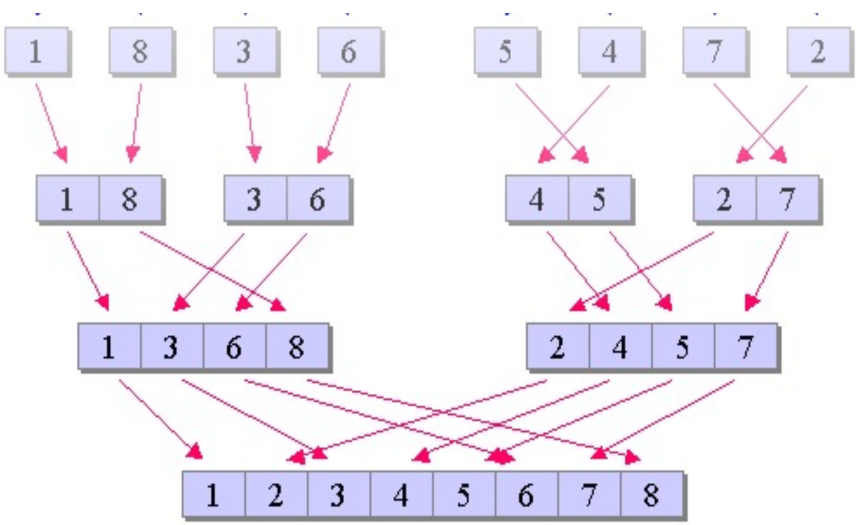


Start here: consider the original array as n singleton arrays or lists

Then do the merging process

Merge Sort: Bottom-Up with linked lists

Suppose that we use linked lists to store the data elements. That is, we don't have random access to the elements (typical case when data reside in external memory).

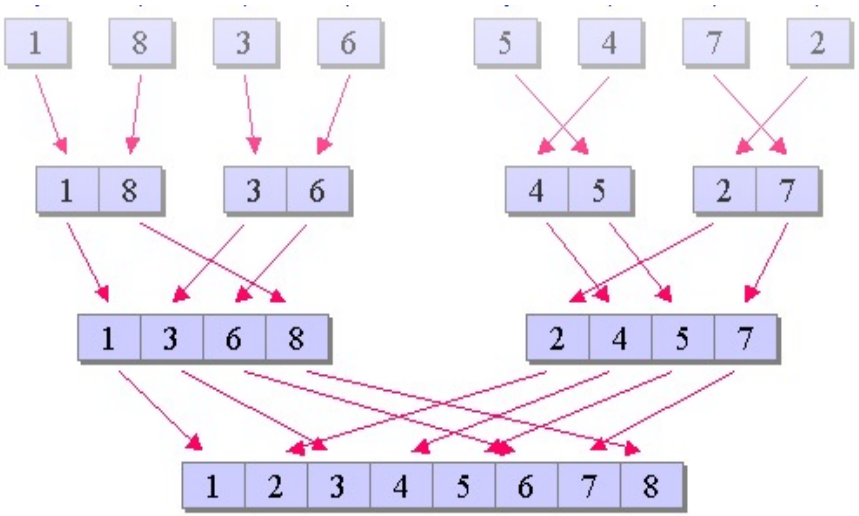


Start here: consider the original data as n singleton lists.
Note: a box represent a list node

Then do the merging process and finally join the data into a single sorted linked list.

But: how to control the merging process, especially when n is not a power of 2?

Merge Sort: Bottom-Up



But: how to control the merging process, especially when n is not a power of 2?

Use a queue!

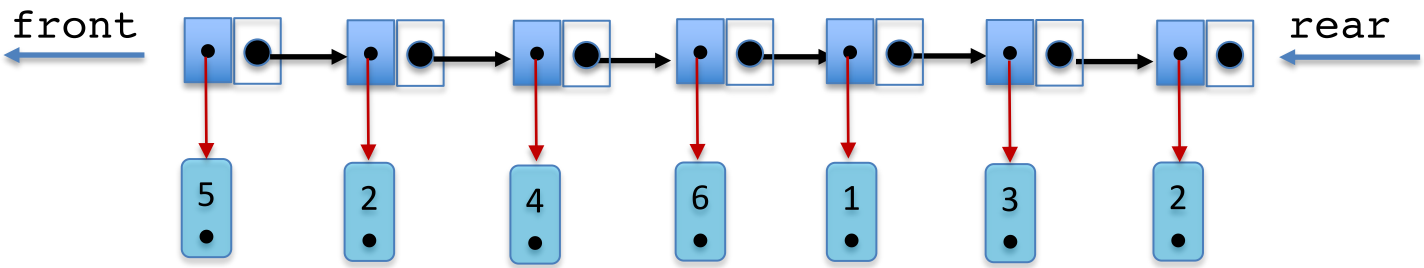
initially put all singletons into an empty queue Q

```
while (Q has 2 or more elements) {
   dequeue 2 elements
   merge them into one
   enqueue the merged element
}
// Q should have only one element
dequeue to get the sorted solution
```

Merge Sort: Bottom-Up for 5,2,4,6,1,3,2

How to implement: using a (linked-list based) queue Q:

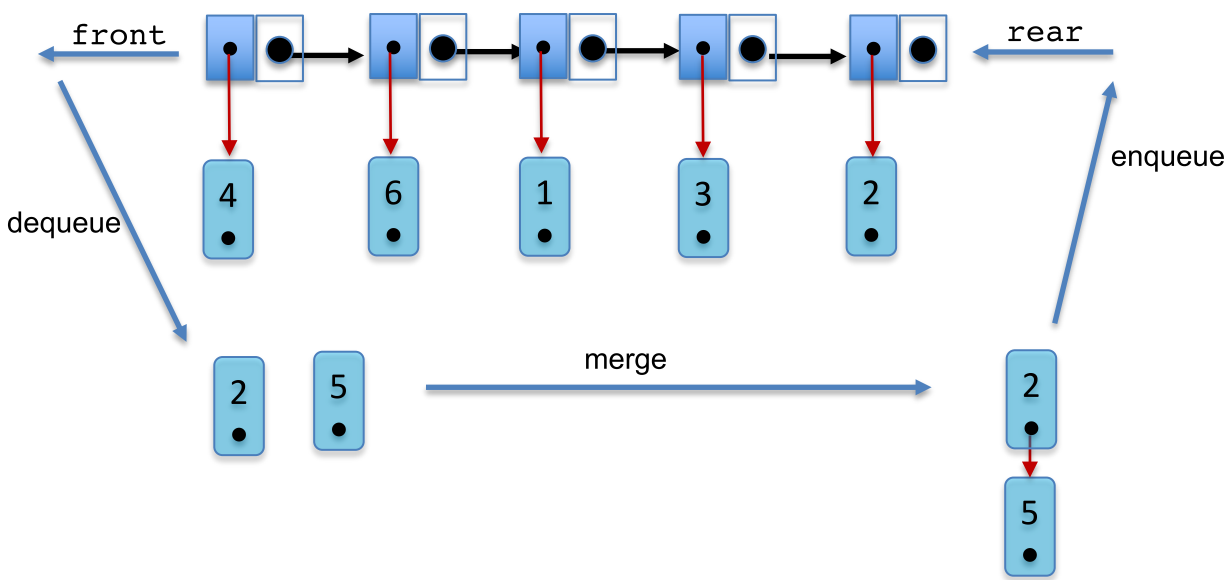
Start with enqueuing all singleton (sorted) lists into Q:



Merge Sort: Bottom-Up for 5,2,4,6,1,3,2

Then: while Q has at least 2 elements:

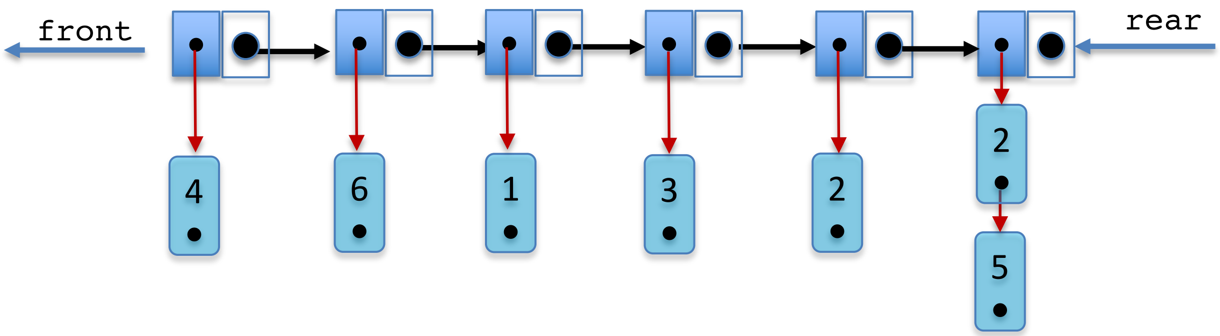
- dequeue 2 sorted lists
- merge them into a single sorted list
- enqueue the merged list



Merge Sort: Bottom-Up for 5,2,4,6,1,3,2

Then: while Q has at least 2 elements:

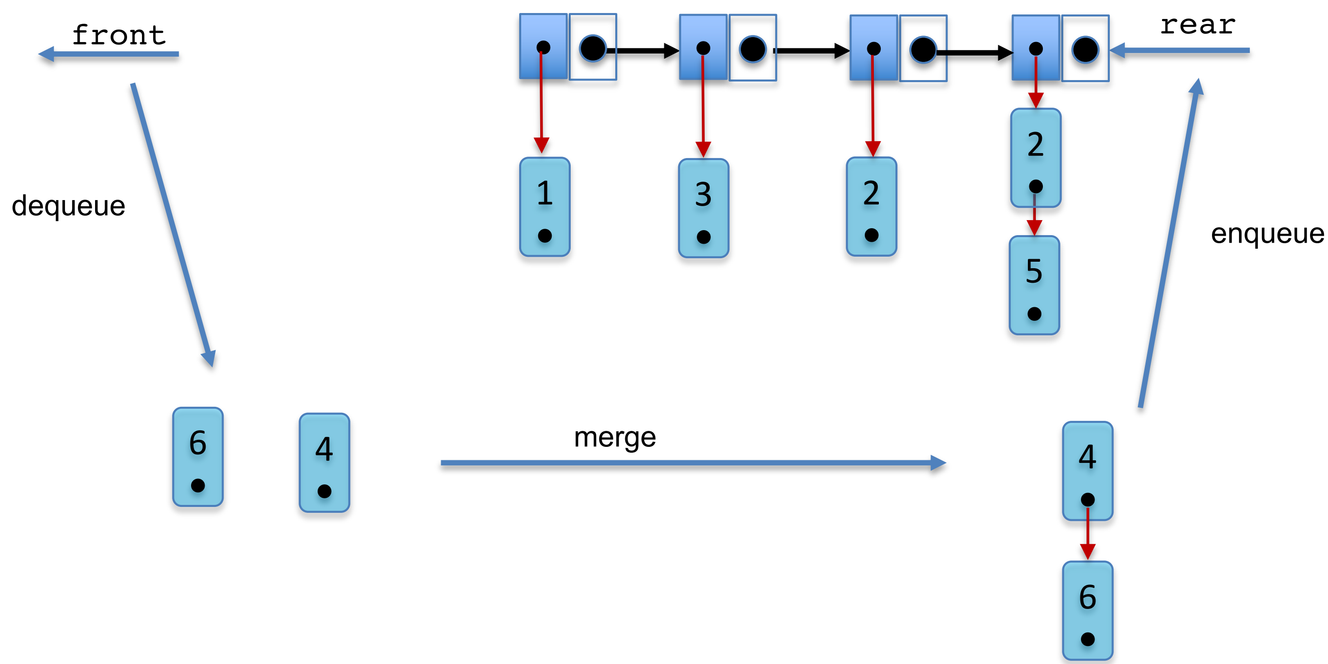
- dequeue 2 sorted lists
- merge them into a single sorted list
- enqueue the merged list



Merge Sort: Bottom-Up for 5,2,4,6,1,3,2

Then: while Q has at least 2 elements:

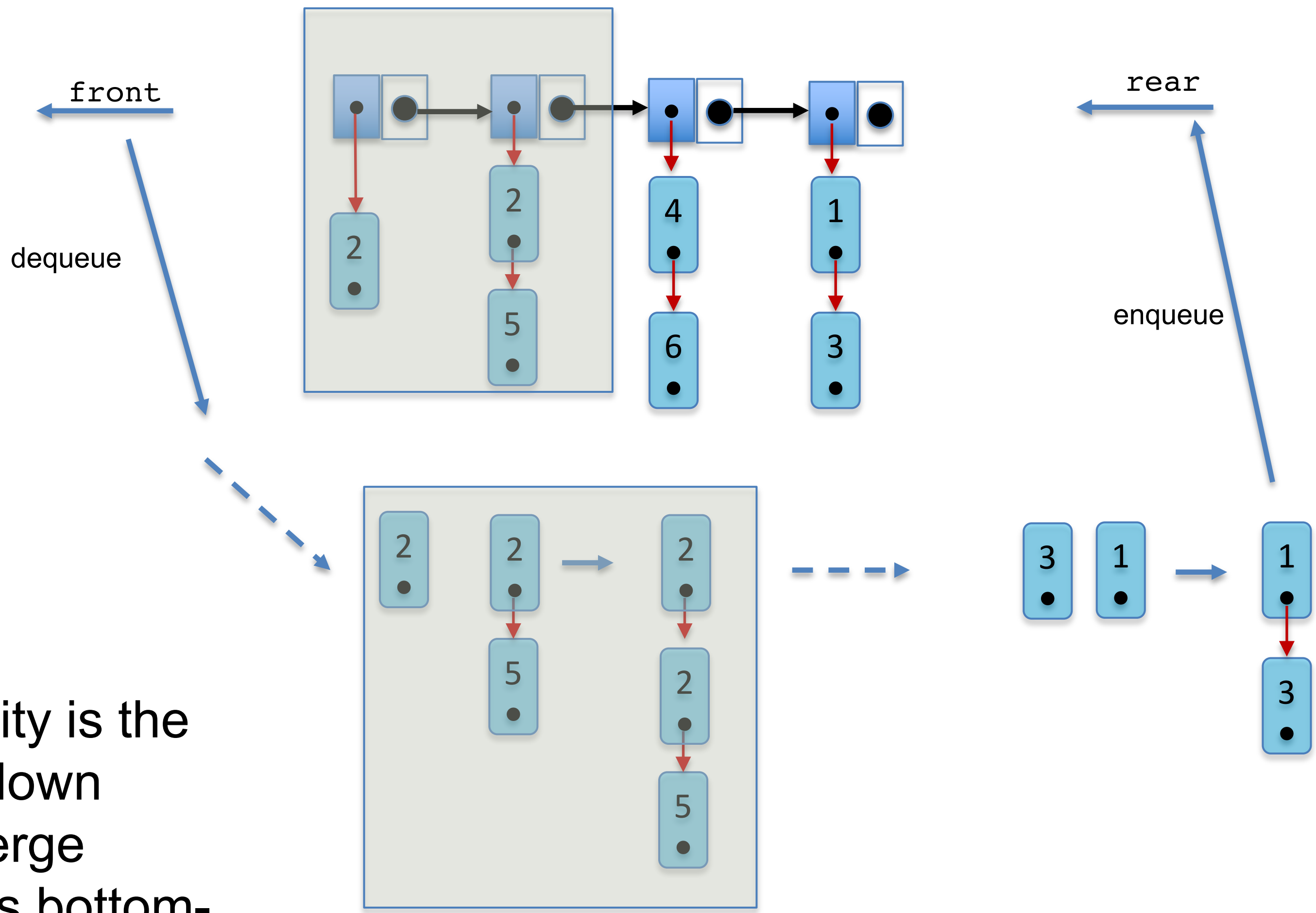
- dequeue 2 sorted lists
- merge them into a single sorted list
- enqueue the merged list



Merge Sort: Bottom-Up for 5,2,4,6,1,3,2

Then: while Q has at least 2 elements:

- dequeue 2 sorted lists
- merge them into a single sorted list
- enqueue the merged list



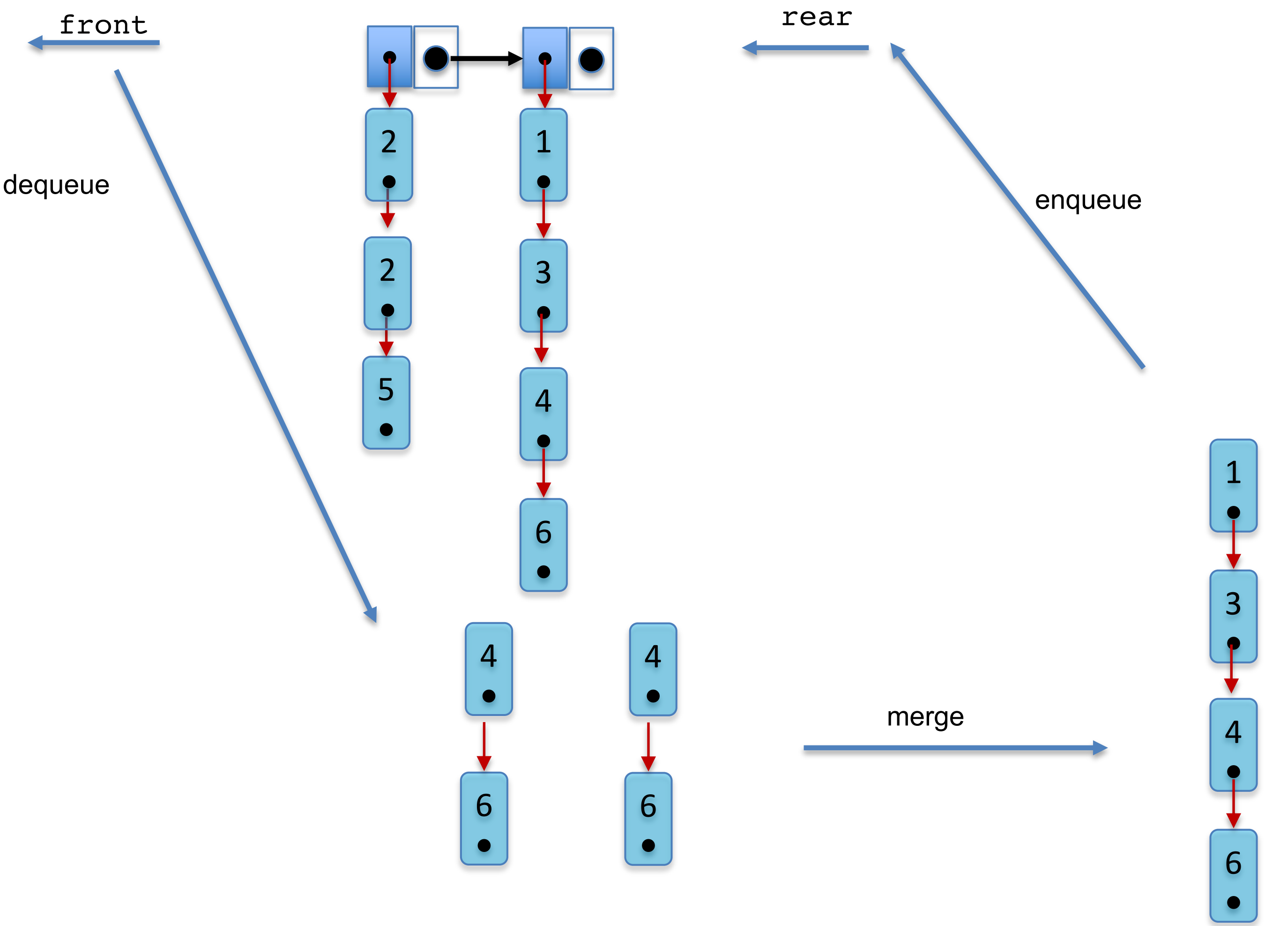
Note:

- Time complexity is the same as top-down
- The boxed merge shows that this bottom-up algorithm could be not stable

Merge Sort: Bottom-Up for 5,2,4,6,1,3,2

Then: while Q has at least 2 elements:

- dequeue 2 sorted lists
- merge them into a single sorted list
- enqueue the merged list

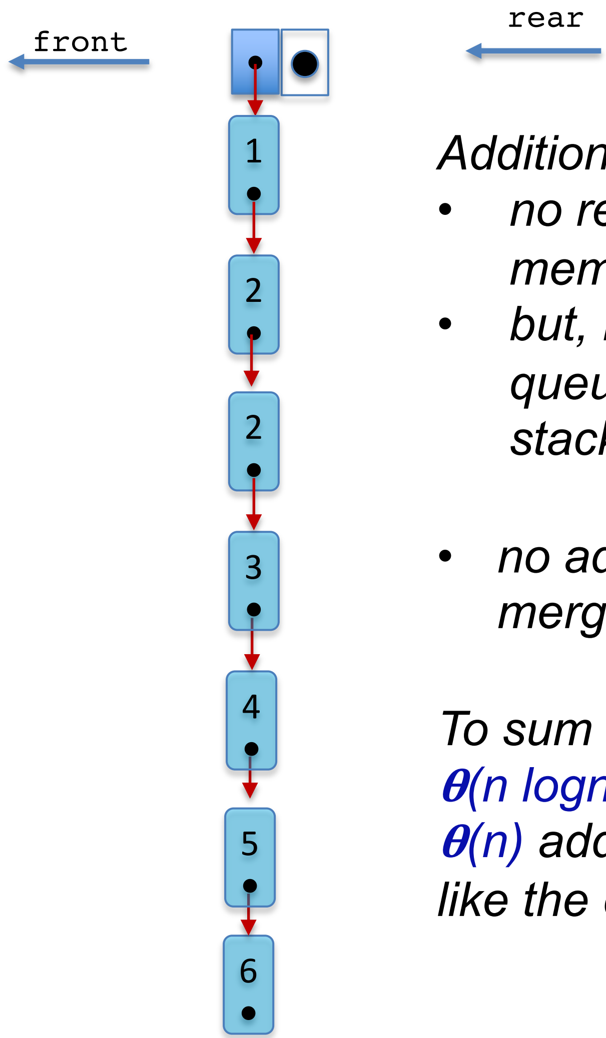


Merge Sort: Bottom-Up for 5,2,4,6,1,3,2

Then: while Q has at least 2 elements:

- dequeue 2 sorted lists
- merge them into a single sorted list
- enqueue the merged list

At the end, the queue has only a single element. Dequeue that to get the final sorted list.



Additional memory need:

- *no recursive, no stack memory needed*
- *but, need $\theta(n)$ for the queue (more than for the stack ☹)*
- *no additional memory for merging*

To sum up:
$\theta(n \log n)$ time complexity, $\theta(n)$ additional memory, just like the case of top-down

Bottom-up Merge Sort using Arrays for elements

With a bit of care, we can organize the merging process using a single additional array of size n (see algorithm in lecture). In this case:

Additional memory need:

- *no recursive, no stack memory needed*
- *no queue, no memory for queue*

But:

- *need $\theta(n)$ for merging the arrays*

At the end, all implementations of merge sort, including top-down and bottom-up:

- $\theta(n \log n)$ *time complexity,*
- $\theta(n)$ *additional memory,*

Lab: P7.1 and P7.2

Programming 7.2 Write code for bottom-up mergesort where the data are contained in an initially unsorted array. You will have to construct an artificial array to test your code. You can populate your array with random numbers before sorting.

Notes:

- a skeleton main() is supplied
- compare your merge function with the one in the lecture slides

Programming 7.1 Write code for bottom-up mergesort where the data are contained in an initially unsorted linked list. You will have to construct an artificial linked list to test your code. You can populate your linked list with random numbers before sorting.

Notes:

- supplied: tools for linked lists, main()
- not supplied: tools for queues
- you need a queue of linked lists

Additional Materials

The Master Theorem

When? A task of size n is divided into

- a tasks of size n/b and:
- and if

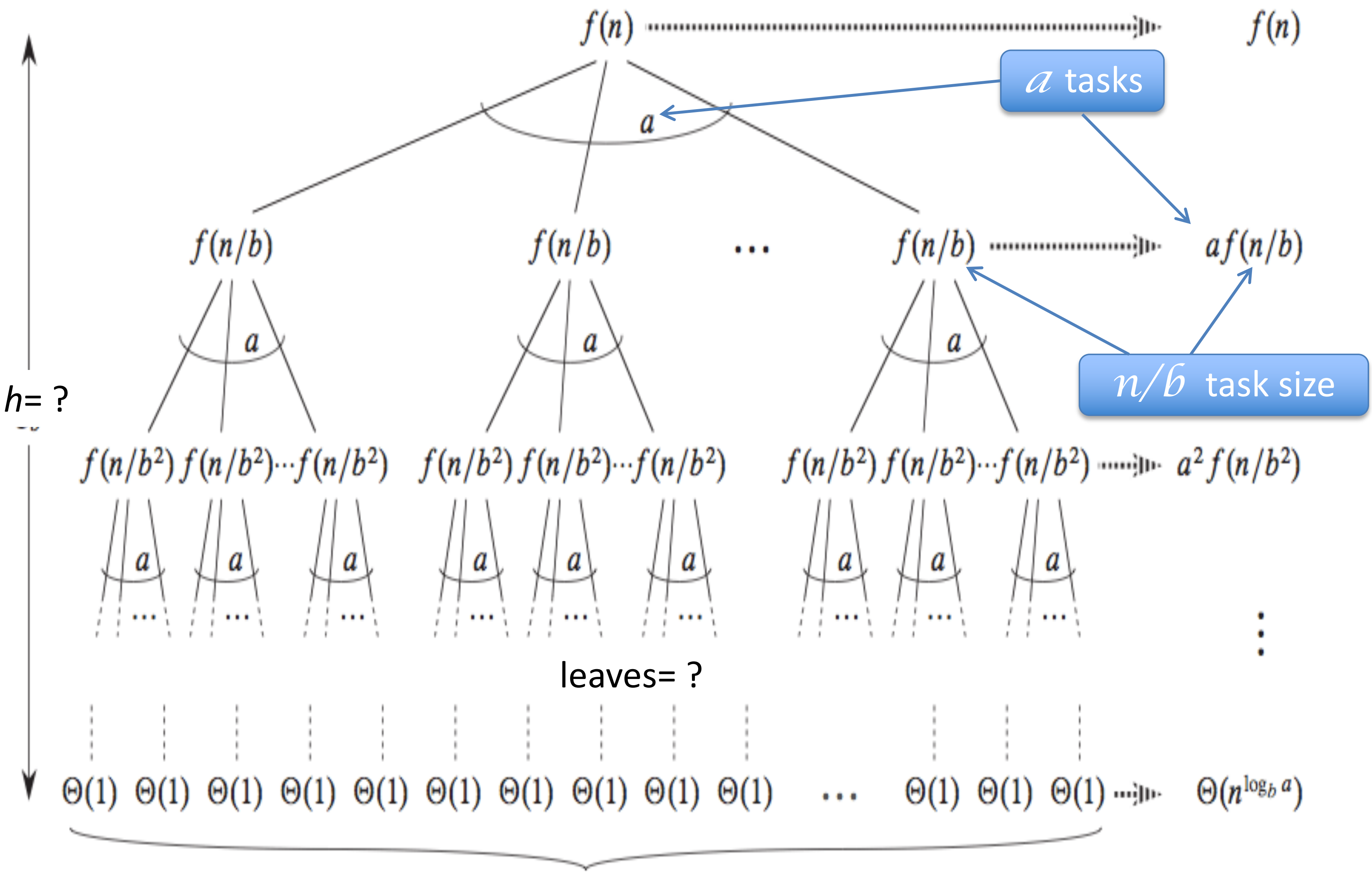
$$T(n) = aT(n/b) + \Theta(n^d)$$

$$T(1) = \Theta(1)$$

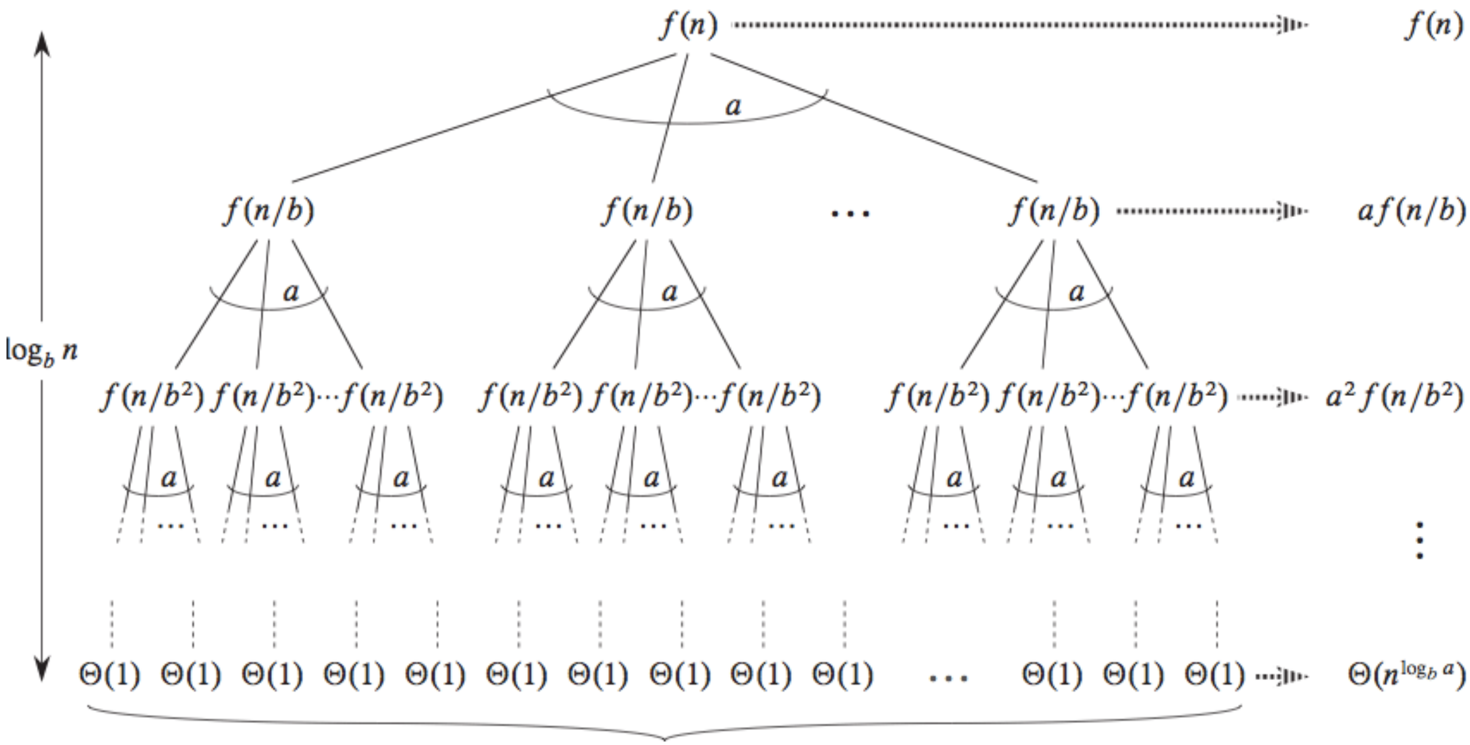
where $a \geq 1$, $b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } d > \log_b a \\ \Theta(n^d \log n) & \text{if } d = \log_b a \\ \Theta(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

aster Theorem is for Divider & Conquer with f(n)=n^d



Master Theorem: Complexity Computation



Note: $leaves = a^h = a^{\log_b n} = n^{\log_b a}$

Total time:

$$n^d + a(n/b)^d + a^2(n/b^2)^d + \ldots + a^h(n/b^h)^d$$

$$= n^d + n^d(a/b^d) + n^d(a/b^d)^2 + \ldots + n^d(a/b^d)^{\log_b n}$$

Master Theorem: Complexity Computation

The running time:

$$= \quad n^d + \; n^d\,(a/b^d) \; + \quad n^d\,(a/b^d)^2 \; + \ldots \; + n^d\,(a/b^d)^{\log_b n}$$

$$= \quad n^d\,(\;\; 1 + \qquad \ldots \qquad + \; (a\,/\,b^d)^{\log_b n} \;)$$

Remember sum of geometric sequence:

$$1 + c + c^2 + \ldots + c^n = (1-c^{n+1})/(1-c) \quad = \quad \Theta(\;1\;) \qquad \text{when } c<1$$

$$\Theta(\;c^n\;) \qquad \text{when } c>1$$

$$c=a/b^d$$

$$\Theta(\;n\;) \qquad \text{when } c=1$$

Winner	Condition	Equivalent condition	Time complexity
Conquer	$a < b^d$	$\log_b a < d$	$\Theta(n^d)$
Divider	$a > b^d$	$\log_b a > d$	$\Theta(n^{\log_b a})$
none	$a = b^d$	$\log_b a = d$	$\Theta(n^d \log n)$

Note:

$$S = 1 + c + c^2 + \ldots + c^n$$

$$Sc = c + c^2 + c^3 + \ldots + c^{n+1} = S - 1 + c^{n+1}$$

$$S(c-1) = c^{n+1} + 1 - 1$$

$$S = (c^{n+1} - 1) / (c-1)$$

Hidden Space Complexity of Recursive Algorithms

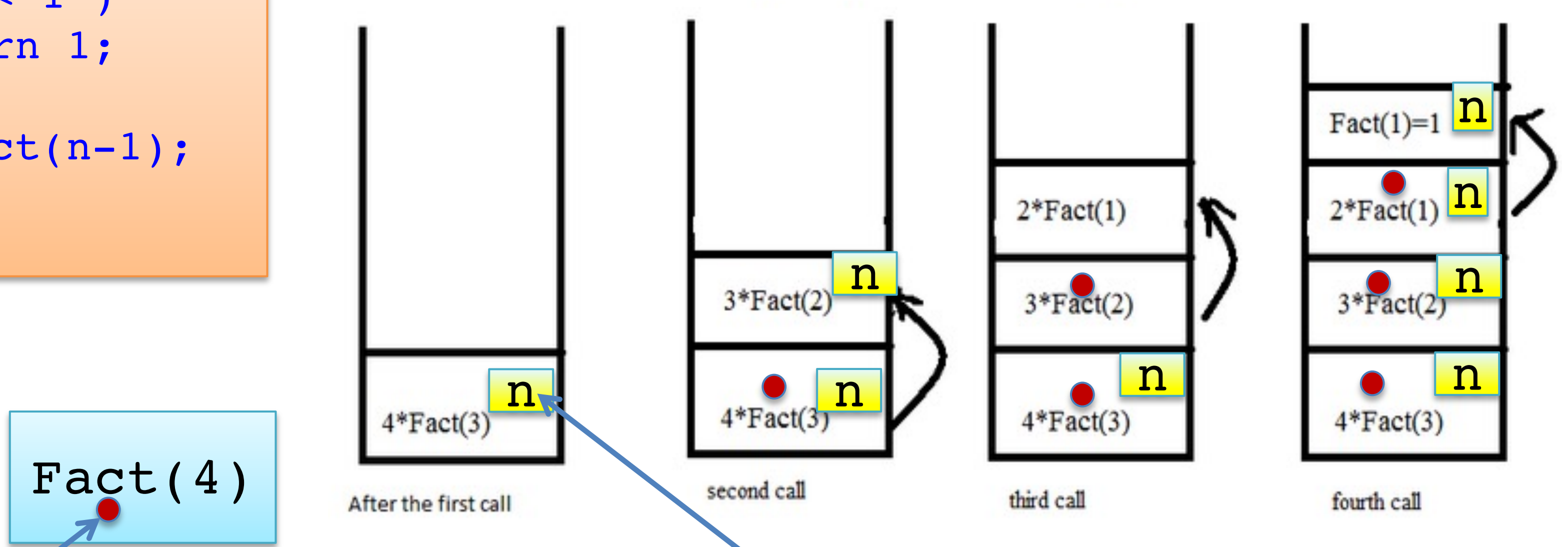
Space complexity of function Fact: is it $\boldsymbol{\theta}(1)$?

```
int Fact(int n) {
   if ( n<=1 )
      return 1;
   return
      n*Fact(n-1);
}
```

Hidden Space Complexity of Recursive Algorithms

Space complexity of function Fact: it is not $\theta(1)$?

```
int Fact(int n) {
  if ( n<=1 )
    return 1;
  return
    n*Fact(n-1);
}
```



returned address

stack frame, containing all local variables of the current execution of Fact

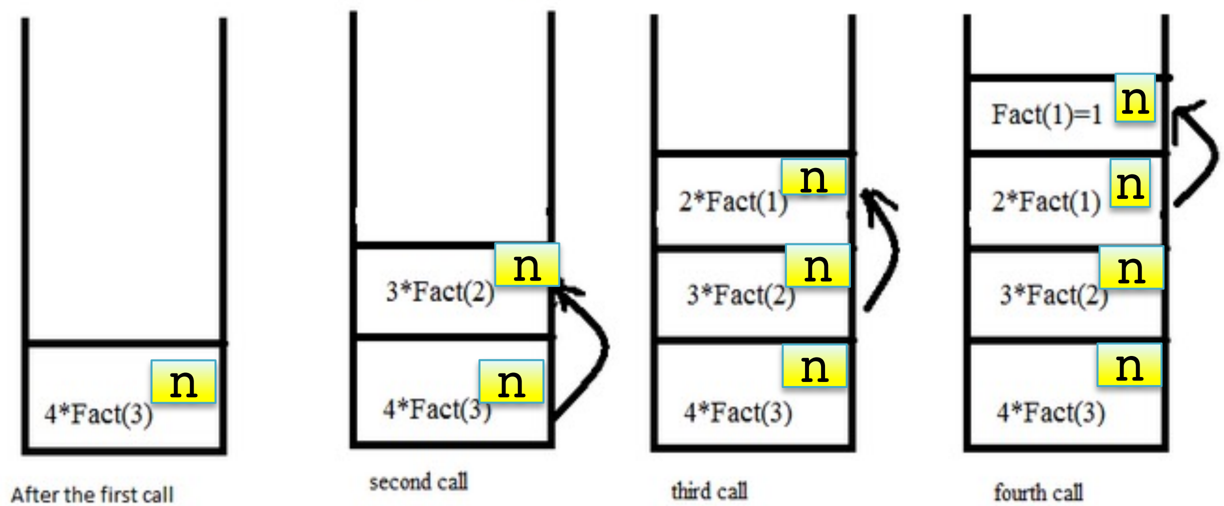
Hidden Space Complexity of Recursive Algorithms

Memory incurred with (recursive) function calls: an example

Space complexity of function Fact =

```
int Fact( int n ) {
  if ( n<=1 )
     return 1;
  return n*Fact(n-1);
}
```

```
Fact(4)
```



Hidden Space Complexity of Recursive Algorithms

Space complexity of Fact: $\boldsymbol{\theta}(n)$

Space complexity of recursive function =

space for local variables ✕ depth of rec calls.

```
int Fact(int n) {
  if ( n<=1 )
    return 1;
  return
    n*Fact(n-1);
}
```

```
Fact(4)
```