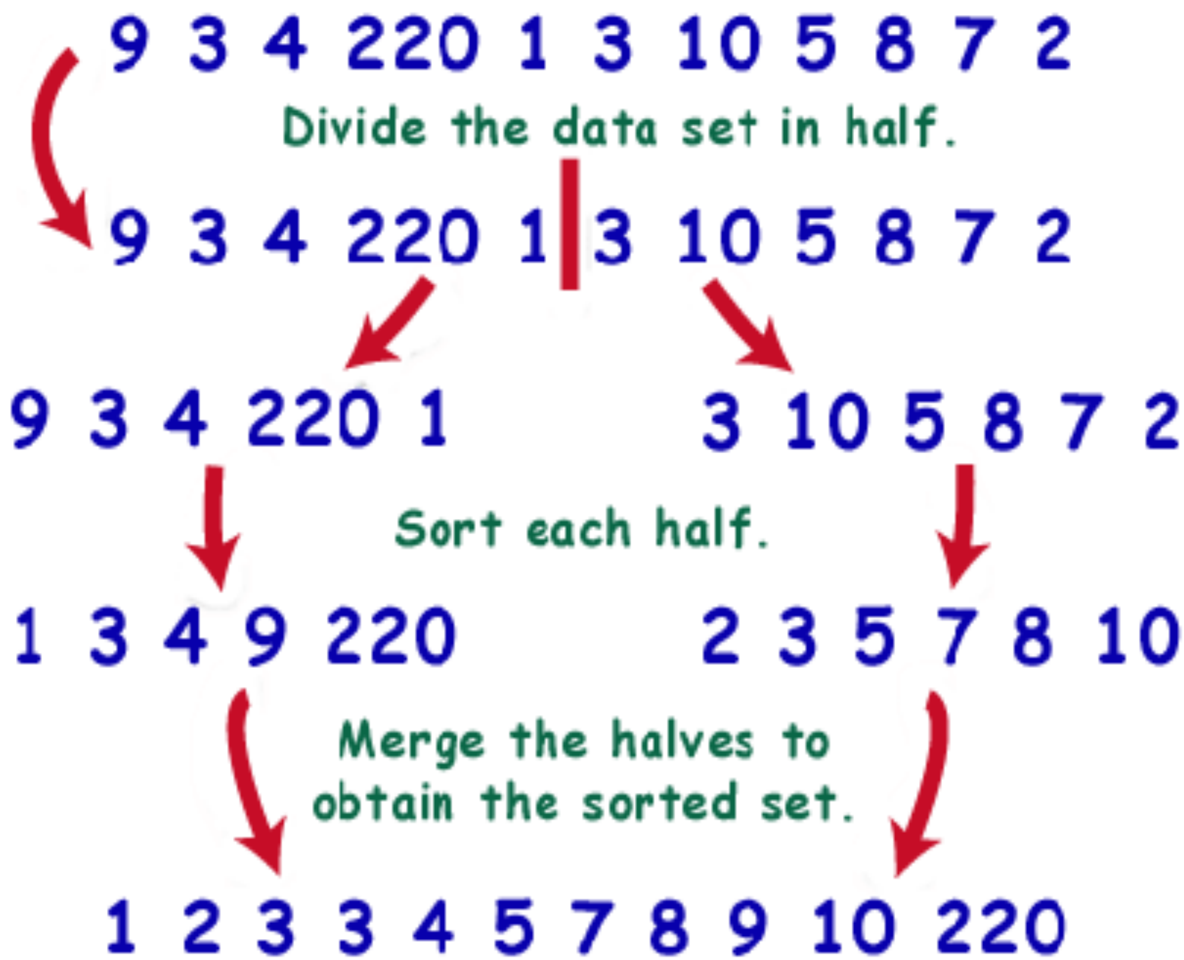
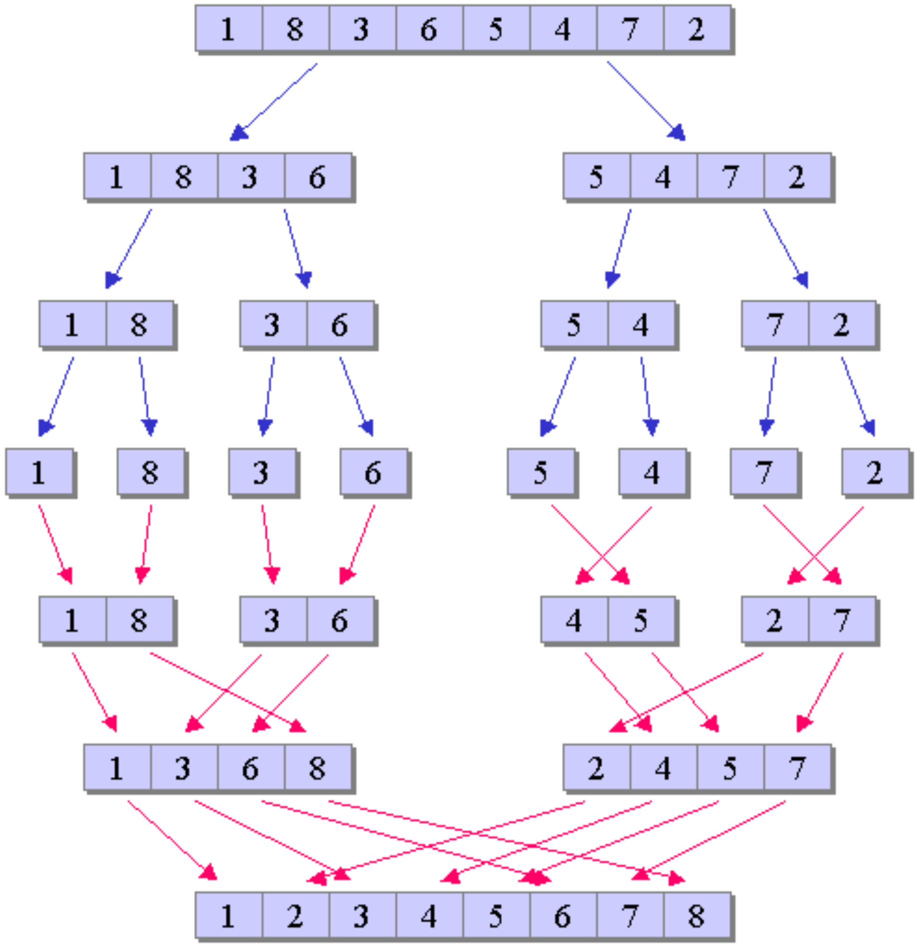
COMP20003 Workshop Week 8

1	Merge Sort: divide-and-conquer
2	Master Theorem
3	Merge Sort: bottom-up algorithms
4	Group exercises
5	Lab: implementing bottom-up mergesort (P 7.1, 7.2)

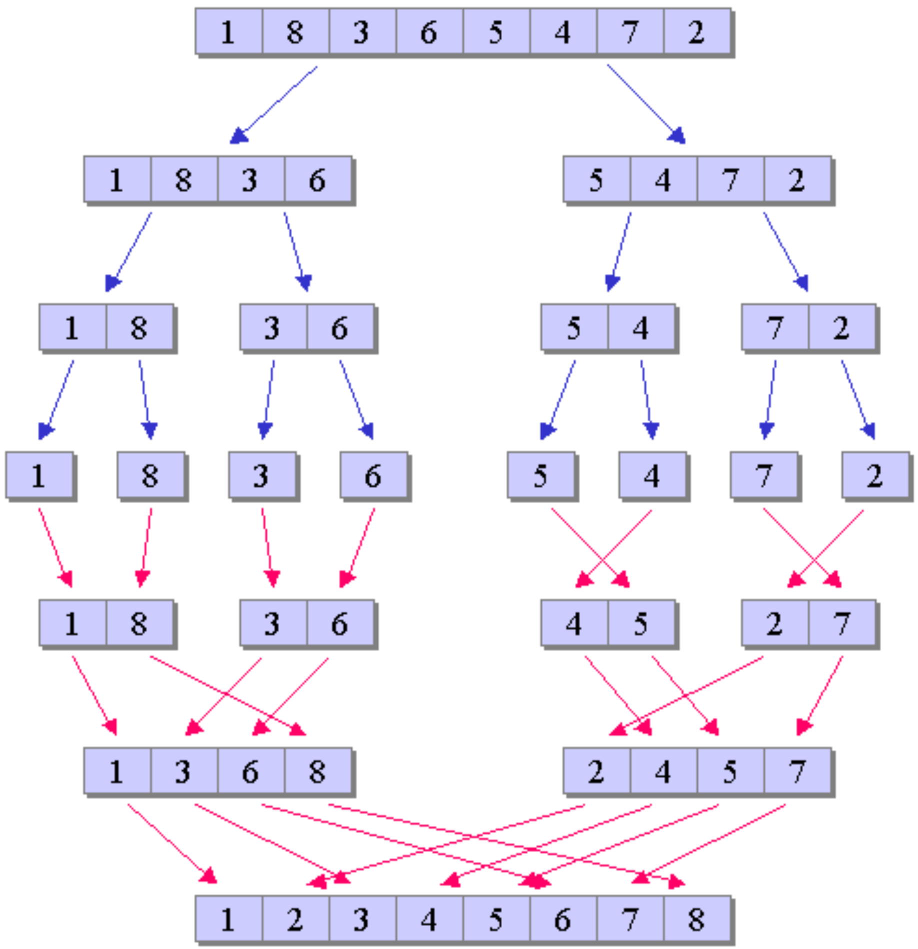
Merge Sort: Main Idea



Top-Down MergeSort: Divide-And-Conquer!



Top-Down MergeSort: Implementation Notes

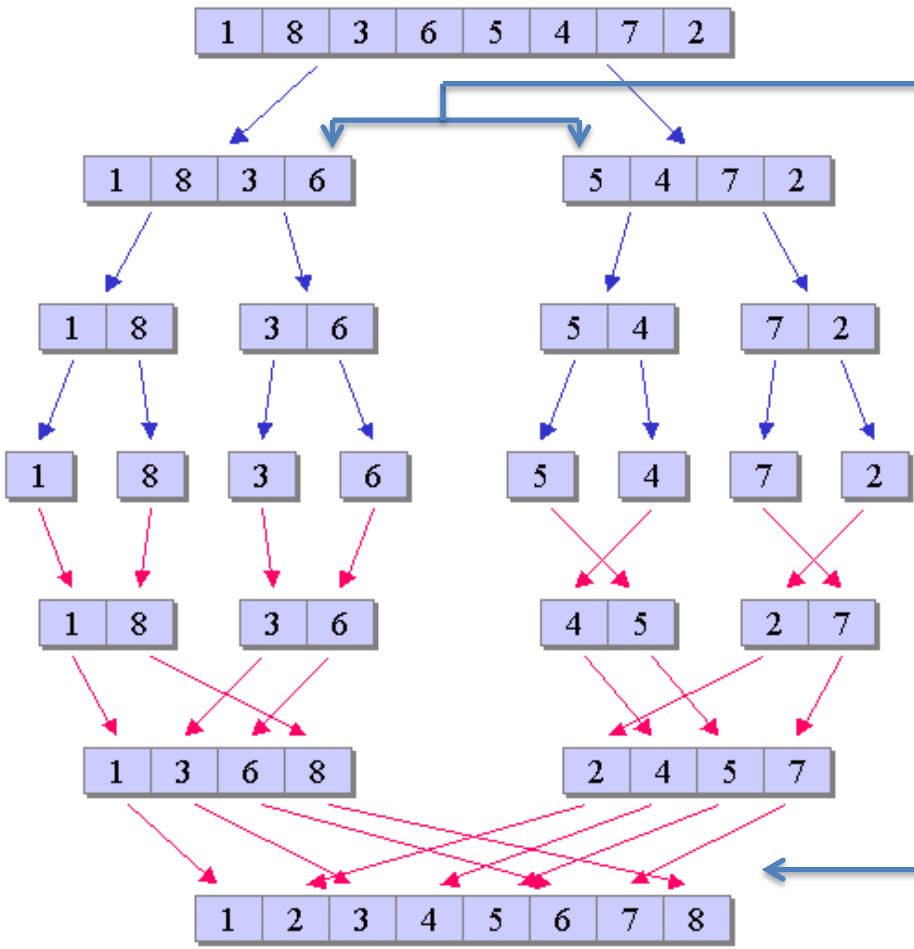


the sorting algorithm is simple?

"divide"
is
simple!

"conquer"= merge
and is more complicated
Need to use additional array(s) for the merging

Complexity of mergesort: Recurrences



Recurrence:

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

$$T(1) = 1$$

$T(n) = ?$

The Master Theorem

If

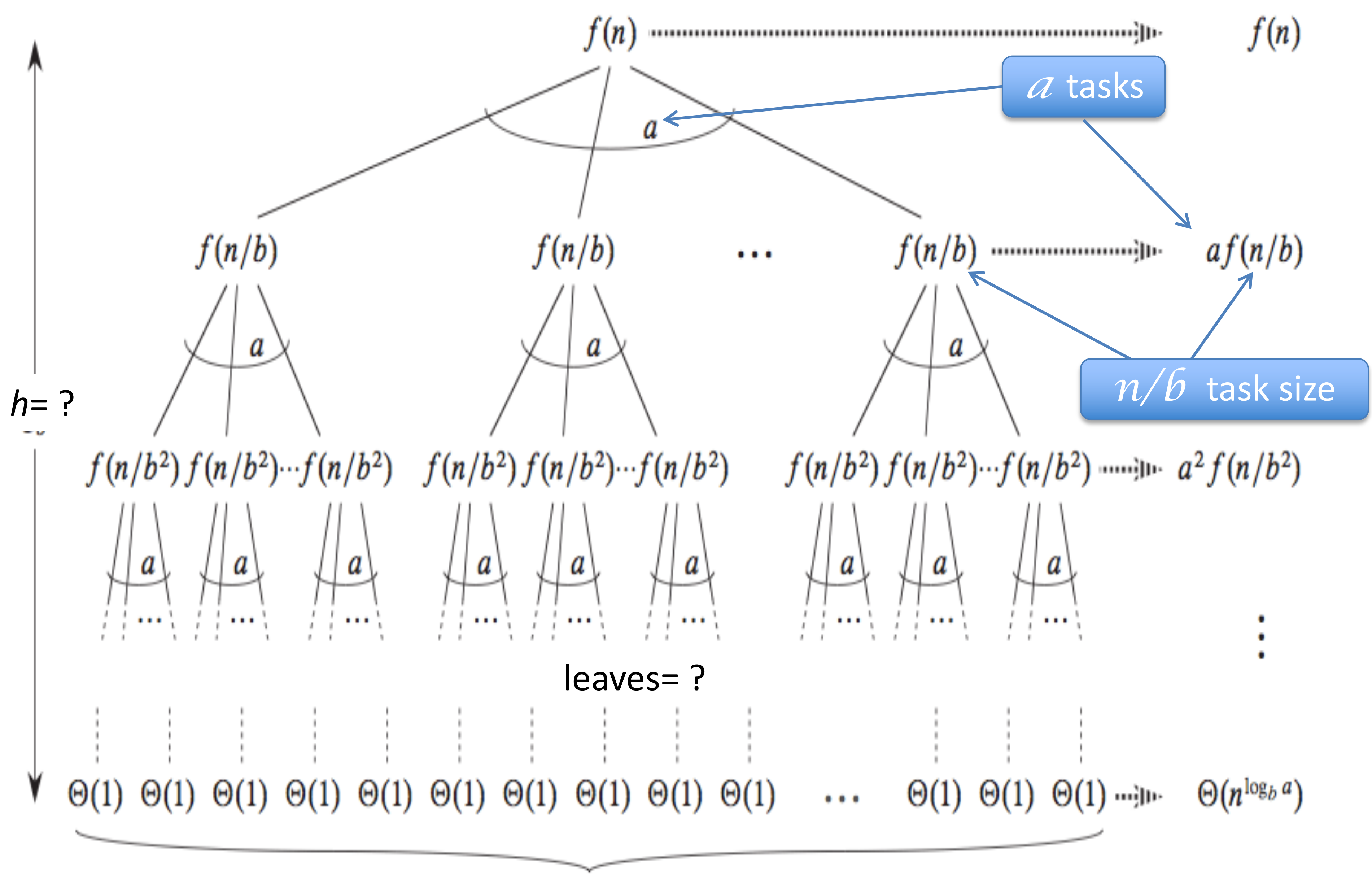
$$T(n) = aT(n/b) + \Theta(n^d)$$

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

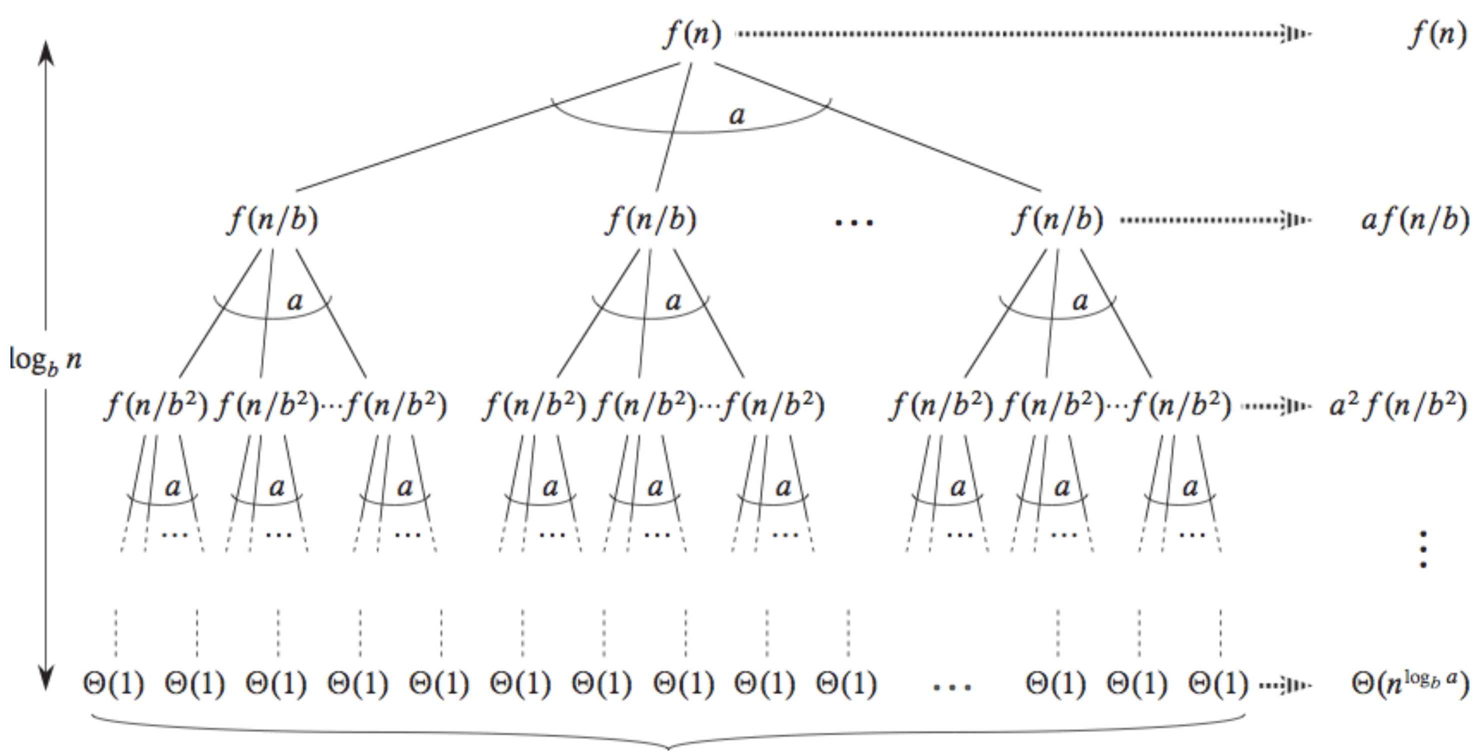
where $a \geq 1$, $b > 1$, and $d \geq 0$, then

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

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:

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

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

Remember sum of geometric sequence:

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

$$\Theta(\quad) \qquad \text{when } c>1?$$

$$c = a/b^d$$

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

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:

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

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

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

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

Master Theorem Examples:

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

Applying master theorem for merge sort:

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

$a = 2, b=2, d=1 \rightarrow d= \log_b a$

$\rightarrow T(n) = n \log n$

Other example:

- Binary Search:
- Quick sort best case:
- T(n)= 5T(n/2) + n2 + 9nlogn

Examples:

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

Applying master theorem for merge sort:

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

$a = 2, b=2, d=1 \rightarrow d= \log_b a$

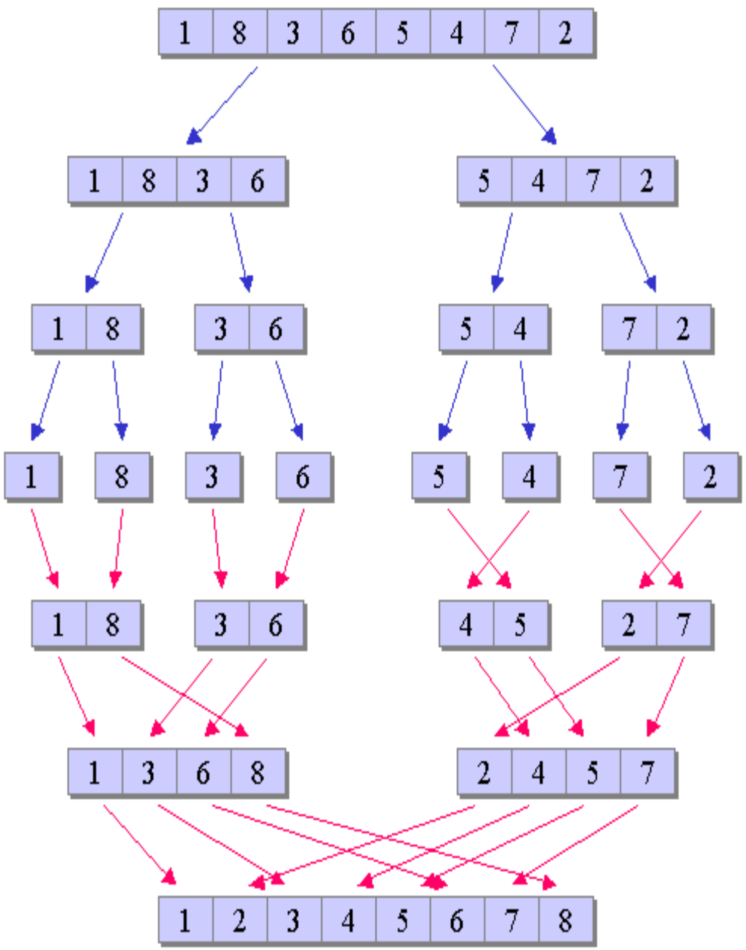
$\rightarrow T(n) = n \log n$

Other example:

- Binary Search: $T(n)= T(n/2) +1 \rightarrow a=1,b=2,d=0$ $d=\log_b a \rightarrow \Theta(\log n)$
- Quick sort best case: $T(n)= 2T(n/2) + n \rightarrow$

  $a=2,b=2,d=1$ $d=\log_b a \rightarrow \Theta(n\log n)$
- $T(n)= 5T(n/2) + n^2 + 9n\log n \rightarrow a=5,b=2,d=2$ $d<\log_b a \rightarrow \Theta(n^{\log_2 5})$

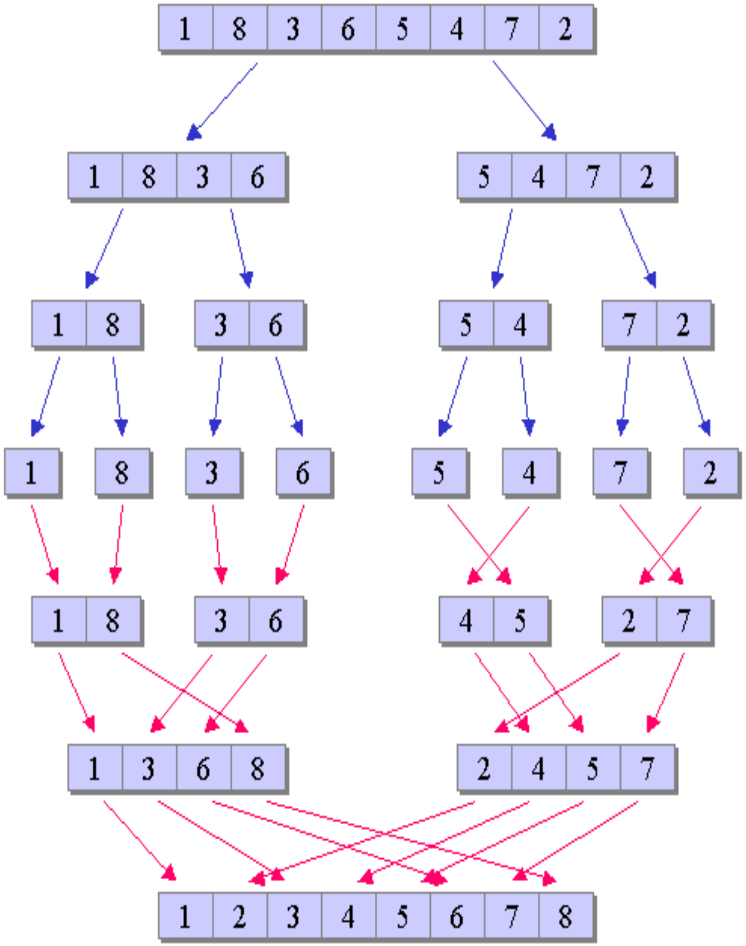
Top-Down MergeSort: Implementation Notes



using recursive calls
that is, using stacks!

```
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;
}
```

Top-Down MergeSort: space complexity when using arrays



Additional memory need:

$n + \log n = \boldsymbol{\theta}(n)$

```
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?)

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);
}
```

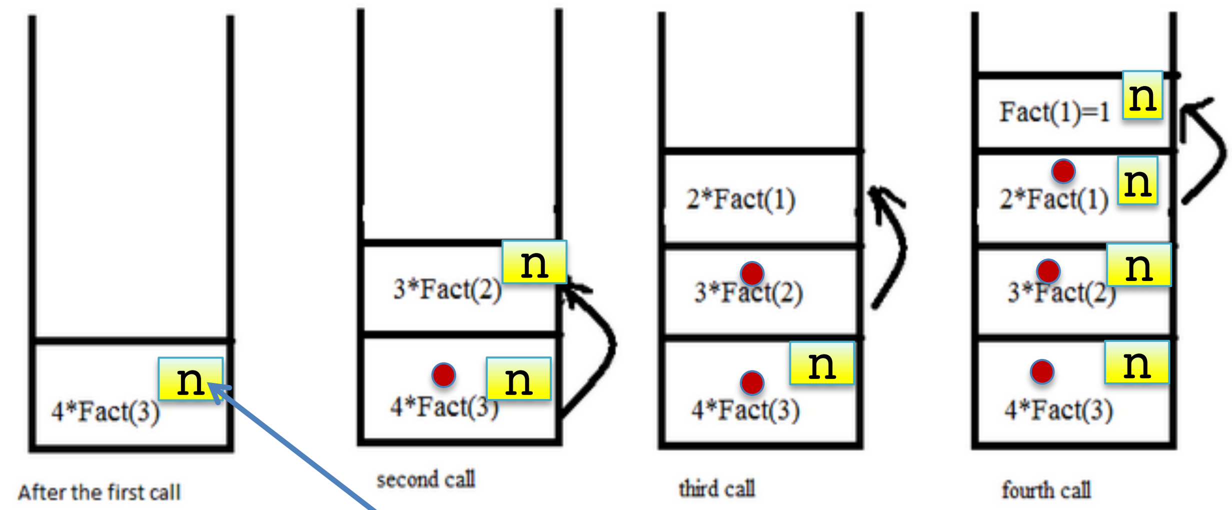
Hidden Space Complexity of Recursive Algorithms

Space complexity of function Fact: it is not **θ**(1) ?

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

Fact(4)

returned address



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

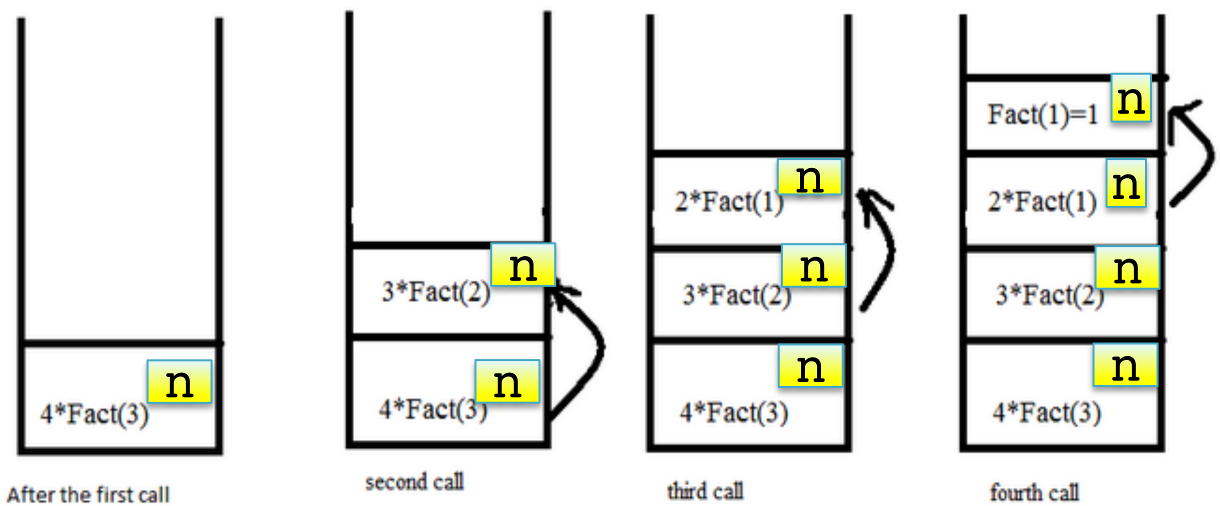
Hidden Space Complexity of Recursive Algorithms

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)
```



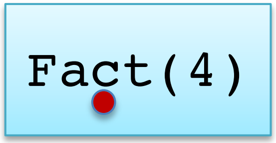
Hidden Space Complexity of Recursive Algorithms

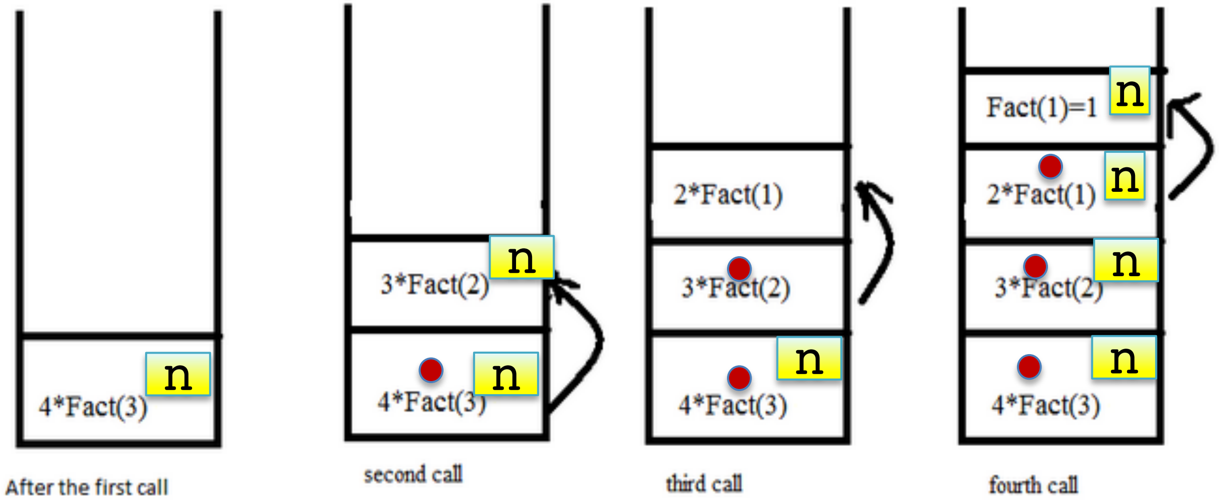
Space complexity of Fact: $\theta(n)$

Space complexity of recursive function =

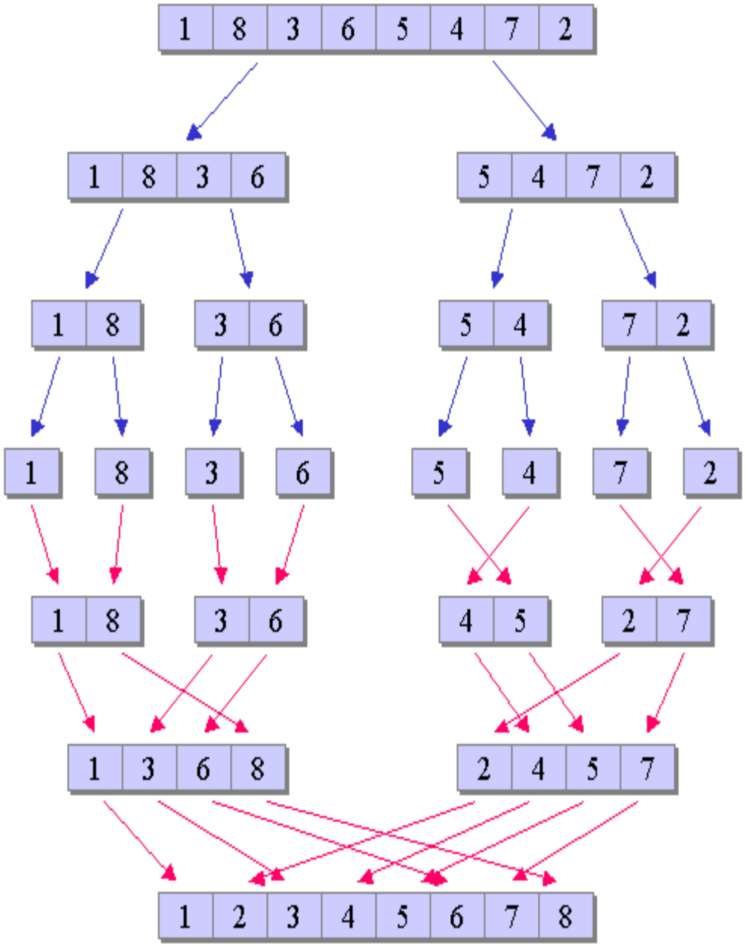
space for local variables $\times$ depth of rec calls.

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





Top-Down MergeSort: space complexity when using arrays

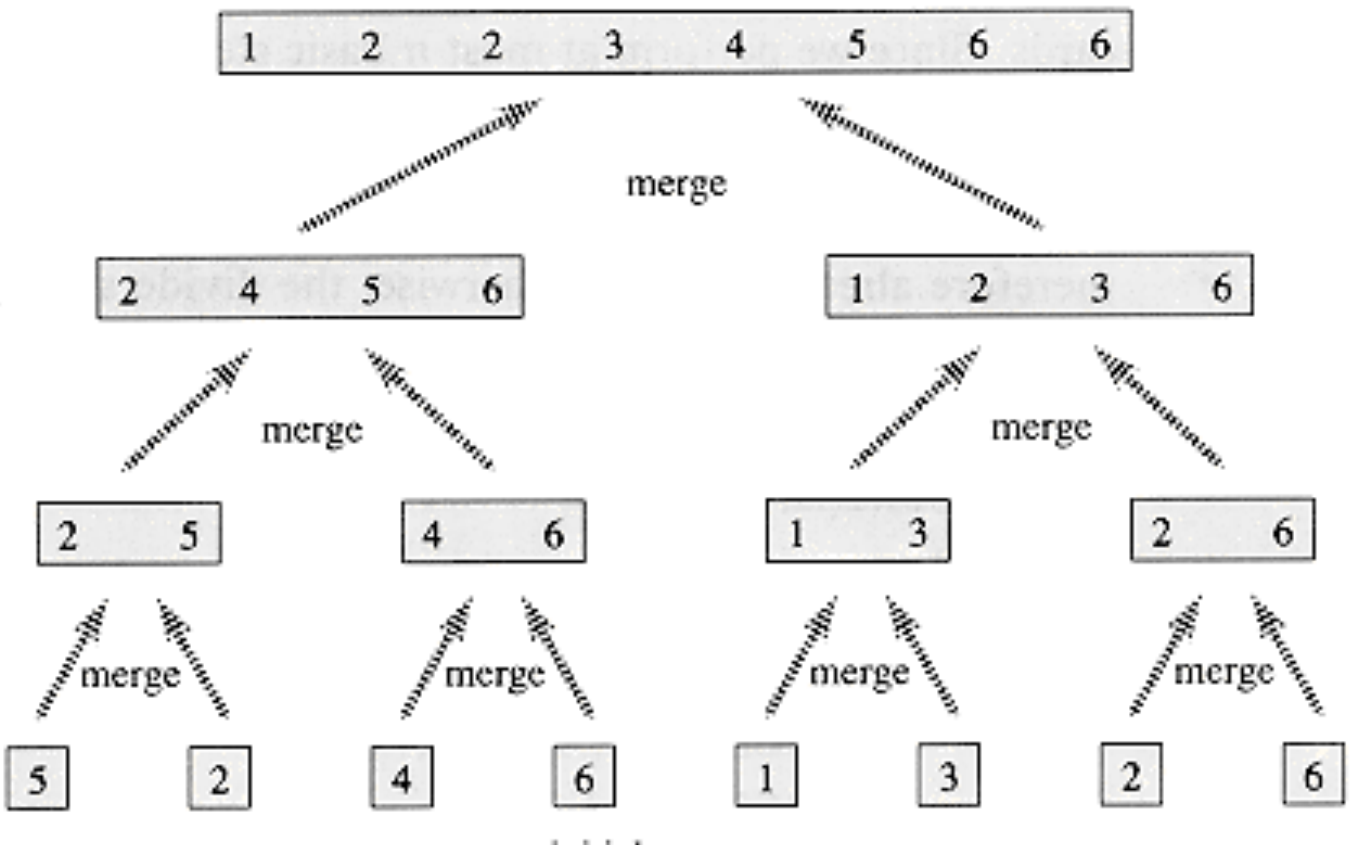


Additional memory need:

```
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?)

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



How to implement?

Merge Sort: Bottom-Up

array= {5,2,4,6,1,3,2,6}

How to implement: using a queue Q:

Q= `{ [5],[2],[4],[6],[1],[3],[2],[6] }`

→ `{ [4],[6],[1],[3],[2],[6], {2,5} }`

→ `...`

→ `{ {2,5}, {4,6}, {1,3}, {2,6} }`

→ `{ {1,3}, {2,6}, {2,4,5,6} }`

→ `{{2,4,5,6} , {1,2,3,6} }`

→ `{ {1,2,2,3,4,5,6,6} }`

`Note: additional memory = n for merging + n for the queue =` θ(n)

Merge Sort: Complexity Bottom-Up

Time Complexity:

Space complexity:

- If using arrays
- if using linked lists (preferable, why?)

How to implement: using a queue Q:

Q= `{ [5],[2],[4],[6],[1],[3],[2],[6] }`

→ `{ [4],[6],[1],[3],[2],[6], {2,5} }`

→ `...`

→ `{ {2,5}, {4,6}, {1,3}, {2,6} }`

→ `{ {1,3}, {2,6}, {2,4,5,6} }`

→ `{{2,4,5,6} , {1,2,3,6} }`

→ `{ {1,2,2,3,4,5,6,6} }`

Merge Sort: Complexity Bottom-Up

Time Complexity:

Space complexity:

- If using arrays: θ(n)
- if using linked lists : θ(n), but only for the queue, no additional memory for merging

How to implement: using a queue Q:

Q= `{ [5],[2],[4],[6],[1],[3],[2],[6] }`

→ `{ [4],[6],[1],[3],[2],[6], {2,5} }`

→ `...`

→ `{ {2,5}, {4,6}, {1,3}, {2,6} }`

→ `{ {1,3}, {2,6}, {2,4,5,6} }`

→ `{{2,4,5,6} , {1,2,3,6} }`

→ `{ {1,2,2,3,4,5,6,6} }`

Group Exercises || MST prep || JupyterHub 7.1, 7.2

1. Trace the action of bottom-up mergesort on the array:

`1 45 17 29 15 2 4 23 29 16 27 10 12`

2. Solve recurrences (assuming T(1)= 1):

   a) $T(n)= T(n-1) + n$

   b) $T(n)= 2T(n/3) + n$

2. Write a function:

```
void pair_merge(int A[], int B[], int C[], int m, int n)
```

that merges `A` (a sorted array of `m` elements) with `B` (a sorted array of `n` elements) into `c` (a sorted array of `m+n` elements).

3. Suppose that `A[]` has `n` elements and is a sequence of sorted chunks of size `k`, write a code fragment that employs `pair_merge()` to turn `A[]` into a sequence of sorted chunks of size `2k`. Beware that depending on `n`:

- the number of chunks might be not even
- the last chunk might have less than `k` elements

Lab: P7.1 and P7.2 (JupyterHub)

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.

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.