

Recursion Techniques - Mergesort

Bernd Burgstaller
Yonsei University



Pitfalls Of Recursion

- If the recursion **never reaches the base case**, the recursive calls will continue until the computer runs out of memory and the program crashes. The message “stack overflow error” or “heap storage exhaustion” indicates a possible runaway recursion.



```
printf("%d\n", fact(-1));
```

```
int fact(int n) {  
    if (n == 1)  
        return 1;  
    else  
        return n * fact(n - 1);  
}
```

- When programming recursively, you need to make sure that the algorithm is moving towards the base case. Each successive call of the algorithm must be solving a simpler version of the problem.
- Any recursive algorithm can be implemented iteratively, but sometimes only with great difficulty. However, a recursive solution will always run more slowly than an iterative one because of the overhead for procedure calls.
 - Tail recursion and related compiler optimizations can help.

Divide and Conquer

- Break the problem into several subproblems that are similar to the original problem but smaller in size, solve subproblems recursively, and combine solutions to create solution to the original problem.

At each level of the recursion:

- 1) **Divide** the problem into a number of subproblems.
- 2) **Conquer** subproblems:
 - directly if subproblem is simple enough (base case)
 - recursively otherwise
- 3) **Combine** subproblem solutions into solution for the original problem.

Mergesort

1) **Divide** array into 2 halves:

Assume we want to sort an array of characters, integers or floats.

A	L	G	O	R	I	T	H	M	S
---	---	---	---	---	---	---	---	---	---

A	L	G	O	R
---	---	---	---	---

I	T	H	M	S
---	---	---	---	---

divide

Mergesort

- 1) **Divide** array into 2 halves.
- 2) **Conquer**: recursively sort each half:

A	L	G	O	R	I	T	H	M	S
---	---	---	---	---	---	---	---	---	---

A	L	G	O	R
---	---	---	---	---

I	T	H	M	S
---	---	---	---	---

divide

A	G	L	O	R
---	---	---	---	---

H	I	M	S	T
---	---	---	---	---

sort

Mergesort

- 1) **Divide** array into 2 halves.
- 2) **Conquer**: recursively sort each half.
- 3) **Combine**: merge two halves to make sorted whole:

A	L	G	O	R	I	T	H	M	S
---	---	---	---	---	---	---	---	---	---

A	L	G	O	R
---	---	---	---	---

I	T	H	M	S
---	---	---	---	---

divide

A	G	L	O	R
---	---	---	---	---

H	I	M	S	T
---	---	---	---	---

sort

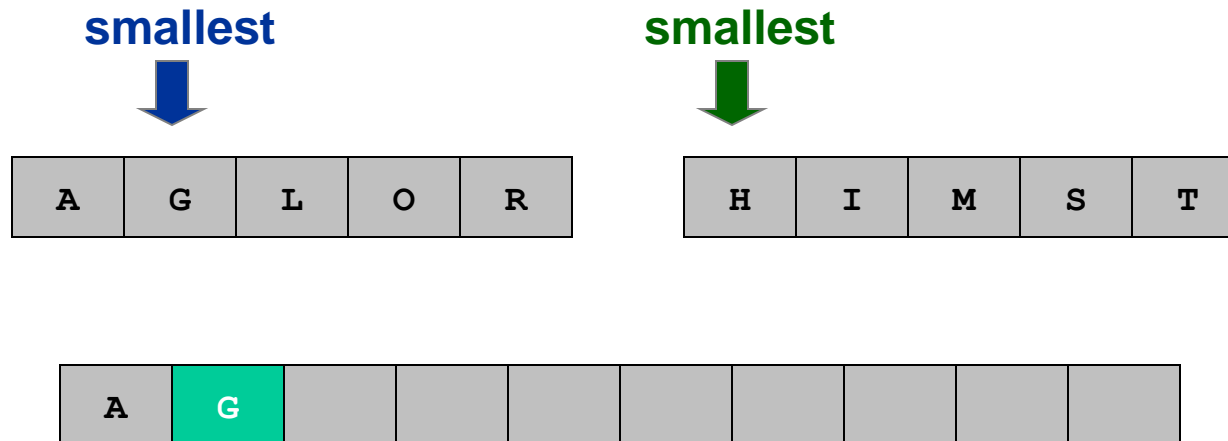
A	G	H	I	L	M	O	R	S	T
---	---	---	---	---	---	---	---	---	---

merge

Merging

Combine:

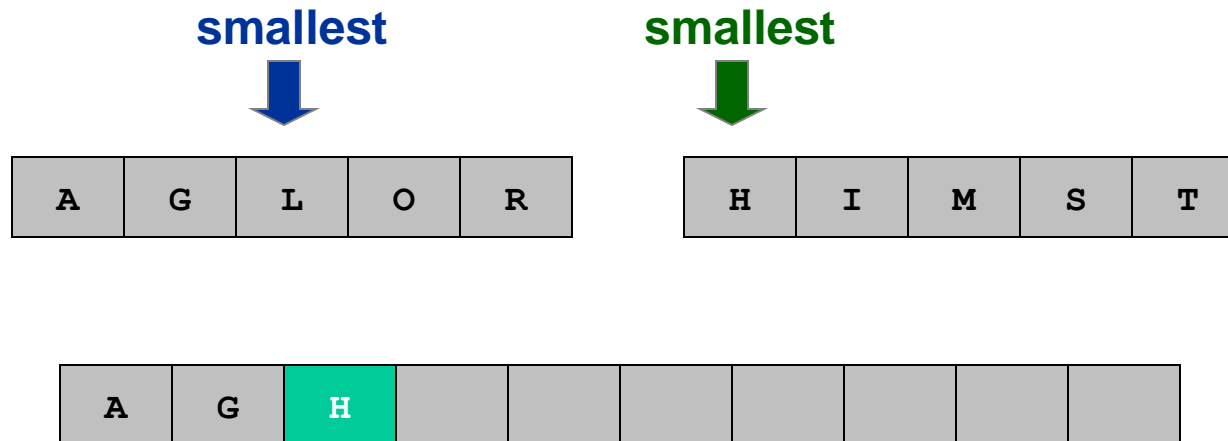
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Combine:

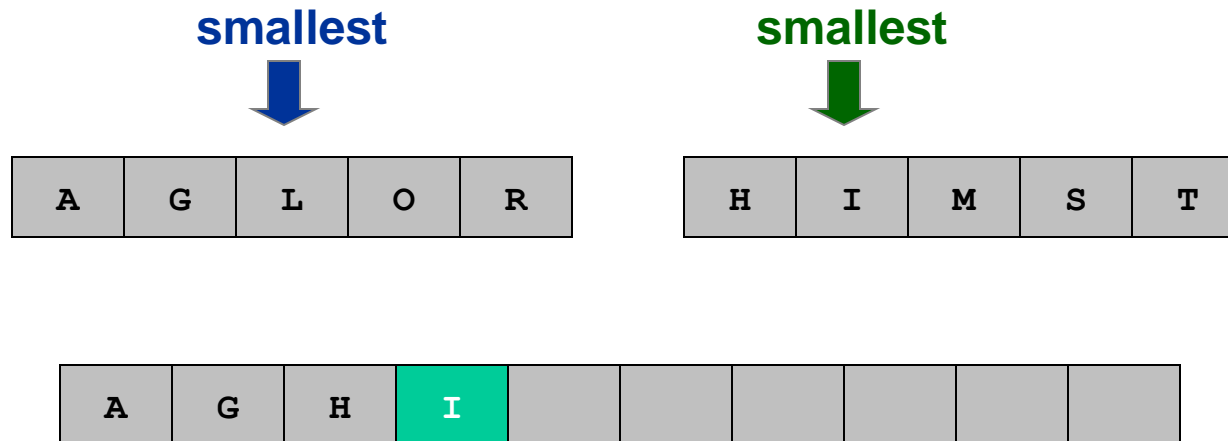
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Combine:

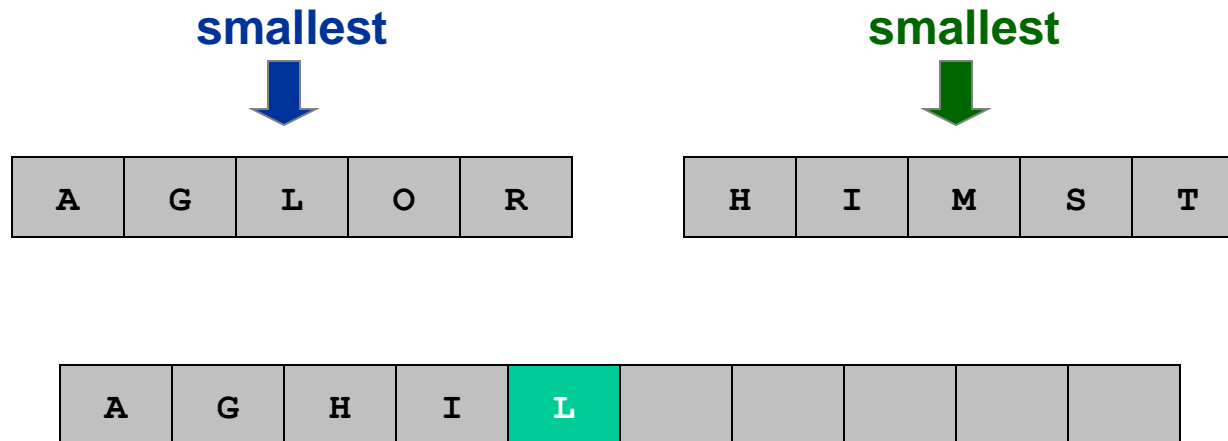
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Combine:

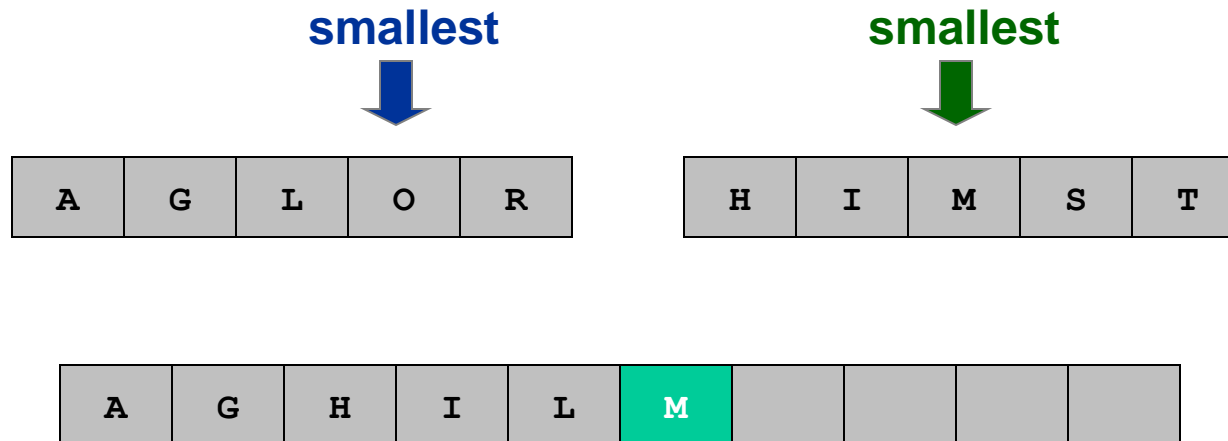
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Combine:

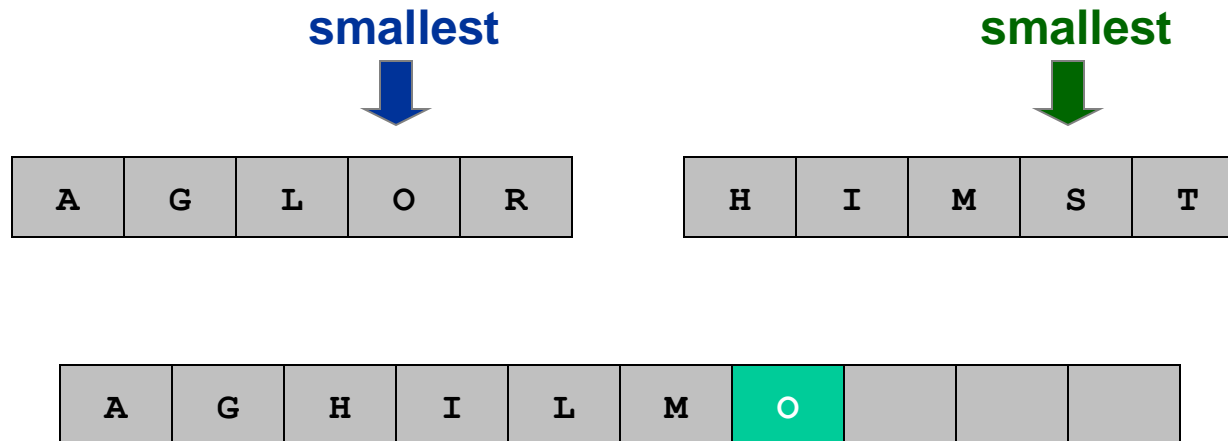
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Combine:

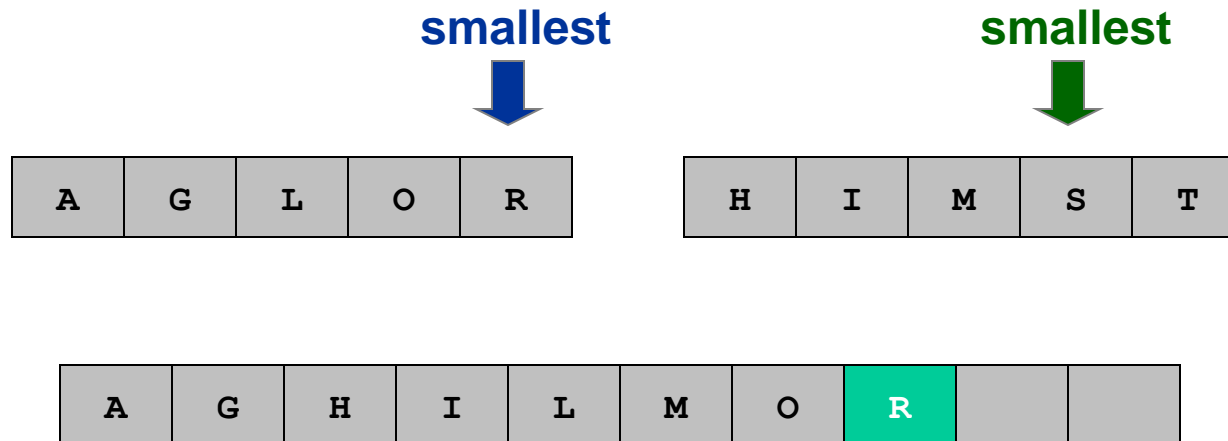
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Combine:

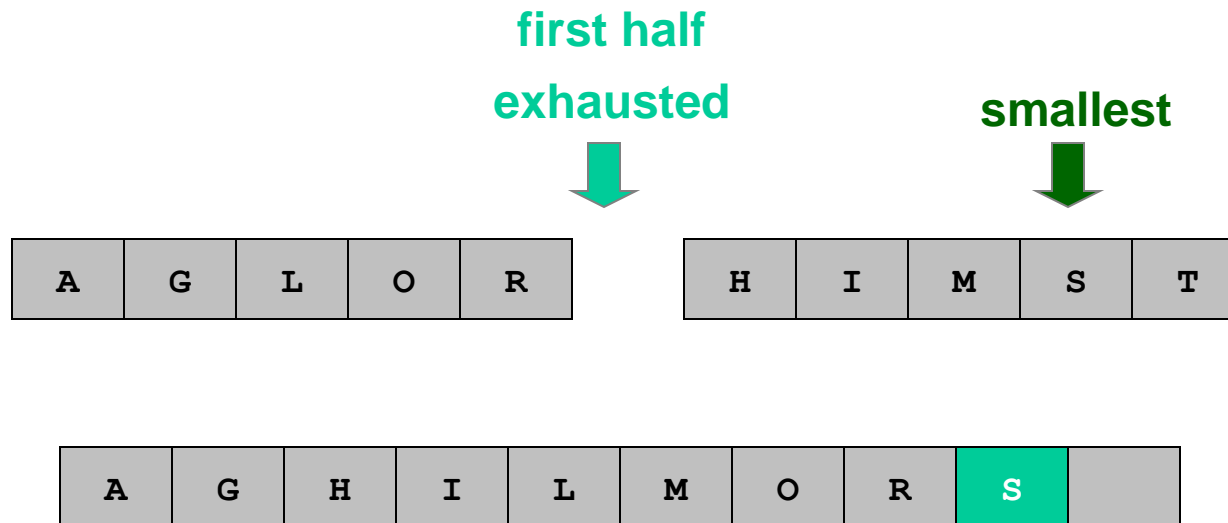
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Combine:

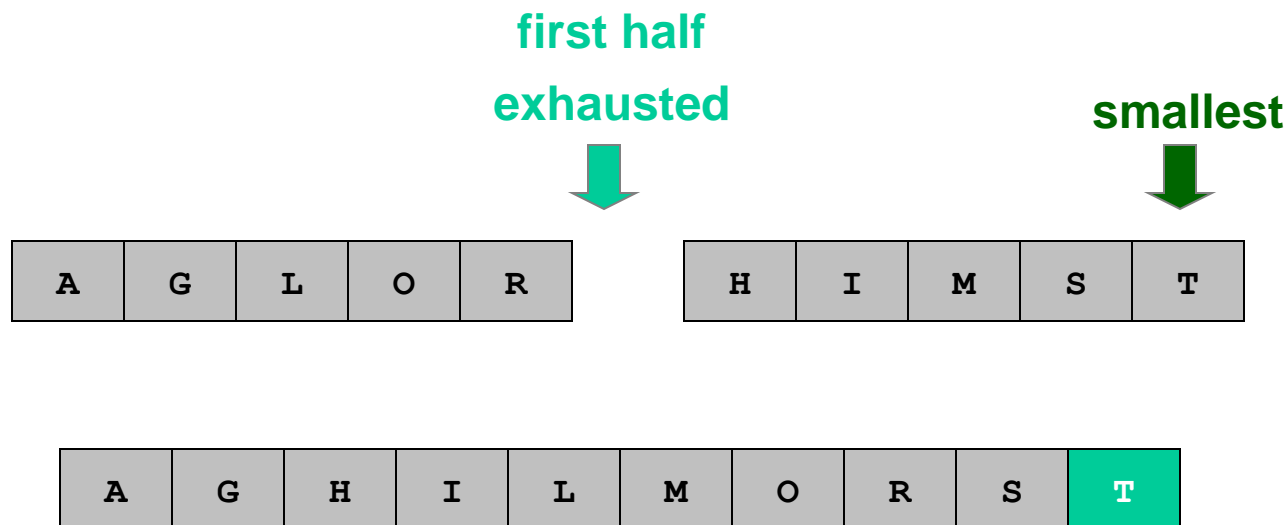
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



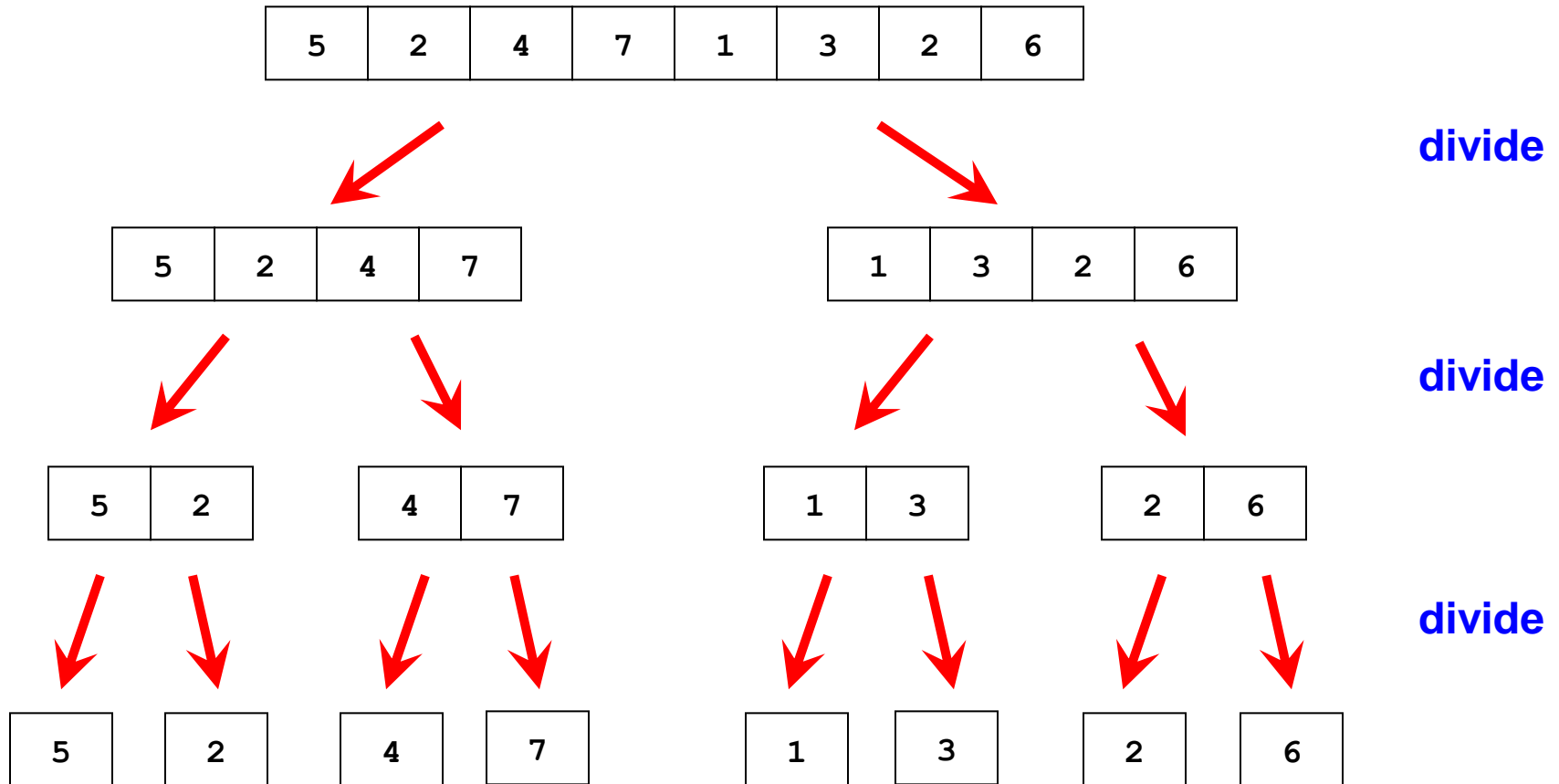
Merging

Combine:

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Mergesort (Divide)



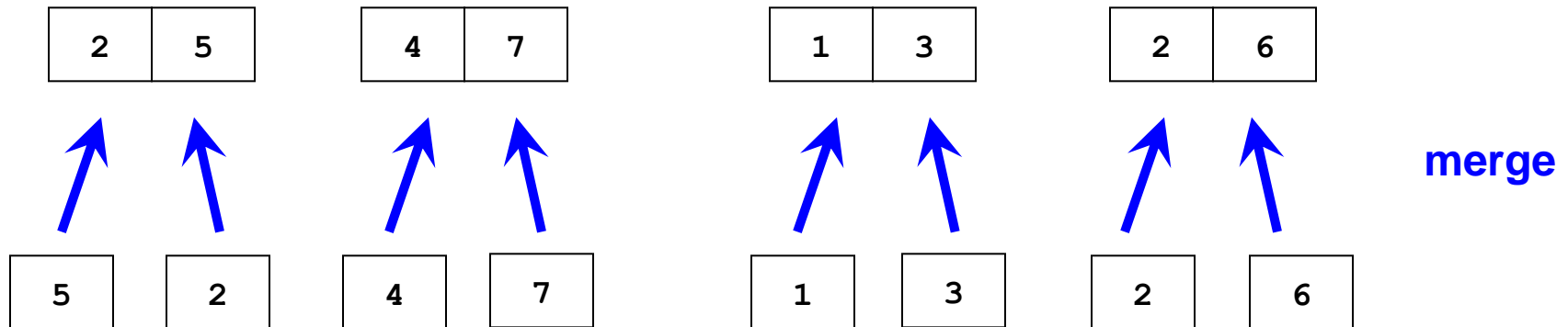
- Divide until we reach 1-element base-case.
 - One-element array is trivially sorted.

Mergesort (Combine)

5	2	4	7	1	3	2	6
---	---	---	---	---	---	---	---

5	2	4	7
---	---	---	---

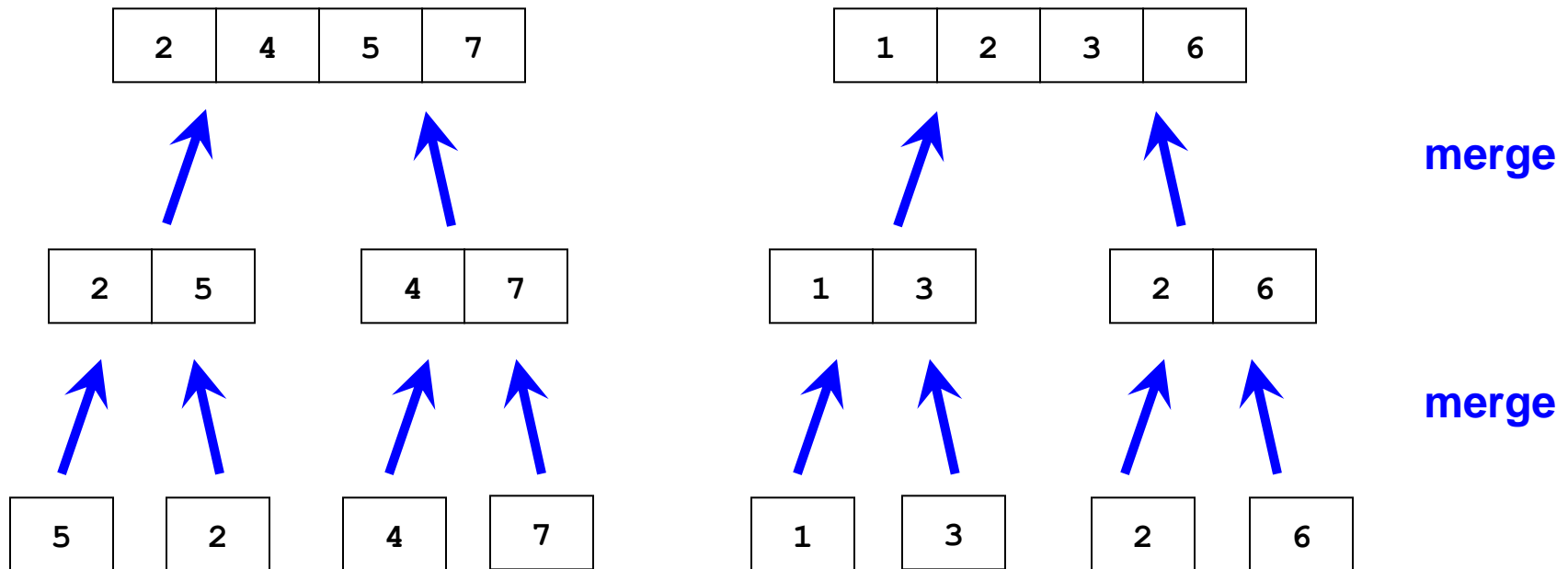
1	3	2	6
---	---	---	---



- Merge, starting from the 1-element base-cases,

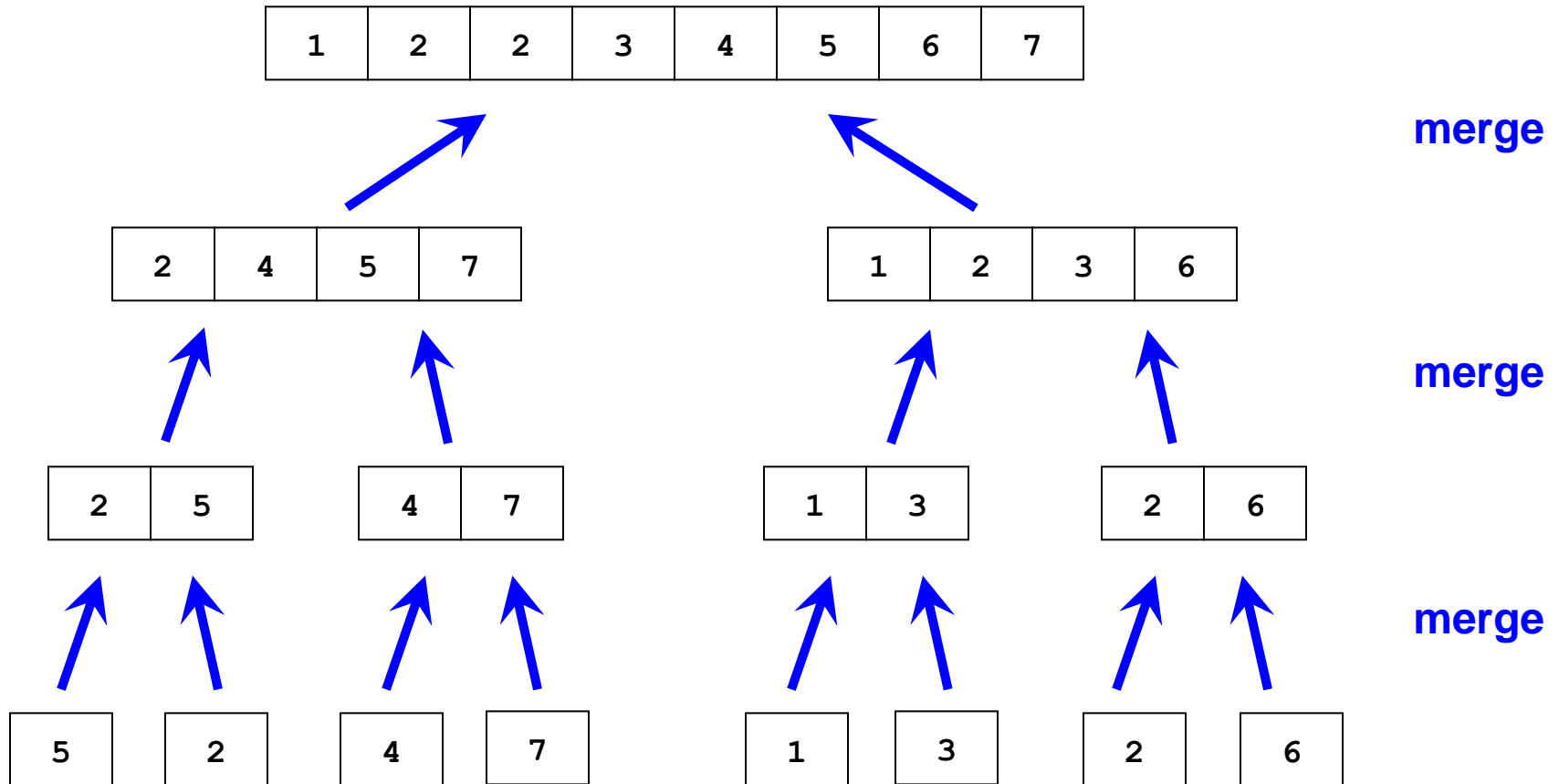
Mergesort (Combine)

5	2	4	7	1	3	2	6
---	---	---	---	---	---	---	---



- ... merging the 2-element arrays, ...

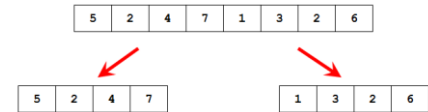
Mergesort (Combine)



- ... until we merge the final result array.

Implementing Mergesort

```
void MergeSort(float A[], int p, int r)
{
    int q;
    if (p < r) {
        q = (p+r)/2;
        MergeSort(A, p, q);
        MergeSort(A, q+1, r);
        Merge(A, p, q, r);
    }
}
```



5	2	4	7	1	3	2	6	A
0	1	2	3	4	5	6	7	index

MergeSort (A, 0, 7)

MergeSort (A, 0, 3)

MergeSort (A, 4, 7)

Implementing Mergesort (cont.)

```
void MergeSort(float A[], int p, int r)
{
    int q;
    if (p < r) {
        q = (p+r)/2;
        MergeSort(A, p, q);
        MergeSort(A, q+1, r);
        Merge(A, p, q, r);
    }
}
```

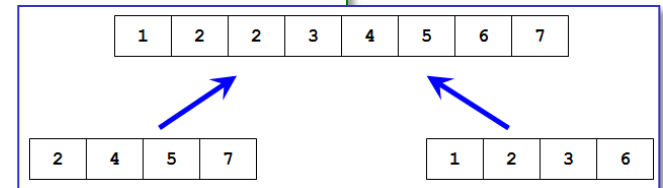
- MergeSort is a recursive procedure
- p and r are indices that mark the range of array A[] to be sorted.
- if to-be-sorted range larger than 1:
 - compute index **q** of middle element
 - **recursively** sort halves (p . . q) and (q+1 . . r) and then **merge** sorted halves.
- If to-be-sorted range == 1:
 - do nothing (base case, arrays of length 1 are trivially sorted!)

Implementing the Merge Step of Mergesort (p. 1)

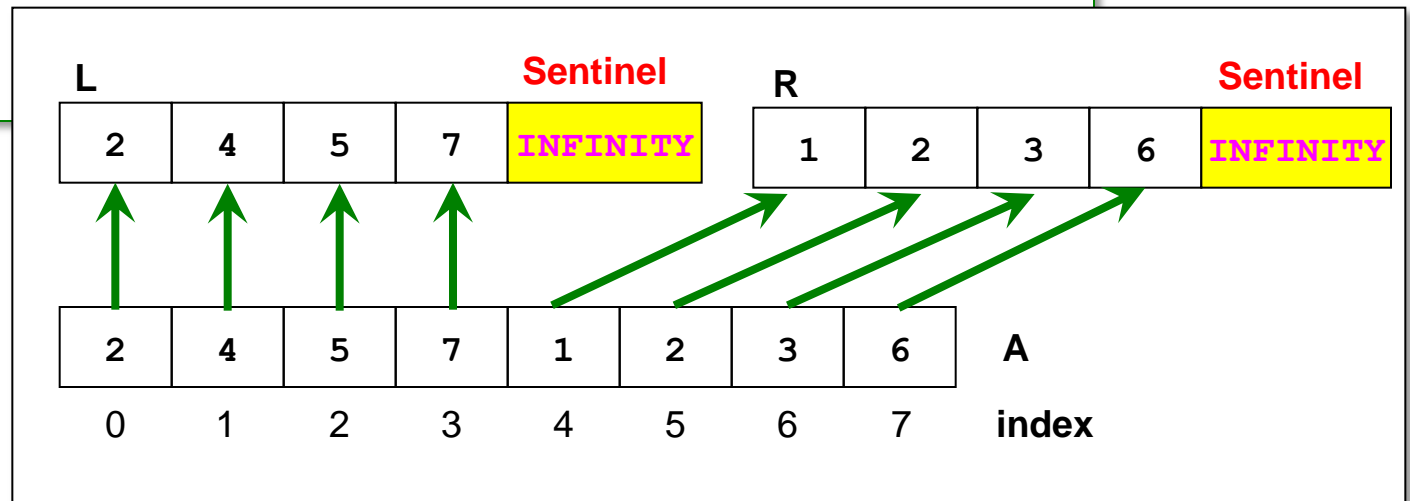
```

void Merge(float A[], int p, int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    for (i=0; i<n1; i++) {
        L[i] = A[p+i];
    }
    for (j=0; j<n2; j++) {
        R[j] = A[q+j+1];
    }
    L[n1] = INFINITY; R[n2] = INFINITY;
    ...
}

```



L and R are
auxiliary
arrays to temporarily
store the array data

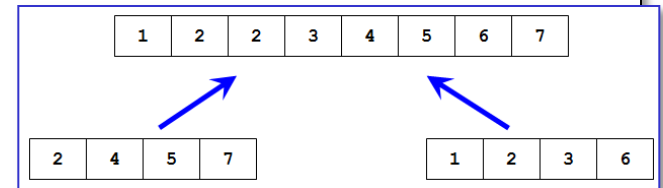


Implementing the Merge Step of Mergesort (p. 2)

```

void Merge(float A[], int p, int q, int r) {
    ...
    int i = 0; int j = 0;
    for (int k = p; k <= r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}

```



smallest (i)



L					Sentinel
2	4	5	7	INFINITY	

smallest (j)



R					Sentinel
1	2	3	6	INFINITY	

--	--	--	--	--	--	--	--

0

1

2

3

4

5

6

7

A

index



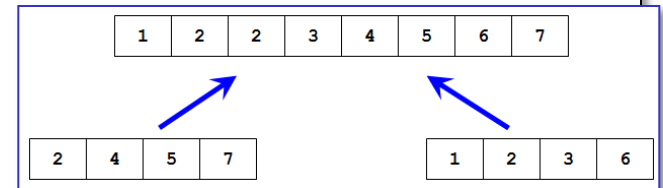
k

Implementing the Merge Step of Mergesort (p. 2)

```

void Merge(float A[], int p, int q, int r) {
    ...
    int i = 0; int j = 0;
    for (int k = p; k <= r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}

```



smallest (i)

L ↓

2	4	5	7	Sentinel INFINITY
---	---	---	---	-----------------------------

smallest (j)

R ↓

1	2	3	6	Sentinel INFINITY
---	---	---	---	-----------------------------

1							
---	--	--	--	--	--	--	--

0 1 2 3 4 5 6 7 **A** index

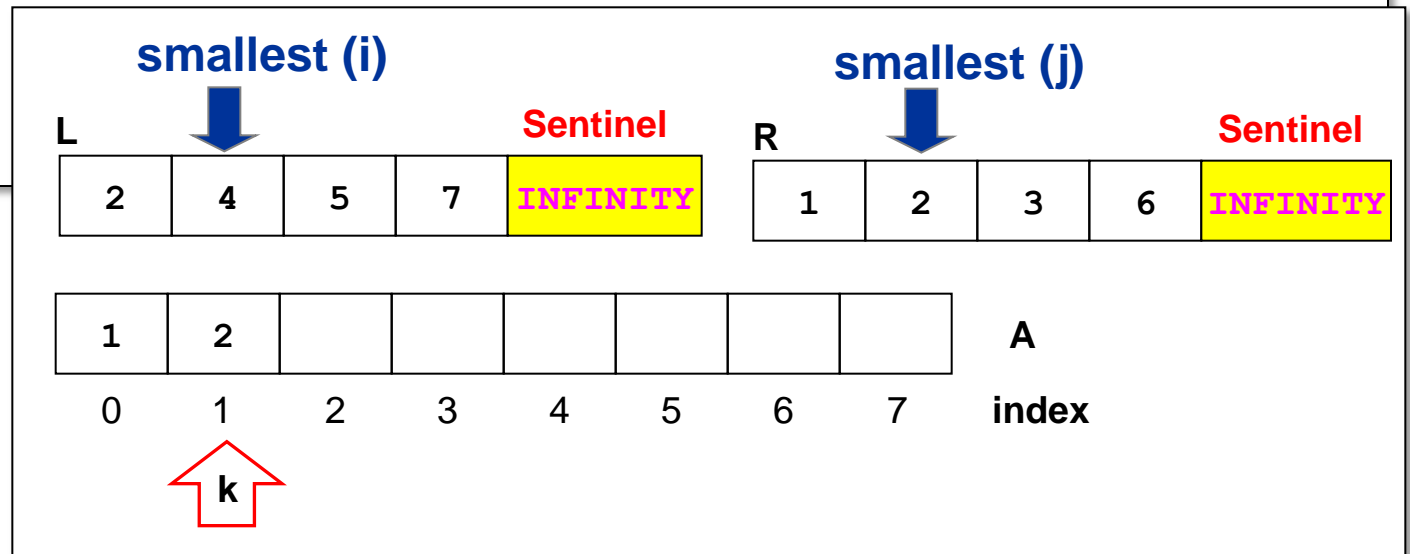
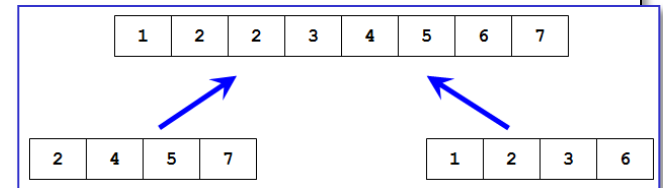
↑
k

Implementing the Merge Step of Mergesort (p. 2)

```

void Merge(float A[], int p, int q, int r) {
    ...
    int i = 0; int j = 0;
    for (int k = p; k <= r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}

```

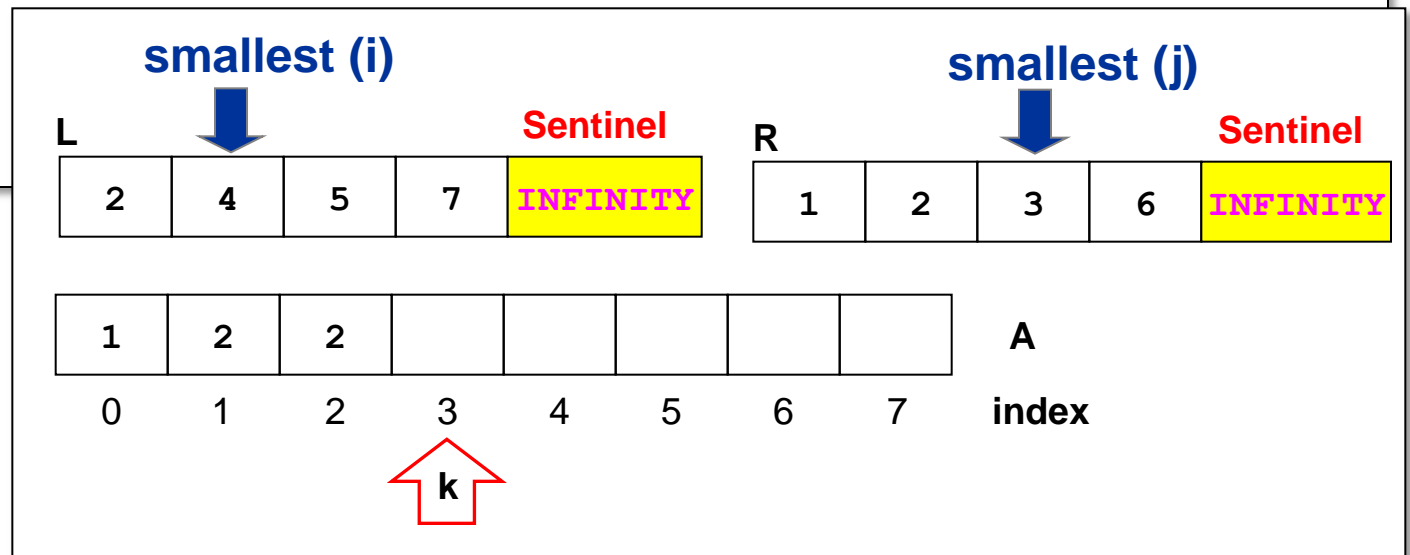
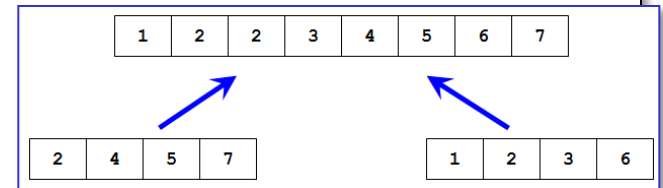


Implementing the Merge Step of Mergesort (p. 2)

```

void Merge(float A[], int p, int q, int r) {
    ...
    int i = 0; int j = 0;
    for (int k = p; k <= r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}

```

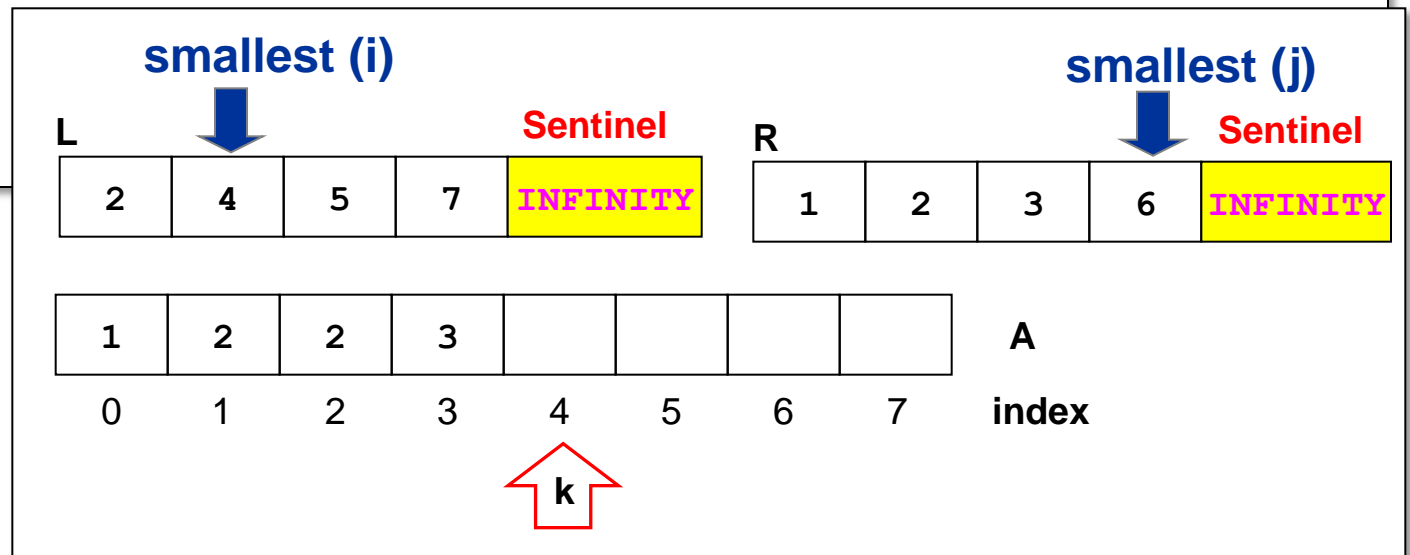
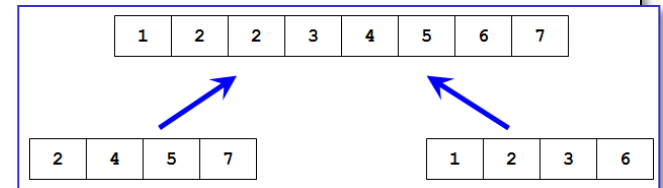


Implementing the Merge Step of Mergesort (p. 2)

```

void Merge(float A[], int p, int q, int r) {
    ...
    int i = 0; int j = 0;
    for (int k = p; k <= r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}

```

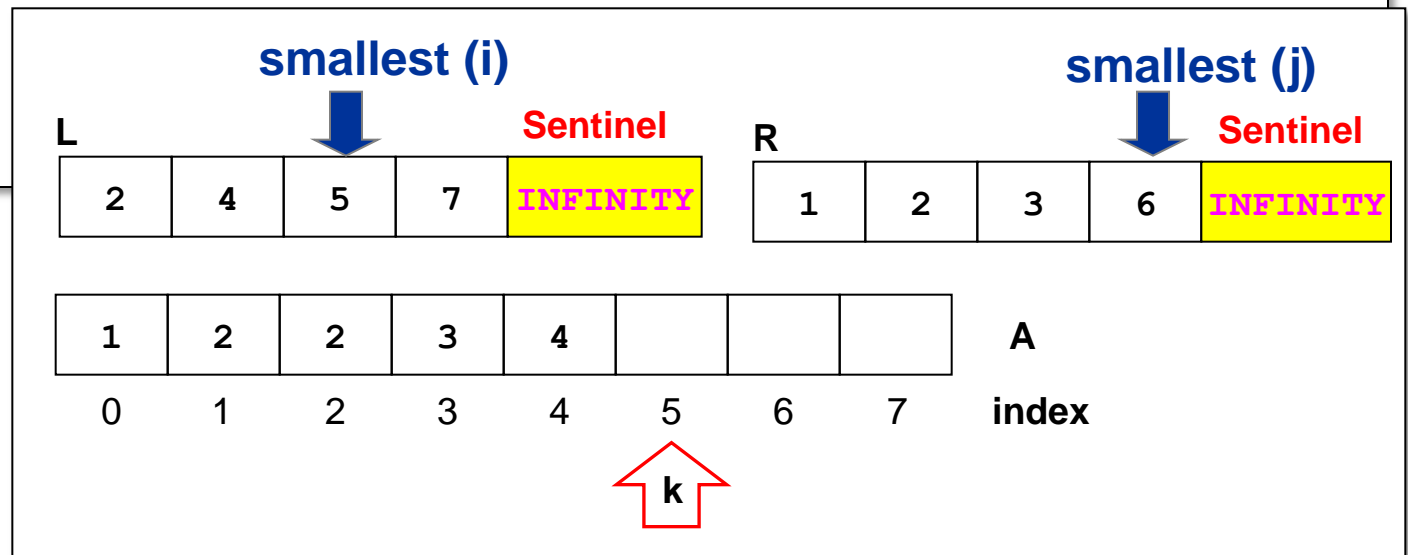
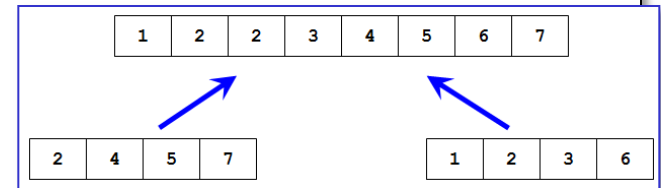


Implementing the Merge Step of Mergesort (p. 2)

```

void Merge(float A[], int p, int q, int r) {
    ...
    int i = 0; int j = 0;
    for (int k = p; k <= r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}

```

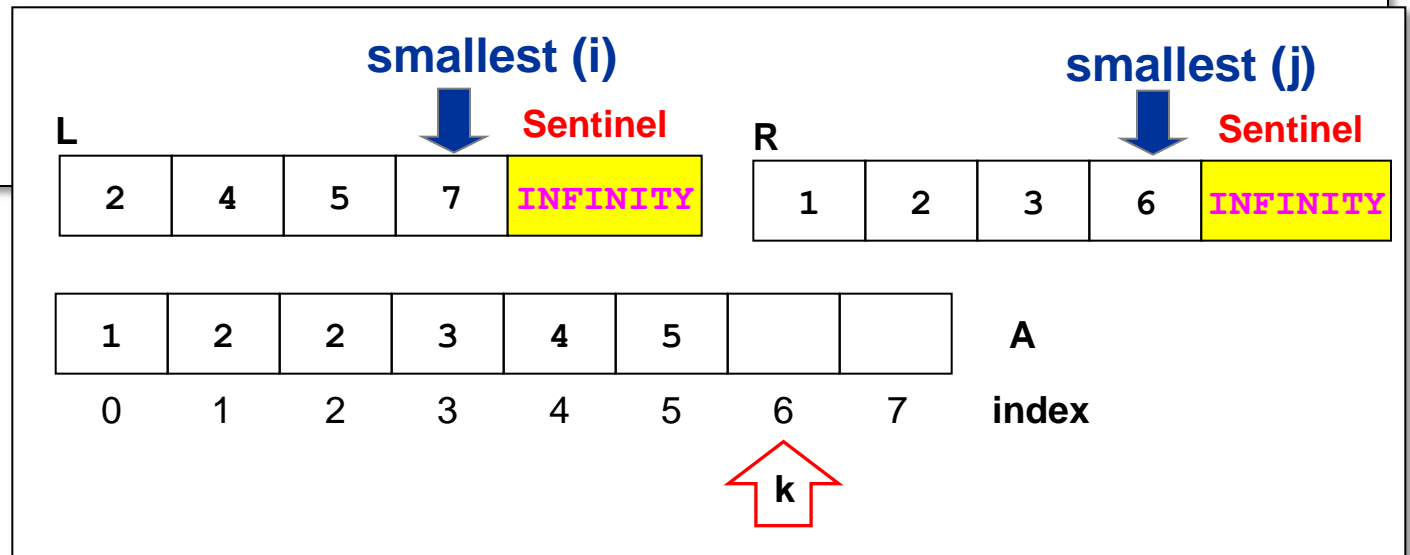
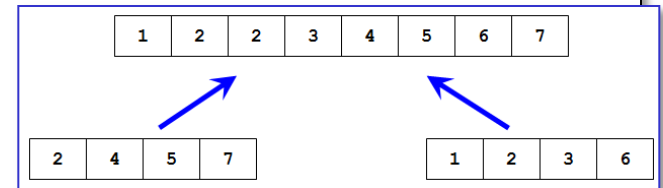


Implementing the Merge Step of Mergesort (p. 2)

```

void Merge(float A[], int p, int q, int r) {
    ...
    int i = 0; int j = 0;
    for (int k = p; k <= r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}

```

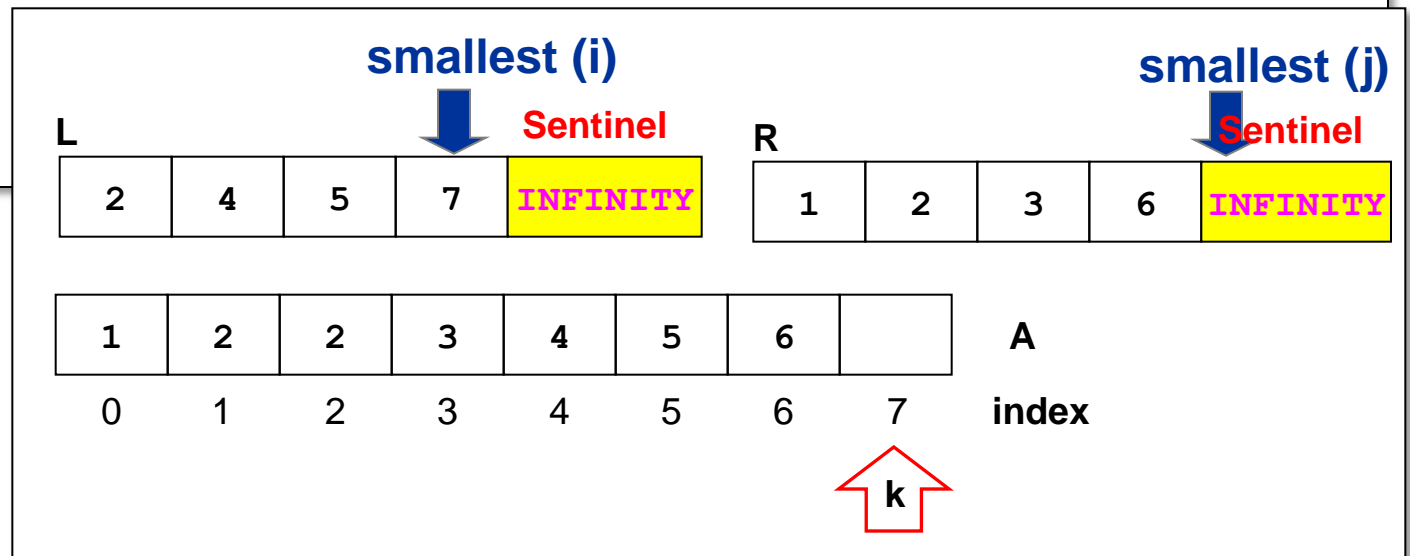
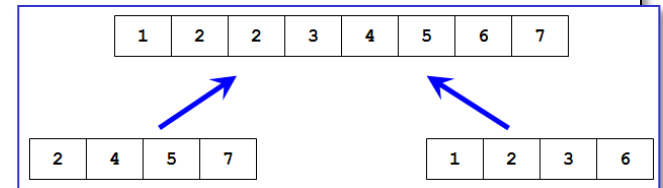


Implementing the Merge Step of Mergesort (p. 2)

```

void Merge(float A[], int p, int q, int r) {
    ...
    int i = 0; int j = 0;
    for (int k = p; k <= r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}

```

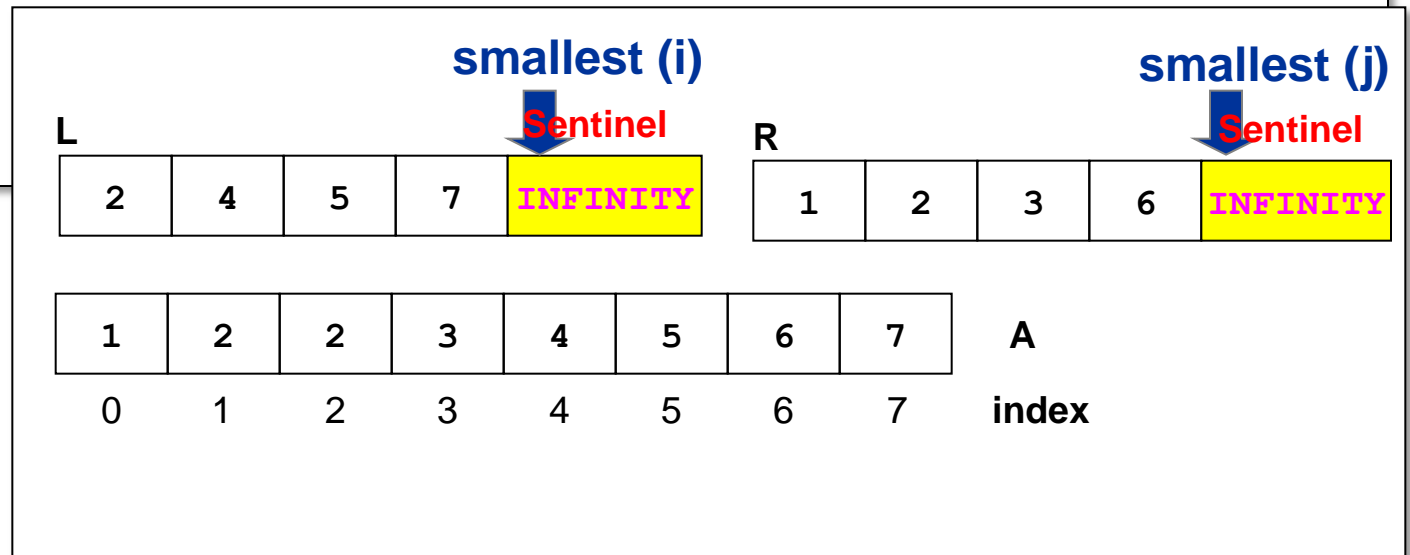
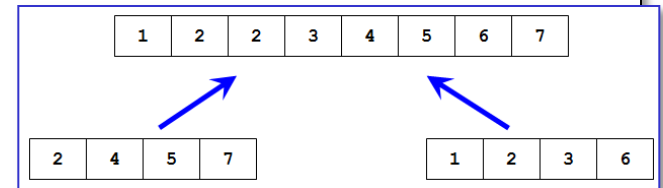


Implementing the Merge Step of Mergesort (p. 2)

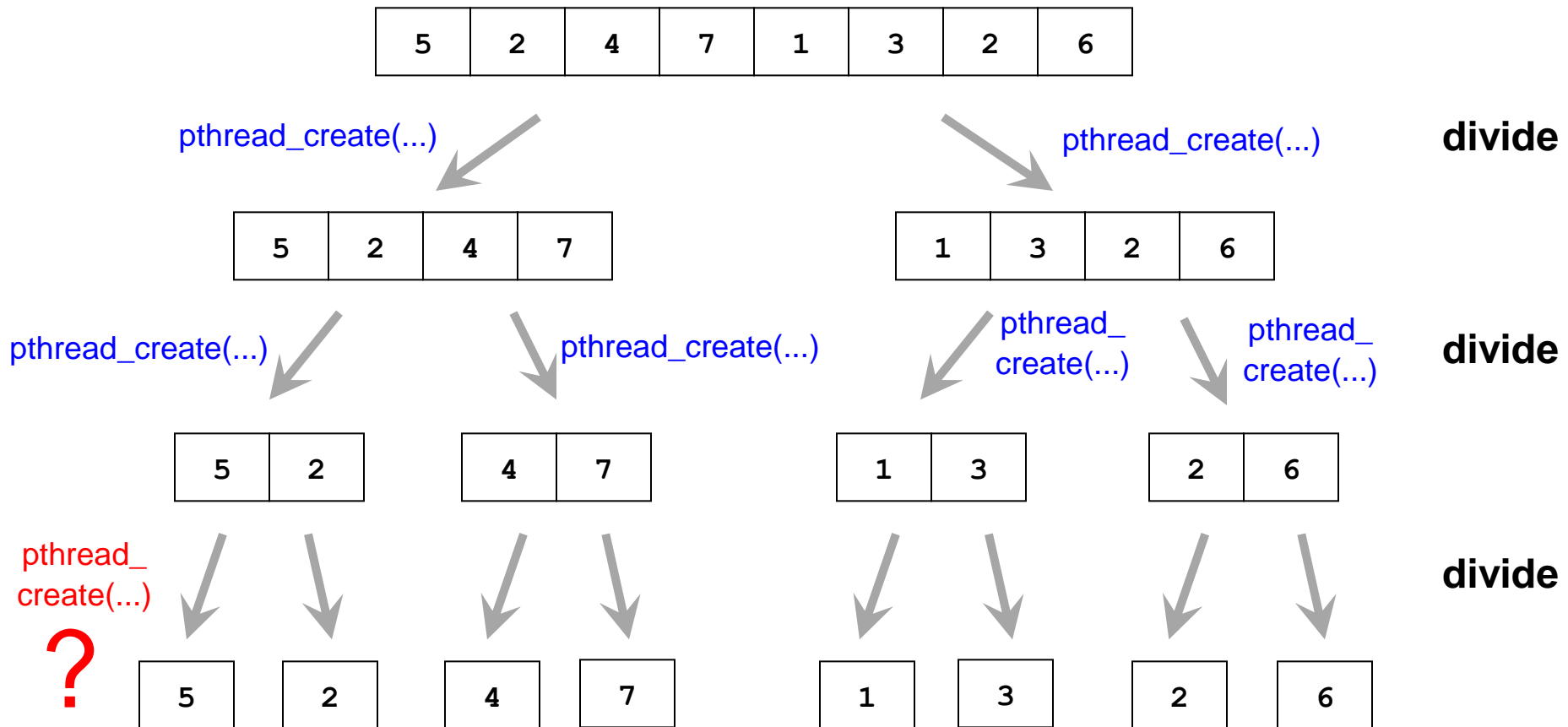
```

void Merge(float A[], int p, int q, int r) {
    ...
    int i = 0; int j = 0;
    for (int k = p; k <= r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}

```

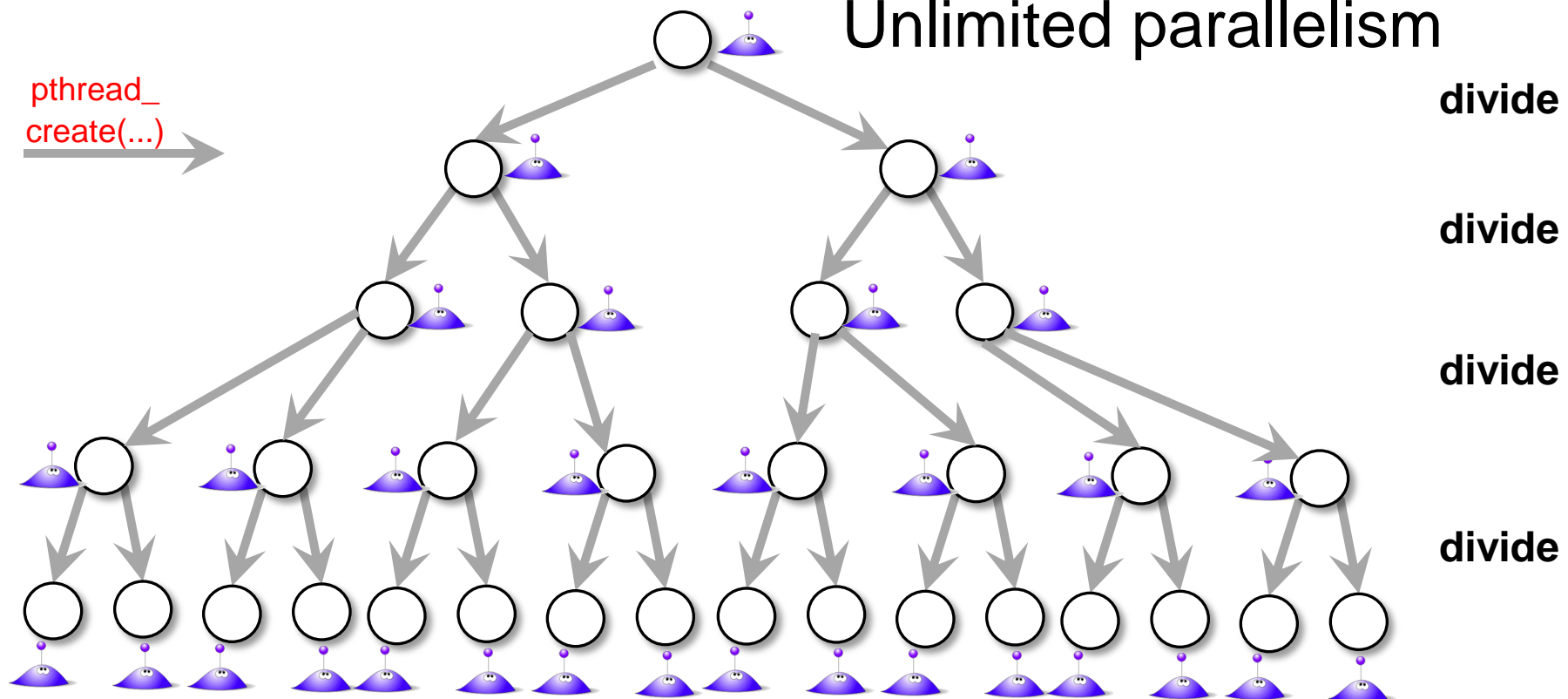


Parallelization of Mergesort



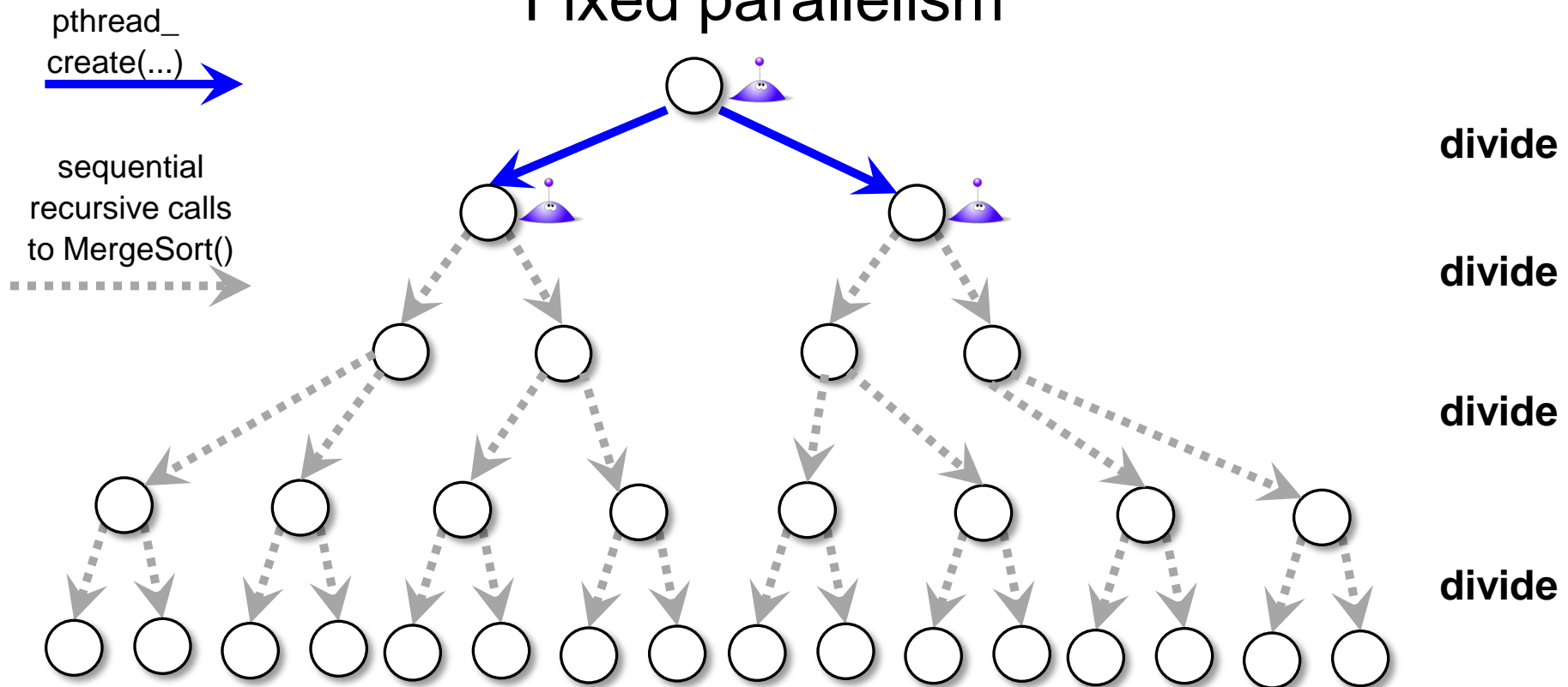
- For each dividing step, create 2 pthreads to sort in parallel.
 - At what recursion depth shall we **switch back to sequential mergesort**?

Unlimited parallelism



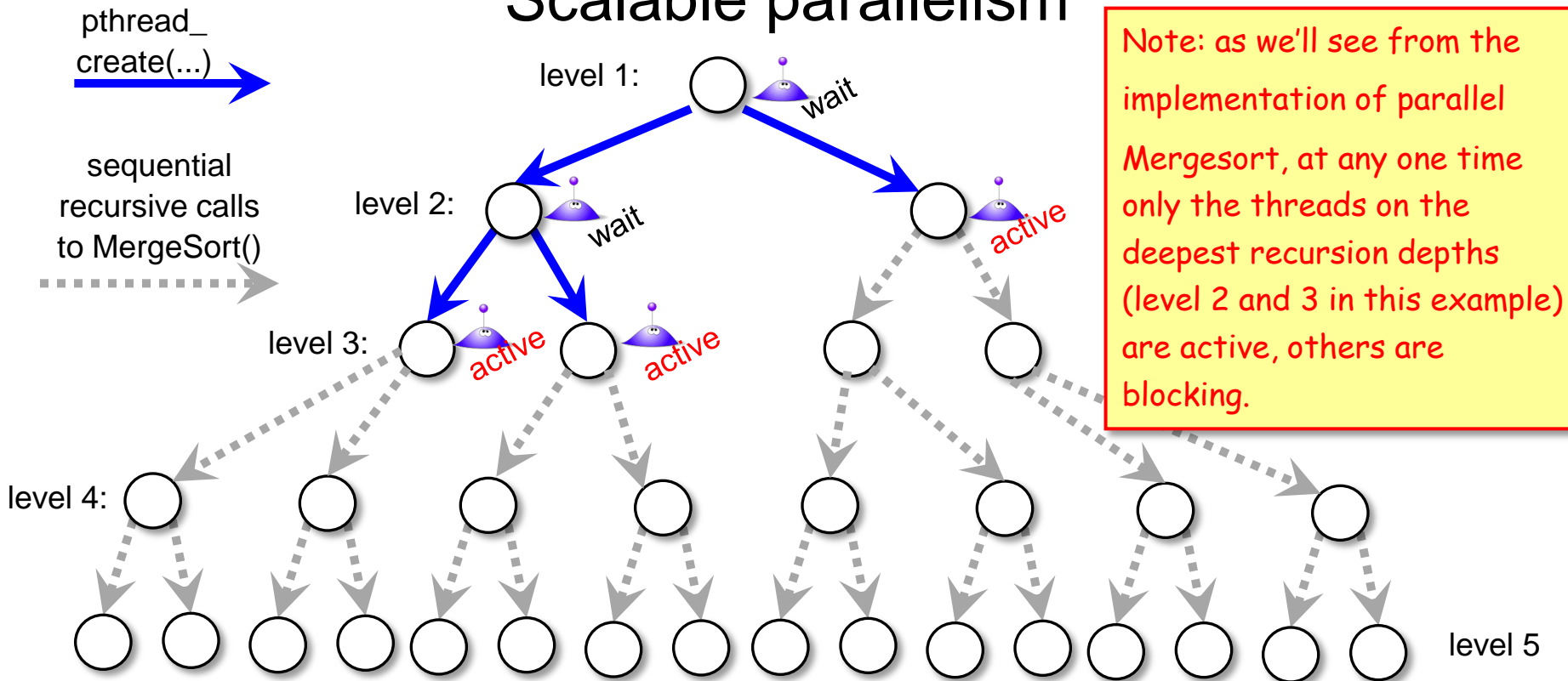
- With unlimited parallelism, we generate new threads with every dividing step, down to the base case.
 - Highest amount of **logical** parallelism (N threads, $N=2^{\text{levels}}$), **but**
 - towards base cases, work gets very small, thread creation overhead dominates
 - we have only $p \ll N$ cores. Only p threads can execute in parallel at any one time (**physical** parallelism). If number of ready-to-execute threads exceeds number of cores, Linux will context-switch threads (overhead!)

Fixed parallelism



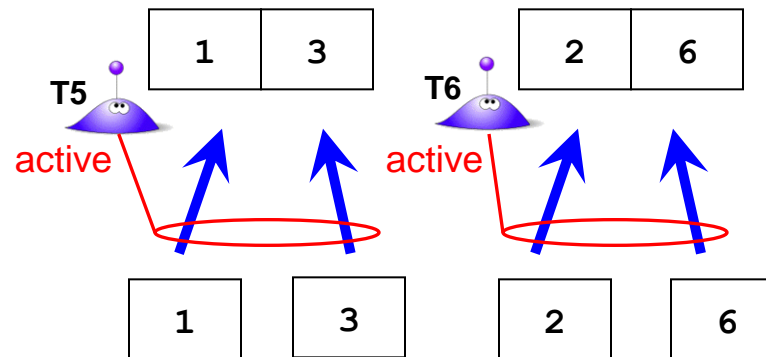
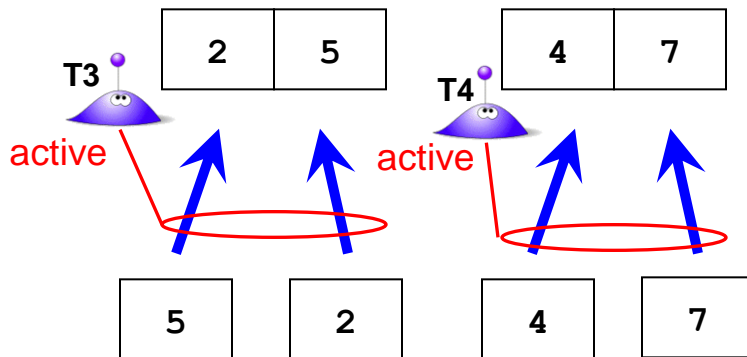
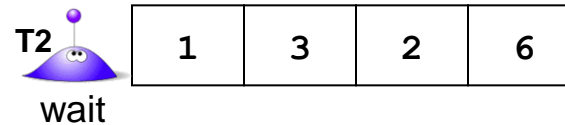
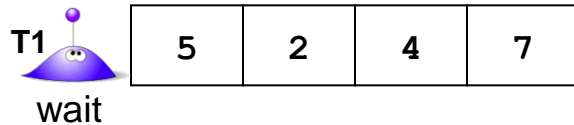
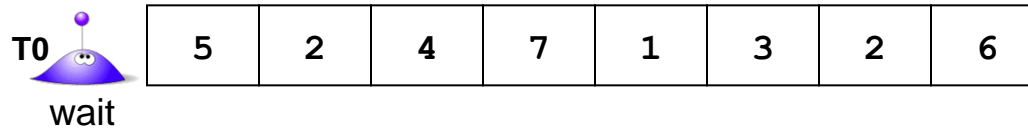
- With fixed parallelism, we generate new threads only until we reach a fixed, hard-coded level (e.g., 2 above). After that, we continue sequentially.
 - **Pro:** More efficient than 'unlimited parallelism', although less amount of 'logical' parallelism.
 - does not flood the computer with threads.
 - **Con:** **fixed behavior**, does not scale to higher numbers of cores.

Scalable parallelism



- With scalable parallelism, the level where we switch to sequential recursive calls depends on the number of available cores. E.g., 3 above.
 - **Pro:** Most efficient, scales to higher numbers of cores.
 - **Con:** Algorithms often harder to program for scalable parallelism.

Parallelization of Mergesort (cont.)

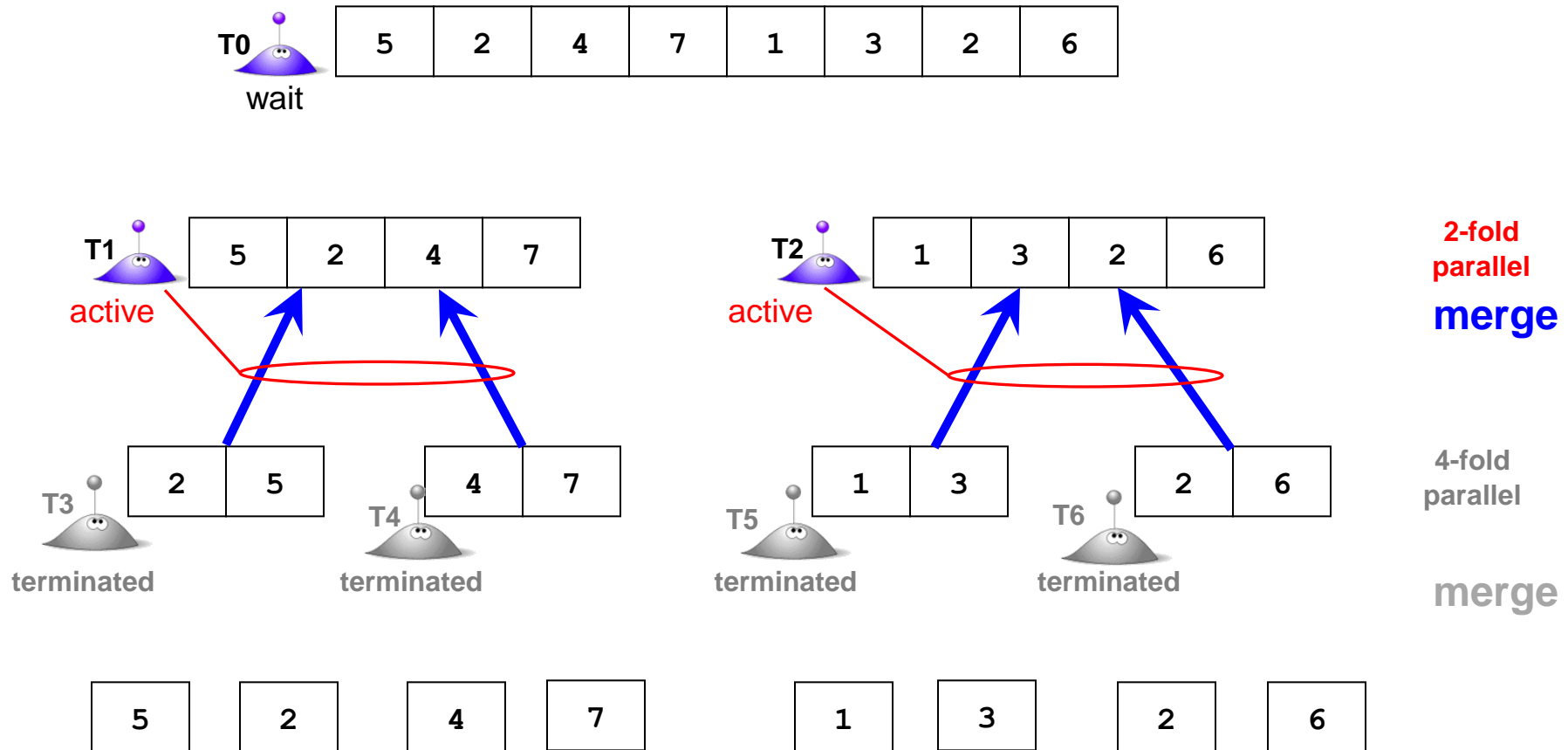


4-fold
parallel

merge

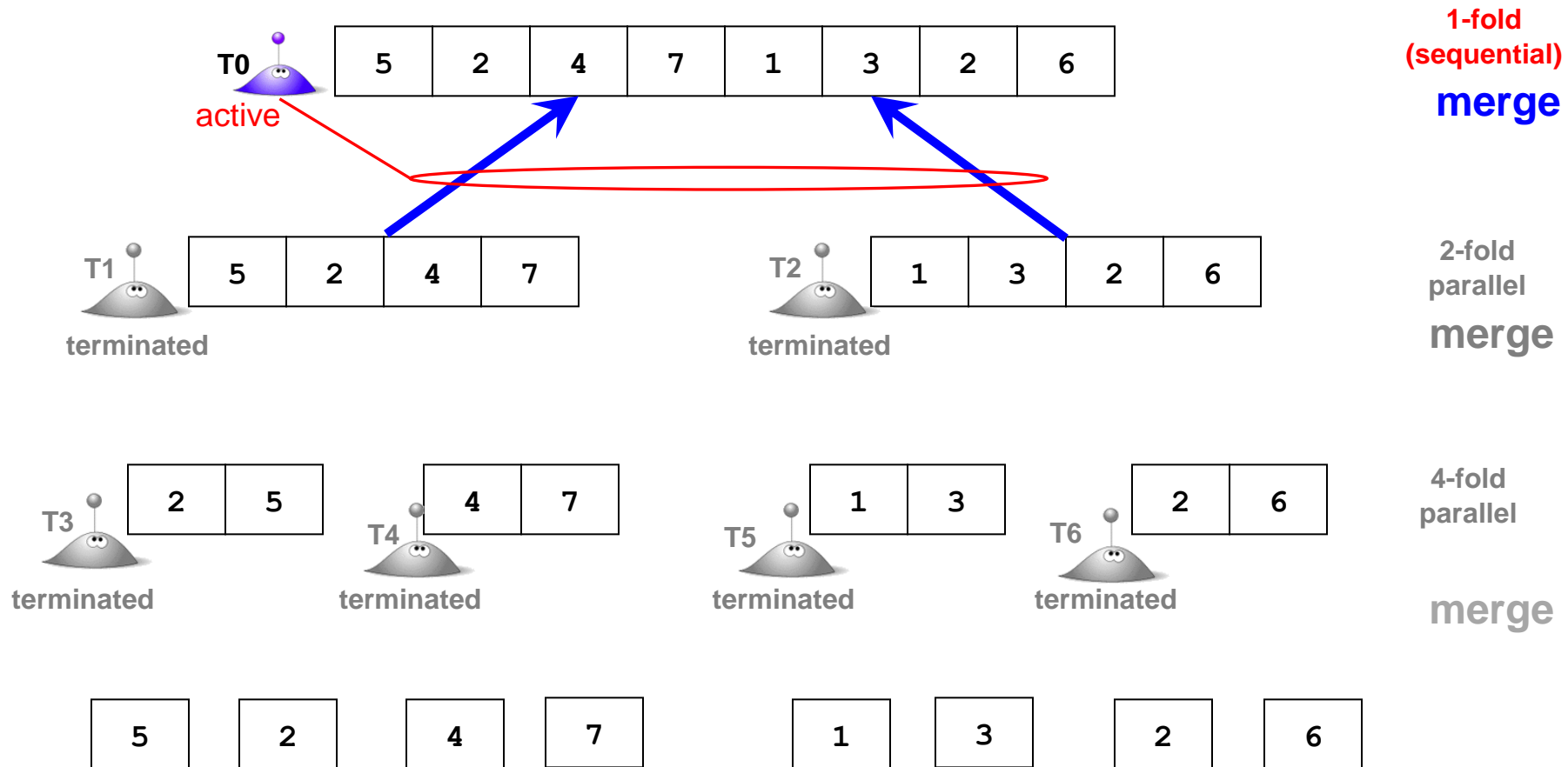
- Merging happens in parallel except on recursion depth level 1:
 - 4-fold parallelism, then 2-fold, then sequential.

Parallelization of Mergesort (cont.)



- Merging happens in parallel except on recursion depth level 1:
 - 4-fold parallelism, then 2-fold, then sequential.

Parallelization of Mergesort (cont.)



- Merging happens in parallel except on recursion depth level 1:
 - 4-fold parallelism, then 2-fold, then sequential.

Implementation


```
void ParallelMergeSort(float A[], int p, int r, int depth, int max_depth) {
    if (depth==max_depth) { // cut-off on specific level, differs from Sl#45!
        // Max depth reached, revert to sequential version:
        MergeSort(A, p, r);
    } else {
        // 1) Spawn 2 threads for left and right sub-array
        // 2) Join the 2 threads
        // 3) Perform the Merge
        int q;
        if (p < r) {
            q = (p+r)/2; struct arg LeftArg, RightArg;
            LeftArg.A = A; LeftArg.p = p; LeftArg.r = q;
            LeftArg.depth = depth+1;
            LeftArg.max_depth = max_depth;
            RightArg.A = A; RightArg.p = q+1; RightArg.r = r;
            RightArg.depth = depth+1;
            RightArg.max_depth = max_depth;
            pthread_create(&LThread, NULL, PMSort, (void *)LeftArg);
            pthread_create(&RThread, NULL, PMSort, (void *)RightArg);
            pthread_join(LThread, NULL);
            pthread_join(RThread, NULL);
            Merge(A, p, q, r);
        }
    }
}
```

```
struct arg {
    float * A;
    int p;
    int r;
    int depth;
    int max_depth;
};
```

Both sequential and parallel
Mergesort implementations are c_

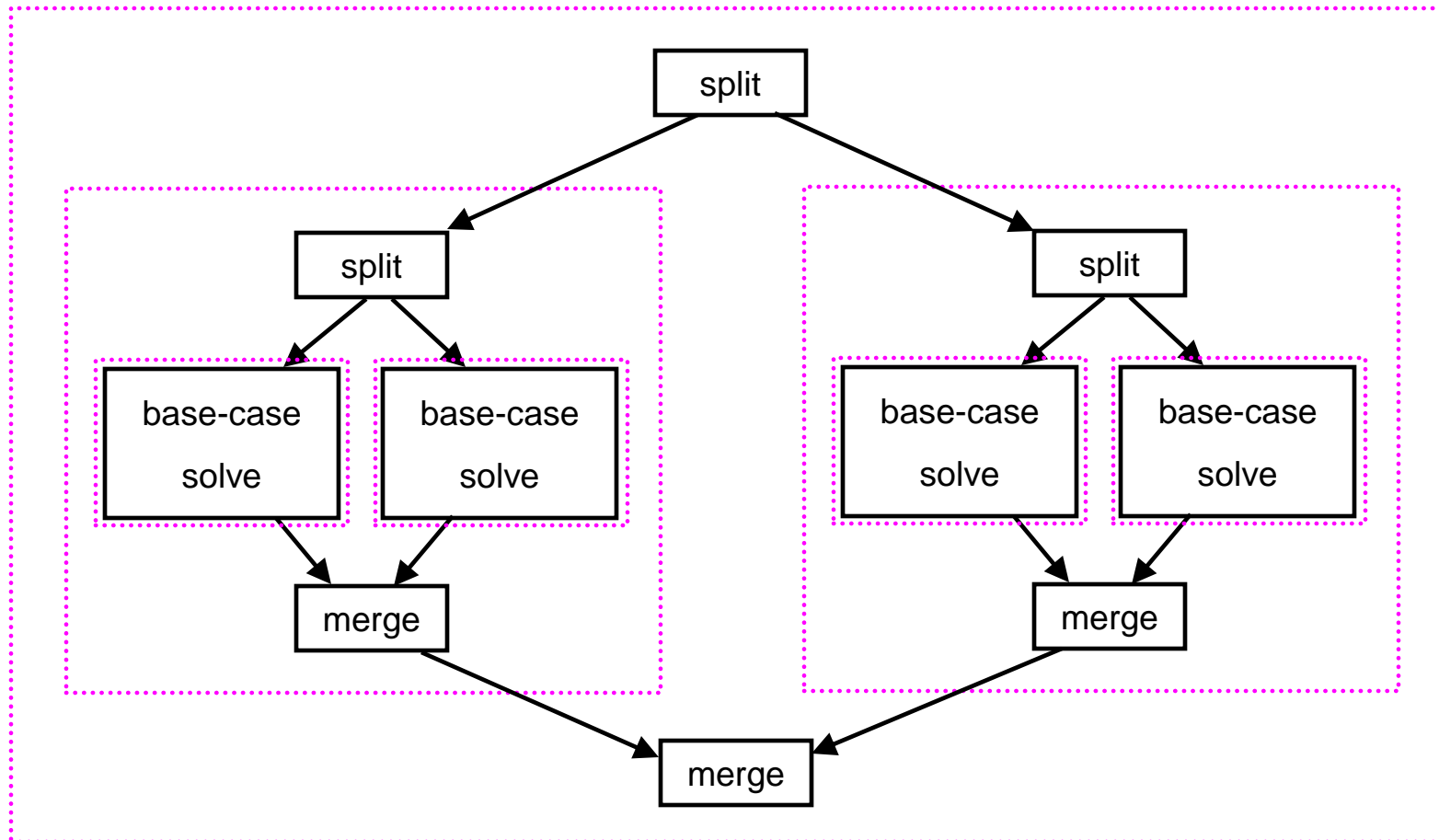
Implementation

```
void ParallelMergeSort(float A[], int p, int r, int depth, int max_depth) {  
    if (depth==max_depth) {  
        // Max depth reached, revert to sequential version:  
        MergeSort(A, p, r);  
    } else {  
        // 1) void * PMSort (void * ptr) {  
        // 2)     struct arg * MyArg = (struct arg *) ptr;  
        // 3)     ParallelMergeSort(MyArg->A, MyArg->p,  
        int q,  
        MyArg->r, MyArg->depth, MyArg->max_depth);  
        if (p < r) {  
            q = (p+r)/2;  
            LeftArg->A = A; LeftArg->p = p; LeftArg->r = q;  
            LeftArg->depth = depth+1;  
            LeftArg->max_depth = max_depth;  
            RightArg->A = A; RightArg->p = q+1; RightArg->r = r;  
            RightArg->depth = depth+1;  
            RightArg->max_depth = max_depth;  
            pthread_create(&LThread, NULL, PMSort, (void *)LeftArg);  
            pthread_create(&RThread, NULL, PMSort, (void *)RightArg);  
            pthread_join(LThread, NULL);  
            pthread_join(RThread, NULL);  
            Merge(A, p, q, r);  
        }  
    }  
}
```



Note: the PMSort thread functions
is just a wrapper for function ParallelMergeSort,
due to the Pthread void * arg limitation.

Divide & Conquer Parallelization



- Each dashed-line box represents a thread.
 - Example: Merge-sort