Week 11: Scalable Algorithmic Templates

Outline

- Recursion
- Divide and Conquer
  - How to parallelize ‘Divide and Conquer’ algorithms
  - Example: Mergesort
- Reductions

Recursion (informally)

''To understand recursion, you need to understand recursion first.''

Definition of the acronym 'GNU':

GNU is not Unix

This is a recursive acronym (geeky!).

From Merriam-Webster:

> ***Recursion:*** *a computer programming technique involving the use of a procedure, subroutine, function, or algorithm that calls itself one or more times until a specified condition is met at which time the rest of each repetition is processed from the last one called to the first — compare iteration.*

What is Recursion?

- **Recursion** occurs when a procedure calls itself to solve a simpler version of the problem. With each recursive call, the problem is different from, and simpler than, the original problem.

- At some stage, the problem is so simple that we can solve it---the recursion terminates.
  - this is the **base-case** of the recursion

- Previous problem instance solved in terms of the simple problem(s).

- And so on, back to the initial procedure call…

Recursion Example – Solving Factorials

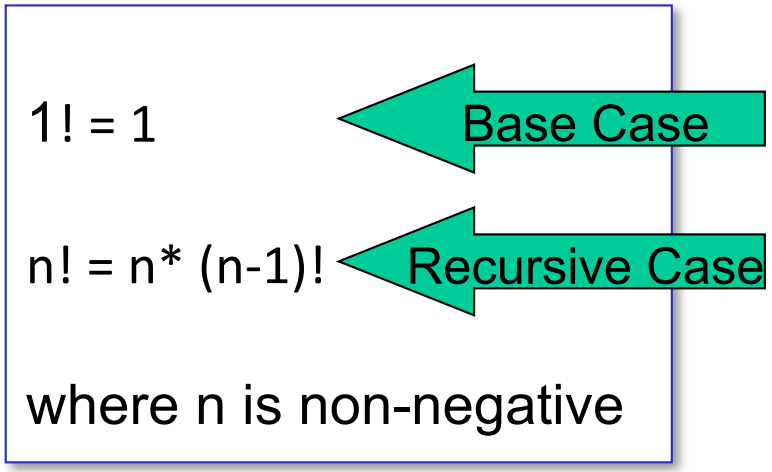
- The problem of solving factorials is our first example of recursion.
- The factorial operation (!) in mathematics is illustrated below.

1! = 1

*2! = 2 * 1* or *2 * 1!*

*3! = 3 * 2 * 1* or *3 * 2!*

4! = 4 * 3 * 2 *1 or 4 * 3!



Notice that each successive line can be solved in terms of the previous line. For example, 4! is equivalent to the problem

4 * 3!

Using A Recursive Method

- A recursive procedure to solve the factorial problem is given below. Notice the recursive call in the last line.
- The procedure calls itself to solve a smaller version of the factorial problem.

```
int fact(int n) {
   //  returns the value of n!
   //  precondition:  n >= 1
   if (n == 1)
      return 1;
   else
      return  n * fact(n - 1);
}
```

- The **base case** is a fundamental situation for which we know the answer.
- In the case of finding factorials, the answer of 1! is by definition 1. No further work is needed for the base-case.

Recursion – Step By Step

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

Suppose we call the method to solve fact(4).

- This will result in **four** calls of method fact:

fact(4): This is not the base case (n==1) so we return the result of 4 * fact(3). This multiplication will not be carried out until an answer is found for fact(3). This leads to the second call of fact to solve fact(3).

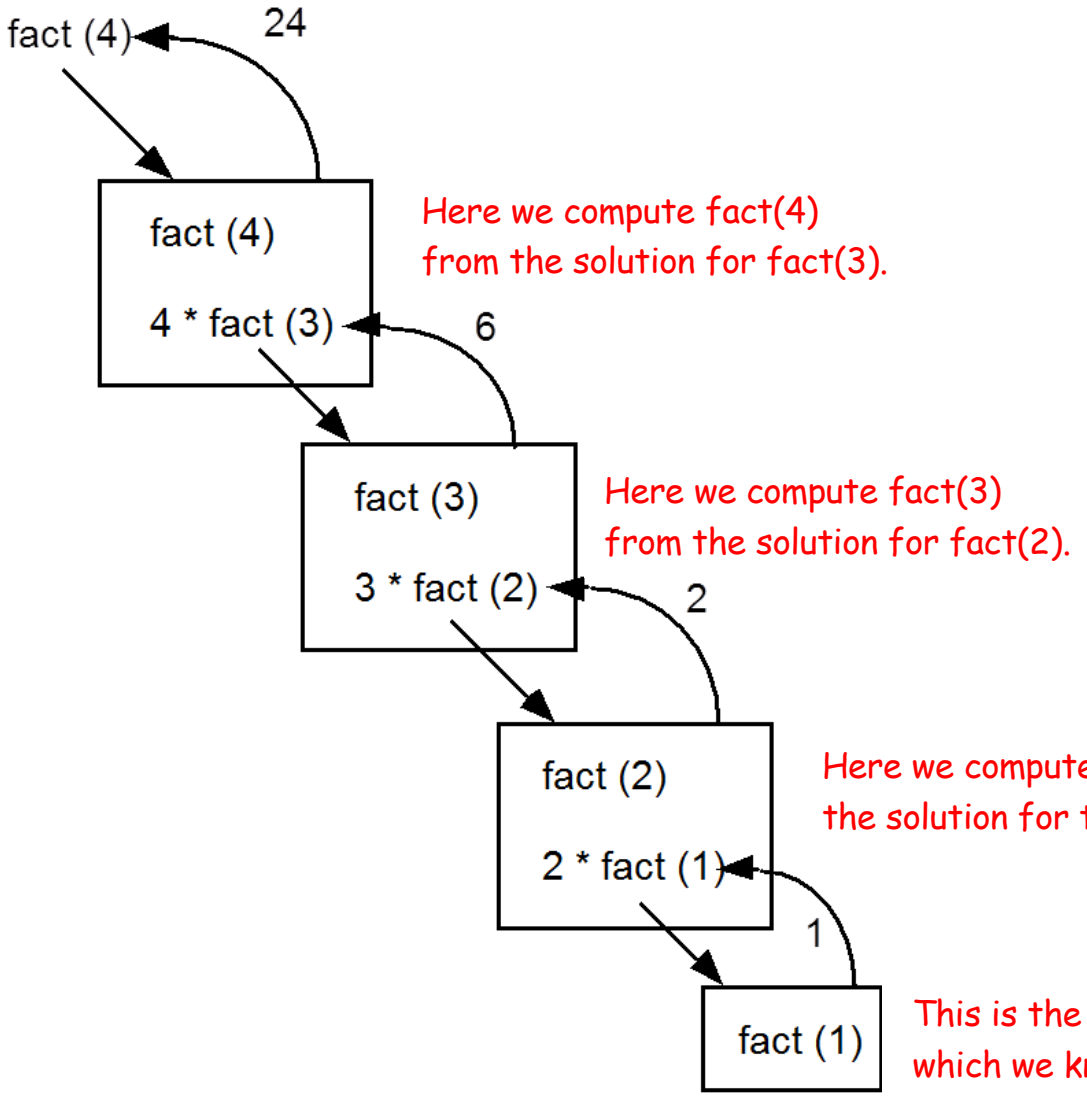
fact(3): Again, this is not the base case and we return

3 * fact (2). This leads to another recursive call to solve fact(2).

fact(2): Still, this is not the base case, we solve 2 * fact(1).

fact(1): Finally we reach the base case, which returns the value 1.

Recursion – The Big Picture



- Each box represents a call of procedure fact(). To solve fact(4) requires four calls of procedure fact.
- Notice that when the recursive calls were made inside the **else** statement, the value fed to the recursive call was (n-1). This is where the problem is getting smaller and simpler with the eventual goal of solving 1!.

Recursion – Step By Step

- When a recursive call is made, the current computation is temporarily suspended with all its current information available for later use.
  - Same as for any other function call.
- A completely new copy of the procedure's data is used to evaluate the recursive call.
  - Fresh set of local variables,
  - Own set of procedure arguments
- When the recursive call is completed, the value returned by the recursive call is used to complete the suspended computation. The suspended computation's work now proceeds.
- When the base case is encountered, the recursion will terminate by returning a value to the caller. The caller will then return a value to the caller's caller aso until the final answer becomes available in the initial recursive call.

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.

Outline

- Recursion ✓
- Divide and Conquer
  - How to parallelize ‘Divide and Conquer’ algorithms
  - Example: Mergesort
- Reductions

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.

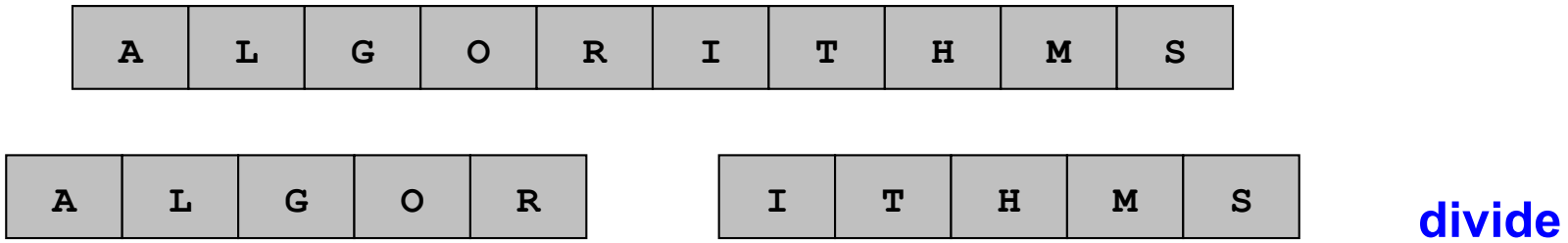
At each level of the recursion:

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

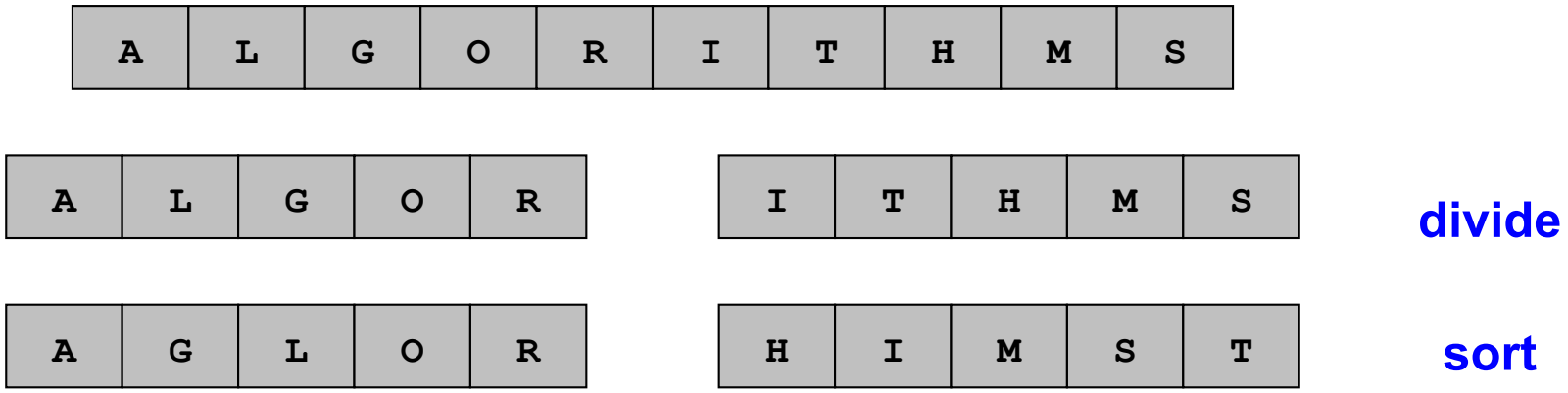
1) **Divide** array into 2 halves:

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



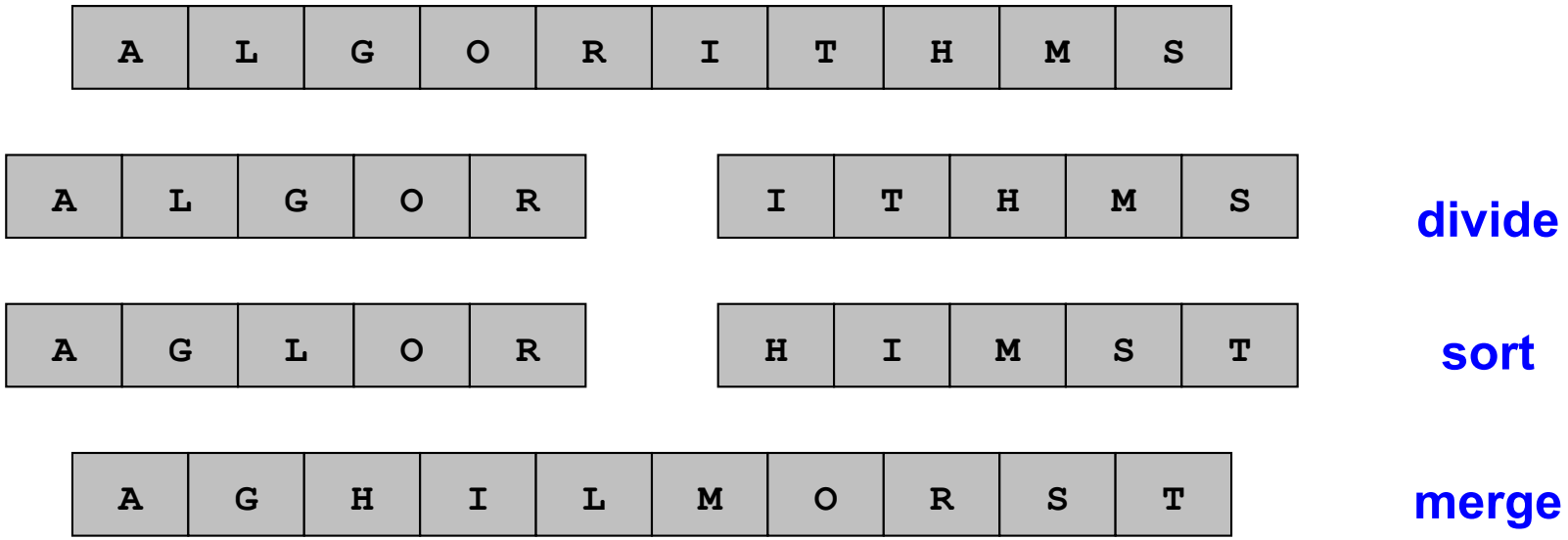
Mergesort

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



Mergesort

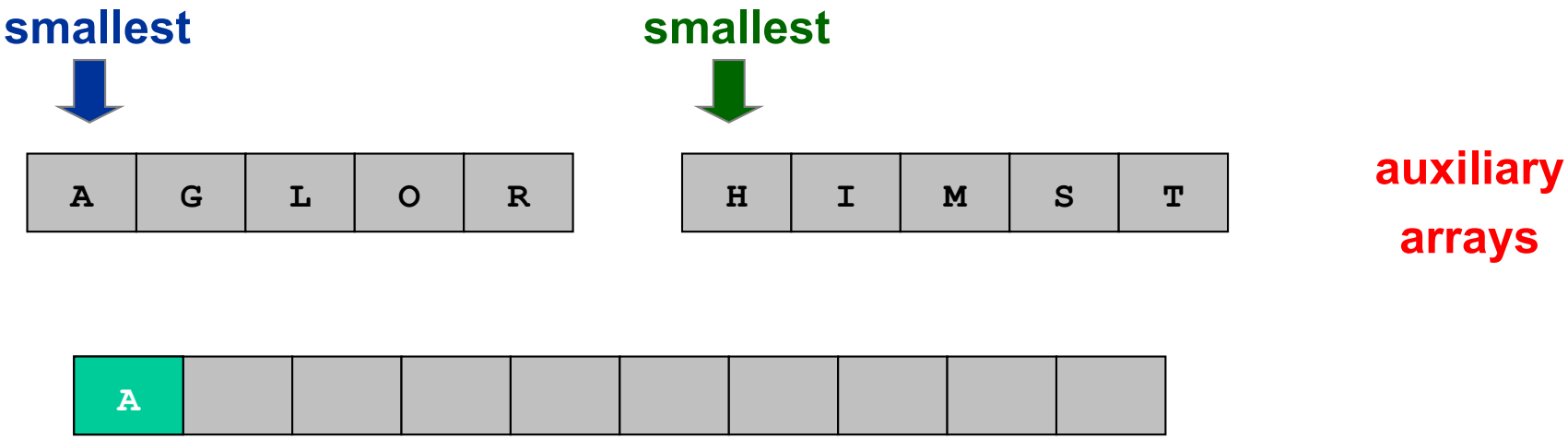
1) **Divide** array into 2 halves.
2) **Conquer:** recursively sort each half.
3) **Combine:** merge two halves to make sorted whole:



Merging

Combine:

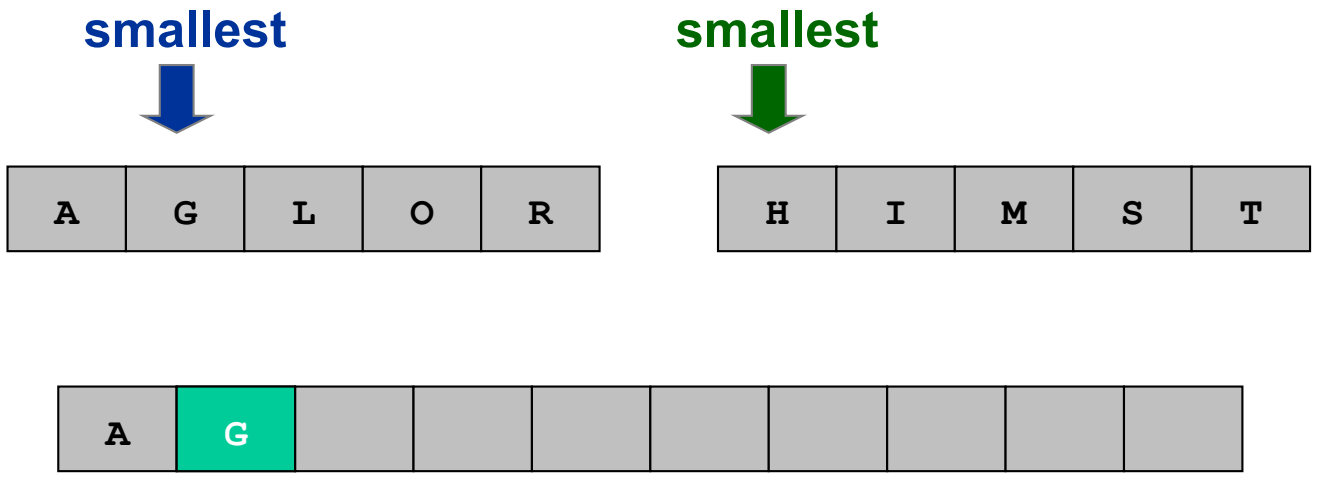
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into array.
- Repeat until done.



Merging

Combine:

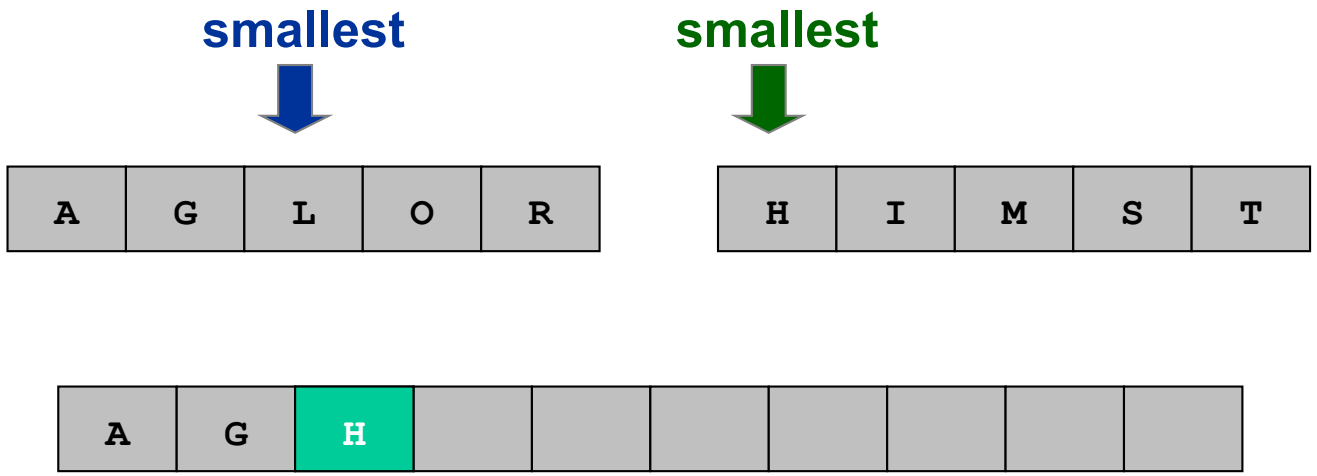
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Combine:

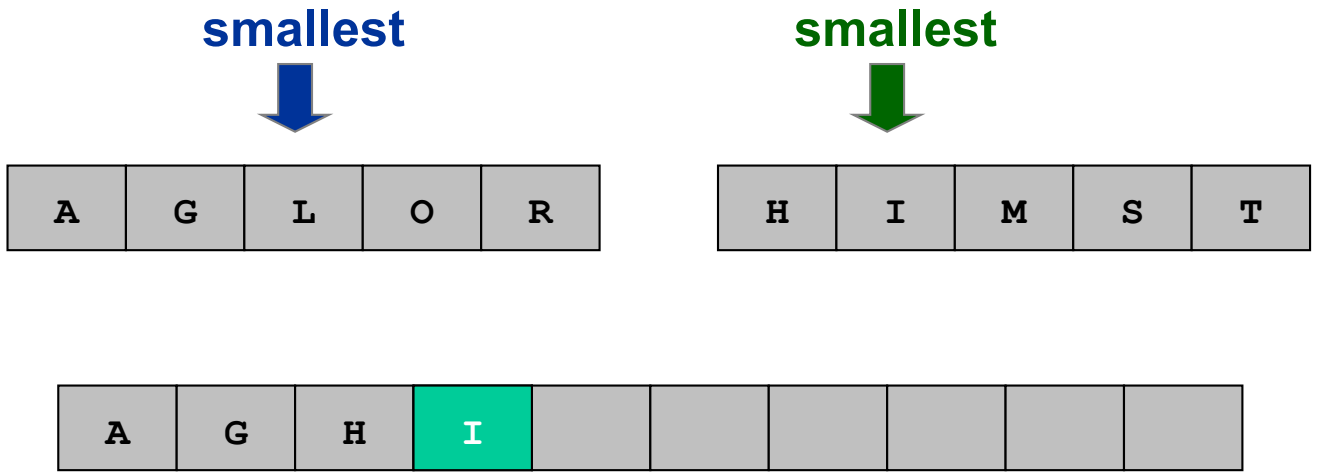
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Combine:

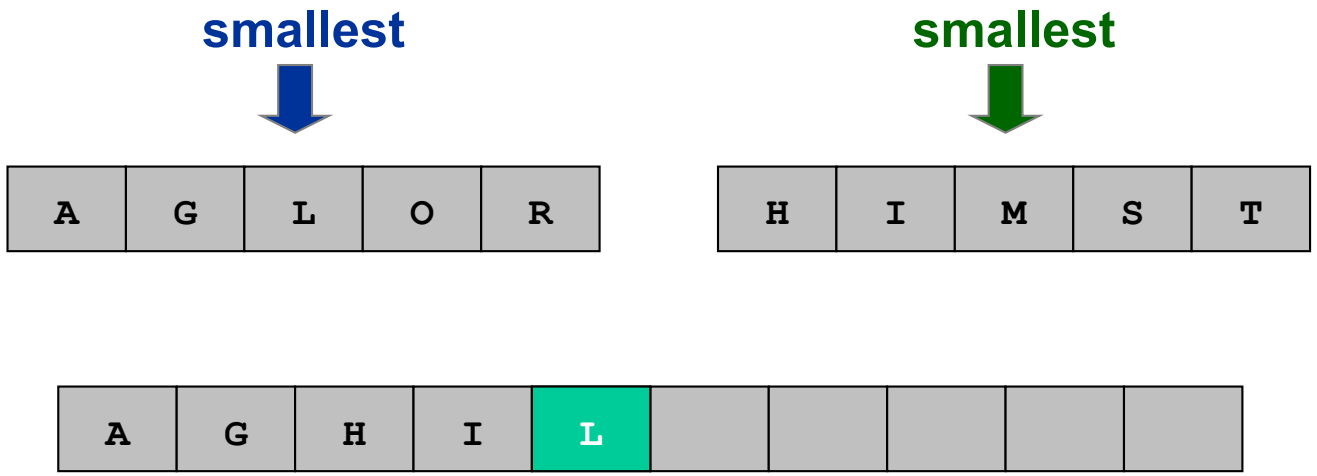
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Combine:

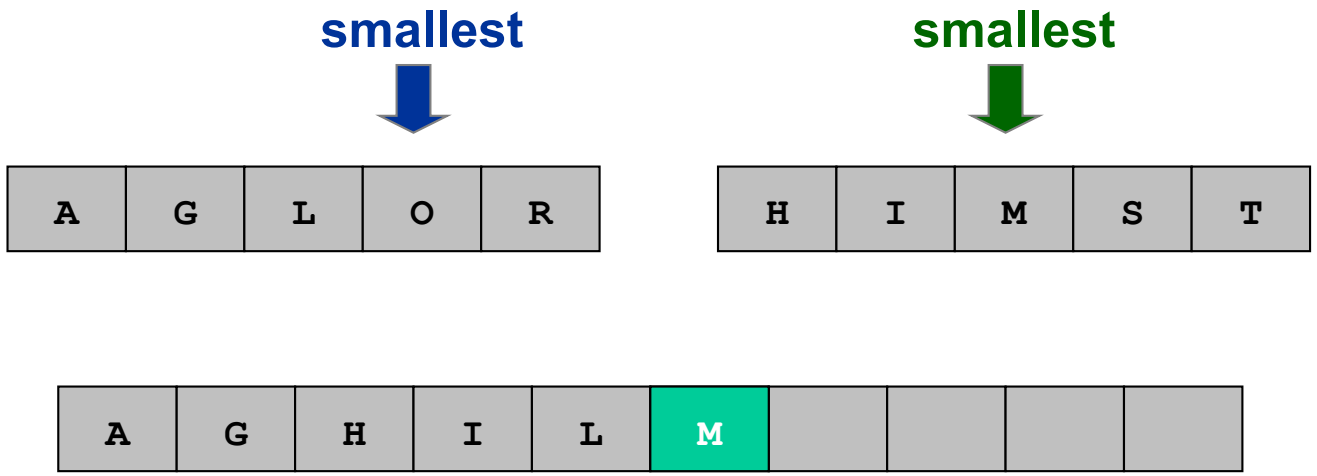
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Combine:

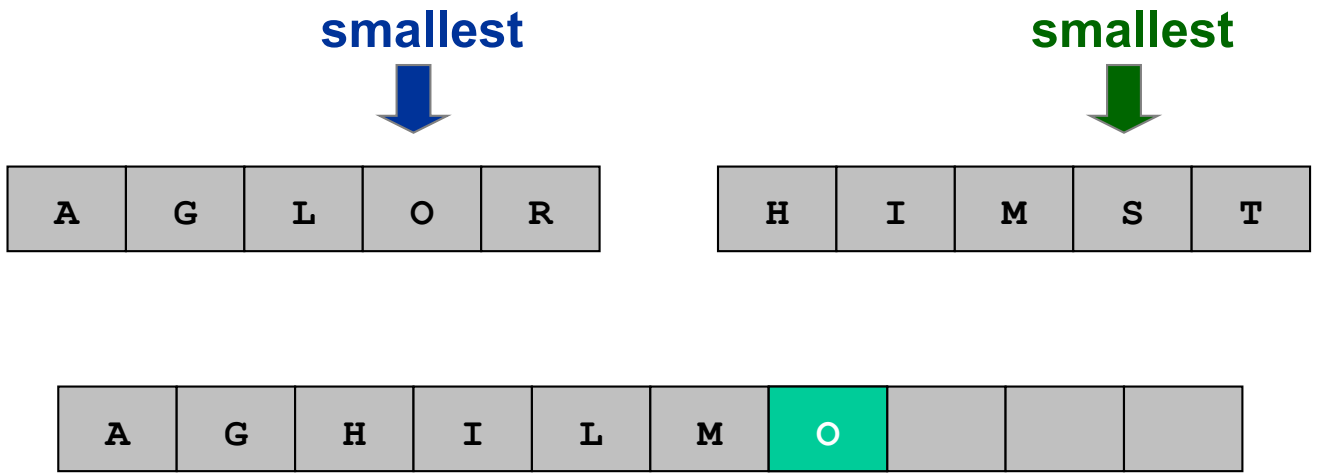
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Combine:

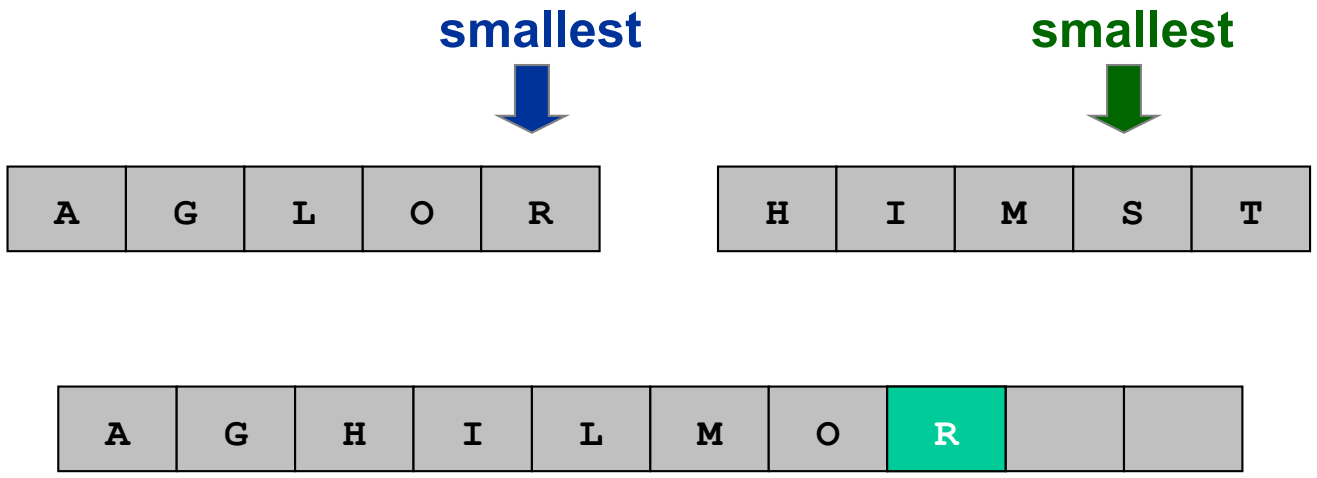
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Combine:

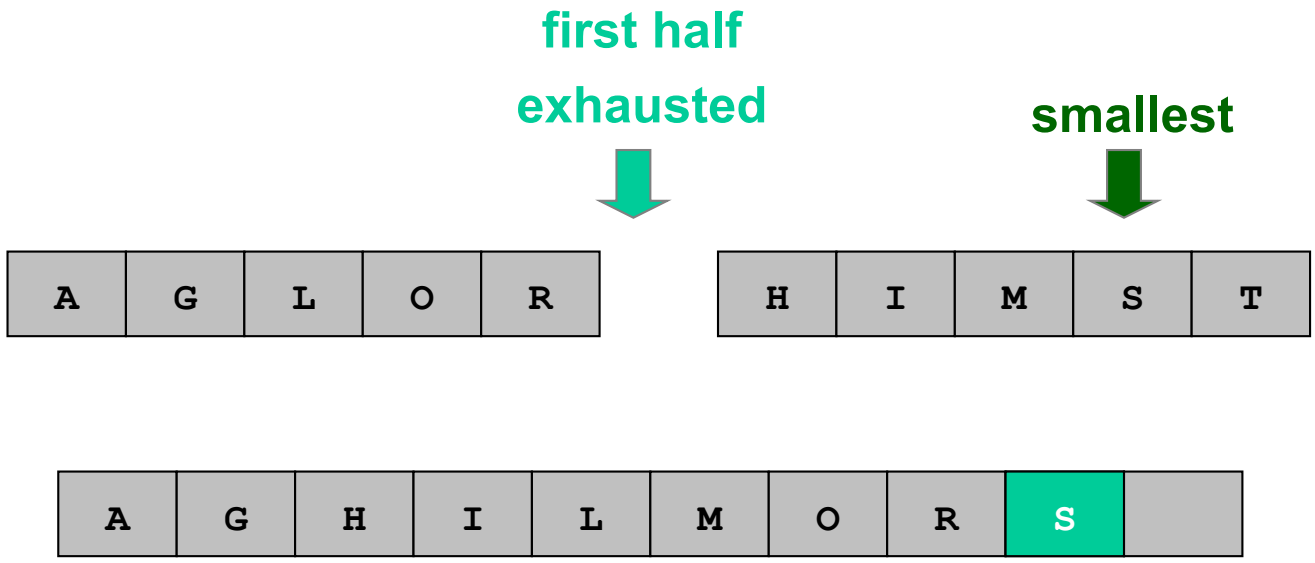
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Combine:

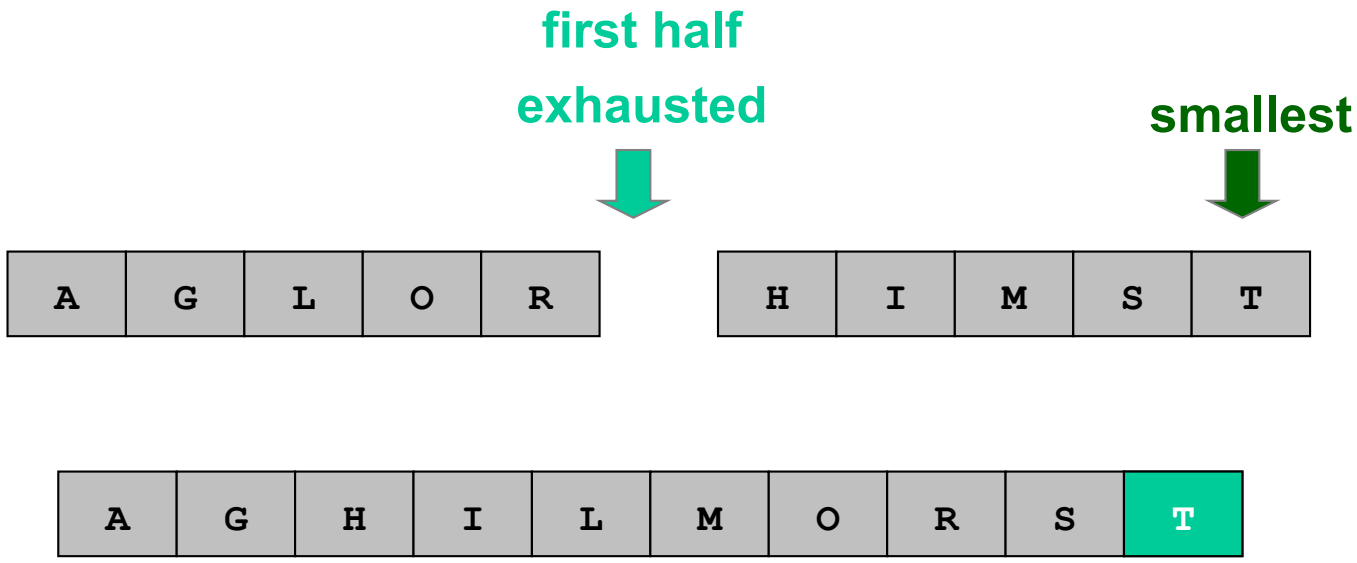
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



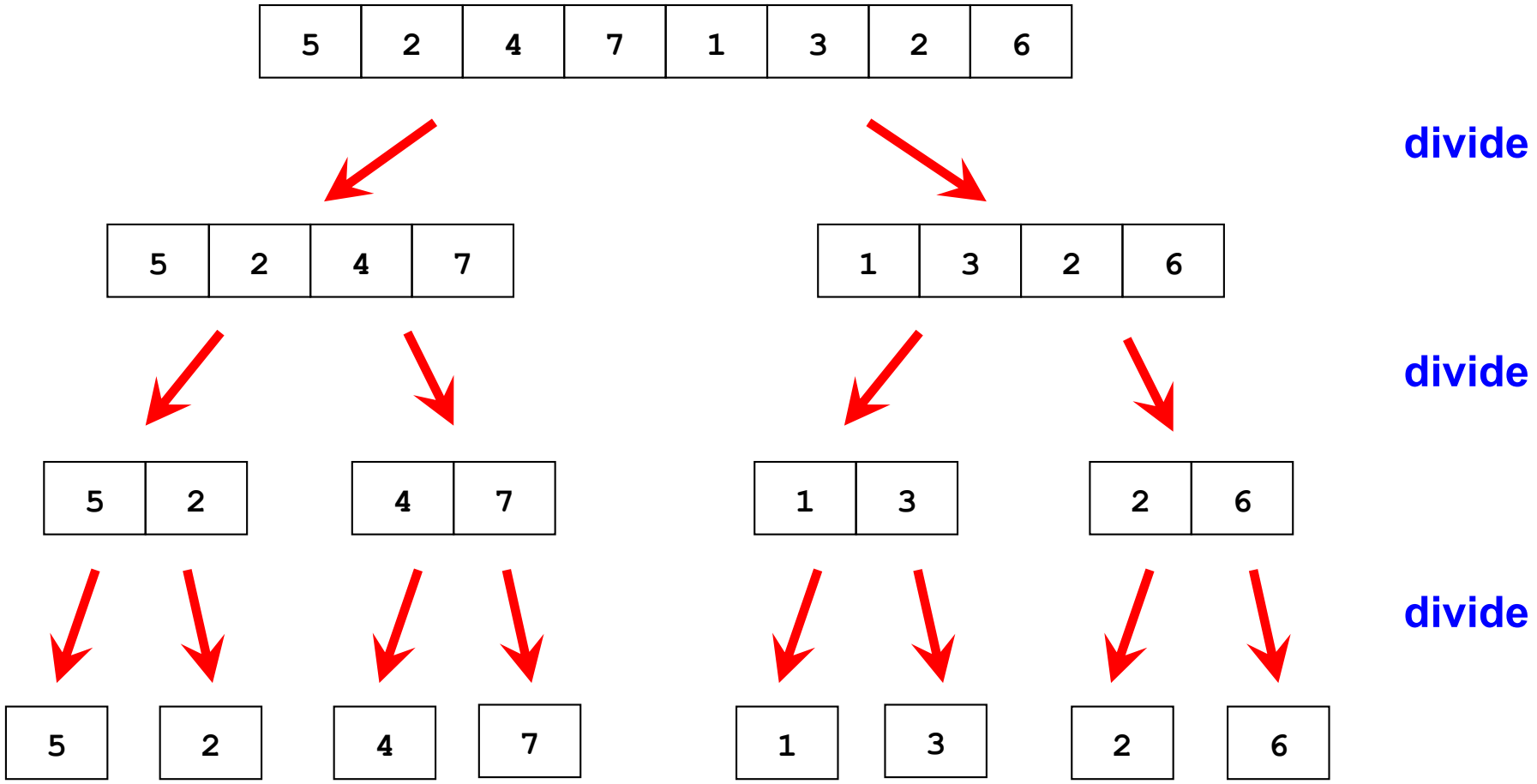
Merging

Combine:

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

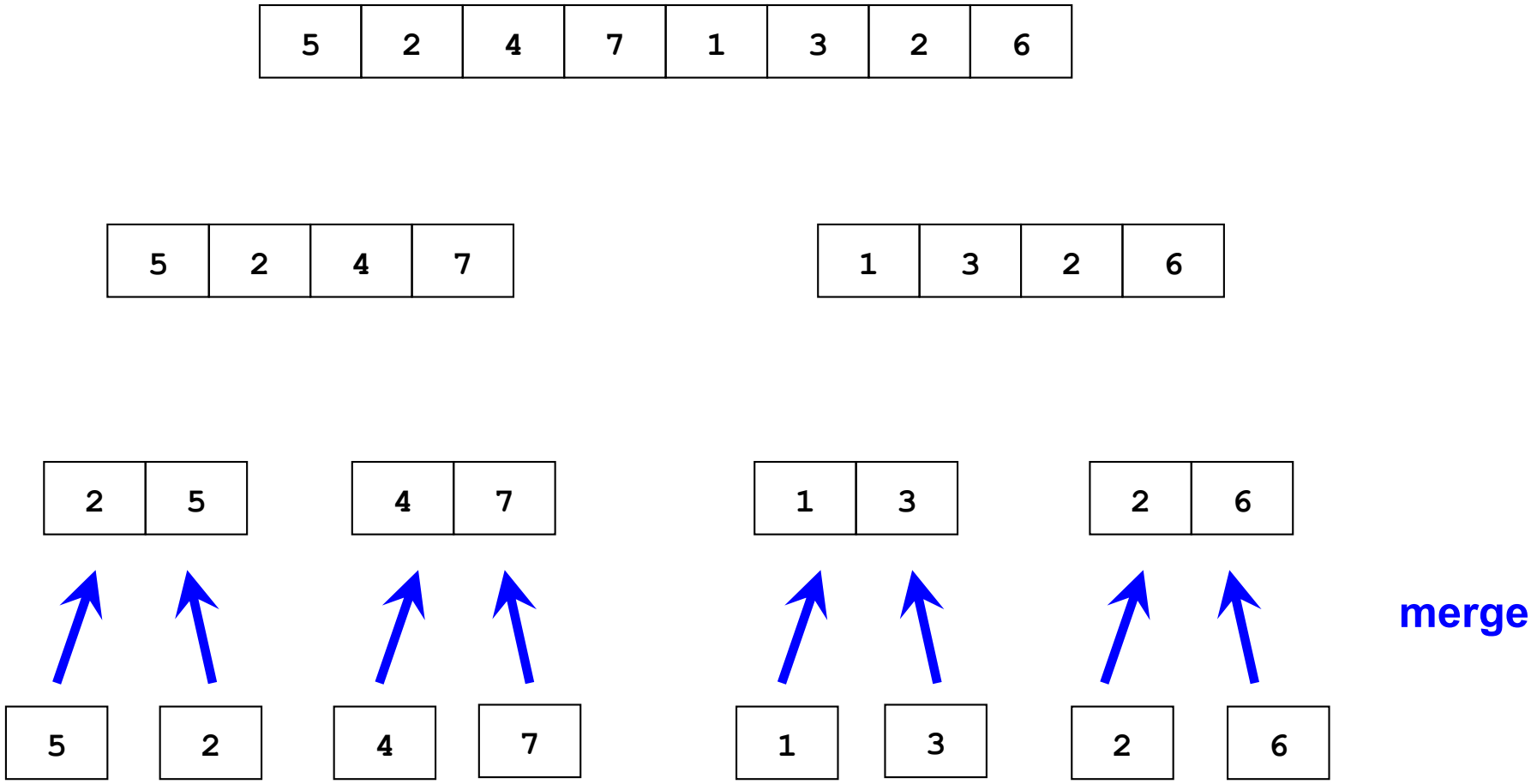


Mergesort (Divide)



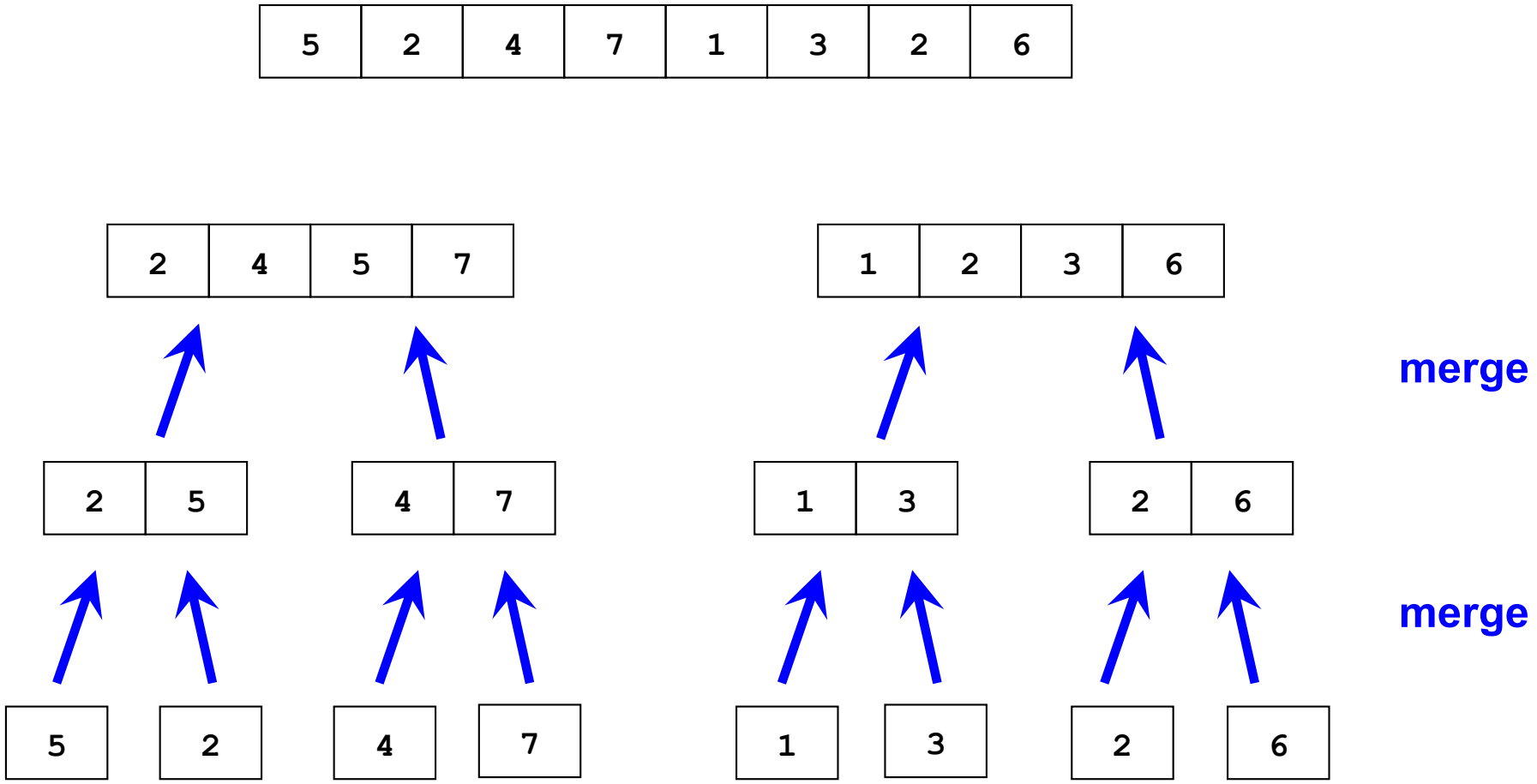
- Divide until we reach 1-element base-case.
  - One-element array is trivially sorted.

Mergesort (Combine)



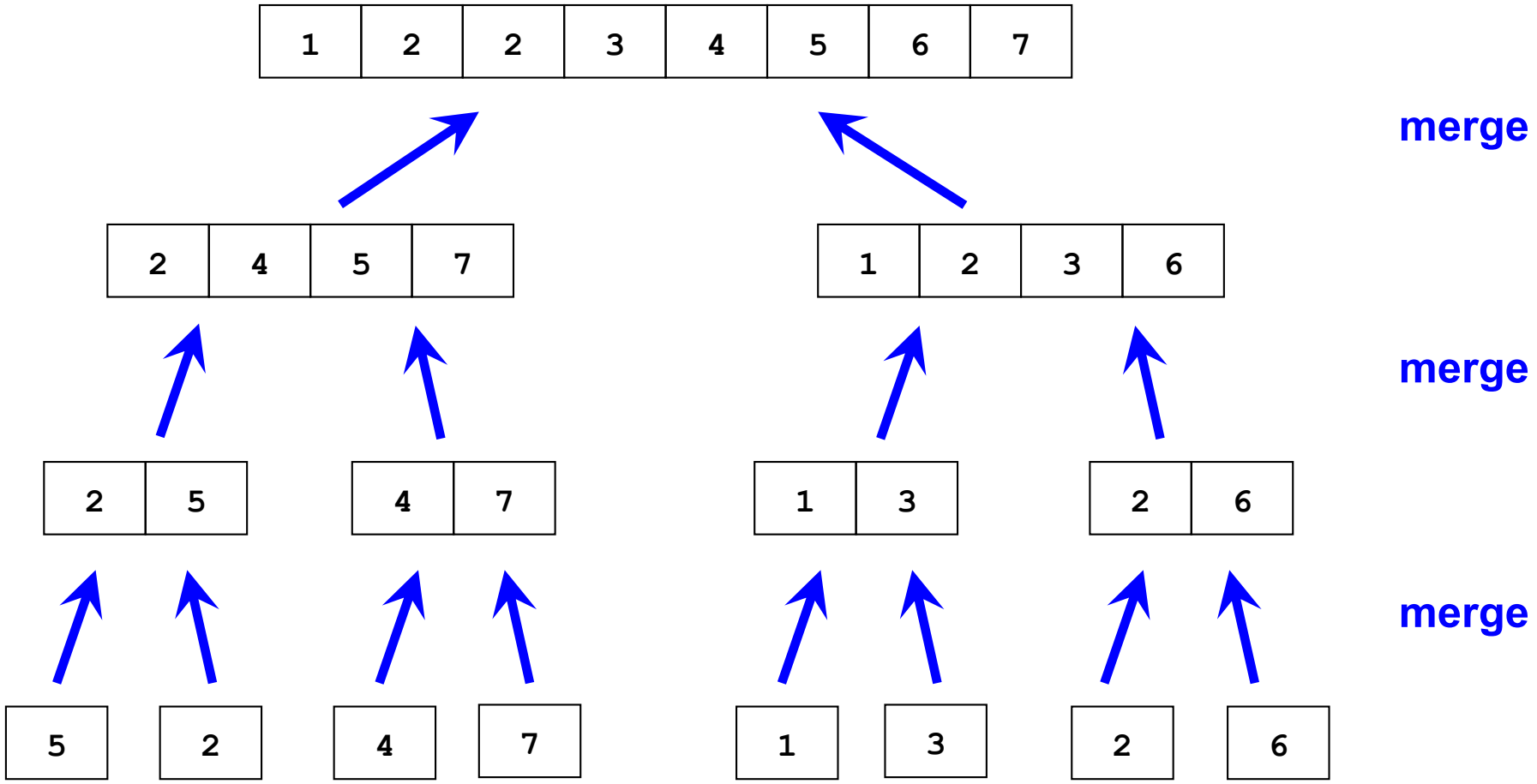
- Merge, starting from the 1-element base-cases, ...

Mergesort (Combine)



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

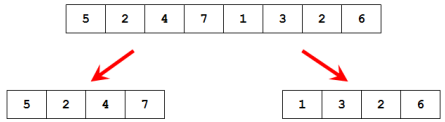
Mergesort (Combine)



- ... 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)

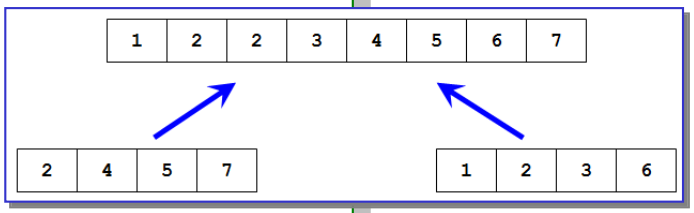
Implementing Mergesort (cont.)

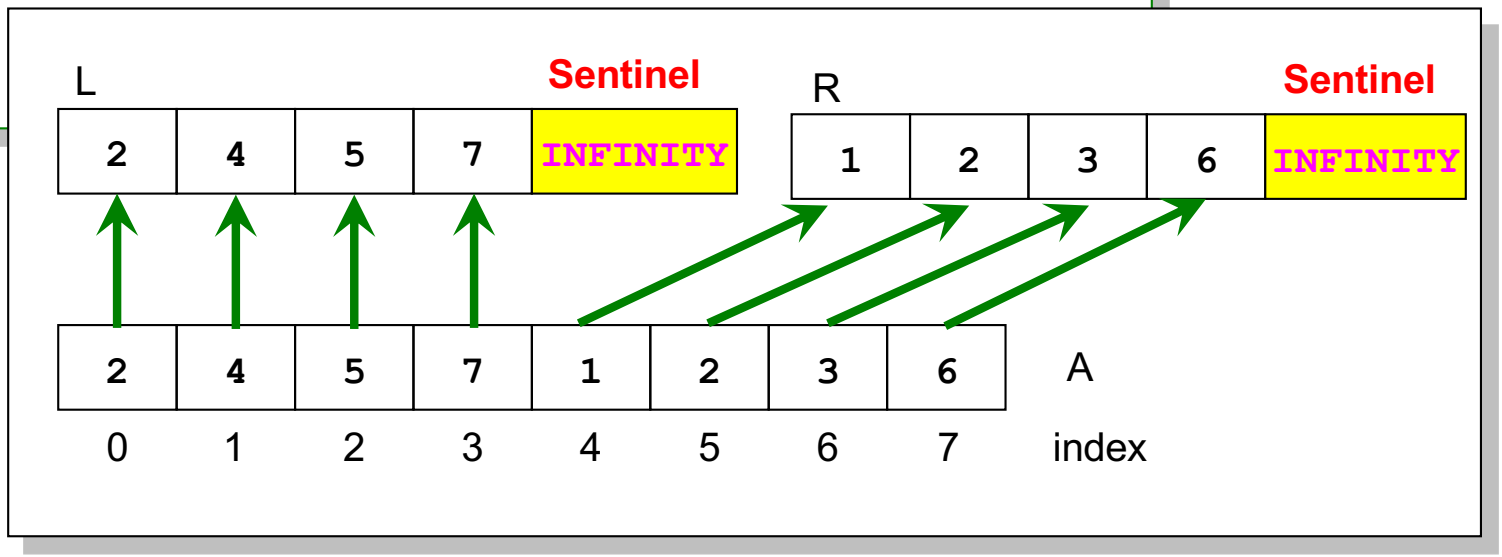
```
void MergeSort(float A[], int p, int r)
{  int q;
   if (p < r) {
      q = (p+r)/2;
      MergeSort(A, p, q);
      MergeSort(A, q+1, r);
      Merge(A, p, q, r);
   }
}
```

- 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 halfs (**p..q**) and (**q+1..r**) and then **merge** sorted halfs.
- 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)

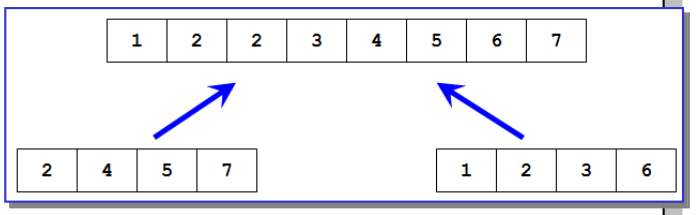
```
void Merge(float A[], int p, int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    for (i=0; i<n1; i++) {
        L[i] = A[p+i];
    }
    for (j=0; j<n2; j++) {
        R[j] = A[q+j+1];
    }
    L[n1] = INFINITY; R[n2] = INFINITY;
    ...
}
```

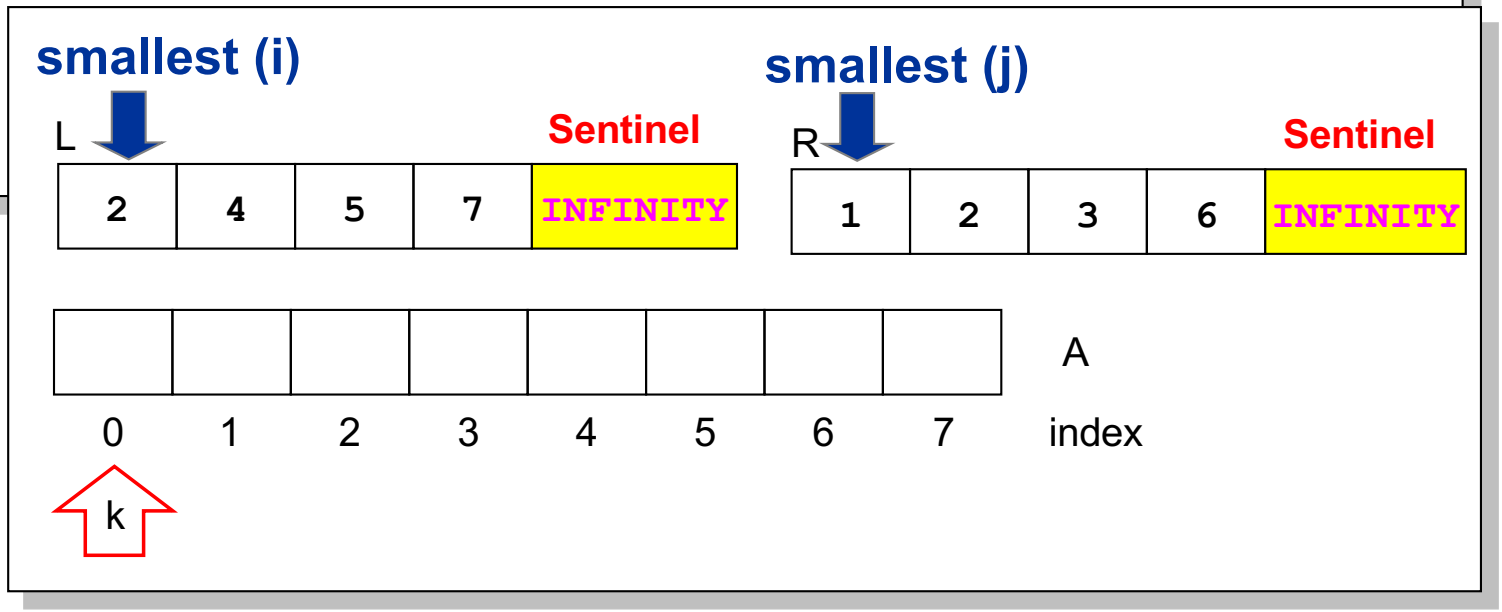




Implementing the Merge Step of Mergesort (p. 2)

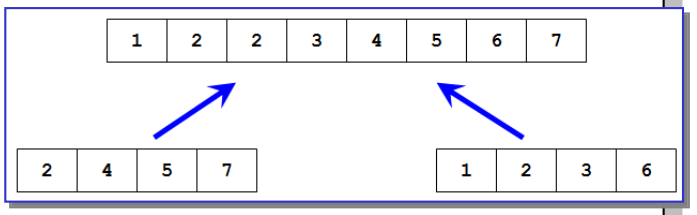
```
void Merge(float A[], int p, int q, int r) {
    ...
    int i = 0; int j = 0;
    for (int k = p; k<=r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}
```

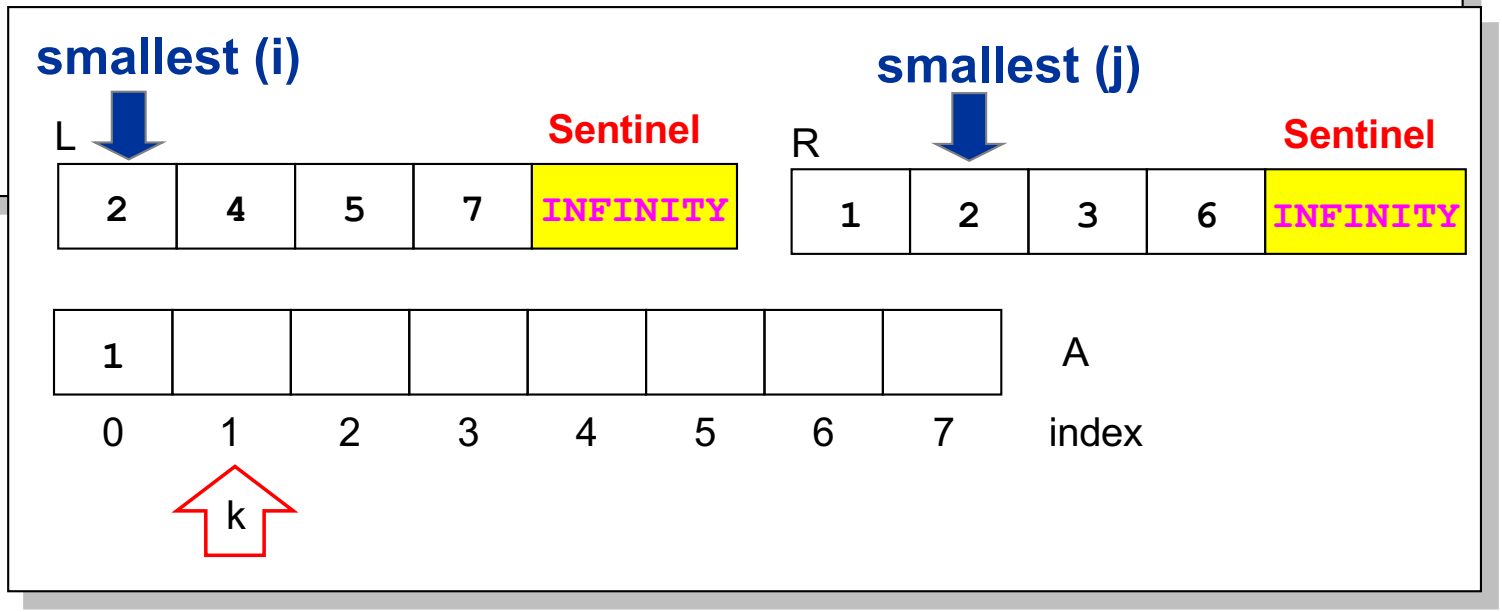




Implementing the Merge Step of Mergesort (p. 2)

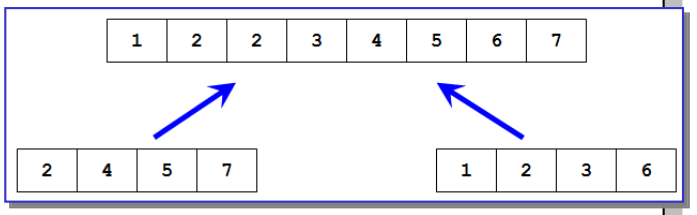
```
void Merge(float A[], int p, int q, int r) {
    ...
    int i = 0; int j = 0;
    for (int k = p; k<=r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}
```

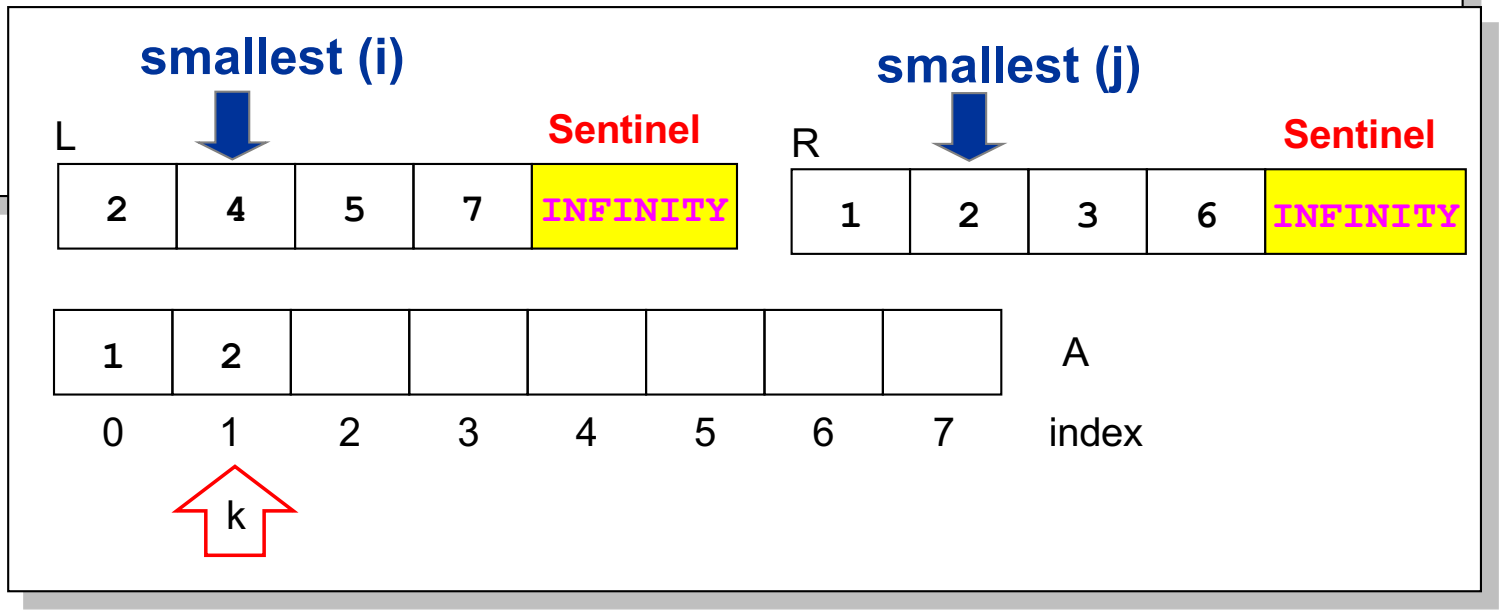




Implementing the Merge Step of Mergesort (p. 2)

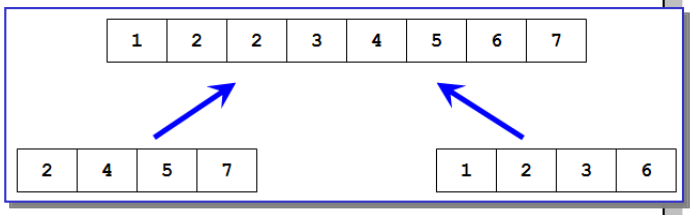
```
void Merge(float A[], int p, int q, int r) {
    ...
    int i = 0; int j = 0;
    for (int k = p; k<=r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}
```

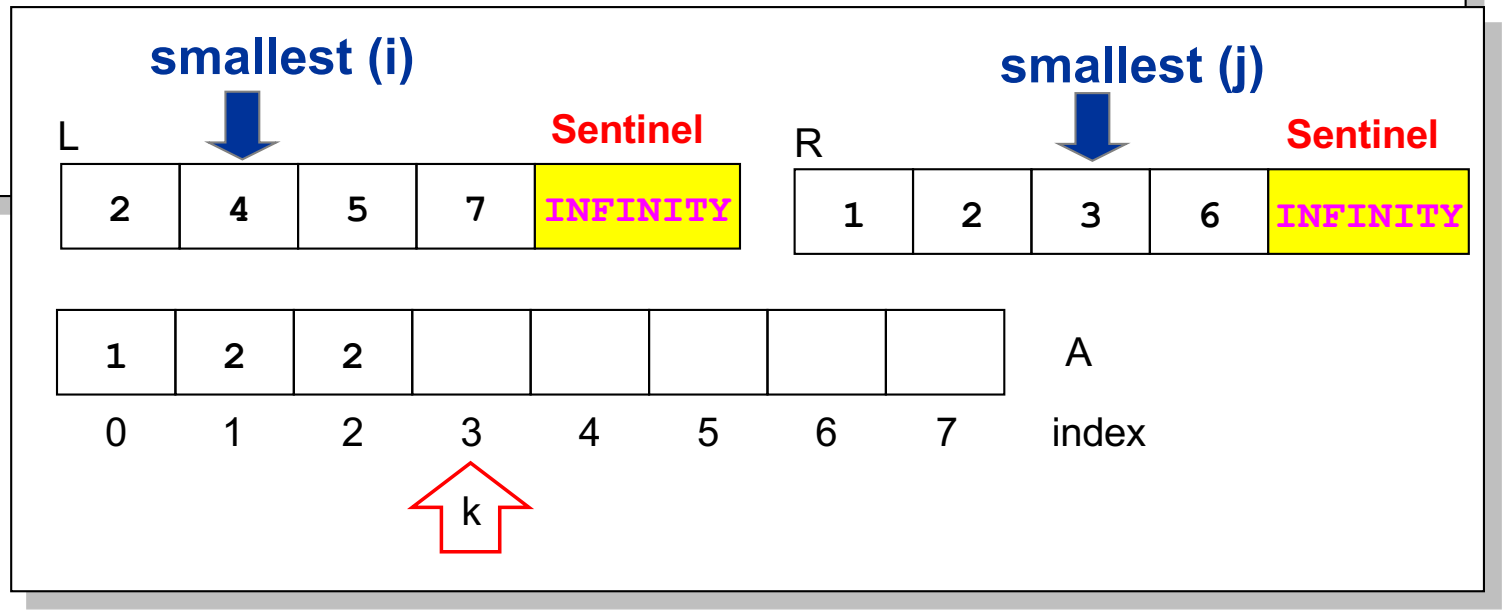




Implementing the Merge Step of Mergesort (p. 2)

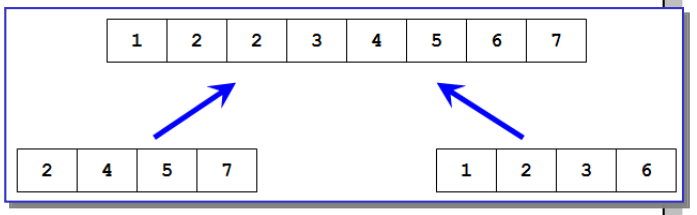
```
void Merge(float A[], int p, int q, int r) {
    ...
    int i = 0; int j = 0;
    for (int k = p; k<=r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}
```

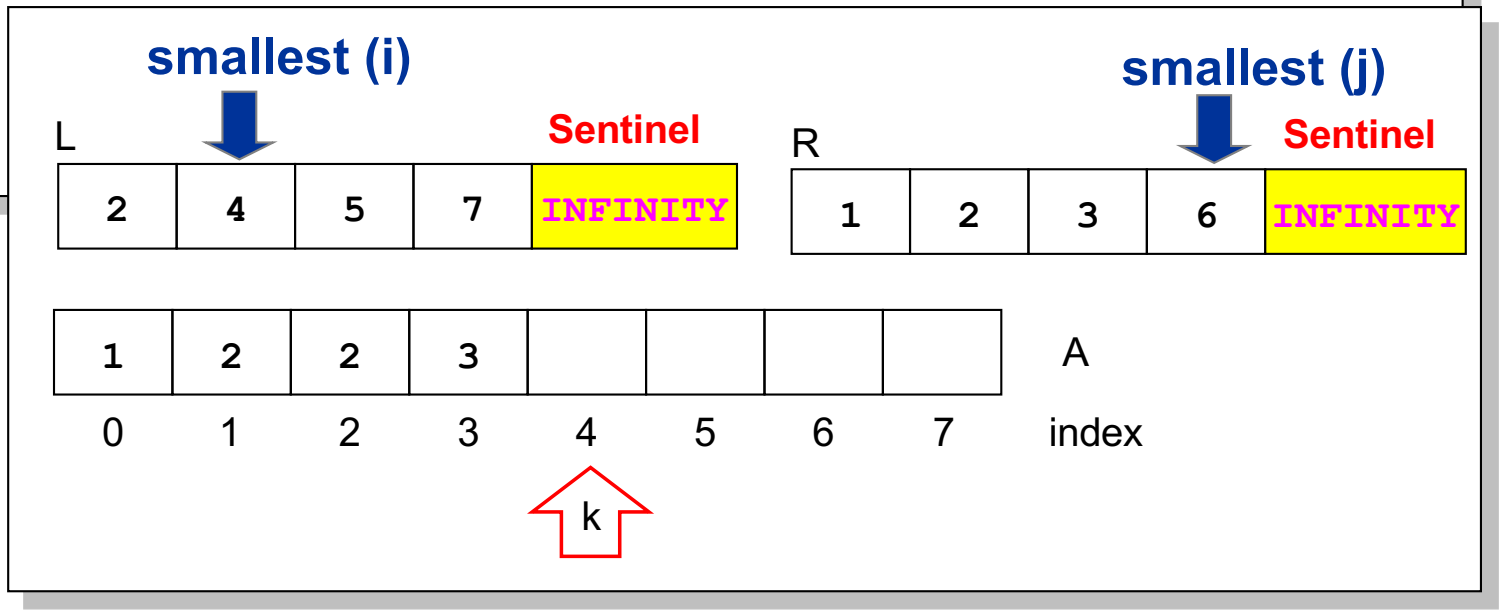




Implementing the Merge Step of Mergesort (p. 2)

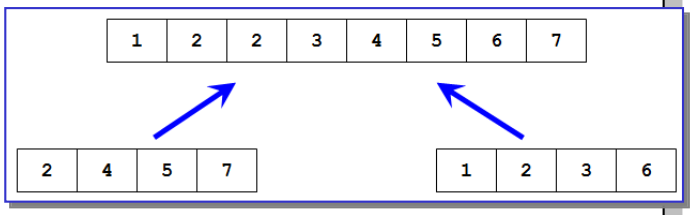
```
void Merge(float A[], int p, int q, int r) {
    ...
    int i = 0; int j = 0;
    for (int k = p; k<=r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}
```

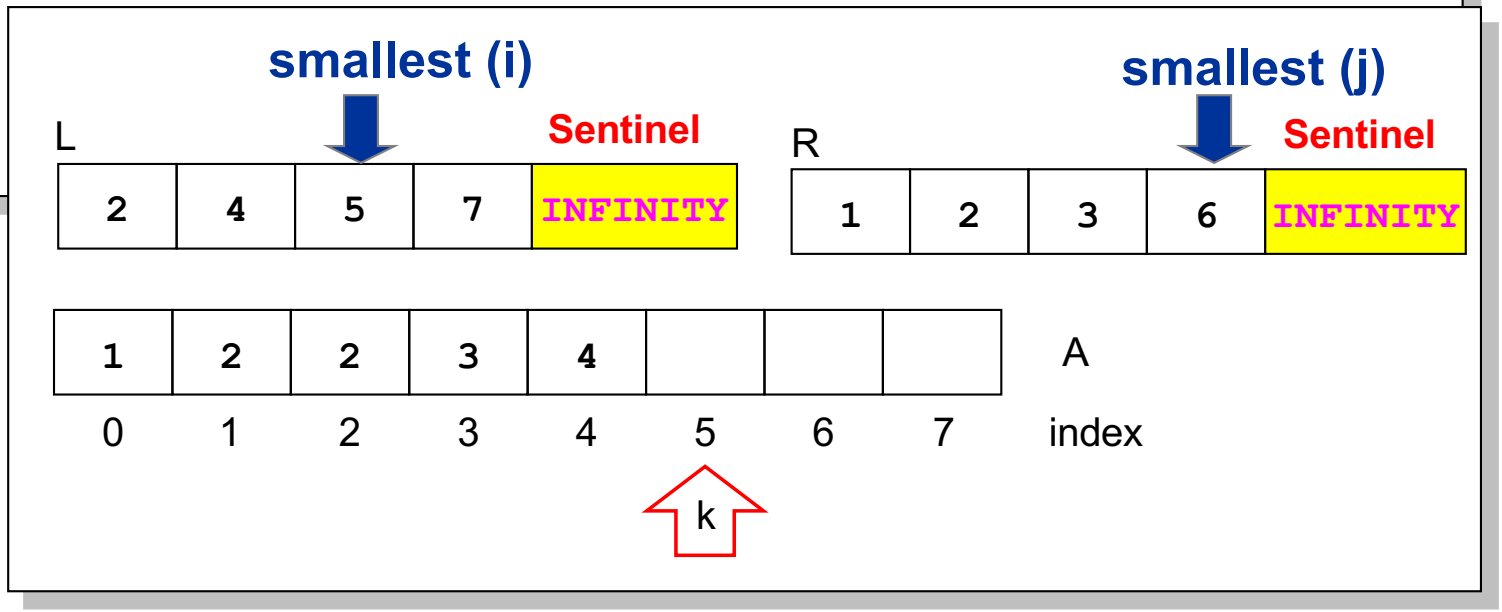




Implementing the Merge Step of Mergesort (p. 2)

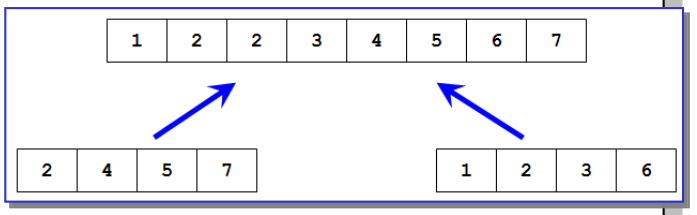
```
void Merge(float A[], int p, int q, int r) {
    ...
    int i = 0; int j = 0;
    for (int k = p; k<=r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}
```

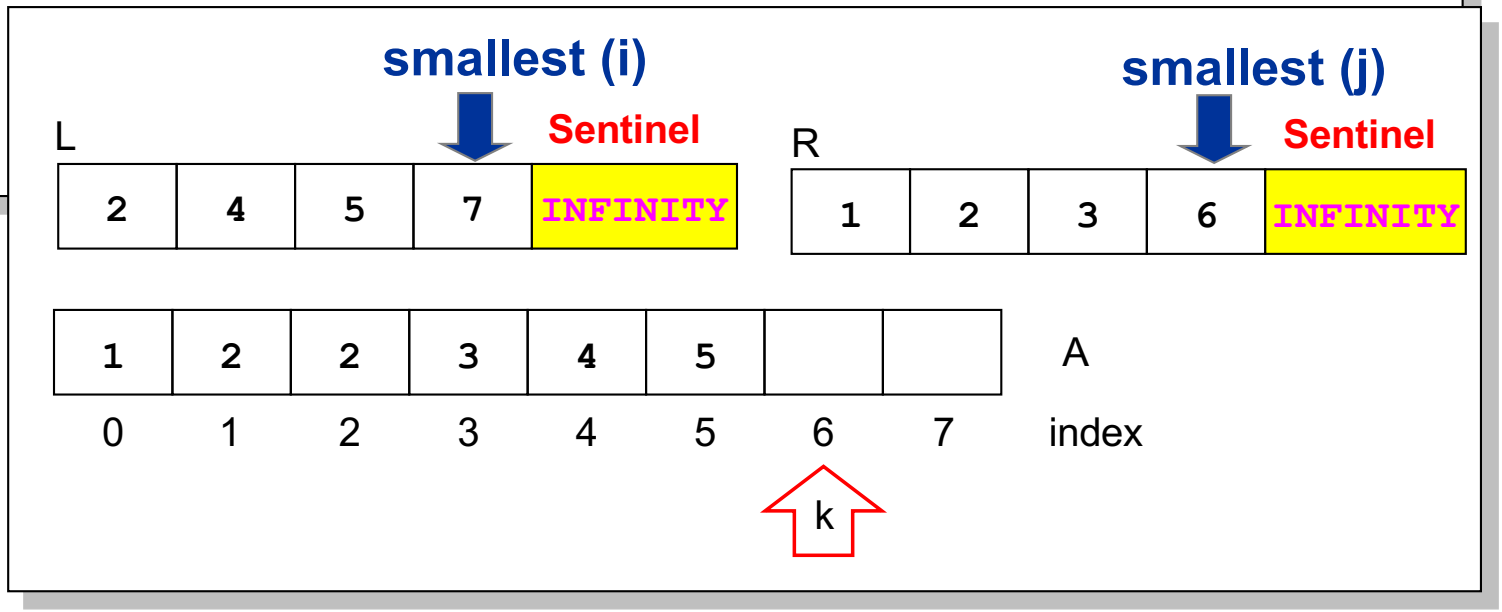




Implementing the Merge Step of Mergesort (p. 2)

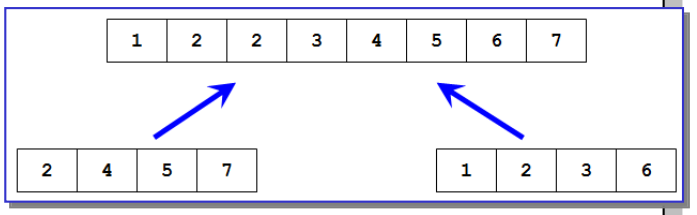
```
void Merge(float A[], int p, int q, int r) {
    ...
    int i = 0; int j = 0;
    for (int k = p; k<=r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}
```

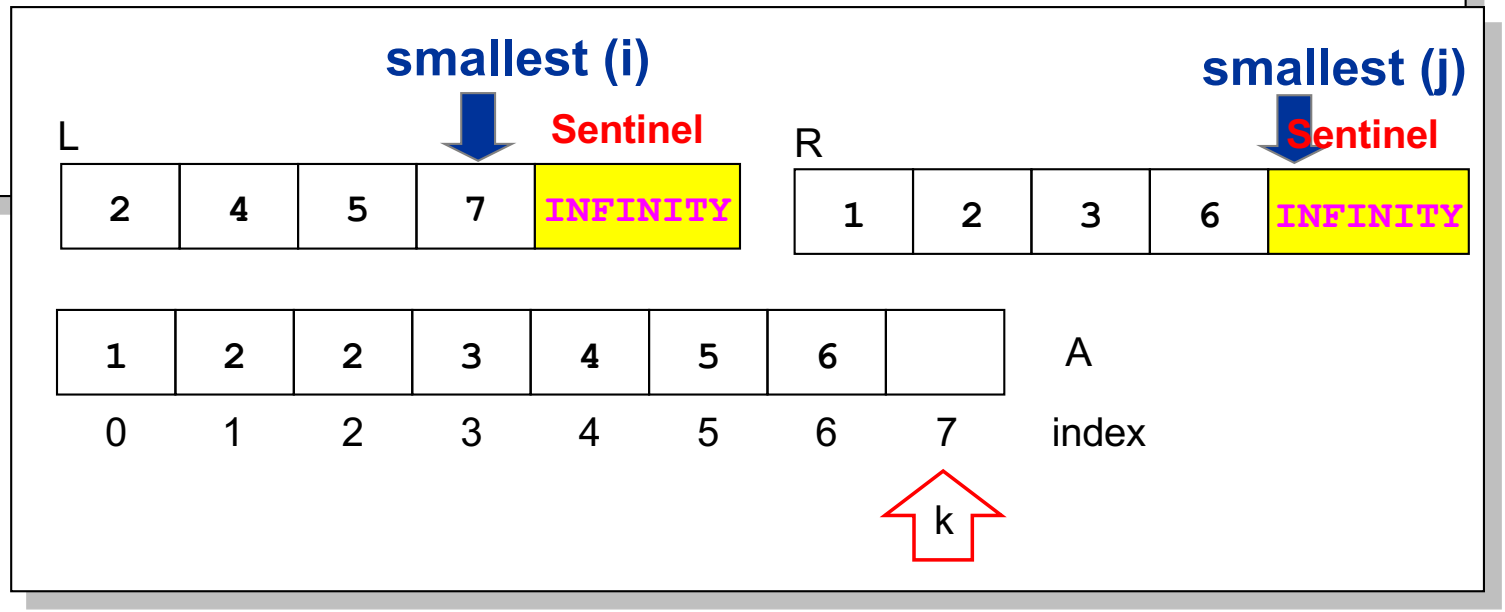




Implementing the Merge Step of Mergesort (p. 2)

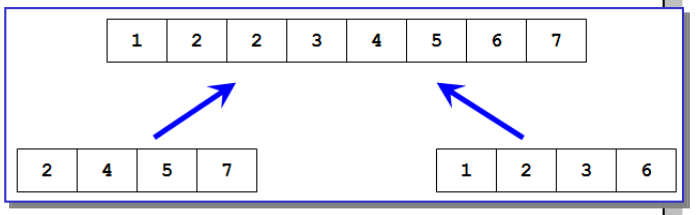
```
void Merge(float A[], int p, int q, int r) {
    ...
    int i = 0; int j = 0;
    for (int k = p; k<=r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}
```

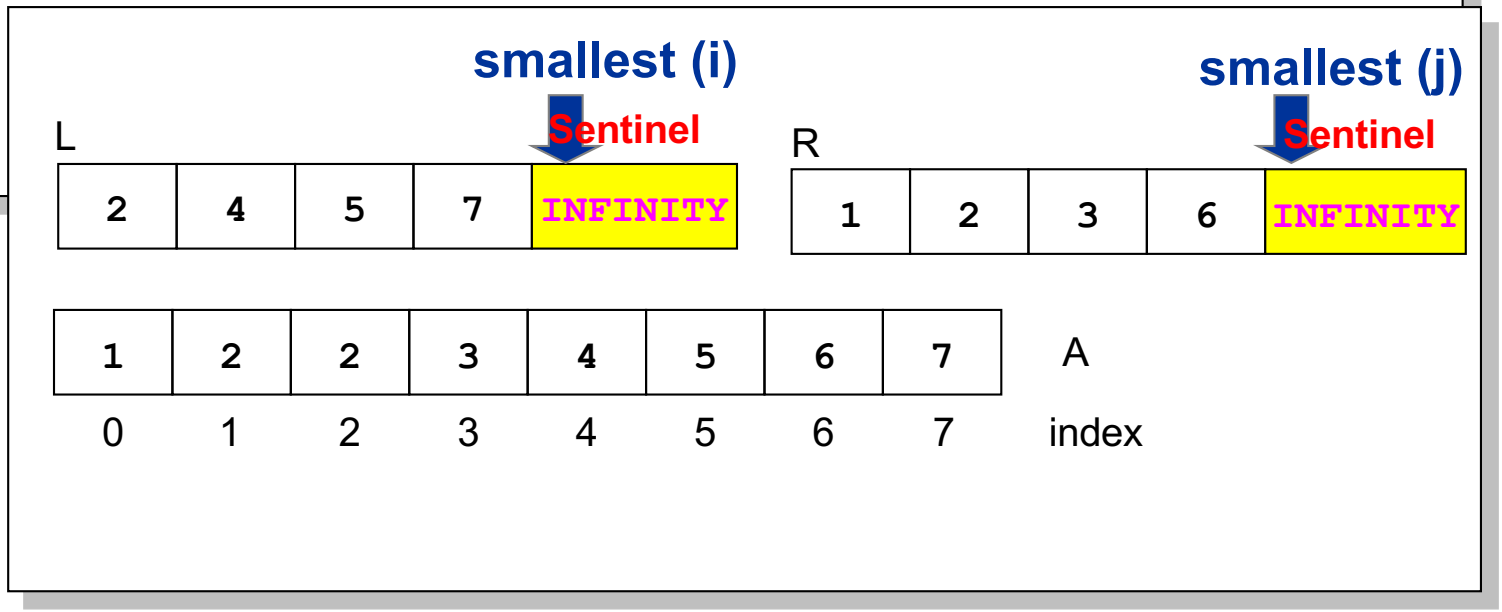




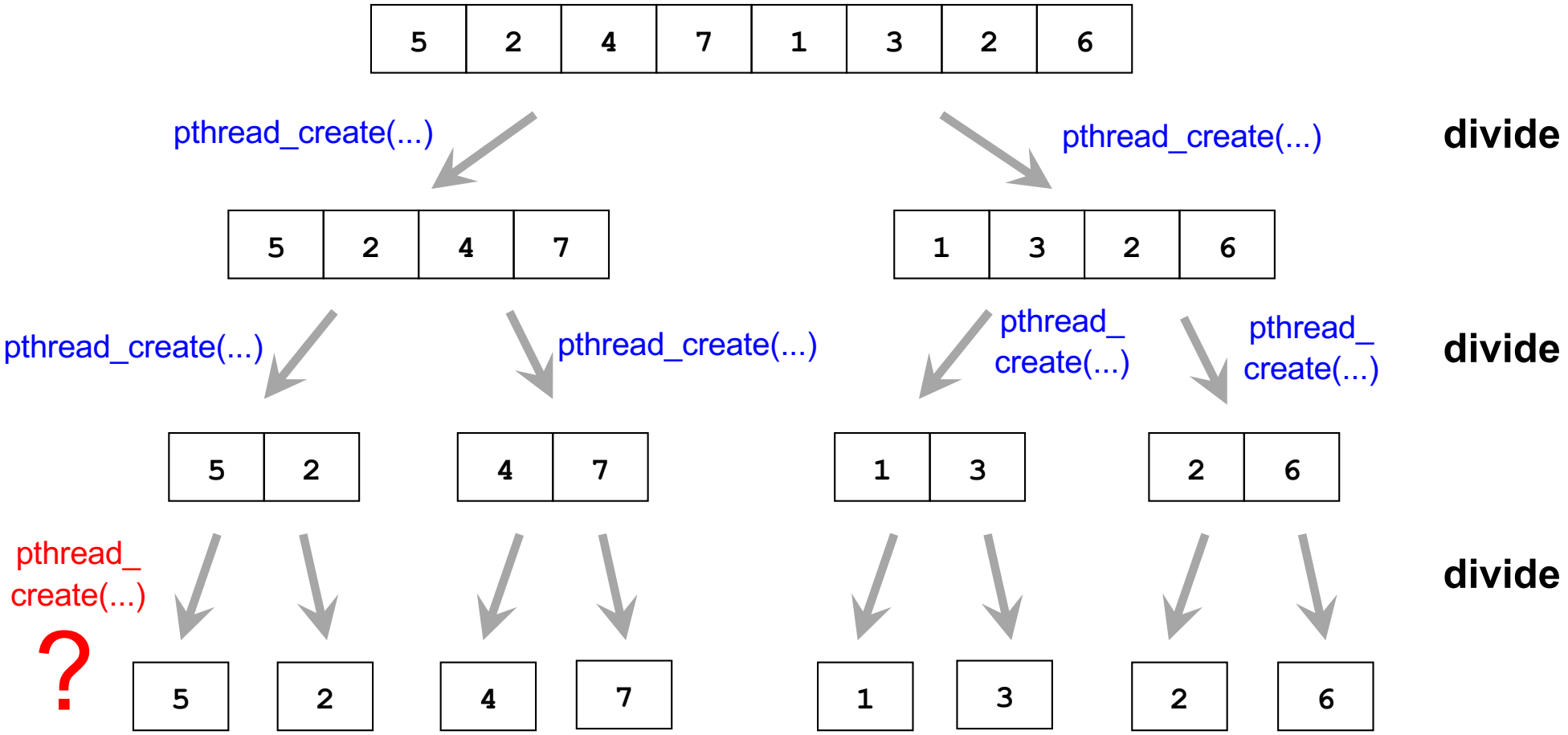
Implementing the Merge Step of Mergesort (p. 2)

```
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++;
        }
    }
}
```

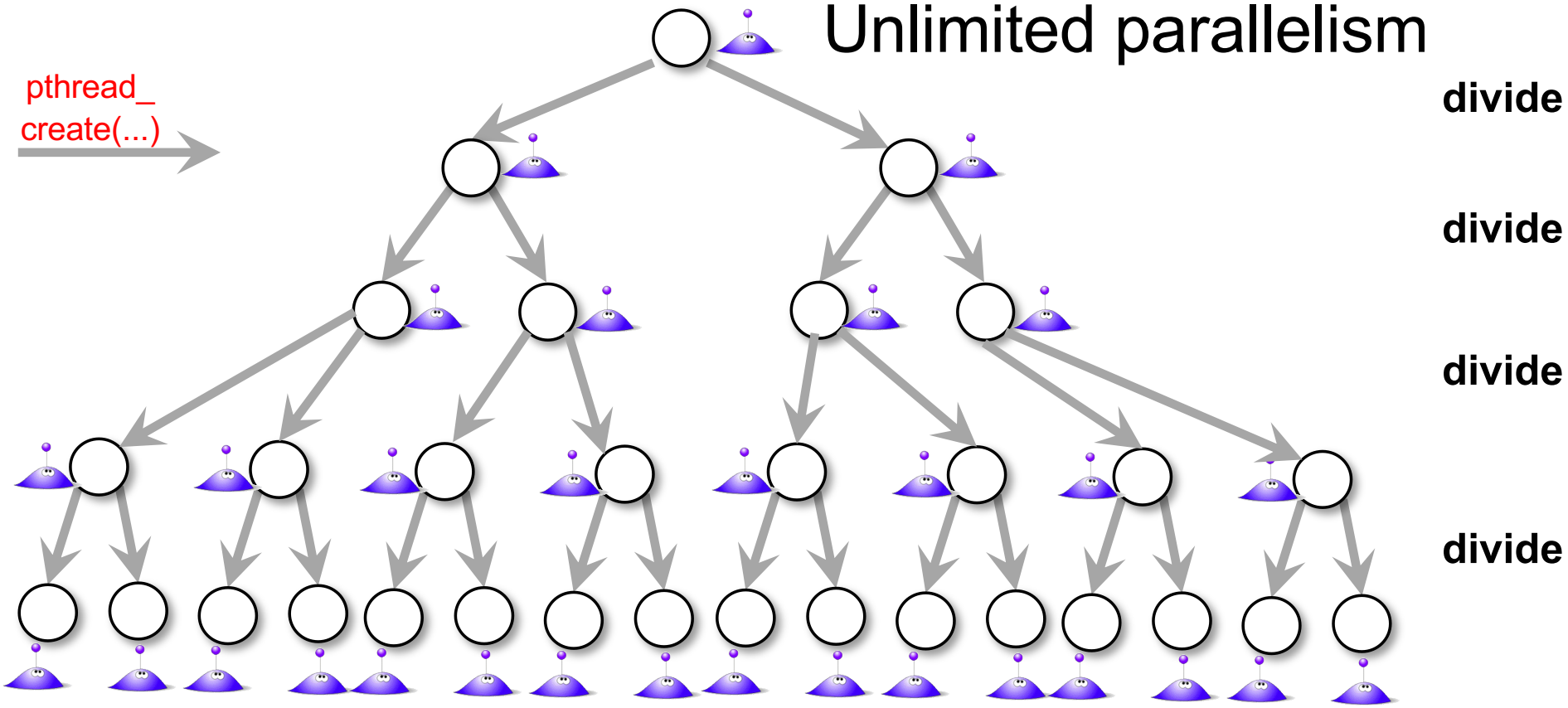




Parallelization of Mergesort

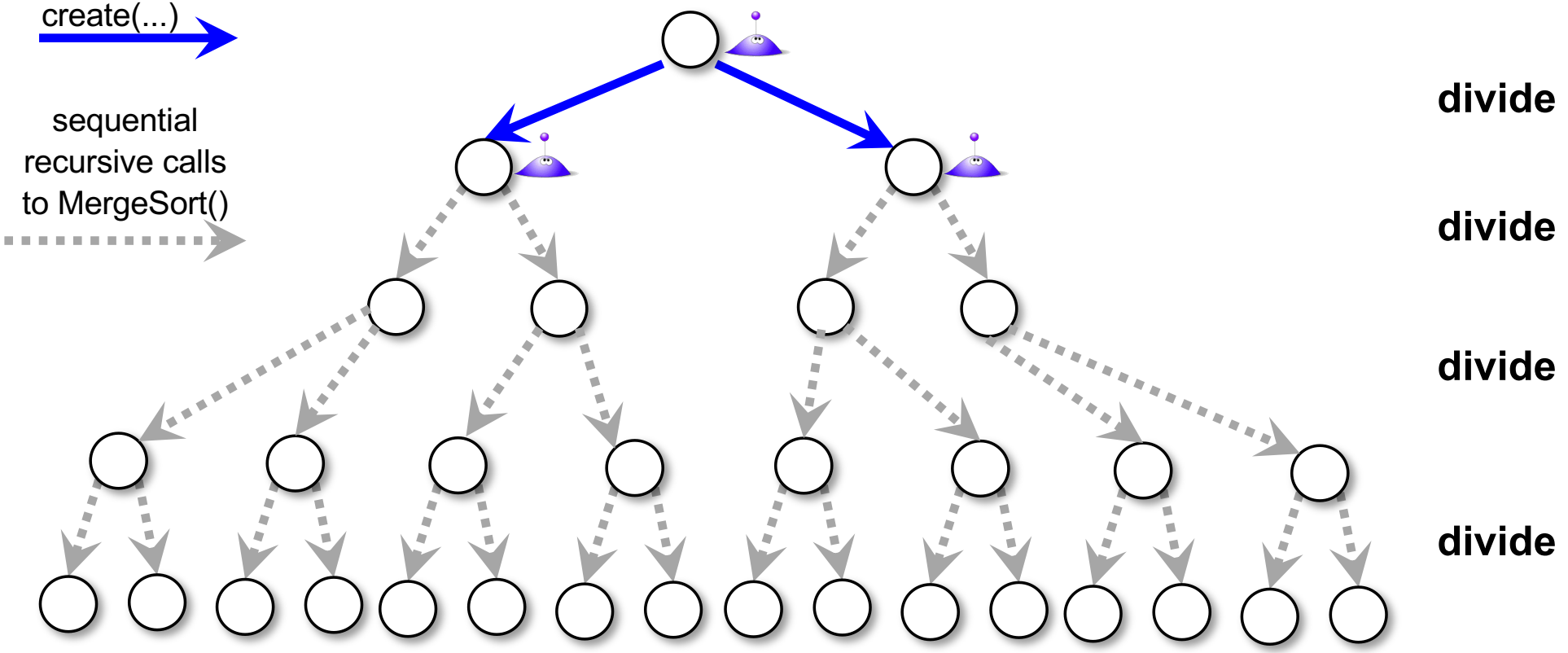


- For each dividing step, create 2 pthreads to sort in parallel.
  - At what recursion depth shall we **switch back to sequential mergesort?**



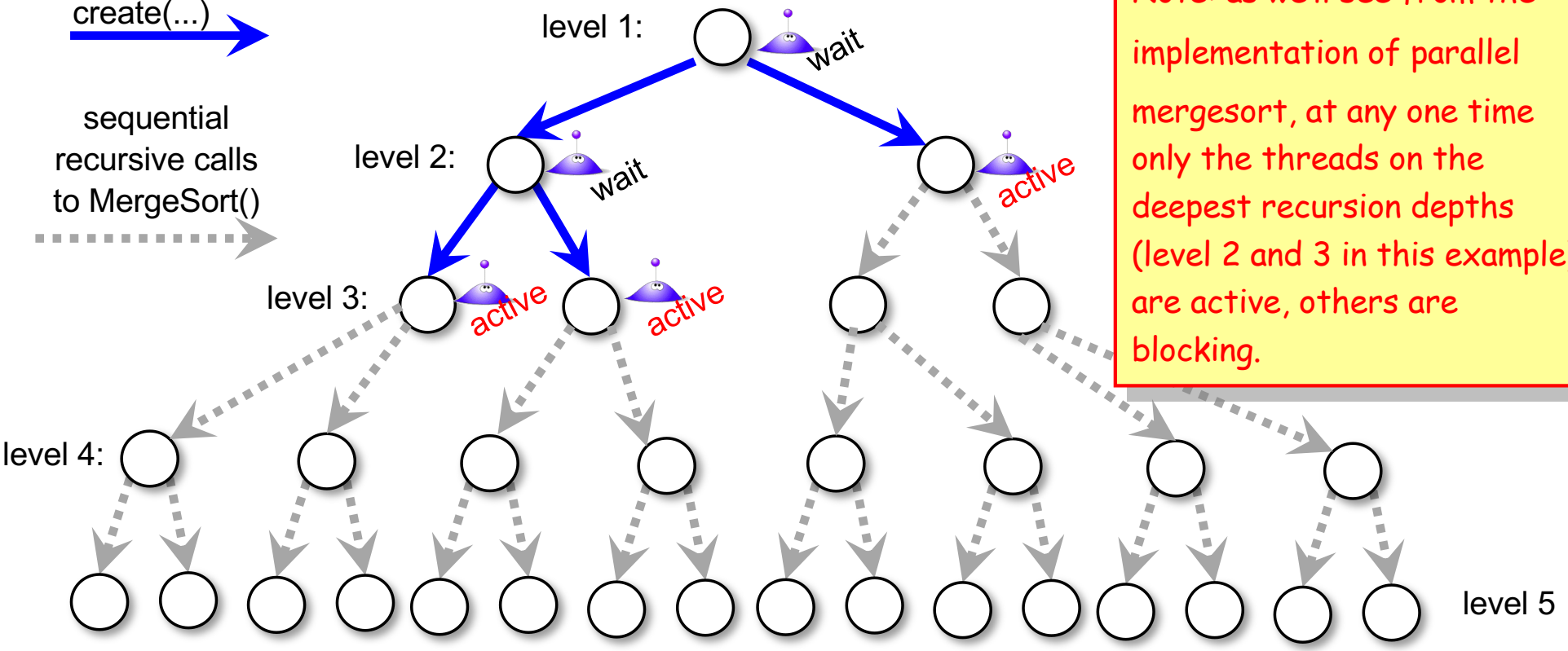
- With unlimited parallelism, we generate new threads with every dividing step, down to the base case.
  - Highest amount of **logical** parallelism (N threads, N=2^levels), **but**
    - towards base cases, work gets very small, thread creation overhead dominates
    - we have only p << N cores. Only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

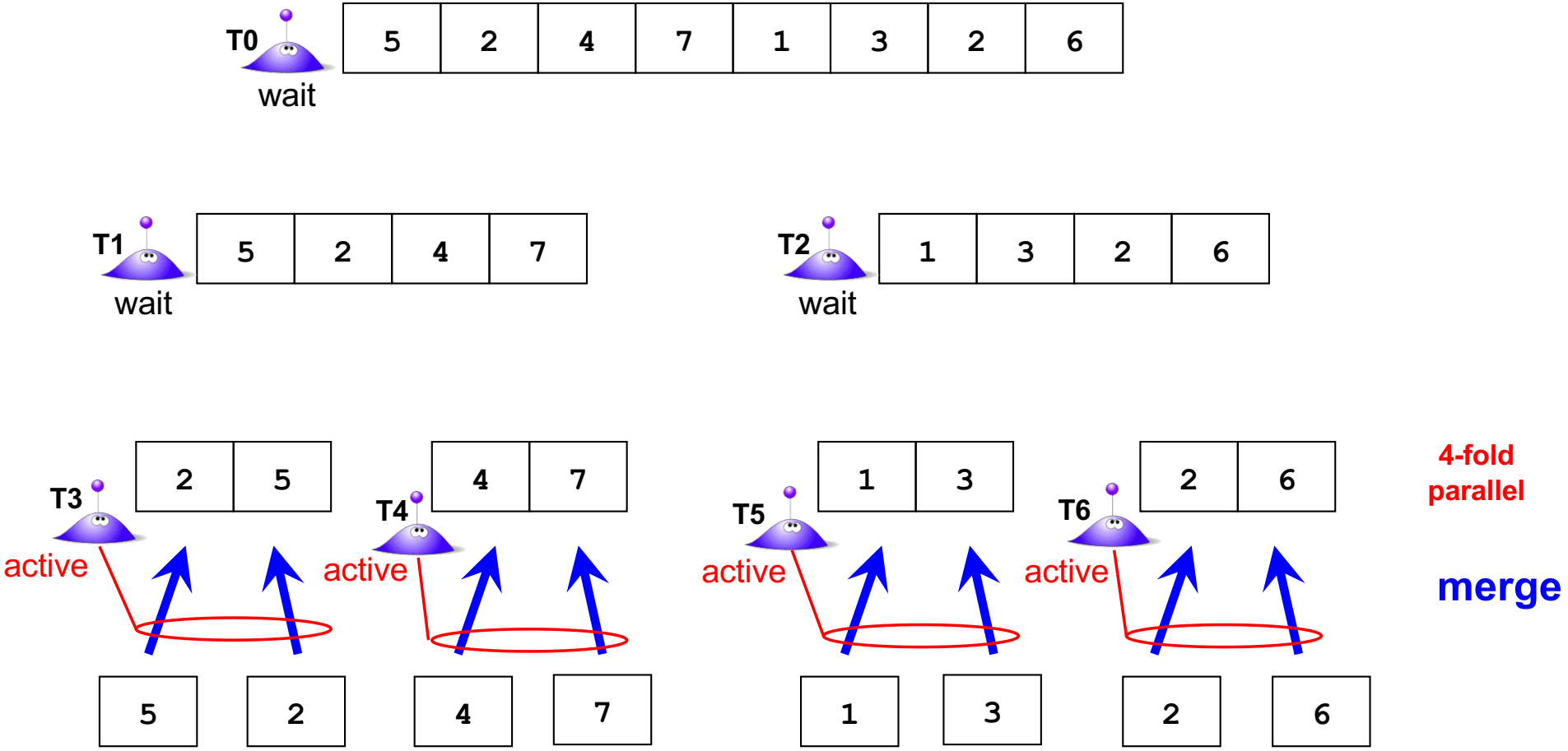
Scalable parallelism



Note: as we'll see from the implementation of parallel mergesort, at any one time only the threads on the deepest recursion depths (level 2 and 3 in this example) are active, others are blocking.

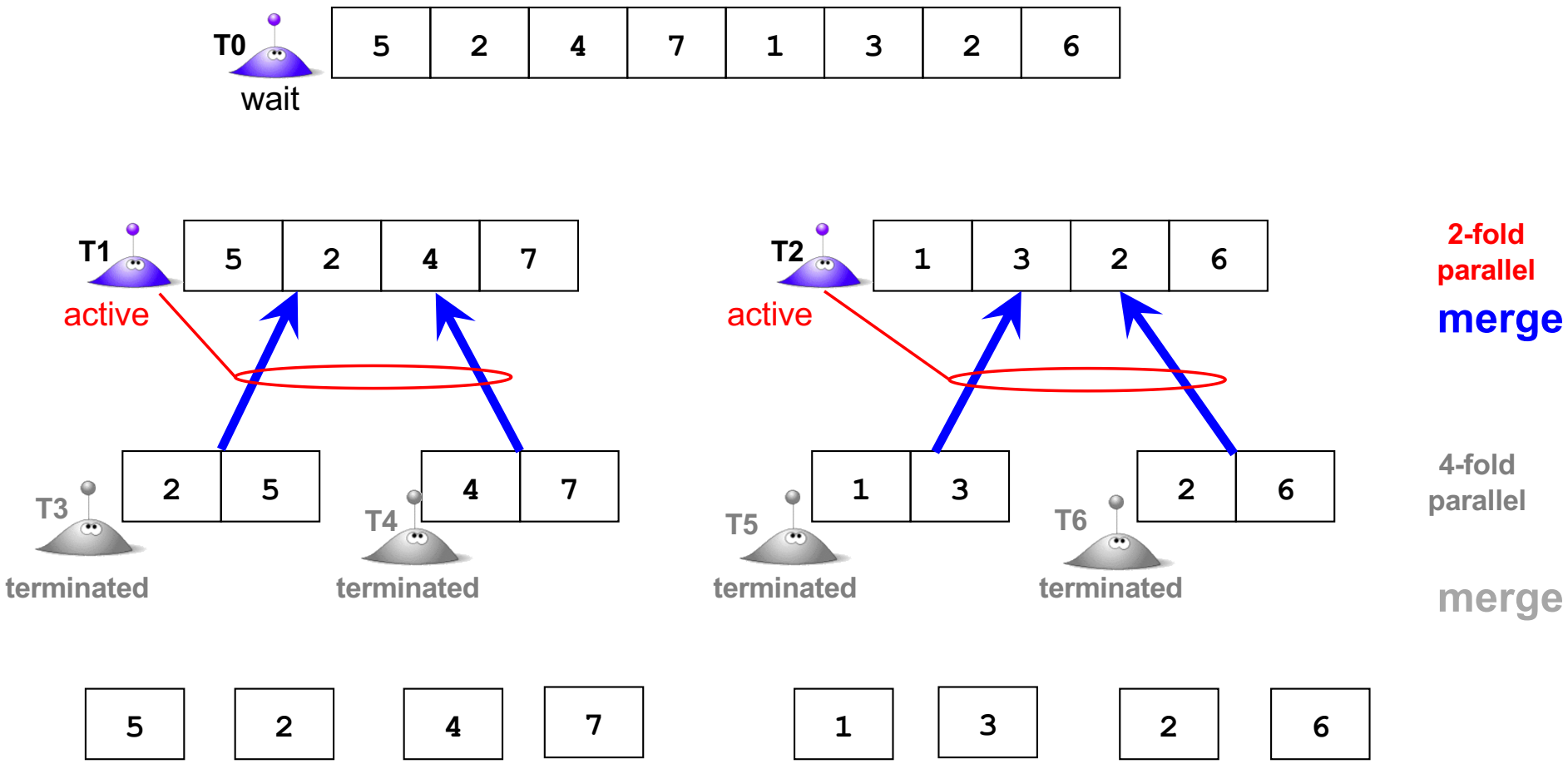
- With scalable parallelism, the level where we switch to sequential recursive calls depends on the number of available cores. E.g., 3 above.
  - **Pro:** Most efficient, scales to higher numbers of cores.
  - **Con:** Algorithms often harder to program for scalable parallelism.

Parallelization of Mergesort (cont.)



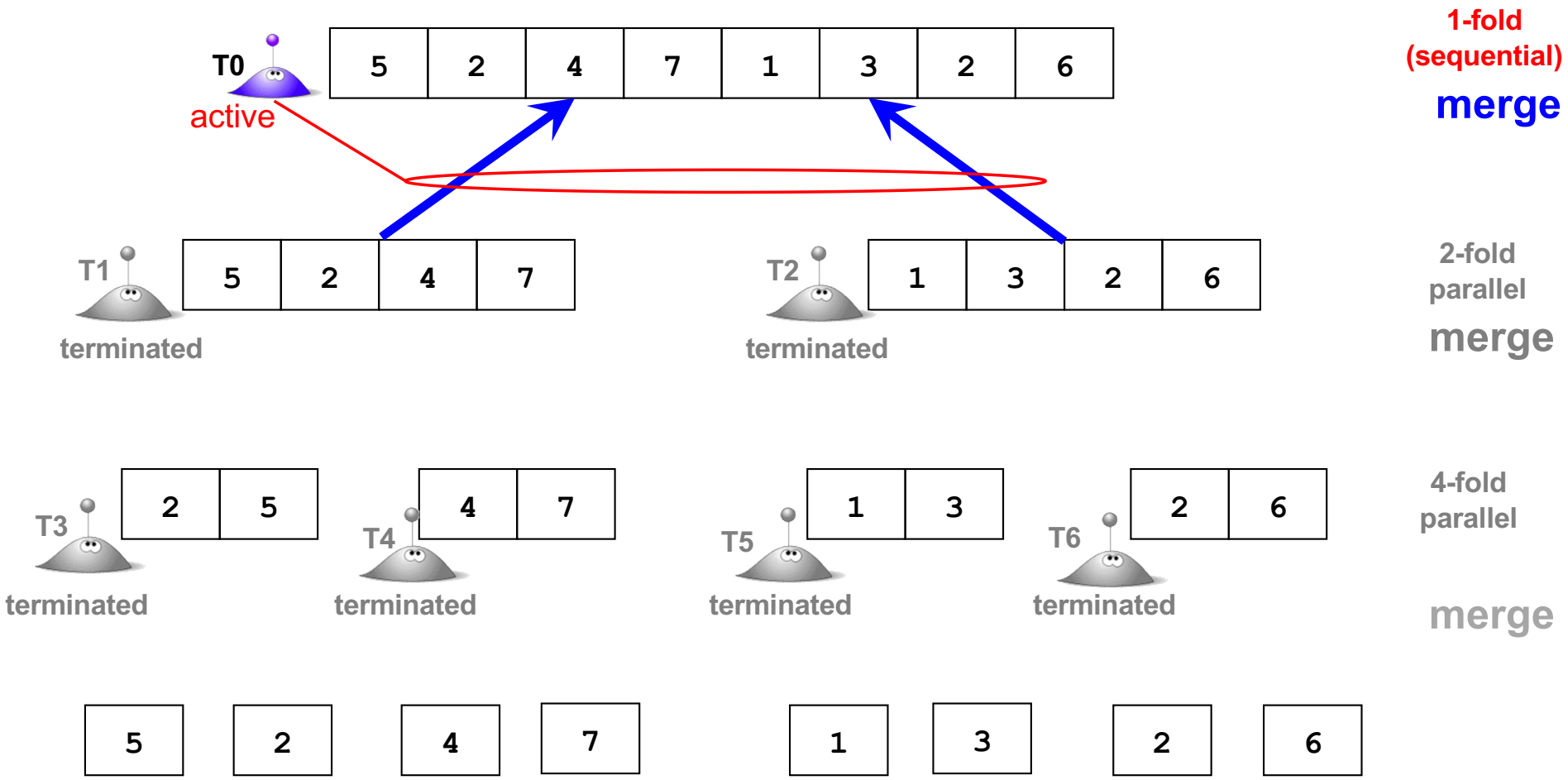
- Merging happens in parallel except on recursion depth level 1:
  - 4-fold parallelism, then 2-fold, then sequential.

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);
      }
   }
}
```

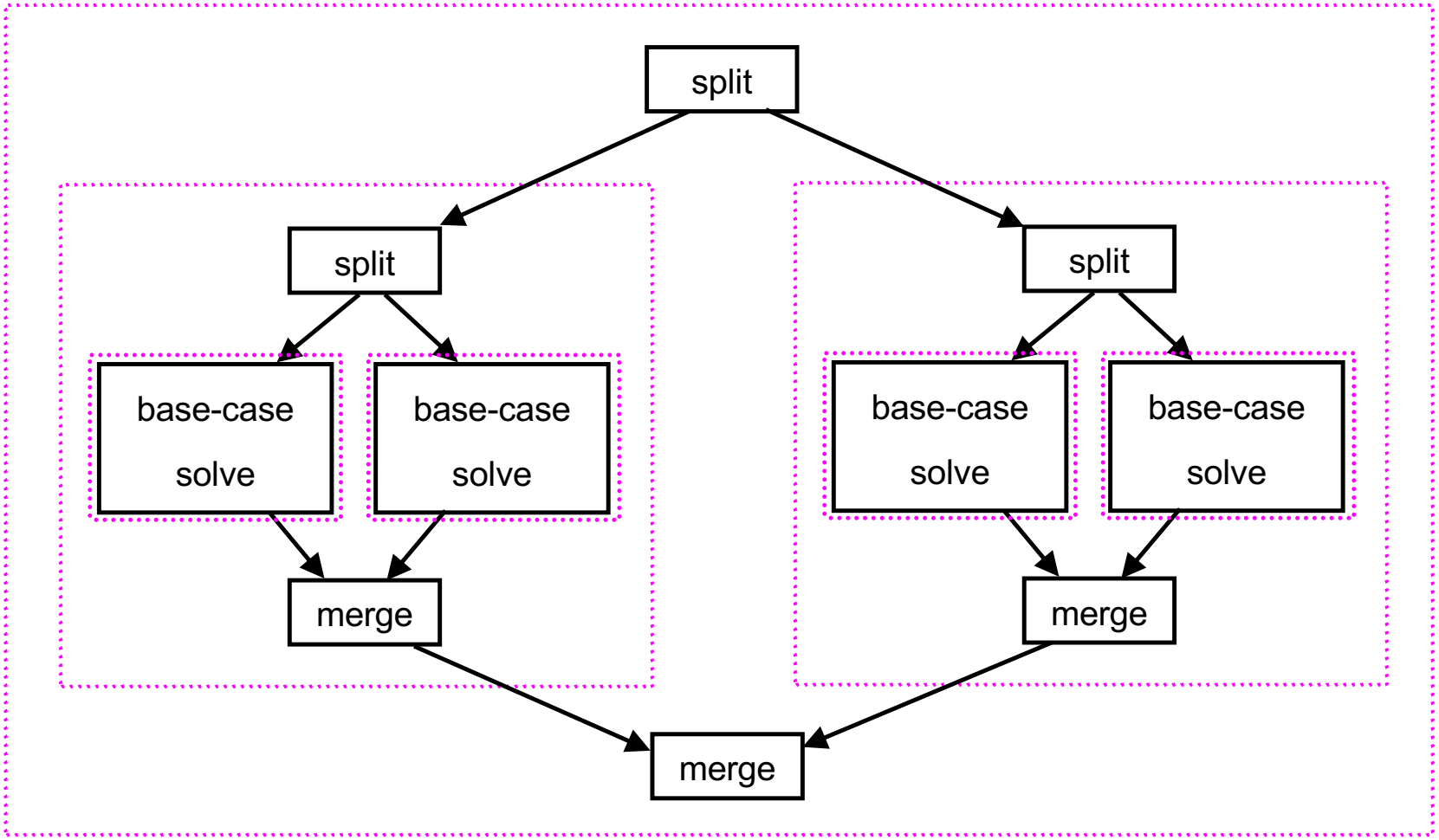
```
struct arg {
   float * A;
   int p;
   int r;
   int depth;
   int max_depth;
};
```

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)
      // 2)
      // 3)
      int q
      if (p
        q =
        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);
      }
   }
}
```

```
void * PMSort (void * ptr) {
   struct arg * MyArg = (struct arg *) ptr;
   ParallelMergeSort(MyArg->A, MyArg->p,
                     MyArg->r, MyArg->depth, MyArg->max_depth);
}
```

Divide & Conquer Parallelization



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

Outline

- Recursion ✓
- Divide and Conquer ✓
  - How to parallelize ‘Divide and Conquer’ algorithms ✓
  - Example: Mergesort ✓
- Reductions

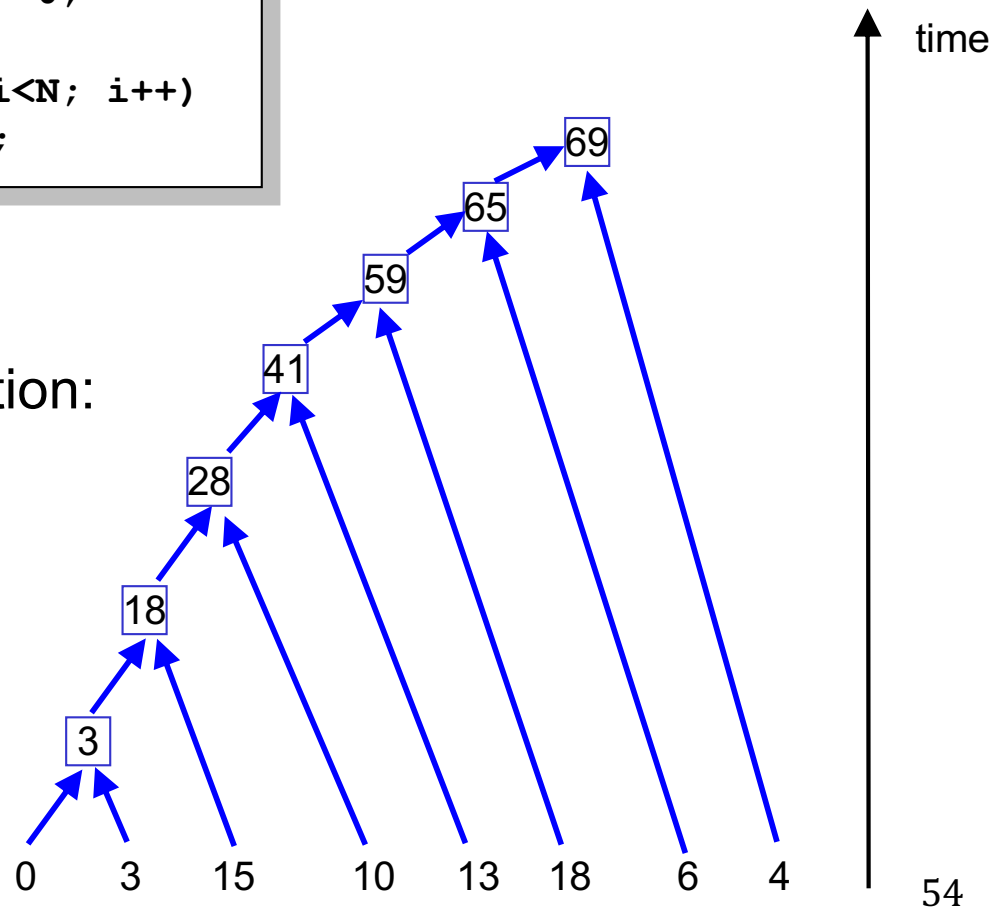
Sequential sum computation

- Assume we want to add up all elements in an array.
- Sequential computation:

```
int A[N], sum = 0;

for (int i=0; i<N; i++)
    sum += A[i];
```

- Graph of sequential computation:

7 steps:

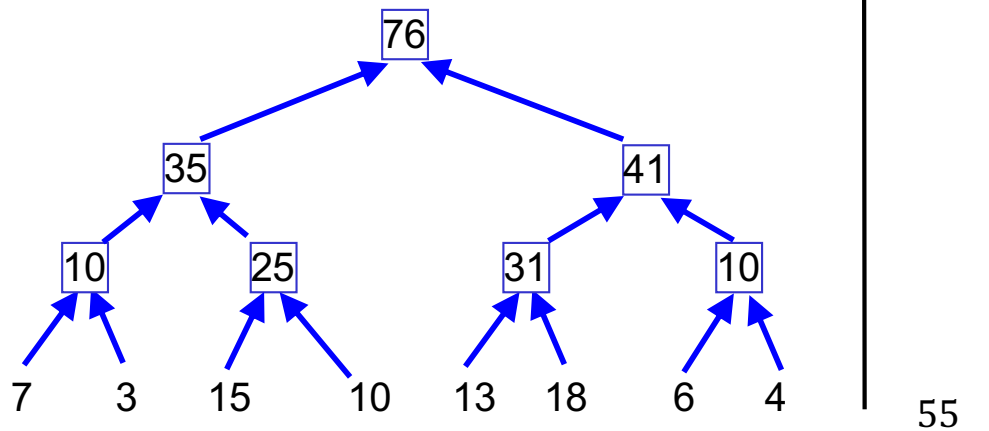


Pairwise summation (parallel!)

- Assume we want to add up elements *x0, x1, ..., x7* in an array.
- Pairwise summation adds even/odd pairs of data values
  (x0+x1), (x2+x3), (x4+x5), (x6+x7)
- Those pairs are again added in pairs:
  ((x0+x1) + (x2+x3)), ((x4+x5) + (x6+x7))
- And so on….

- Graph of computation:

- Same number of operations
  - No advantage with 1 CPU
- On multicore, all additions on same level can be performed in parallel
  - 3 steps vs. 7 sequentially
- Requires that operation is associative:
  - a + b + c + d = (a + b) + (c + d)



Reduction Definition

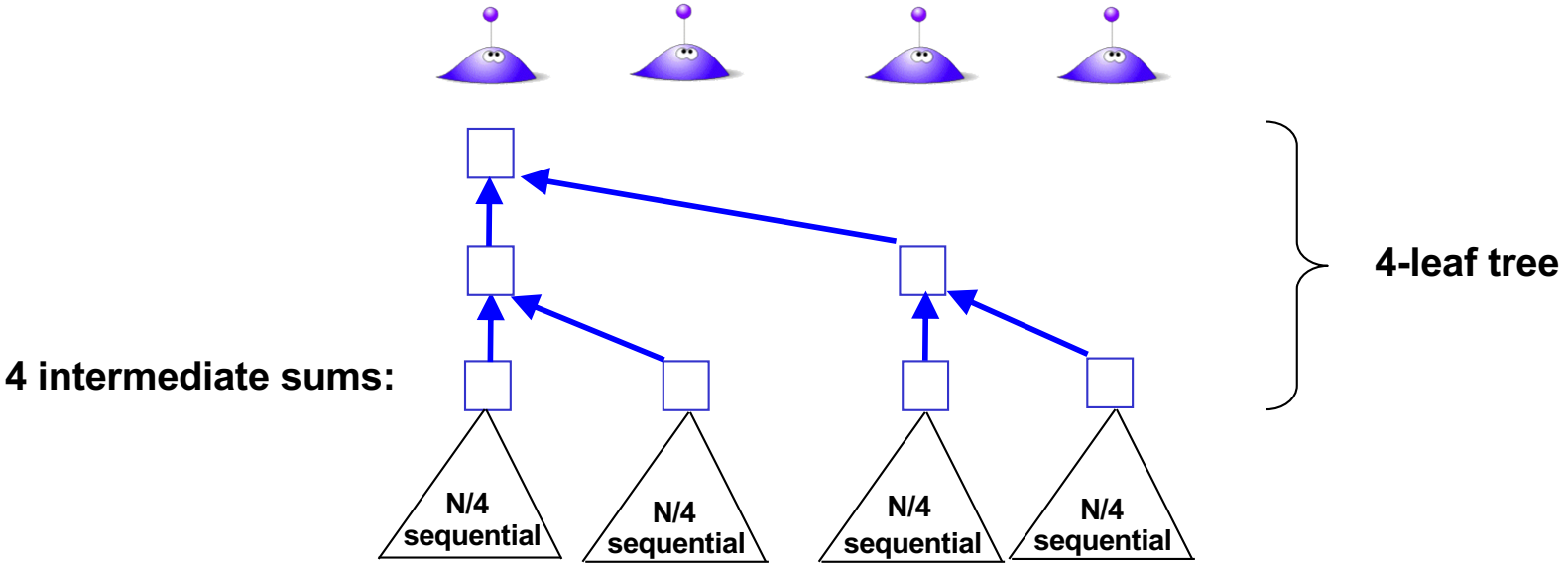
- A **reduction operation** reduces a collection of data items to a single data item by repeatedly combining the data items pairwise with a binary operator.
  - Operator usually associative and commutative
- Examples of reduction operators:
  - Sum (+), product (*), maximum
- Further reduction operators:
  - Smallest array element
  - Second-smallest array element
  - K-way histogram
  - Index of first occurrence of x

Schwartz' Algorithm for + Reduction

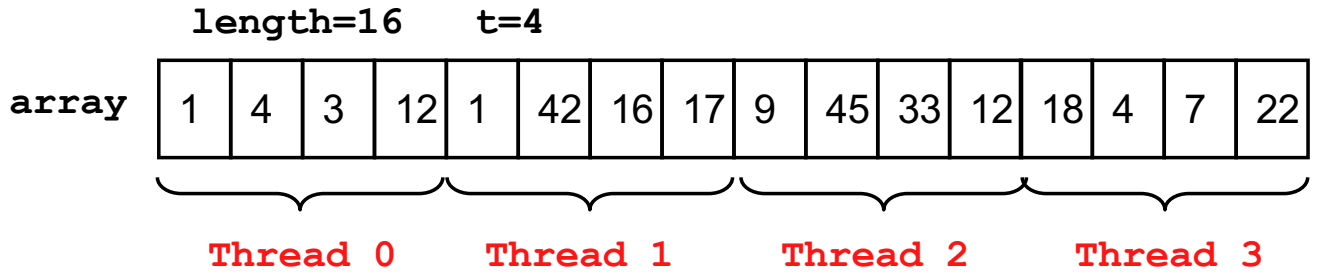
Given P cores and N values, P < N:

1) Have P threads add N/P items sequentially
2) Combine the P intermediate sums with a P-leaf tree

Example for P=4:



Motivation to use + Reduction:

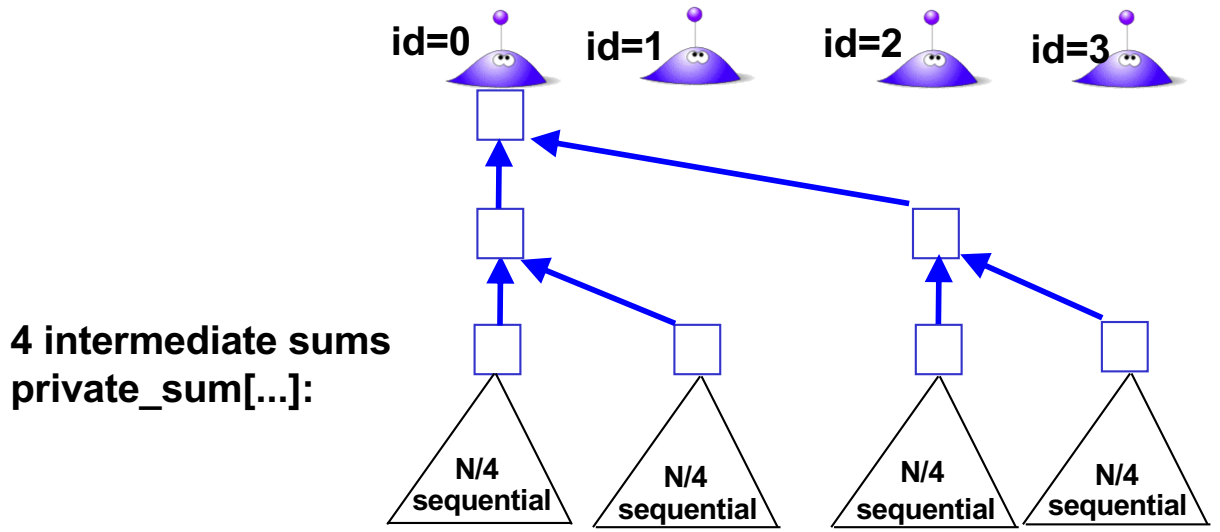



```
unsigned long long private_sum[4]={0, ...};
unsigned long long sum = 0;
pthread_mutex_t m;

for(i=start; i<start+length_per_thread; i++){
   private_sum[id] = private_sum[id] + array[i];
}
// P pieces: each trying to write sum
pthread_mutex_lock(&m);
   sum = sum + private_sum[id]; // combine
pthread_mutex_unlock(&m);
```

- Each thread adds into its own memory (variable private_sum[]).
- Combine sums at the end → **serialization!**

Code example for +Reduction

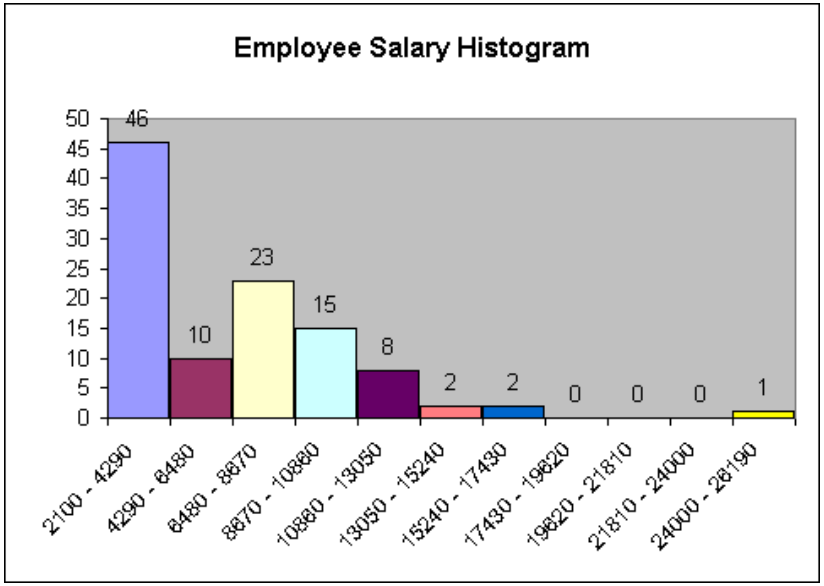
```
for(i=start; i<start+length_per_thread; i++){
   private_sum[id] = private_sum[id] + array[i];
}

barrier_wait(b4); //wait for all 4 threads to complete seq.
if((id % 2) == 0)
   private_sum[id]+=priviate_sum[id+1]; // combine
barrier_wait(b4);
if(id==0)
   sum = private_sum[0] + private_sum[2];
```



Points to note for reductions

- A + Reduction of a shallow tree (not many levels) contains not much computation.
  - Might not be much faster than the sequential sum

- However, deeper trees or more complicated reduction operations will yield more speed-ups.
  - Example: k-way histogram reduction



Outline

- Recursion ✓
- Divide and Conquer ✓
  - How to parallelize ‘Divide and Conquer’ algorithms ✓
  - Example: Mergesort ✓
- Reductions ✓