Divide and Conquer Sorting Algorithms and Non-comparison-based Sorting Algorithms

COMP1927 17x1

Sedgewick Chapters 7 and 8
Sedgewick Chapter 6.10, Chapter 10

Divide and Conquer Sorting algorithms

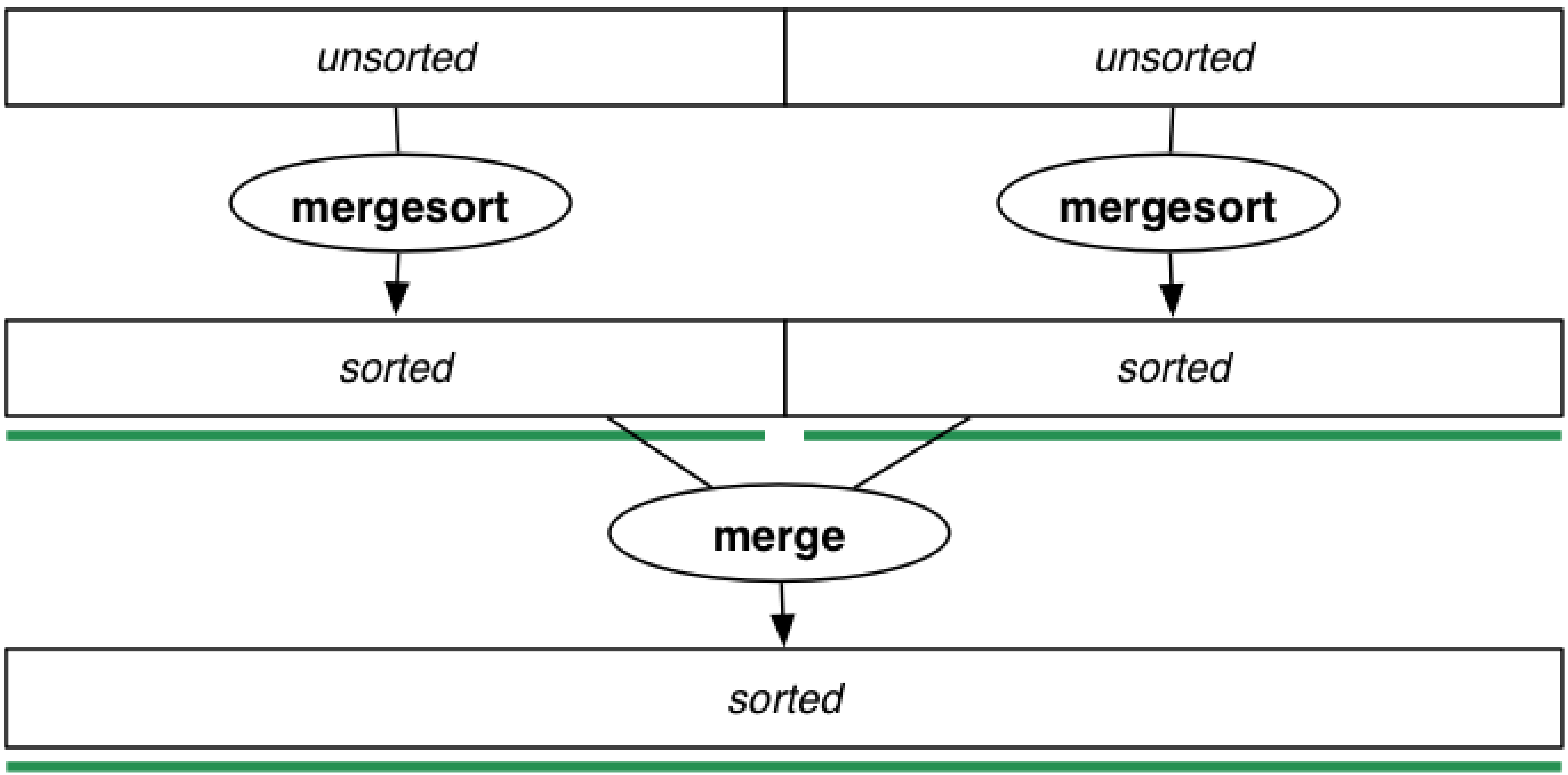
- Step 1
  - If a collection has less than two elements, it's already sorted
  - Otherwise, split it into two parts
- Step 2
  - Sort both parts separately
- Step 3
  - Combine the sorted collections to return the final result

Merge Sort

- Basic idea: Divide and Conquer
  - split the array into two equal-sized partitions
  - (recursively) sort each of the partitions
  - merge the two sorted partitions together

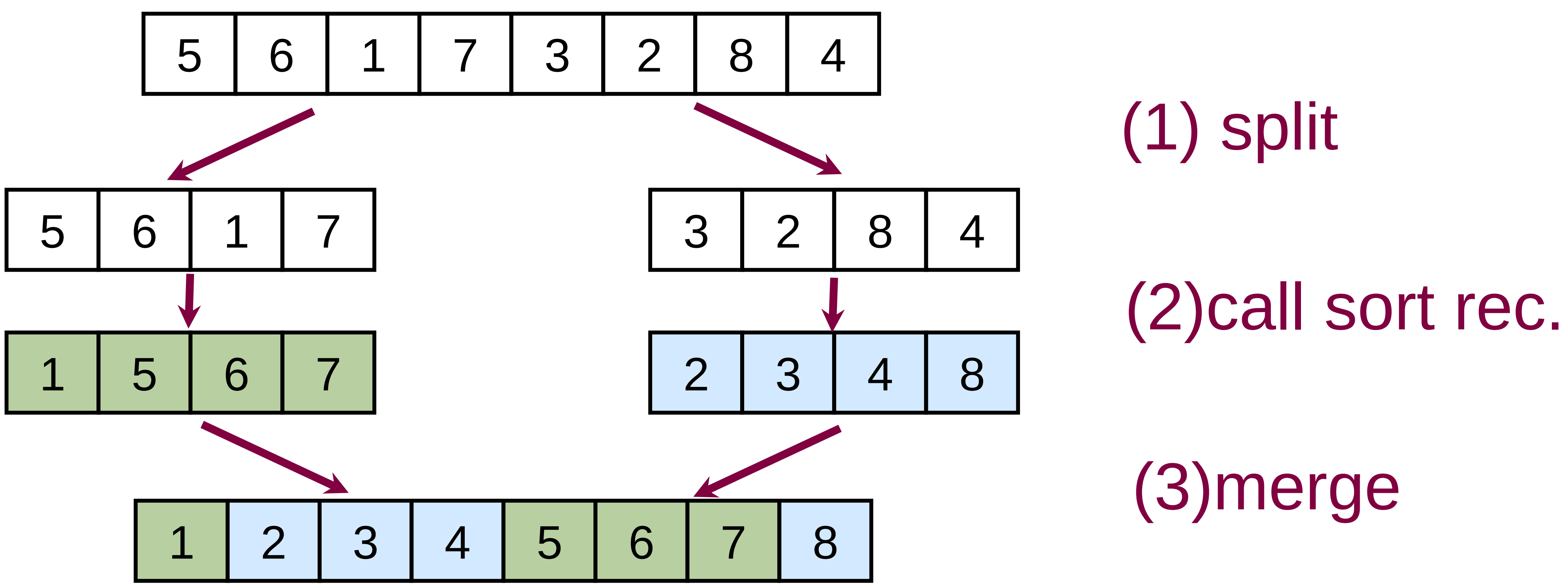
- Merging: Basic idea
  - copy elements from the inputs one at a time
  - give preference to the smaller of the two
  - when one exhausted, copy the rest of the other

Phases Of Merge Sort



Divide and Conquer Sorting: Mergesort

- Split the sequence in halves
- Sort both halves independently
- What is the best way to combine them?
  - look at the first element in each sequence, pick the smallest of both, insert in sorted collection, continue until all elements are used up



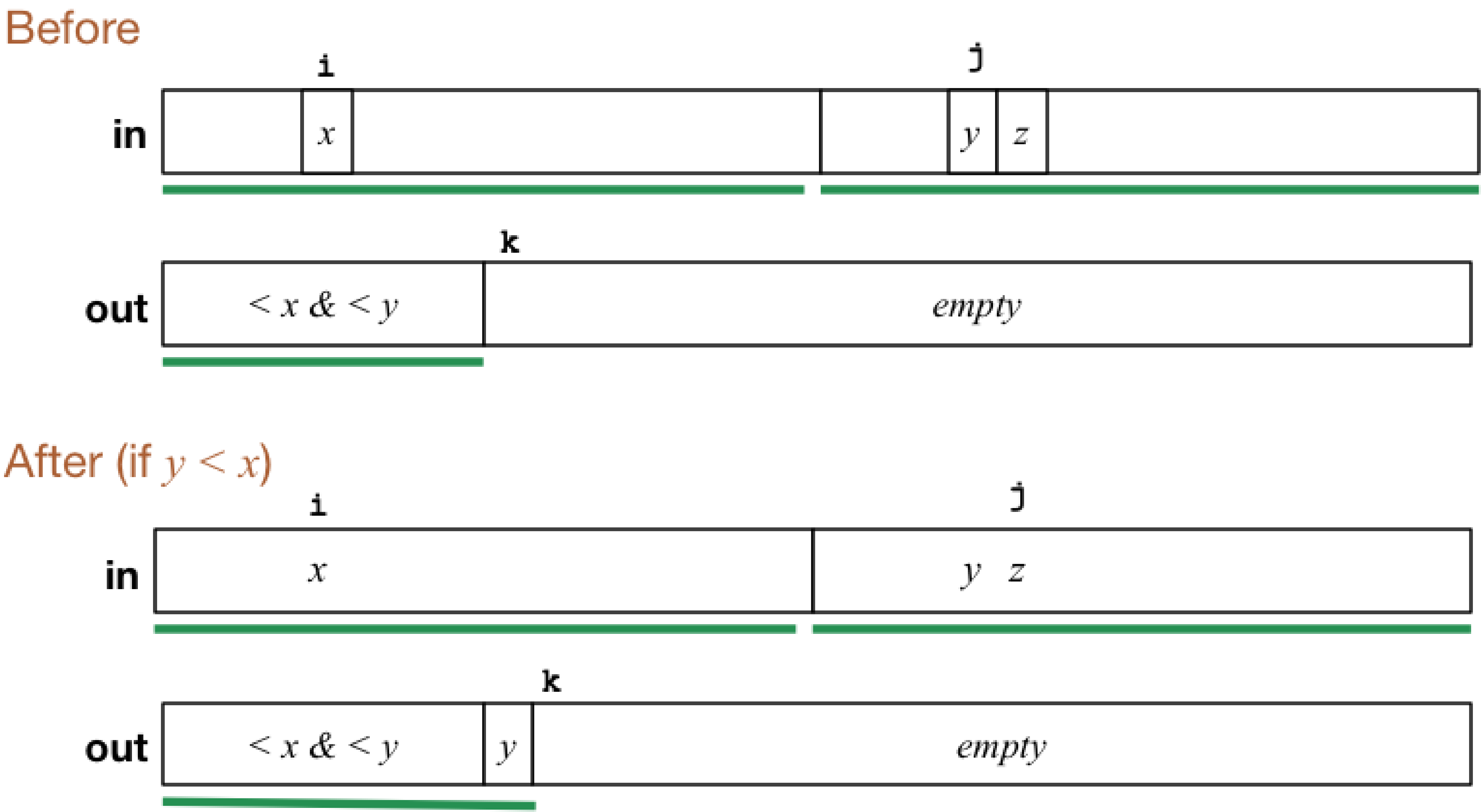
MERGE SORT: ARRAY IMPLEMENTATION

- assuming we have `merge` implemented, `mergesort` can be defined as:

```
void merge (int a[], int lo, int mid, int
hi);

void mergesort (Item a[], int lo, int hi){
     int mid = (lo+hi)/2;  // midpoint
     if  (hi <= lo) {
          return;
     }
     mergesort (a, lo, mid);
     mergesort (a, mid+1, hi);
     merge (a, lo, mid, hi);
}
```

Merging Process

- The merging process:



Merge Implementation

```
// sorted(a[0..mid]), sorted(a[mid+1..N-1])
int a[N]; // input array
int b[N]; // output array
mid = N/2;
i = 0;
j = mid+1;
k = 0; while (i <= mid && j <= N-1) {

      if (a[i] <= a[j]) b[k++] = a[i++]
      if (a[j] < a[i]) b[k++] = a[j++]
}
while (i <= mid)
      b[k++] = a[i++]
while (j <= N-1)
      b[k++] = a[j++]
```

Mergesort: Work Complexity

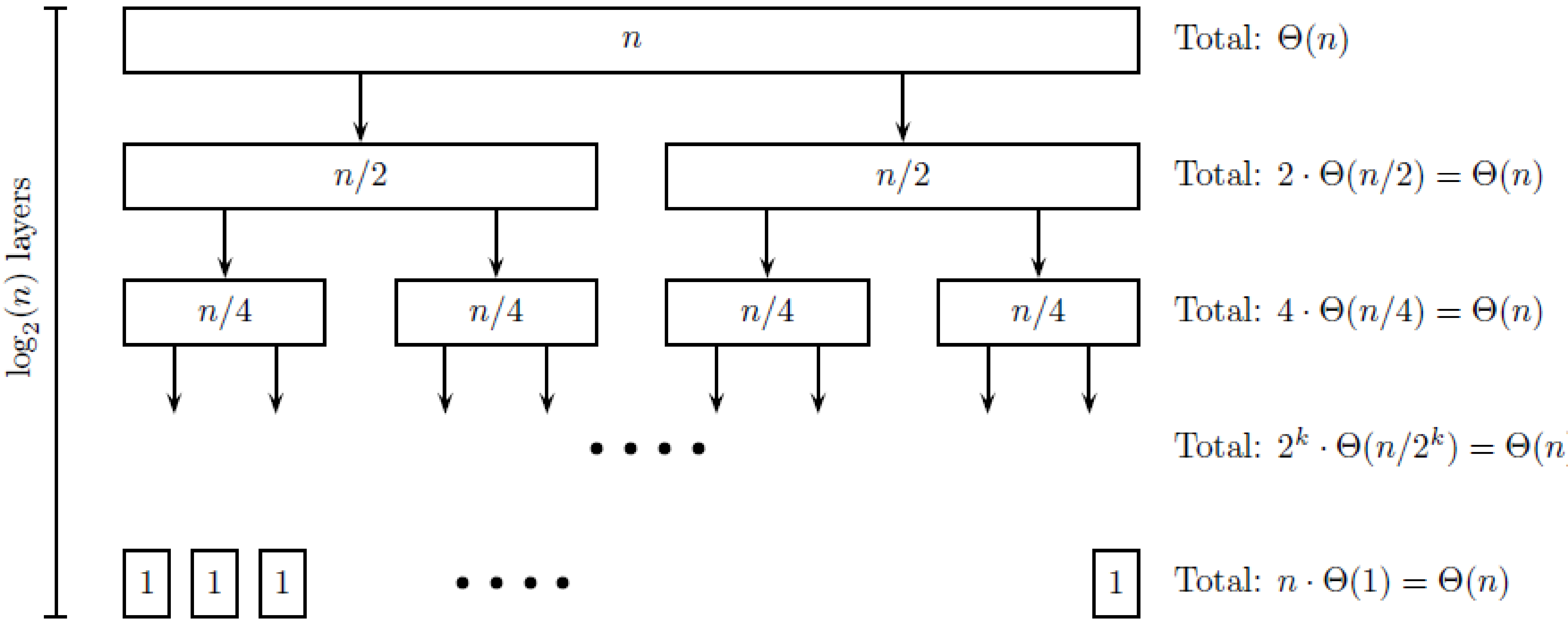
Running time on an array of size n accrues as follows:

1. First, we spend time O(1) for computing m.
2. Then, we make two recursive calls to merge sort, with arrays of sizes $\lfloor (n-1)/2 \rfloor$ and $\lceil (n-1)/2 \rceil$
3. Finally, call **merge**. Merge goes through the two sub-arrays with one loop, always increasing one of i and j. Thus, it takes time (n).

$$T(n) = T(\lfloor (n-1)/2 \rfloor) + T(\lceil (n-1)/2 \rceil) + (n),$$

$$T(1) = (1)\ ,\ T(0) = (1) \text{ (Base case)}$$

MERGESORT: WORK COMPLEXITY



- Total work done at each level = n
- Total number of levels = $log_2 n$ levels (halving at each level, starting at *n* and finishing at *1*)
- Total work over all levels = $n \, log_2 n$
- Disadvantage over quicksort: need extra storage *O(n)*

Merge Sort Work Complexity

- Overall:
  - Merge sort is in O(*nlogn*),
  - Stable – as long as merge implemented to be stable
  - Not in-place: Uses O(n) memory for merge and $O(\log_2 n)$ stack space
  - Non-adaptive : still nlogn for ordered data

Bottom Up Merge Sort

- Basic Idea: Non-recursive
  - On each pass, array contains sorted sections of length m
  - At start treat as n sorted sections of length 1
  - 1st pass merges adjacent elements into sections of length 2
  - 2nd pass merges adjacent elements into sections of length 4
  - continue until a single sorted section of length n
- This approach is used for sorting diskfiles

Bottom-Up Merge sort Array Implementation

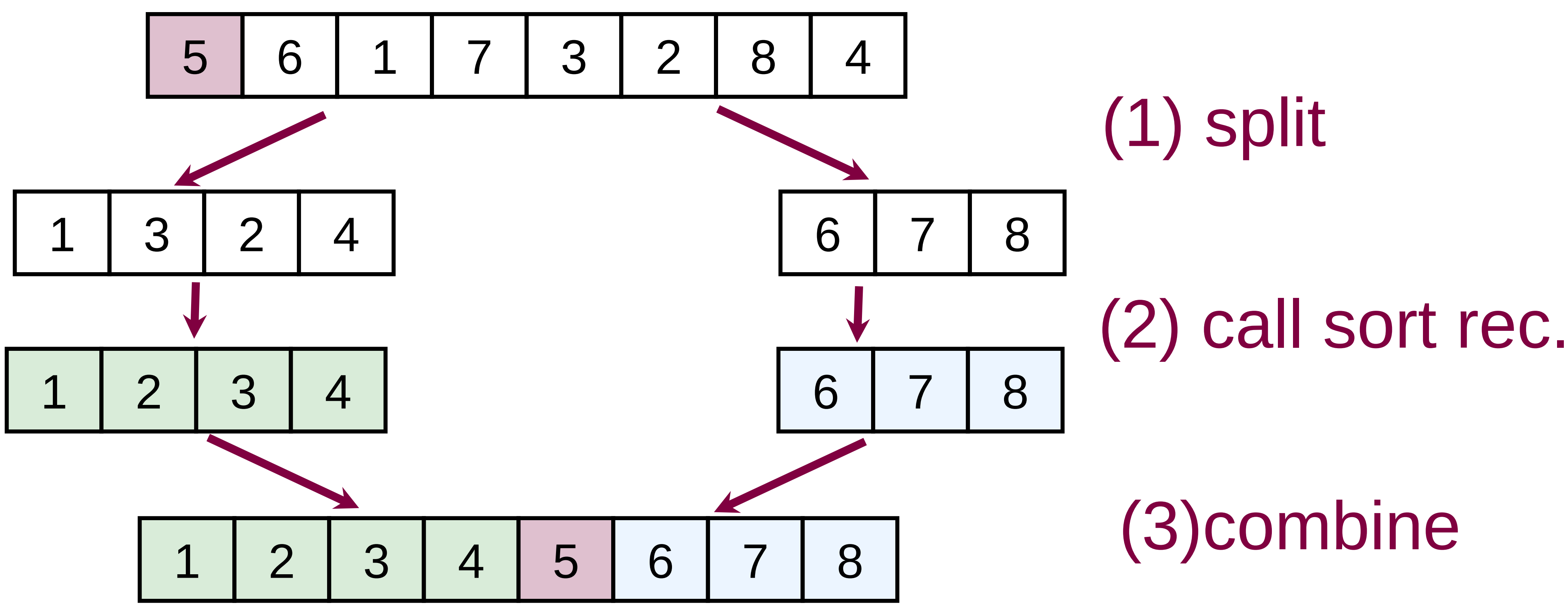
```
#define min(A,B) (A<B ? A : B)

int merge (int a[], int l, int m, int r);

void mergesortBU (int a[], int l, int r){
    int i, m, end;
   for (m = 1; m <= r-l; m = 2*m) {
     for (i = l; i <= r-m; i += 2*m) {
        end = min(i + 2*m – 1, r));
        merge (a, i, i+m-1, end);
     }
   }
}
```

MERGE SORT: IMPLEMENTATION

- Straight forward to implement on lists
  - Traverses its input in sequential order
  - Do not need extra space for merging lists
  - Works for top-down and bottom up versions

Divide and Conquer Sorting: Quicksort

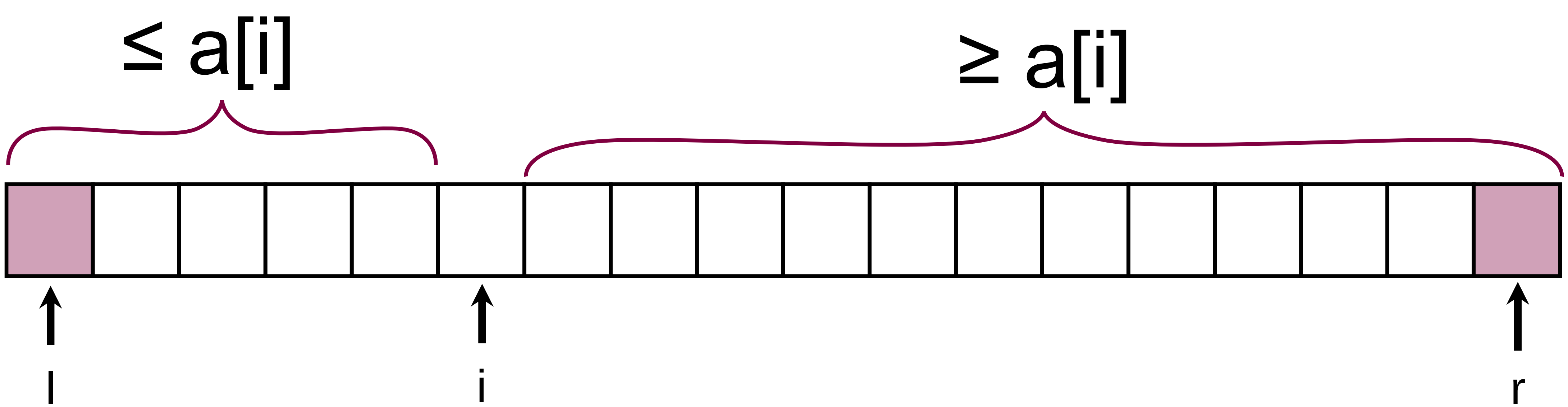
- Mergesort uses a trivial split operation and puts all the work in combining the result
- Can we split the collection in a more intelligent way, such that combining the results is trivial?
  - make sure all elements in one part are less than all the elements in the second part



More on Quick sort: implementation

On arrays, we need in-place partitioning:

- we need to swap elements in the array, such that for some pivot we choose, and some index i, all
  - `j<i, a[j] ≤ a[i], and`
  - `k>i, a[k] ≥ a[i]`



Quick sort

- Given such a partition function, the implementation of quick sort on arrays is easy:
  - However, it's surprisingly tricky to get `partition` right for all cases

```
int partition(int a[], int l, int r);

void quicksort (int a[], int l, int r){
   int i;
   if  (r <= l) {
        return;
   }
   i = partition (a, l, r);
   quicksort (a, l, i-1);
   quicksort (a, i+1, r);
}
```

QUICK SORT: PARTITIONING

```
int partition (int a[], int l, int r) {
    int i = l-1;
     int j = r;
    int pivot = a[r]; //rightmost is pivot

     for (;;) {
          while ( a[++i] < pivot) ;
          while ( pivot <  a[--j] && j != l);
          if (i >= j) {
               break;
          }
          swap(i,j,a);
     }
    //put pivot into place
    swap(i,r a);
    return i; //Index of the pivot
}
```

Quicksort: Work Complexity

- How many steps?
  - N steps to split array in two
  - Combing the sorted sub-results in constant time
  - Best case (both parts have the same size):
    - $T(N) = N + 2 * T(N/2)$ $O(N * \log N)$
  - Worst case (one part contains all elements):
    - $T(N) = N + T(N-1)$
    - $= N + N-1 + T(N-2)$
    - $= N + N-1 + N-2 + \ldots + 1 = N(N+1)/2$
    - $= O(N^2)$

Quick-sort properties

- It is not adaptive: existing order in the sequence only makes it worse
- It is not stable in our implementation. Can be made stable.
- In-place: Partitioning done in place
  - Recursive calls use stack space of
    - O(N) in worst case
    - O(log N) on average

Quick sort - performance problems

- Taking the first or last element as pivot is often a bad choice
  - sequence might be partially sorted already
    - Already ordered data is a worst case scenario
    - Reverse ordered data is a worst case scenario
  - split into parts of size *N-1* and *0*
- Ideally our pivot would be
    - The median value
- In the worst case our pivot is
    - the largest or smallest value

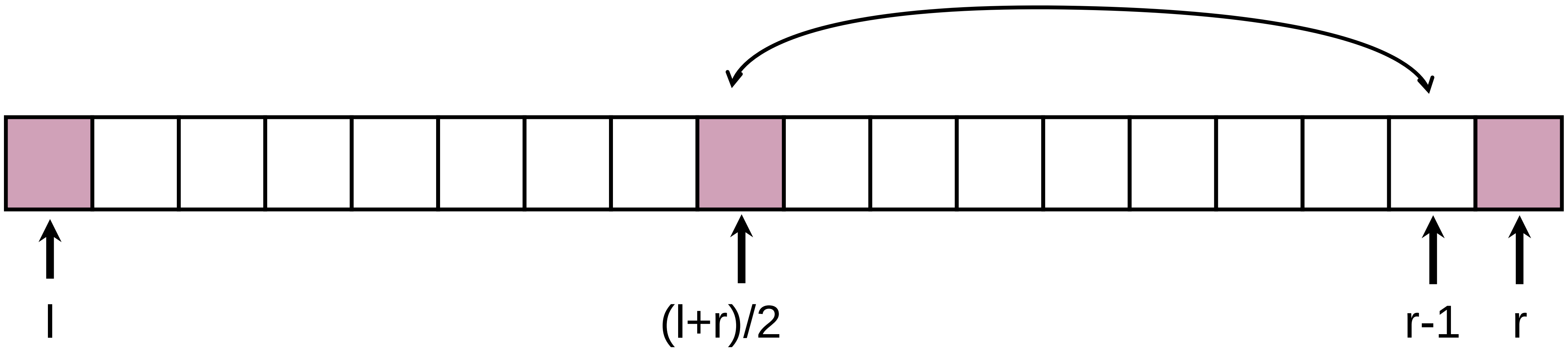
Quick sort choosing better a pivot

- We can reduce the probability of picking a bad pivot
  - picking a random element as the pivot e.g.,

    - int randNum = rand() % (r-l+1);

    - swap elements at (left+randNum) with right-most element before calling partition
  - picking the best out of three (or more)
    - Median of Three partitioning
      - Compare left-most, middle and right-most element
      - Pick the median of these 3 values to be the pivot
  - Does not eliminate the worst case but makes it less likely
  - Ordered data no longer a worst case scenario

Quick sort Median of Three Partitioning- choosing a better pivot



(1) pick a[l],a[r], a[(r+l)/2]

(2) swap a[r-1] and a[(r+l)/2]

(3) sort a[l], a[r-1],a[r] such that a[l]<=a[r-1] <= a[r]

- if a[l] > a[r-1]) then swap a[r-1] and a[l]
- if a[r-1] > a[r]) then swap a[r-1] and a[r]
- if a[l] > a[r-1]) then swap a[r-1]) a[l]

(4) call partition on a[l+1] to a[r-1]

Quick sort: performance and optimisation

- Optimized versions of quick sort are frequently used
- For small sequences, quick sort is relatively expensive because of the recursive calls
  - Quick sort with sub-file cutoff
    - Handle small partitions less than a certain threshold length differently
    - Switch to insertion sort for the small partitions
    - Don’t sort. Leave and do insertion sort at the end
- Use median of five or more elements
- Handling duplicates more efficiently by using three way partitioning.

Quicksort on Linked Lists

- Straight forward to do if we just use first or last element as the pivot
  - Picking the pivot via randomisation or median of 3 is now O(n) instead of O(1).

Quick Sort vs Merge Sort

- On typical modern architectures, **efficient** quicksort implementations generally outperform mergesort for sorting RAM-based arrays.
  - Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.
- On the other hand, merge sort is a stable sort, parallelizes better, and is more efficient at handling slow-to-access sequential media. Merge sort is often the best choice for sorting a linked list and the merging can be done without using extra space that is used during merge for arrays.

How fast can a sort become?

- All the sorts we have seen so far have been comparison based sorts
  - find order by **comparing elements in the sequence**
  - can sort any type of data as long as there is a way to compare 2 items
- Theoretical lower bound on worst case running time of comparison based sorts
  - O(nlog(n)).
  - Algorithms such as quicksort and mergesort are really about as fast as we can go for unknown types of data.

Sorting has a theoretical nlogn lower bound

- If there is 3 items, then 3! = 6 possible permutations or 6 possible different inputs
  - If there are n items, then n! possible permutations or inputs
- If we do 1 comparison we can divide into 2 different categories
  - If we do k comparisons we can divide into 2^k different categories
- We need to do enough comparisons so
  - $n! <= 2^K$
    - $\log n! <= \log 2^k$
    - $\log n! <= k$
    - $n \log n <= k$ (using stirling's approximation)

Non-comparison Based Sorting

- Non-comparison based sorting
  - We may not actually have to compare pairs of elements to sort the data.
- Specialised sorts can be implemented if additional information about the data to be sorted is known.
  - Take advantage of special properties of keys
- We can do some kinds of sorts in linear time!

Key Indexed Counting Sort

- Basic Idea:
  - Using an array, count up number of times each key appears
  - Use this information as an index of where the item belongs in the final sorted array
  - Place items in the final sorted array based on their index
- For example: Sorting numbers from 0..10
  - If I knew there were three 0's and two 1's
    - If I had a 2, it would go at index 5
    - If I got another 2, it would go at index 6.

KEY INDEXED COUNTING SORT

- May work in O(n) time. How?
  - Because it uses **no** comparisons!
  - But we have to make assumptions about the size and nature of the data
- Assumptions
  - Sequence of size N
  - Each key is in the range of 0 - M-1
- Time Complexity
    - Efficient if M is not too large compared to N
    - O(n + M) - Not good in cases like : 1,2,999999
- In-place? No. Uses temporary arrays of O(n+M)
- Is stable

Radix Sorting

- Comparison based sorting:
  - Sorting based on comparing two whole keys
- Radix sorting:
  - Processing keys one piece at a time
- Keys are treated as numbers represented in base-R (radix) number system
  - Binary numbers R is 2
  - Decimal numbers R is 10
  - Ascii strings R is 128 or 256
  - Unicode strings R is 65,536
- Sorting is done individually on each digit in the key on at a time – digit by digit or character by character

Radix Sort LSD (Least significant digit first)

- Consider characters or digits or bits from **Right to Left** (ie from least significant)
- **Stably** sort using **dth digit as the key**
  - Can use Key Indexed Counting sort.
  - For example: sorting 1019, 2301, 3129, 2122

101**9**, 230**1**, 312**9**, 212**2** -> 230**1**, 212**2**, 101**9**, 312**9**

23**0**1, 21**2**2, 10**1**9, 31**2**9 -> 23**0**1, 10**1**9, 21**2**2, 31**2**9

2**3**01, 1**0**19, 2**1**22, 3**1**29 -> 1**0**19, 2**1**22, 3**1**29, 2**3**01

1019, **2**122, **3**129, **2**301 -> **1**019, **2**122, **2**301,**3**129

RADIX SORT LSD PROPERTIES

- $O(w(n+R))$
  - w is the width of the data ie 987 is 3 digits wide, "aaa" is 3 characters, integers (binary rep) could have w as 32 and R of 2
  - The algorithm makes w passes over all n keys.
- Not in place: extra space: $O(n + R)$
- Stable
- Can modify to use for variable length data
- Imagine sorting strings like
  - "zaaaaaaa" and "aaaaaaaa"
- Can spend lots of work comparing insignificant details

Radix sort MSD (Most Significant Digit first)

- Partition file into R pieces according to first character
  - Can use key-indexed counting
- Recursively sort all strings that start with each character
  - key-indexed counts delineate files to sort
- O(w(n+R)) – in worst case
- Extra space N + DR (D is depth of recursion)
- Don't have to go through all of the digits to get a sorted array. This can make MSD radix sort considerably faster
- Can use insertion sort for small subfiles