Sorting

Department of Electrical and Computer Engineering
McMaster University

January 2020

Textbook Reading

- Chapter 8

Sorting

- Along with searching, sorting is a central operation in computer systems to **organize** quantities of data.
- In this topic we will look at a number of algorithms to sort arrays of "elements".
  - → In this section we will consider sorting arrays of integers
  - → The algorithms can be extended to other data types.
  - → Assume that operators for "==", ">" and "<" are provided i.e., `compareTo` method available – **comparison based sorting**.
  - → Assume that your sorting algorithm knows the number of elements in the array (usually denoted as n).

Sorting

We will consider a number of sorting algorithms:

- Simple algorithms - $\Theta(n^2)$ time at worst and average.
  - → Bubble Sort
  - → Insertion Sort - hidden constant is small; very efficient for **small** n.
- $\Theta(n \log n)$ time on average - best running time for **comparison-based** sorting
  - → Merge Sort
  - → Quick Sort - hidden constant is small; very efficient in practice for **large** n.
  - → Heap Sort - to be discussed in Part II.
- $\Theta(n)$ time at worst and average - with **some assumptions** on the input
  - → Counting Sort
  - → Radix Sort

We will consider a number of sorting algorithms:

- Simple algorithms - $\Theta(n^2)$ time at worst and average.
  - Bubble Sort
  - Insertion Sort - hidden constant is small; very efficient for **small** n.
- $\Theta(n \log n)$ time on average - best running time for **comparison-based** sorting
  - Merge Sort
  - Quick Sort - hidden constant is small; very efficient in practice for **large** n.
  - Heap Sort - to be discussed in Part II.
- $\Theta(n)$ time at worst and average - with **some assumptions** on the input
  - Counting Sort
  - Radix Sort
- For short lists are lists which are partially sorted, the simple algorithms may suffice, however, in the general case they perform poorly (to be seen later)

Classes of Sorting Algorithms

- **Internal Sorting**
  - → The data set fits in the main memory and elements can be read or written at any time
- **External Sorting**
  - → The data set is too large to fit in main memory and is usually stored on a disk or a tape. Used when you can only access blocks of the data set at a given time due to the limitations of main memory.
- ☞ We will consider both cases but focus on internal sorting techniques.

Inversions

Definition: An **inversion** in an array of numbers is any ordered pair $(a[i], a[j])$ such that $i < j$ and $a[i] > a[j]$.

Example: Consider the array 5, 20, 14, 23, 1, 6
The inversions are:

(5,1), (20,14), (20,1), (20,6), (14,1), (14,6), (23,1), (23,6)

Notes:

- The inversions represent elements which are out of place in the list.
- The goal of sorting algorithms is to reduce (or eliminate) the number of inversions
- Here we consider sorting in ascending order

Sorting Algorithm Stability

Definition: A sorting algorithm is **stable** if for $a[i] = a[j]$ and $i < j$ in the original list then in the sorted list, in their new locations $p(i)$, $p(j)$, $p(i) < p(j)$

Example: Consider the input array of marks

Bobby	A
Franky	C
Suzy	A
Loni	B

Sort by mark:

Unstable

Suzy	A
Bobby	A
Loni	B
Franky	C

Stable

Bobby	A
Suzy	A
Loni	B
Franky	C

- Stable algorithms are ones when two keys have the same value the **original** order is preserved.

Types of Internal Sorting Algorithms

- **Exchange Techniques** - exchange elements in the list to reduce the number of inversions
  - → Bubble Sort, Quick Sort
- **Selection Techniques** - select the next largest element in the list and add it to the sorted list
  - → Selection Sort, Heap Sort
- **Insertion Techniques** - take elements one at a time and add them to the sorted list
  - → Insertion Sort, Shell Sort
- **Merging Techniques** - combine or merge two sorted lists into a single list
  - → Merge Sort
- **Special Cases** - if the range of the integers is known beforehand, performance can be improved
  - → Radix Sort, Bucket Sort

Bubble Sort

Basic Idea: Move through the list swapping adjacent elements that are inverted until the list is sorted.

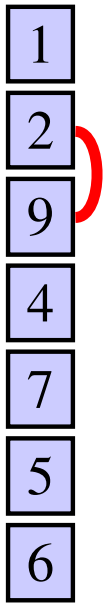
```
1  public void bubble(int[] list){
2      int i;
3      boolean done=false;
4
5      while(done == false){
6          done=true;
7          for(i=1 ;i < list.length ; i++){
8              if(list[i-1]>list[i]){
9                  done=false;
10                 swap(list,i-1,i);
11             }
12         }
13     }
14 }
```

☞ An example of an **exchange technique** sorting algorithm

Example:

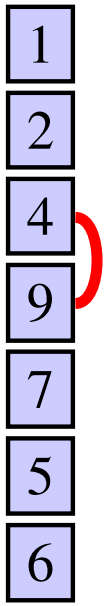
Note:
Top is beginning of array.

Example:



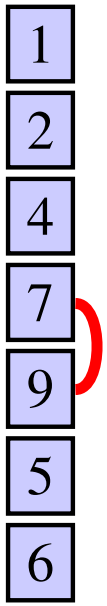
First pass:
Swaps: $(9, 2)$

Example:



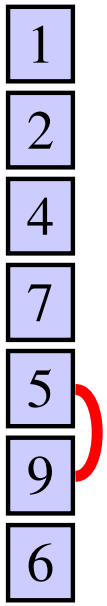
First pass:
Swaps: $(9, 2)$, $(9, 4)$

Example:



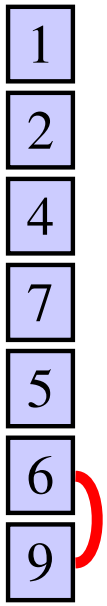
First pass:
Swaps: $(9, 2)$, $(9, 4)$, $(9, 7)$

Example:



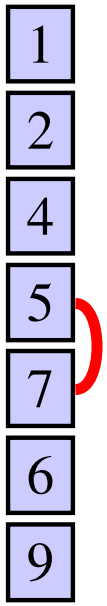
First pass:
Swaps: $(9,2)$, $(9,4)$, $(9,7)$, $(9,5)$

Example:



First pass:
Swaps: $(9,2), (9,4), (9,7), (9,5), (9,6)$ - First pass done.

Example:



Second pass:
Swaps: $(7, 5)$

Example:

Second pass:
Swaps: $(7,5)$, $(7,6)$ - Second pass done.

Example:

Third pass:
Swaps: No swaps needed. Therefore, list sorted. **STOP**

Bubble Sort

- This is a simple but inefficient sorting algorithm.
- The large elements sink the bottom of the array while the smallest elements bubble to the top.
- There are at most $n-1$ iterations of outer loop each with at most $n-1$ swaps
  - → **Time**: $\Theta(n^2)$ worst and average case[1], $\Theta(n)$ best case if input sorted.
  - → **Space**: $\Theta(1)$ extra space required
- Number of swaps: $\Theta(n^2)$ at worst and average, 0 at best.

[1]For the average case analysis it is assumed that: (1) the n array elements are all integers from 1 to n. (2) Different possible arrays correspond to different possible permutations of these n elements. (3) All permutations are equally likely.

Selection Sort

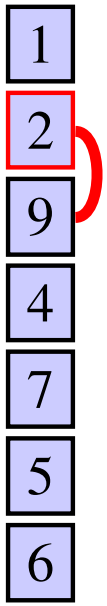
Basic Idea: Move through the list and find next smallest element in the list and move to beginning of the list.

```
1  public void selection(int[] list){
2      int i,j,min;
3      for(i=0; i < list.length-1 ; i++){
4          min=i;
5          for(j=i+1 ; j < n ; j++)
6              if(list[min]>list[j])
7                  min=j;
8          swap(list,i,min);
9      }
10 }
```

☞ An example of a **selection technique** sorting algorithm

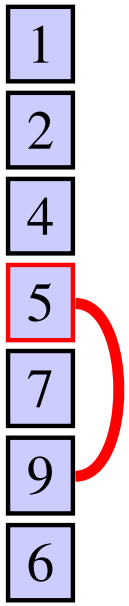
Example:

Example:

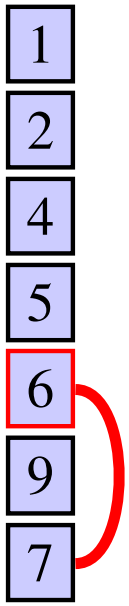


Example:

Example:



Example:



Example:

Example: **Done**

Selection Sort

- This sort is not much better than Bubble Sort
  - → The only advantage is that the number of swaps is fixed in selection sort. This is advantage is swaps are very expensive, e.g, when dealing with large entries in the list.
- The outer loop runs $n-1$ times, and the number of operations is

$$\sum_{i=1}^{n-1} i = \Theta(n^2)$$

  - → **Time**: $\Theta(n^2)$ worst, best and average case.
  - → **Space**: $\Theta(1)$ extra space required
- Number of swaps: $\Theta(n)$ at worst and average, 0 at best.

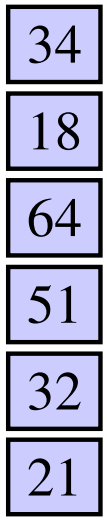
Insertion Sort

Basic Idea: Look at entries one at a time and place them in the correct spot relative to the previous entries

```
1  public void insertion(int[] list){
2      int i,j,temp;
3      for(i=1 ; i < length.list ; i++){
4          temp=list[i];
5          for( j = i ; j > 0 && list[j-1] > temp ; j--)
6              list[j] = list[j-1];
7          list[j] = temp;
8
9      }
10 }
```

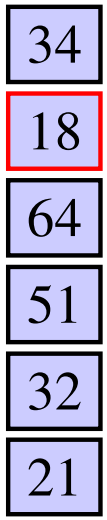
- This technique is analogous to sorting a deck of cards.
- An example of a **insertion technique** sorting algorithm

Example:



→ There are 9 inversions: (34,8), (34,32), (34,21), (64,51), (64,32), (64,21), (51,32), (51,21), (32,21)

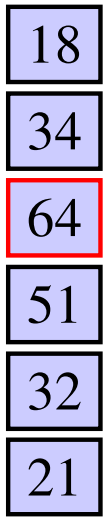
Example:



→ $i = 1$

→ To place $a[i]$ in correct spot relative to previous elements, 1 element needs to be shifted.

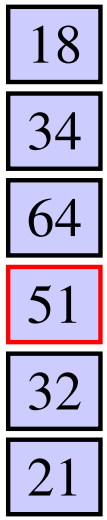
Example:



→ $i = 2$

→ To place $a[i]$ in correct spot relative to previous elements, 0 elements need to be shifted.

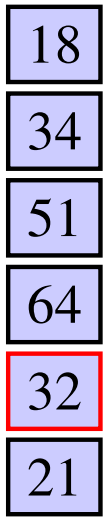
Example:



→ $i = 3$

→ To place $a[i]$ in correct spot relative to previous elements, 1 element needs to be shifted.

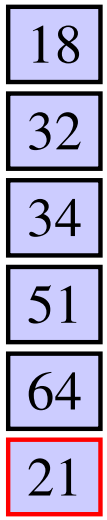
Example:



→ $i = 4$

→ To place $a[i]$ in correct spot relative to previous elements, 3 elements need to be shifted.

Example:



→ $i = 5$

→ To place $a[i]$ in correct spot relative to previous elements, 4 elements need to be shifted.

Example: **Done**

→ In all 9 shifts required - equal to the number of inversions!

Insertion Sort

- Notice that the number of shifted elements (virtual swaps) **equals** the number of inversions.
- The outer loop runs $n-1$ times, and the number of operations to make space and move an element into the correct location is at most n operations
  - → **Time**: $\Theta(n^2)$ worst and average case. At best, $\Theta(n)$ if the list is sorted.
  - → **Space**: $\Theta(1)$ extra space required
- Number of swaps: $\Theta(n^2)$ at worst and average, 0 at best.

- Poor performance for large or unsorted lists.
- **However**, this algorithm is **very fast** if the list is **almost sorted** or if the list is short (n small), because the constant hidden in the big-theta notation is small.

Limitations of Simple Sorting Algorithms

- Simple sorting algorithms are mostly based on repeatedly **swapping adjacent elements** to reduce the number of inversions (bubble sort, insertion sort).
- Algorithms based on repeatedly swapping adjacent elements have running time $\Omega(n^2)$ on average.
  - → A swap eliminates only one inversion.
  - → If I denotes the number of inversions, then running time is $\Omega(I)$.
  - → The average value of I can be shown to be $n(n-1)/4 = \Theta(n^2)$.
- ☞ Note, if the list is already sorted (i.e., there are no inversions), it still takes $\Theta(n)$ time to verify this

Limitations of Simple Sorting Algorithms

- No matter what you do, these simple algorithms require $\Omega(n^2)$ time to run on average (can be proved)
- To improve performance need to eliminate more than one inversion per operation
- Perhaps a good idea is to look at elements which are further apart in an effort to increase the number of inversions we eliminate in each step?
  - This is the basic idea of **Shell Sort**.

Shell Sort

- Insertion Sort is quite fast when the list is nearly sorted
  - → The problem is that with each swap insertion sort only removes a single inversion
- **Key Idea**: Why not consider modifying insertion sort to consider elements which are further apart in an effort to remove more than one inversion per swap?
  - → this is the basic idea behind **Shell Sort** invented by Donald Shell.

Shell Sort

- Rather than performing insertion sort on the entire list, consider breaking the problem into sorting several disjoint shorter lists.
  - → Start off sorting in many small lists where the elements are far apart
  - → Continue sorting with a smaller number of sub-lists each with more elements.

Shell Sort

- Rather than performing insertion sort on the entire list, consider breaking the problem into sorting several disjoint shorter lists.
  - Start off sorting in many small lists where the elements are far apart
  - Continue sorting with a smaller number of sub-lists each with more elements.
- **Definition**: Consider the sub-list formed from elements that are h_k elements apart
  - If $h_k = 4$ then every 4^{th} element is grouped together
  - Example: For the list $0, 1, 2, 3, \ldots, n$ with $h_k = 4$ there are 4 sub-lists:
    - $0, 4, 8, \ldots$
    - $1, 5, 9, \ldots$
    - $2, 6, 10, \ldots$
    - $3, 7, 11, \ldots$

Algorithm:

- Define a series of increments $h_1, h_2, \ldots, h_t$ where $h_1 = 1$ and $h_t > h_{t-1} > \cdots > h_1$.
- For each increment, break the list into a series of disjoint h_k sub-lists.
  - → In each sub-list the elements are the same distance apart, i.e., h_k elements apart.
- For $k = t, (t-1), \ldots, 1$, perform Insertion sort on each of the h_k sub-list

Algorithm:

- Define a series of increments $h_1, h_2, \ldots, h_t$ where $h_1 = 1$ and $h_t > h_{t-1} > \cdots > h_1$.
- For each increment, break the list into a series of disjoint h_k sub-lists.
  - → In each sub-list the elements are the same distance apart, i.e., h_k elements apart.
- For $k = t, (t-1), \ldots, 1$, perform Insertion sort on each of the h_k sub-list

- On initial passes, the algorithm considers many sub-lists with few elements.
- On subsequent passes, the number of sub-lists decreases, but the number of elements in each sub-list increases.
- Notice that since $h_1 = 1$ the last step is the regular insertion sort
- This is why this sorting technique is sometimes referred to as **diminishing increment sort**.

Shell Sort – Example

- Sort the following list using Shell Sort with increments $h_k = 1, 3, 5$

63	43	82	21	19	11	25	75	04	17	60	91	13

Shell Sort – Example

- For increment $h_k = 5$, break the entire list into five sub-lists

Shell Sort – Example

- Now, sort each sub-list using insertion sort ($h_k = 5$).

Shell Sort – Example

- Set increment to $h_k = 3$, and break the list into three sub-lists

Shell Sort – Example

- Sort each sub-list using insertion sort ($h_k = 3$).

Shell Sort – Example

- Now, increment is $h_k = 1$ and there is only one sub-list.

04	17	13	11	25	21	19	63	60	43	75	91	82

Shell Sort – Example

- Sort the entire list with insertion sort ...

04	11	13	17	19	21	25	43	60	63	75	82	91

Shell Sort – Example

- Notice that at each step the list looks increasingly "sorted"
- The last pass of the algorithm is the Insertion sort considered earlier.
  - → However, notice that the performance of the insertion sort at the end is greatly improved by the preprocessing steps

Shell Sort – Example

- Notice that at each step the list looks increasingly "sorted"
- The last pass of the algorithm is the Insertion sort considered earlier.
  - → However, notice that the performance of the insertion sort at the end is greatly improved by the preprocessing steps

Definition: A list is h_k **sorted** if

$$a(i) < a(i + h_k)$$

- The algorithm relies on the fact that for an h_k sorted list **remains** h_k sorted even after it is h_{k-1} sorted, i.e., the algorithm never undoes any of the progress made by earlier steps.

Shell Sort – Increments

- The performance of Shell sort depends on how the increment sequence is selected.
  - → In fact, any increment will work as long as $h_1 = 1$
- **Shell's Increments**
  - → $h_t = \lfloor n/2 \rfloor$ and $h_k = \lfloor h_{k+1}/2 \rfloor$.
- **Hibbard's Increments**
  - → use the elements of the sequence $1, 3, 7, 15, \ldots, 2^k - 1$
  - → consecutive elements do not share common factors (relatively prime).
- **Division by Three**
  - → $h_1 = 1$, $h_k = 3h_{k-1} + 1$
  - → Example: $1, 4, 13, 40, 121, \ldots$
  - → Consecutive elements are relatively prime
- there are many possible increments with varying performance ...

Shell Sort

- Shell sort is easy to code. It is just a modified version of insertion sort

```
1  public void shell(int[] list){
2      int incr=list.length;
3      while(incr=increment(incr)){
4          insert2(list, incr);
5      }
6  }
7  public void insert2(int[] list, int incr){
8      int i,j,temp;
9      for(i=incr; i < list.length; i++){
10         temp=list[i];
11         for(j = i; j >= incr && list[j-incr] > temp; j-=incr)
12             list[j] = list[j-incr];
13         list[j] = temp;
14     }
15 }
```

- The function `increment()` returns the next increment value

Shell Sort – Performance

- In general, the average performance of Shell sort is not known in closed form for any but the simplest increment sequence.

Using Shell's increments, the worst case is when the list length is a power of 2, i.e., $n = 2^m$.

- For each pass with increment h_k, there are h_k sorts on n/h_k elements
- Since insertion sort is $\Theta(p^2)$ worst case for a list of length p, each pass requires

$$O\left(h_k\left(\frac{n}{h_k}\right)^2\right)$$

- Now, for t increments, the total work required is

$$O\left(\sum_{k=1}^{t}\frac{n^2}{h_k}\right) = O\left(n^2\sum_{k=1}^{t}\frac{1}{h_k}\right)$$

- Since, $\sum 1/h_k = 1/(n/2) + 1/(n/4) + \ldots + 1/4 + 1/2 + 1/1 < 2$
- Therefore, the worst case is $\Theta(n^2)$.

Shell Sort – Performance

- In general, the average performance of Shell sort is not known in closed form for any but the simplest increment sequence.

Using Shell's increments, the worst case is when the list length is a power of 2, i.e., $n = 2^m$.

- For each pass with increment h_k, there are h_k sorts on n/h_k elements
- Since insertion sort is $\Theta(p^2)$ worst case for a list of length p, each pass requires

$$O\left(h_k\left(\frac{n}{h_k}\right)^2\right)$$

- Now, for t increments, the total work required is

$$O\left(\sum_{k=1}^{t}\frac{n^2}{h_k}\right) = O\left(n^2\sum_{k=1}^{t}\frac{1}{h_k}\right)$$

- Since, $\sum 1/h_k = 1/(n/2) + 1/(n/4) + \ldots + 1/4 + 1/2 + 1/1 < 2$
- Therefore, the worst case is $\Theta(n^2)$.
- Using **Hibbard's increments** it can be shown that the worst case performance is $\Theta(n^{\frac{3}{2}})$

Shell Sort

- Unlike exchange techniques, Shell sort makes comparisons and swaps between non-adjacent elements in the list.
  - → This is one of the first algorithms to break the quadratic time complexity bound.
- Shell sort makes the list mostly sorted, and then uses Insertion sort to make the list sorted.
- There is a gain in performance if the increments are chosen so that there is a **mixing** of elements in each of the sub-lists.
  - → This is precisely why choosing relatively prime increment values is a good idea
- For small and mid-sized lists, shell sort performs competitively with more advanced algorithms with asymptotically better performance (to be seen shortly).

- Comparison of running times for Insertion sort and Shell sort [2]

N	Insertion Sort	Shell Sort (Shell's Increments)	Shell Sort (Dividing by 2.2)
10,000	575	10	9
20,000	2,489	23	20
40,000	10,635	51	41
80,000	42,818	114	86
160,000	174,333	270	194
320,000	N/A	665	451
640,000	N/A	1,593	939

[2] Taken from textbook, pg.311, Fig. 8.6

Efficient Sorting Algorithms

Merge Sort

- Invented by John von Neumann in 1945
- This technique is based on the **merging** or combining of two sorted lists into a single sorted list.

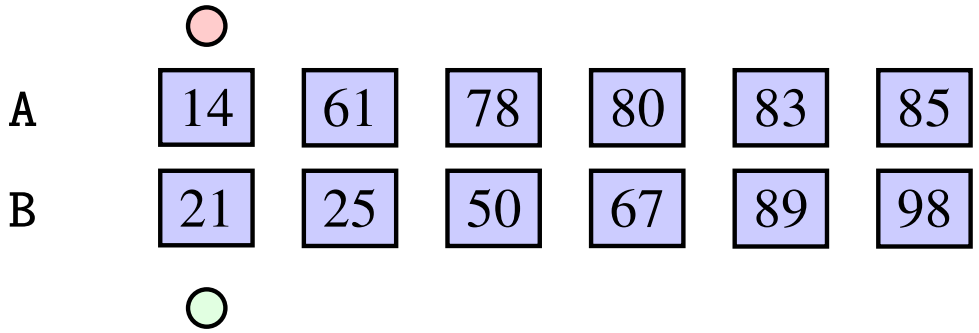
Example: Merge the sorted lists `A` and `B`

`A`	14	61	78	80	83	85
`B`	21	25	50	67	89	98

- What is a good way to do this?

- Have two pointers, one in each sub-list, and copy the smallest element that is being pointed to a temporary array

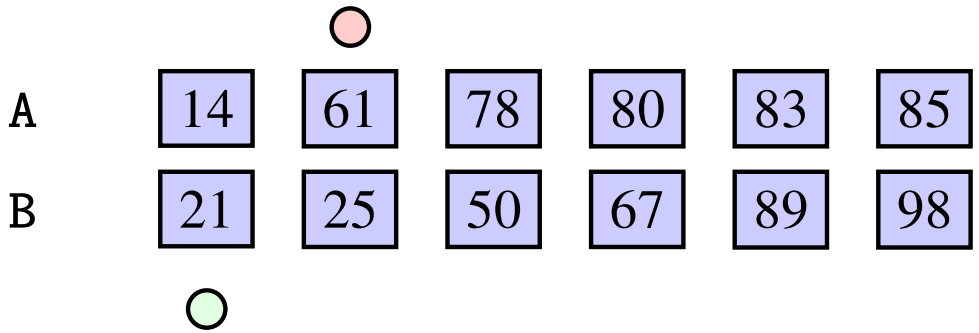
Example: Merge the sorted lists `A` and `B`



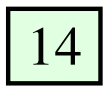
Temporary Array:

- Have two pointers, one in each sub-list, and copy the smallest element that is being pointed to a temporary array

Example: Merge the sorted lists `A` and `B`

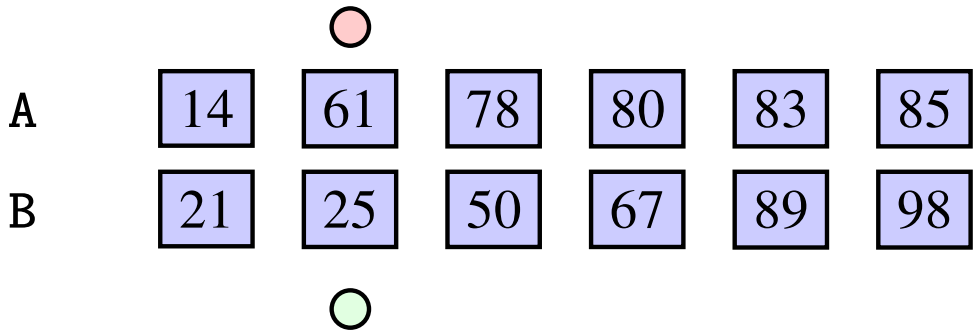


Temporary Array:

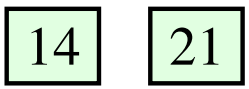


- Have two pointers, one in each sub-list, and copy the smallest element that is being pointed to a temporary array

Example: Merge the sorted lists `A` and `B`

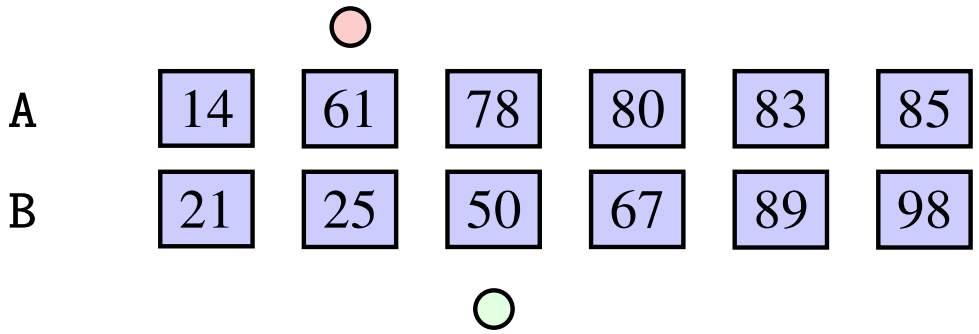


Temporary Array:



- Have two pointers, one in each sub-list, and copy the smallest element that is being pointed to a temporary array

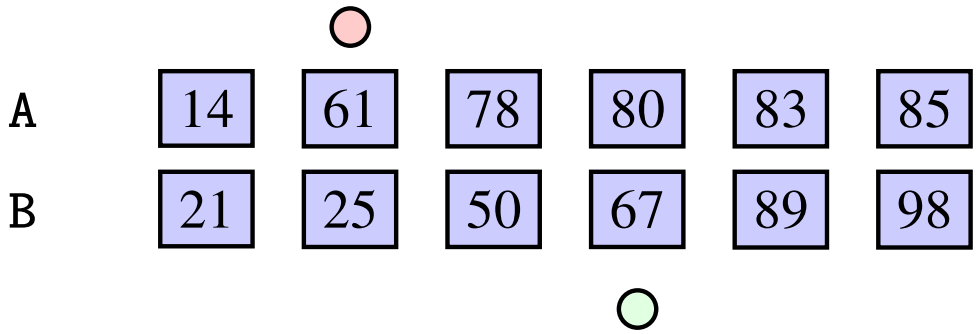
Example: Merge the sorted lists `A` and `B`



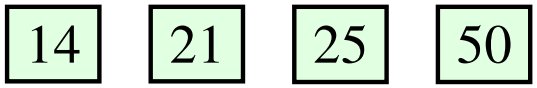
Temporary Array:

- Have two pointers, one in each sub-list, and copy the smallest element that is being pointed to a temporary array

Example: Merge the sorted lists `A` and `B`

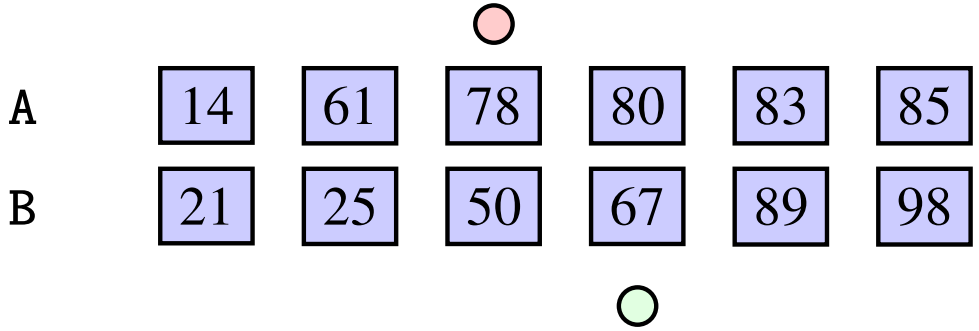


Temporary Array:



- Have two pointers, one in each sub-list, and copy the smallest element that is being pointed to a temporary array

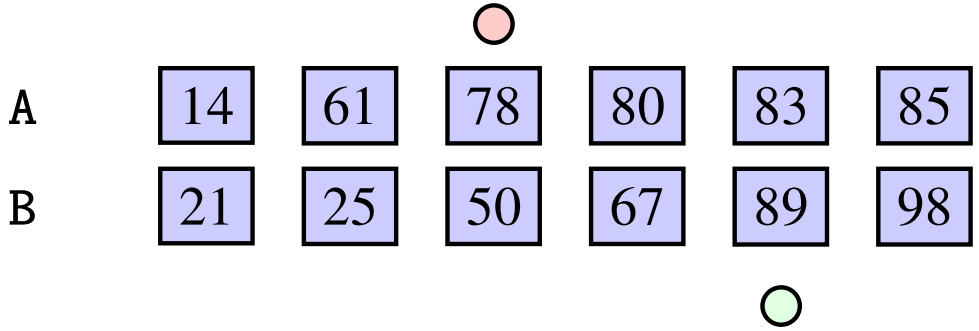
Example: Merge the sorted lists `A` and `B`



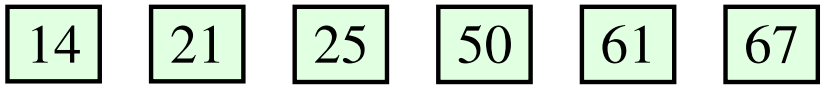
Temporary Array:

- Have two pointers, one in each sub-list, and copy the smallest element that is being pointed to a temporary array

Example: Merge the sorted lists `A` and `B`

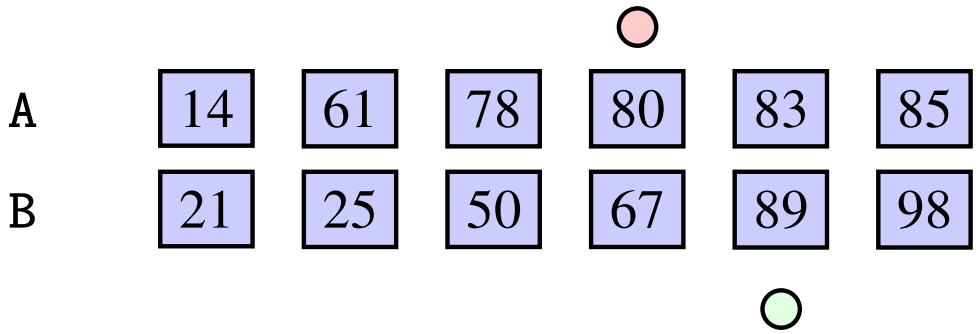


Temporary Array:

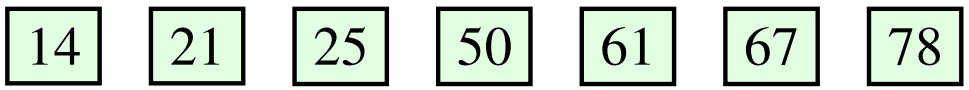


- Have two pointers, one in each sub-list, and copy the smallest element that is being pointed to a temporary array

Example: Merge the sorted lists `A` and `B`

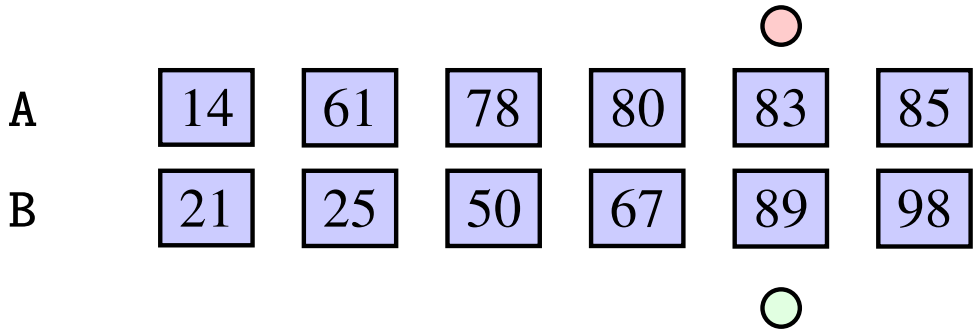


Temporary Array:

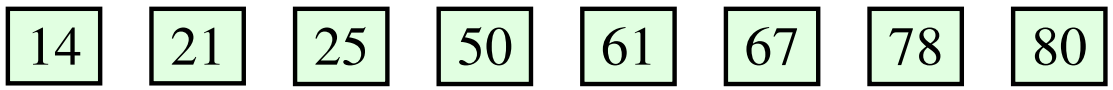


- Have two pointers, one in each sub-list, and copy the smallest element that is being pointed to a temporary array

Example: Merge the sorted lists `A` and `B`

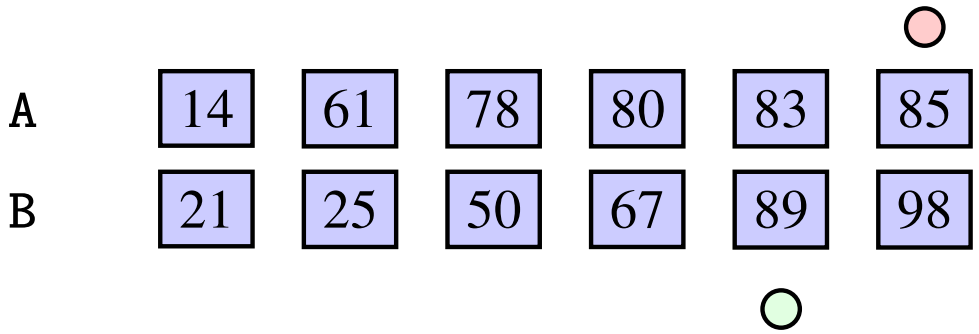


Temporary Array:

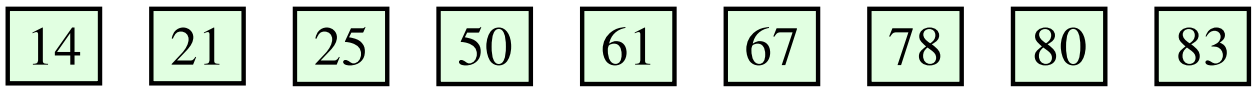


- Have two pointers, one in each sub-list, and copy the smallest element that is being pointed to a temporary array

Example: Merge the sorted lists `A` and `B`

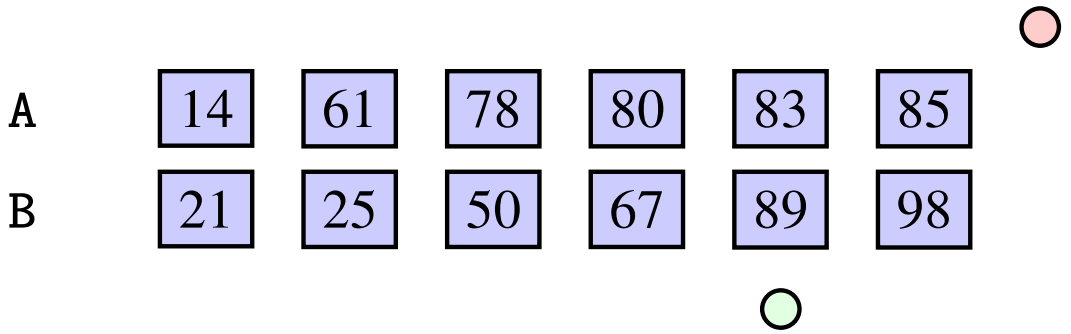


Temporary Array:

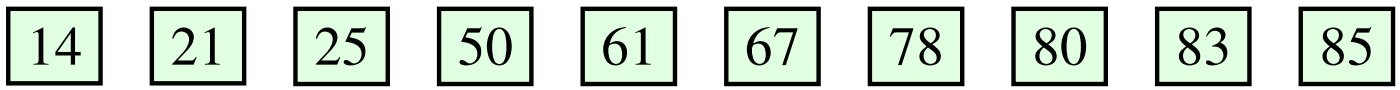


- Have two pointers, one in each sub-list, and copy the smallest element that is being pointed to a temporary array

Example: Merge the sorted lists `A` and `B`

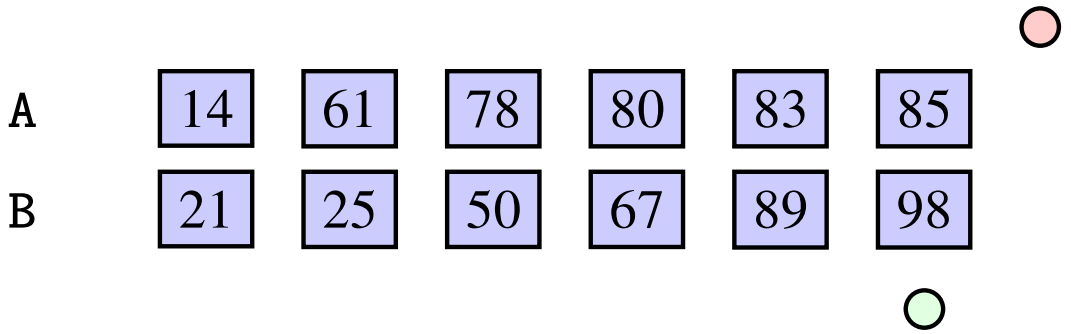


Temporary Array:

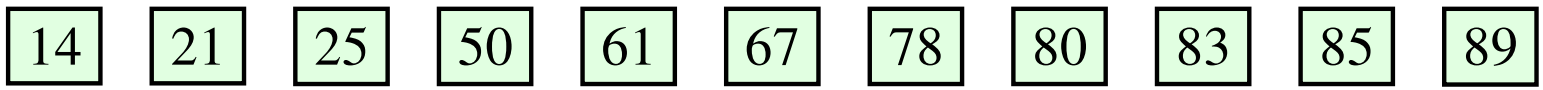


- Have two pointers, one in each sub-list, and copy the smallest element that is being pointed to a temporary array

Example: Merge the sorted lists `A` and `B`

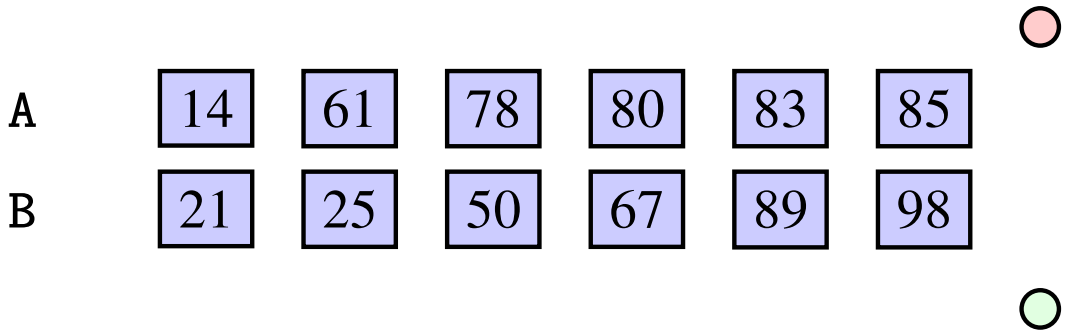


Temporary Array:

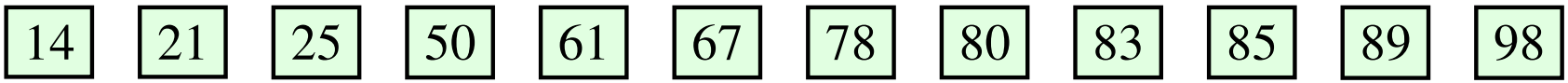


- Have two pointers, one in each sub-list, and copy the smallest element that is being pointed to a temporary array

Example: Merge the sorted lists `A` and `B`



Temporary Array:



Done

- Requires $\Theta(n)$ operations to merge two lists (where n is the total number of elements)
  - At most $n - 1$ comparisons between elements in the two sub-lists

Merge Sort Algorithm

- Problem: Sort an array of n integers.
- **Merge Sort Algorithm**:
  - If $n = 1$, STOP (the list is sorted since there is only one element)
  - Else
    - 1) Recursively Merge Sort left half.
    - 2) Recursively Merge Sort right half.
    - 3) Merge the two sorted sub-arrays into a single sorted array.

Merge Sort

- A C version of the Merge sort algorithm

```
1  void msort(int *a, int *tempA, int left, int right){
2      int center;
3      if(left<right){
4          center=(left+right)/2;                    /* middle of list */
5          msort(a,tempA,left,center);               /* merge sort left half */
6          msort(a,tempA,center+1,right);            /* merge sort right half */
7          merge(a,tempA,left,center+1,right); /* merge sorted lists
8                                                 left to center and center+1 to right */
9      }
10 }
11 void mergesort(int *a, int n){
12     int *tempA=(int *)malloc(n*sizeof(int));
13     if(tempA){
14         msort(a,tempA,0,n-1);
15         free(tempA);
16     } else exit(1);
17 }
```

Merge Sort – Example 1

Example: Sort the list

19	28	10	61	42	17	12

Merge Sort on each half of the list recursively ...

Merge Sort – Example 1

Example: Sort the list

19	28	10	61	42	17	12

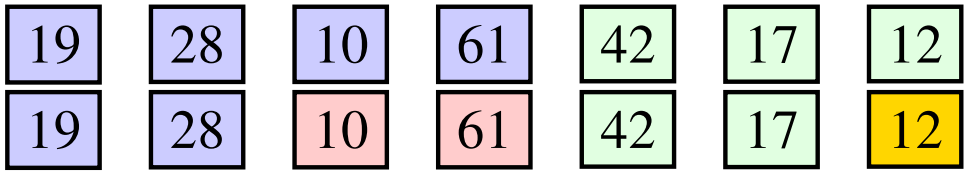
Merge Sort on each half of the list recursively ...

19	28	10	61	42	17	12

Merge Sort – Example 1

Example: Sort the list

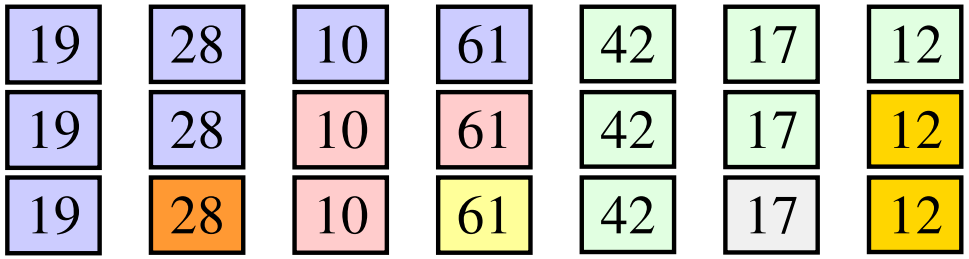
Merge Sort on each half of the list recursively ...



Merge Sort – Example 1

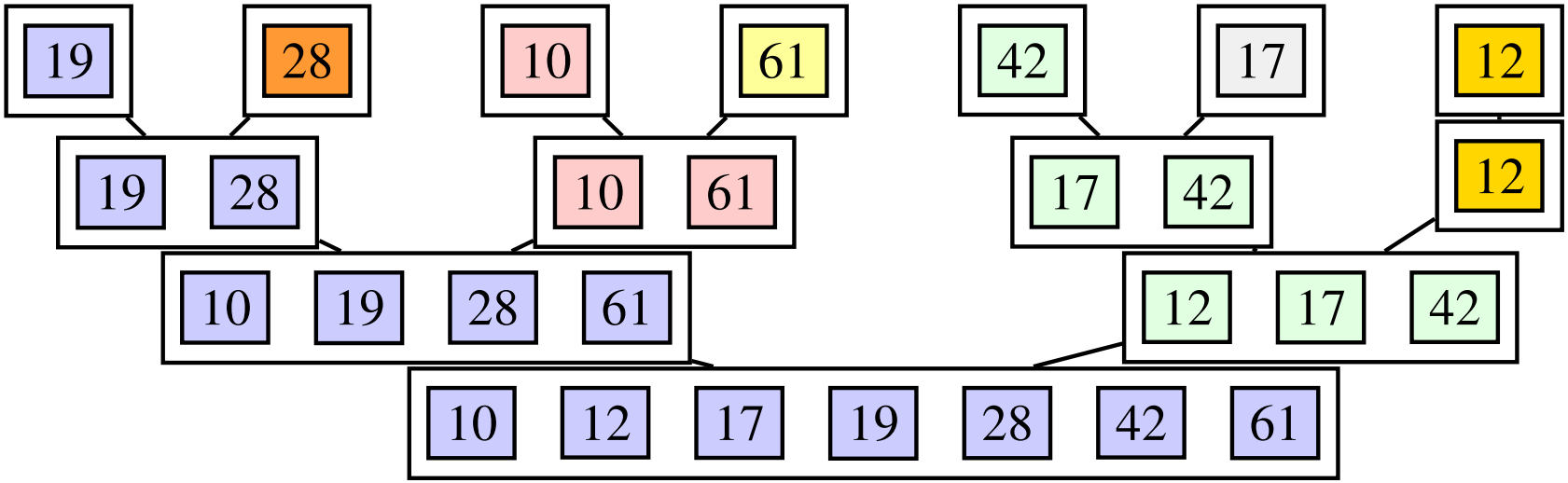
Example: Sort the list

Merge Sort on each half of the list recursively ...



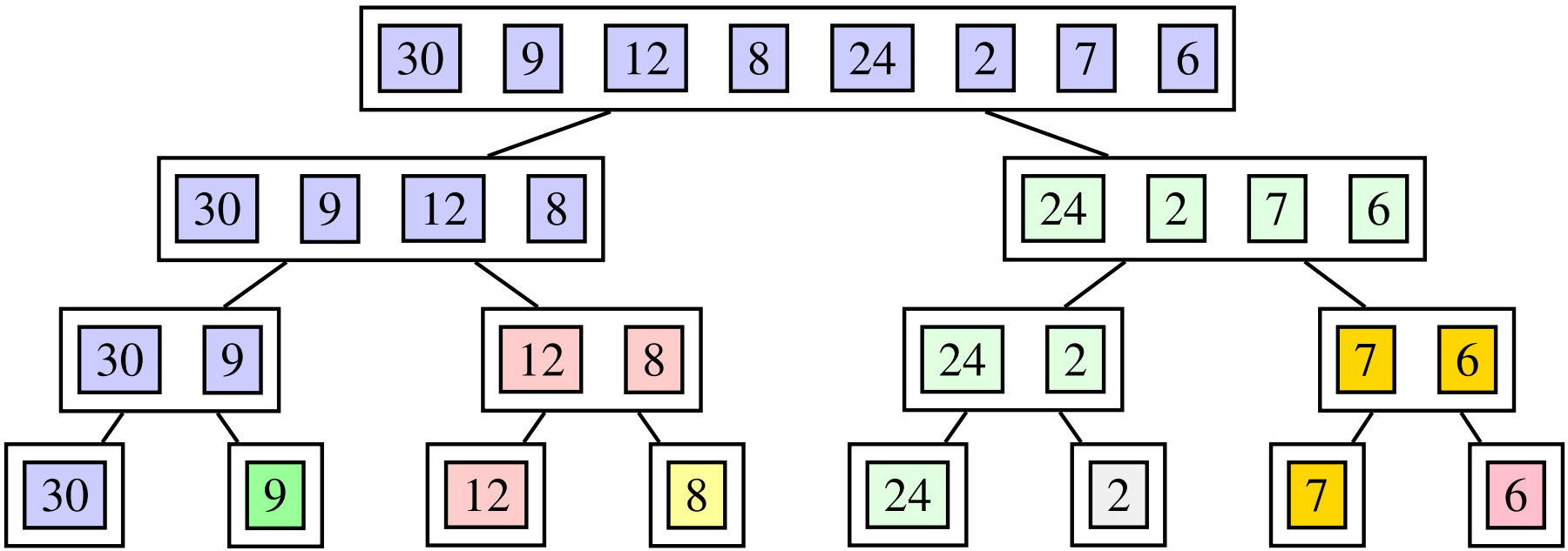
Merge Sort – Example 1

- Now, merge sort on each sub-list



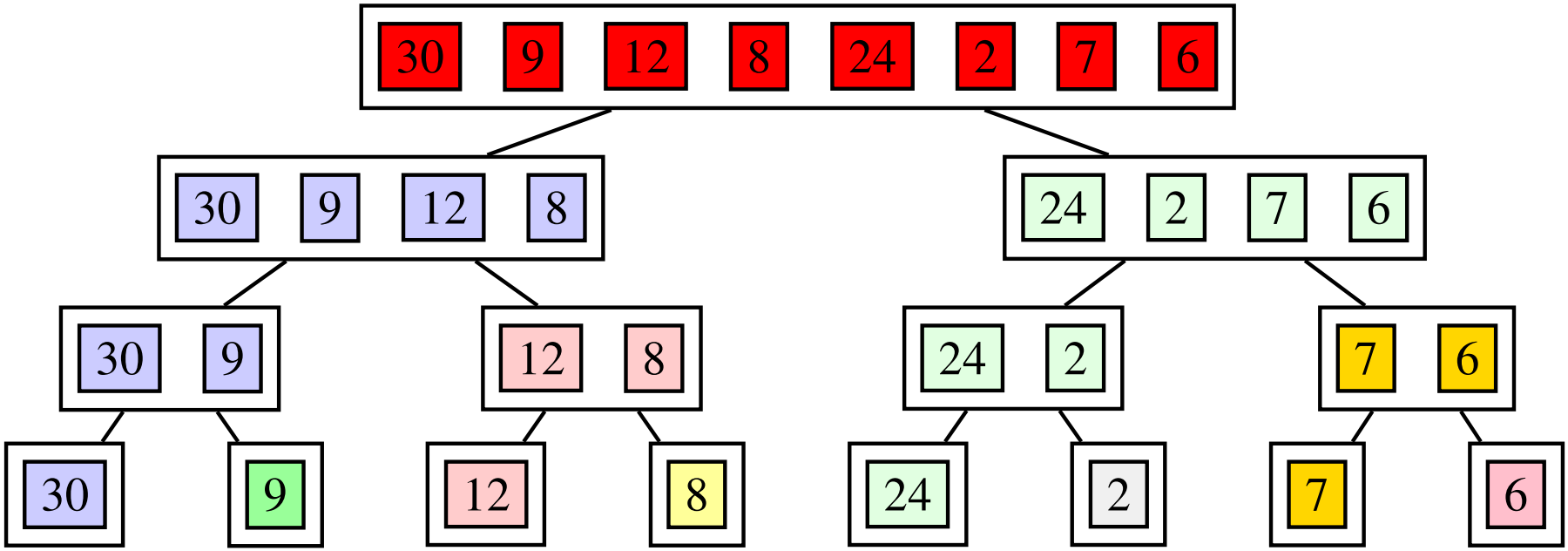
Merge Sort – Example 2

- Let us trace merge sort on an example.
- On the following slides colour red marks the current array considered at step of recursion.



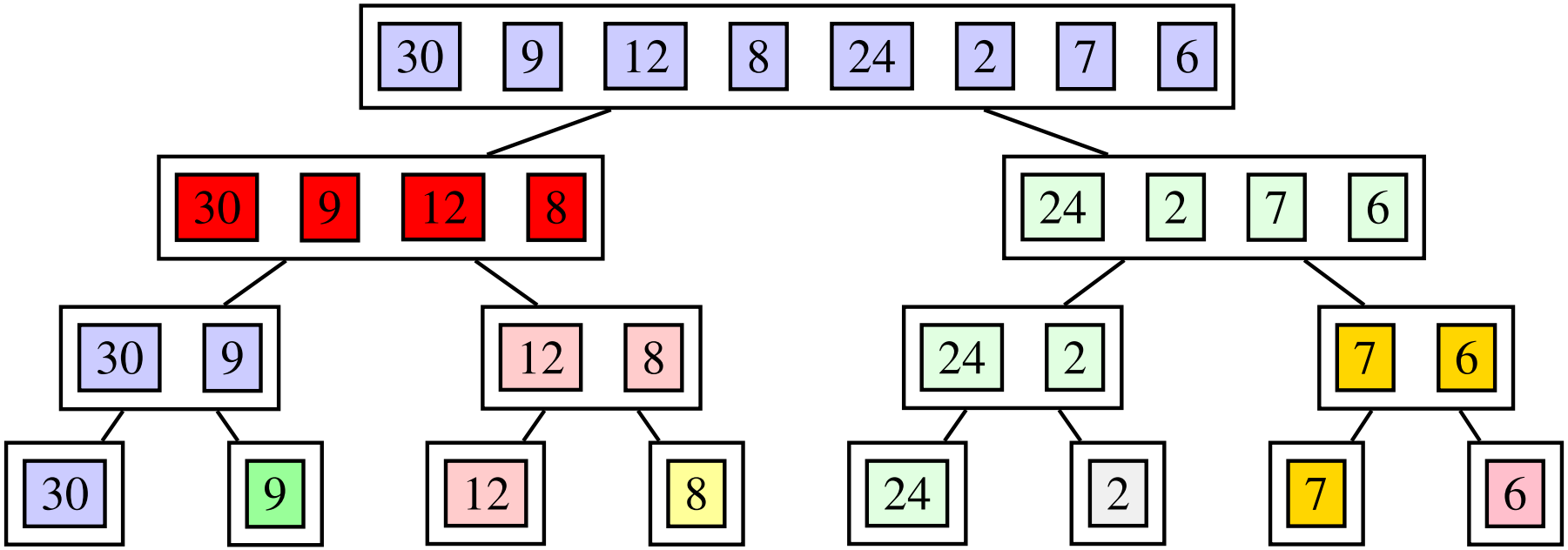
Merge Sort – Example 2

- Initially, an array of 8 elements.
- Invoke merge sort on left sub-array.



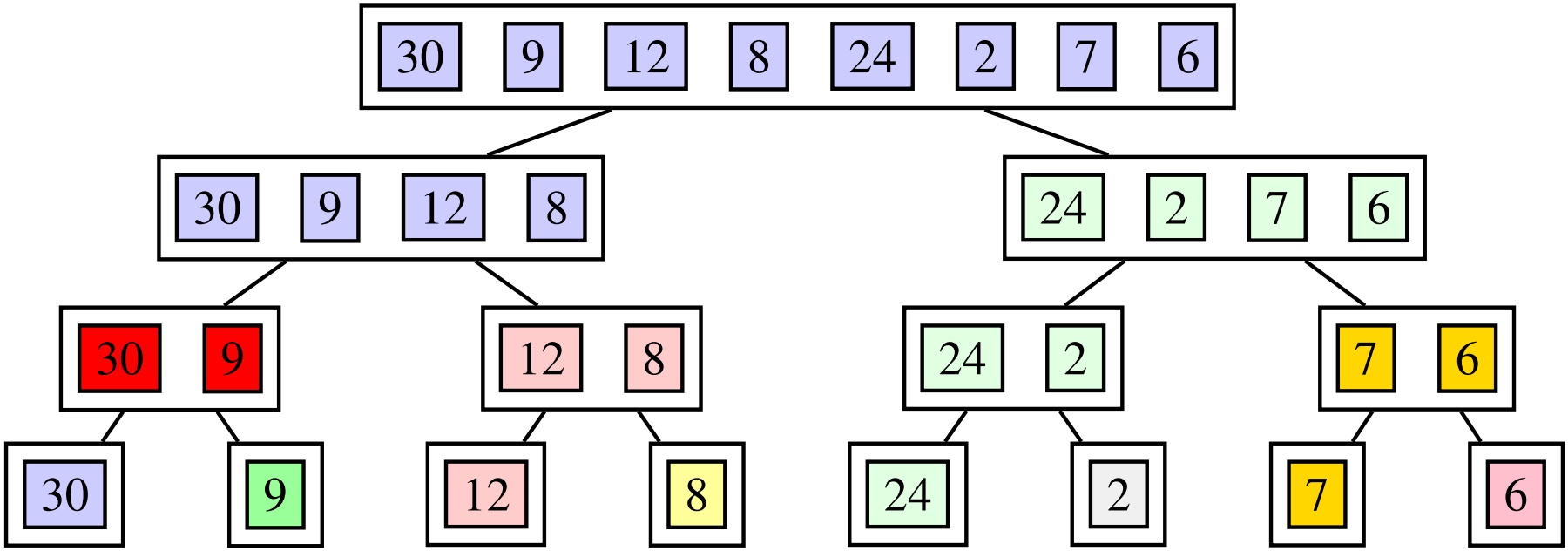
Merge Sort – Example 2

- Now the current array has 4 elements.
- Invoke merge sort on left sub-array.



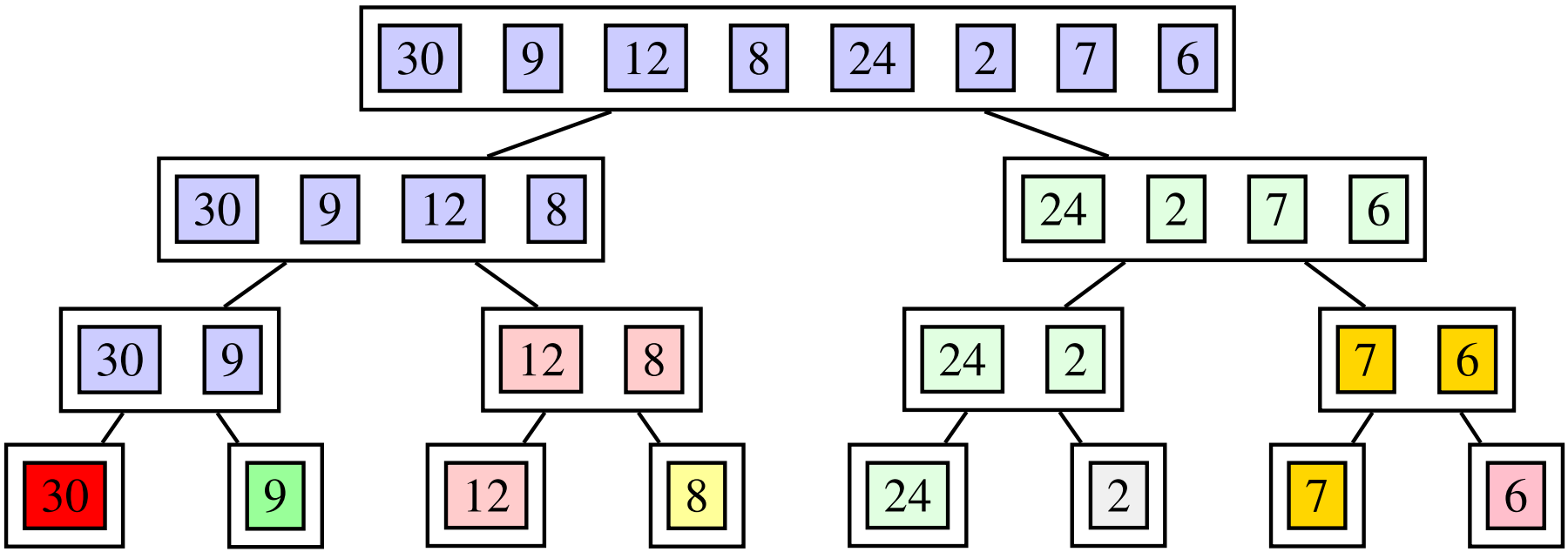
Merge Sort – Example 2

- Current array has 2 elements.
- Invoke merge sort on left sub-array.



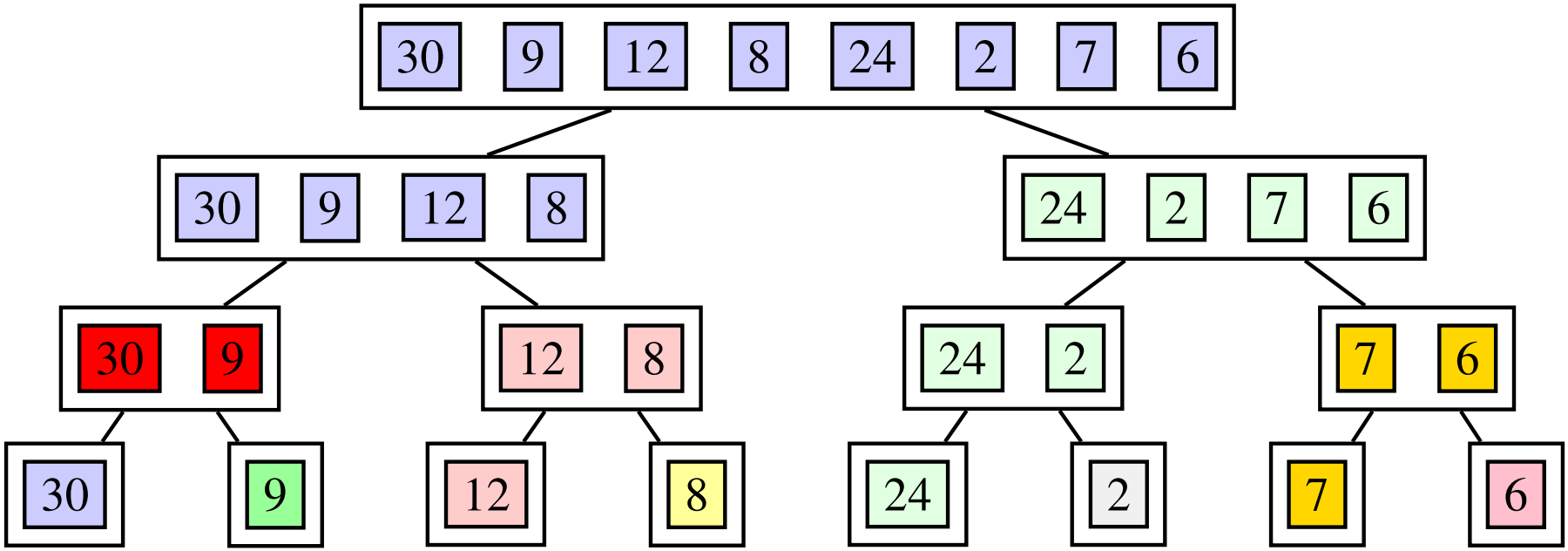
Merge Sort – Example 2

- Current array has 1 element, thus it is sorted.
- Return.



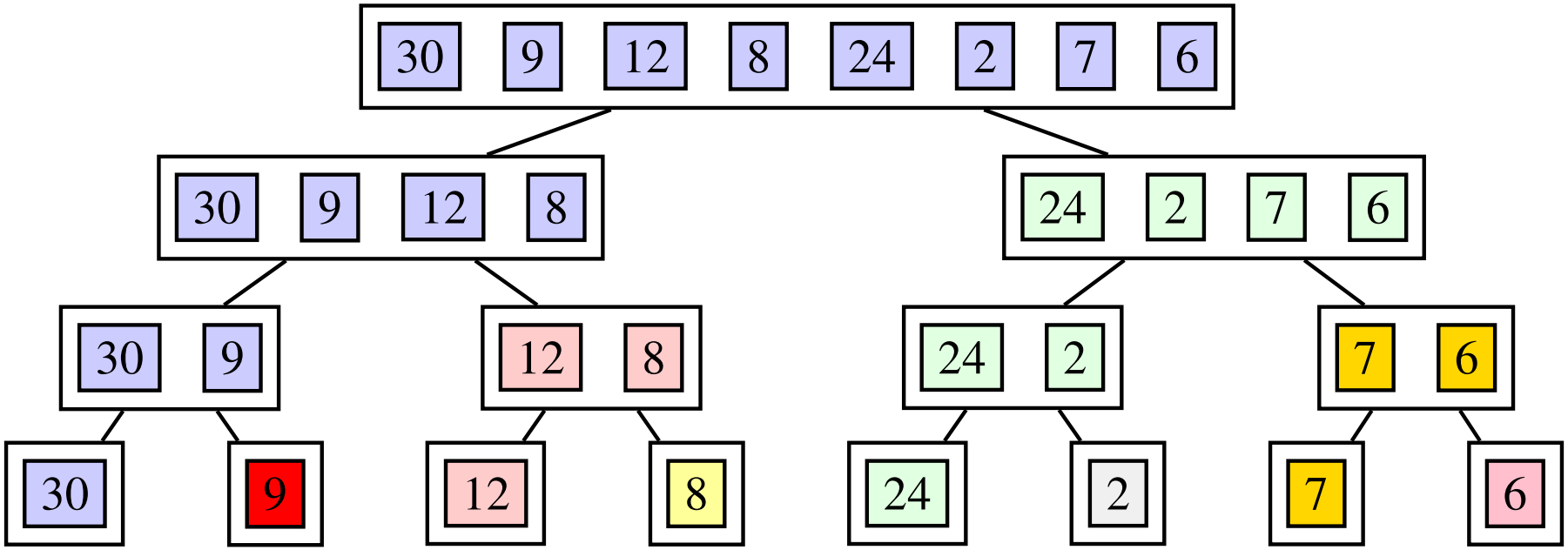
Merge Sort – Example 2

- Current array has 2 elements.
- Invoke merge sort on right sub-array.



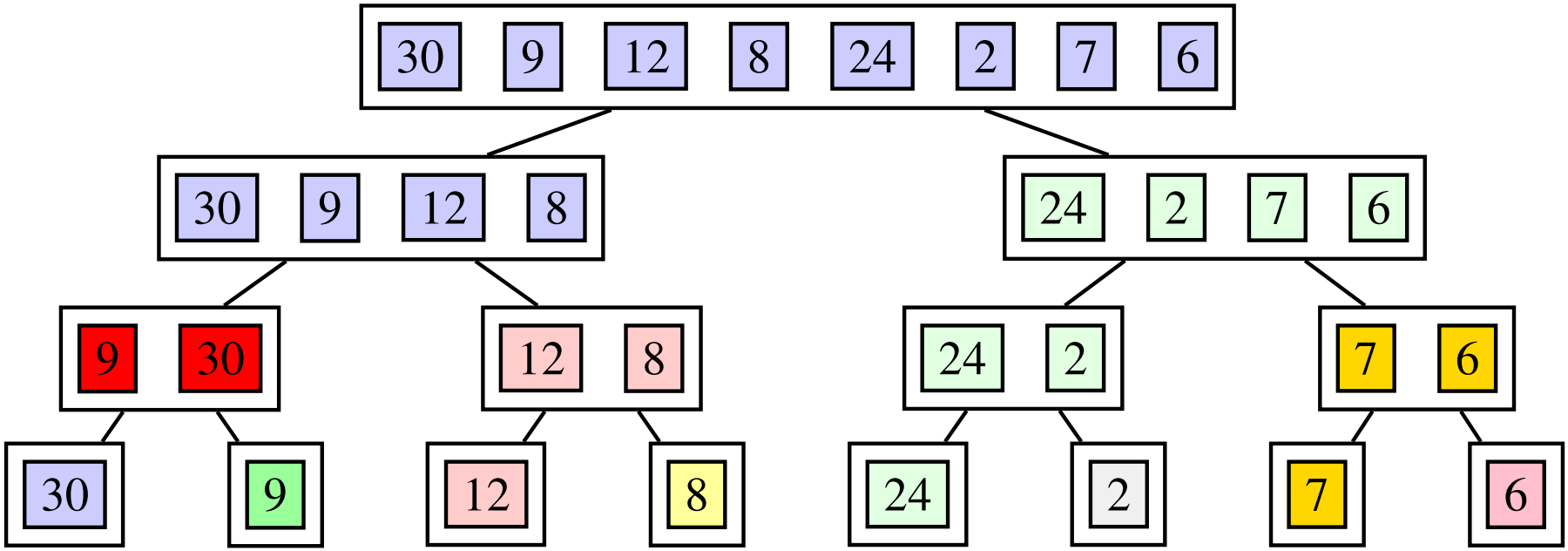
Merge Sort – Example 2

- Current array has 1 element, thus it is sorted.
- Return.



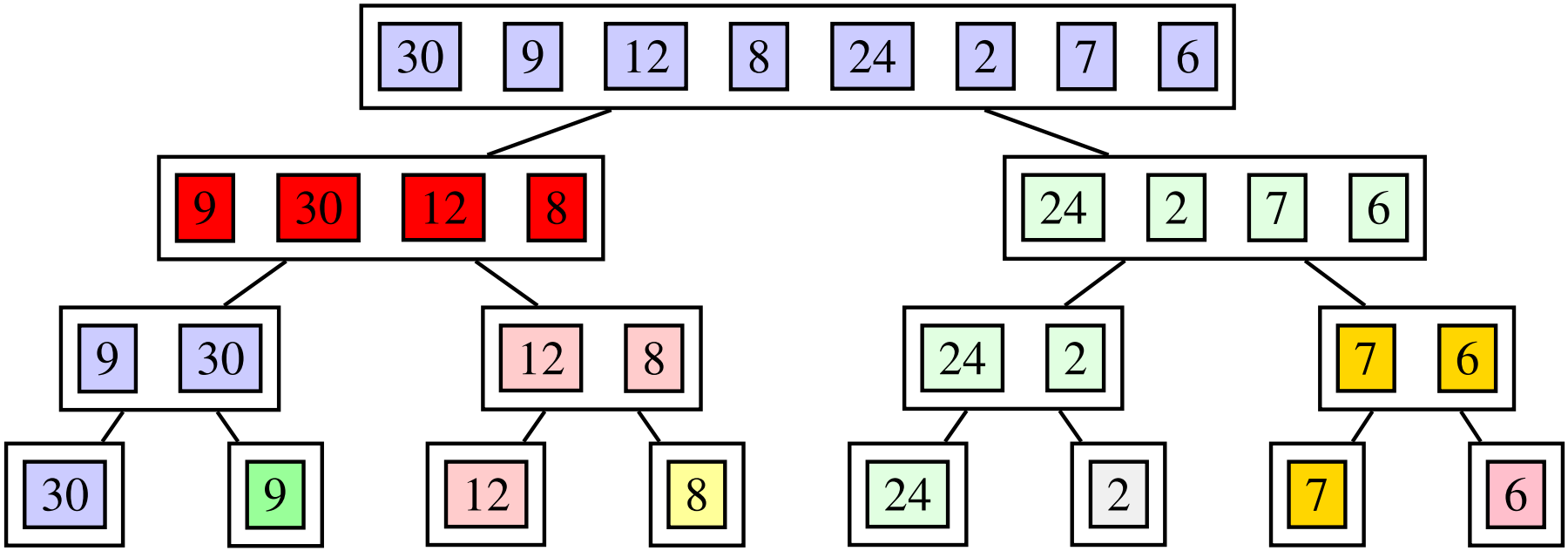
Merge Sort – Example 2

- Merge sorted sub-arrays.
- Return.



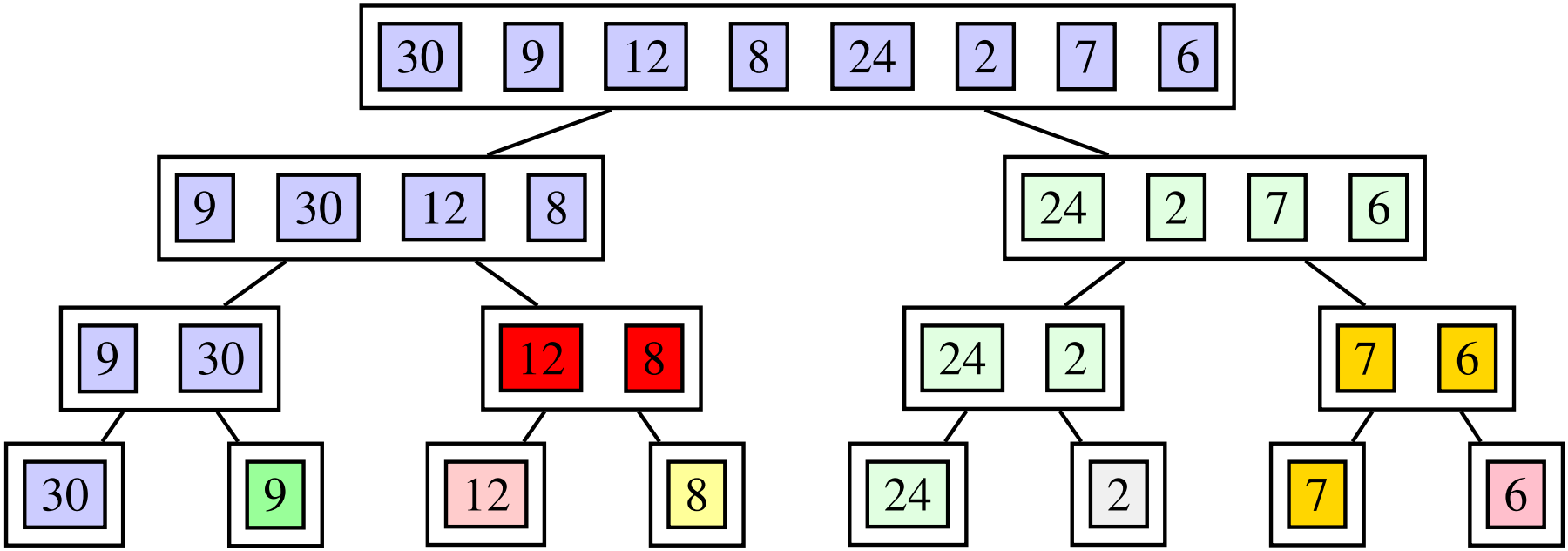
Merge Sort – Example 2

- Current array has 4 elements.
- Invoke merge sort on right sub-array.



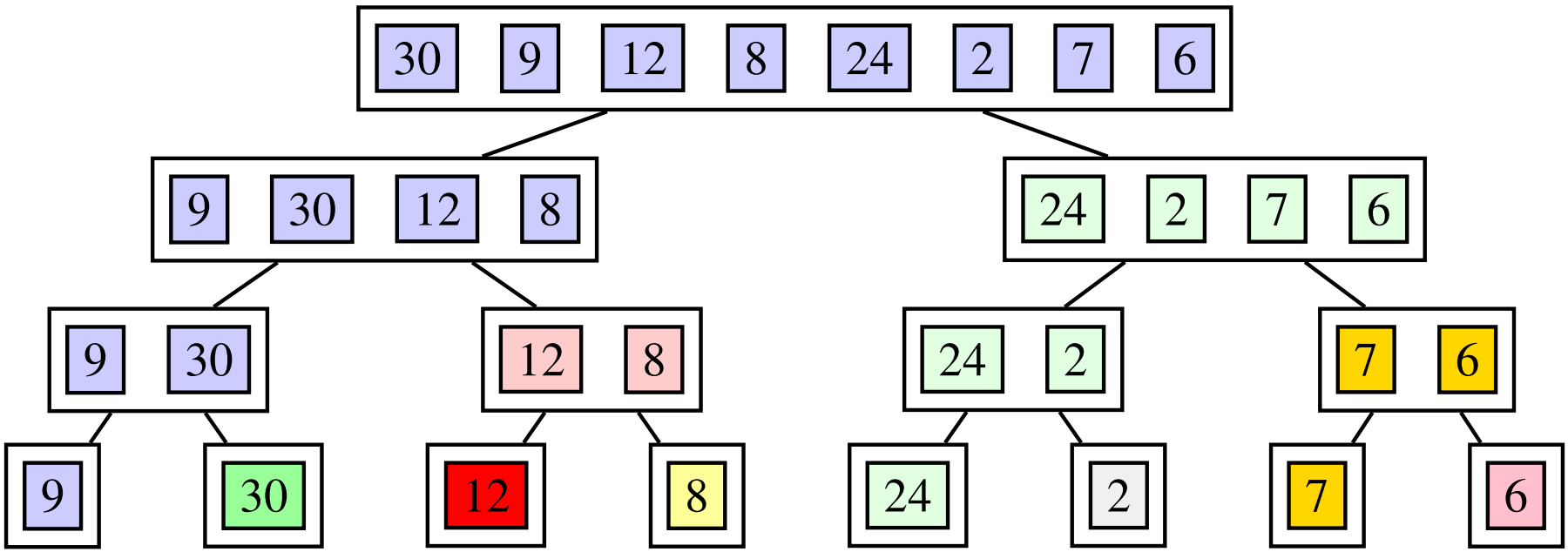
Merge Sort – Example 2

- Current array has 2 elements.
- Invoke merge sort on left sub-array.



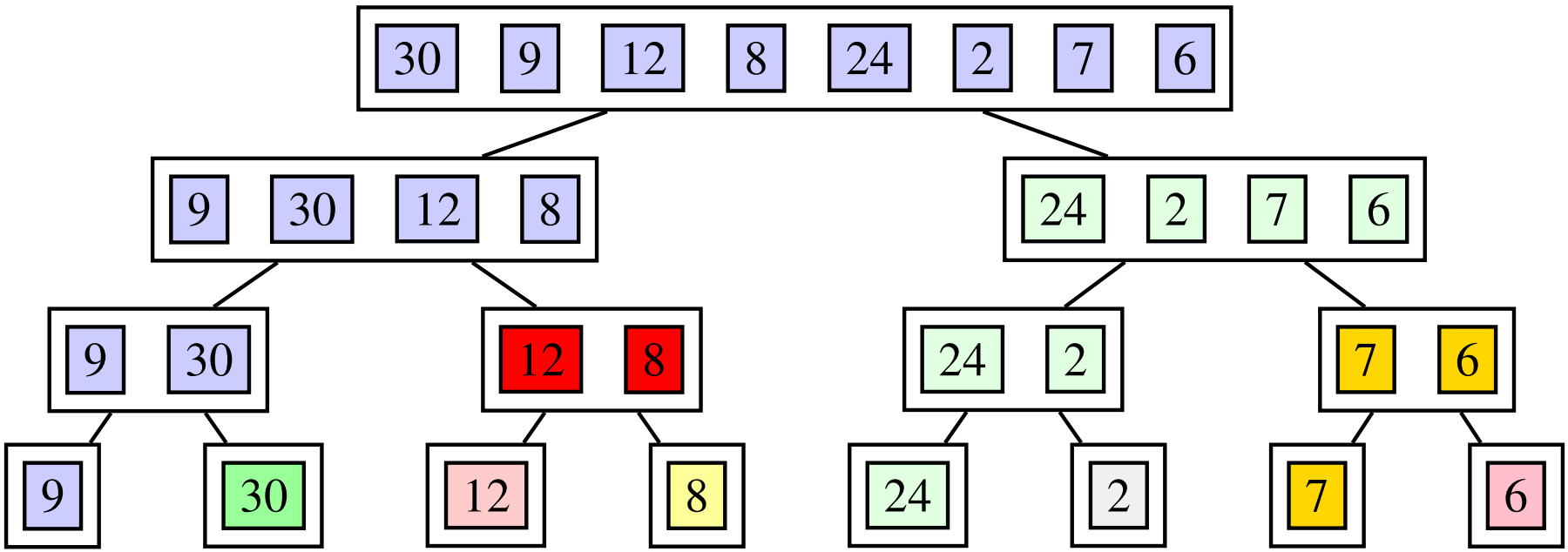
Merge Sort – Example 2

- Current array has 1 element, thus it is sorted.
- Return.



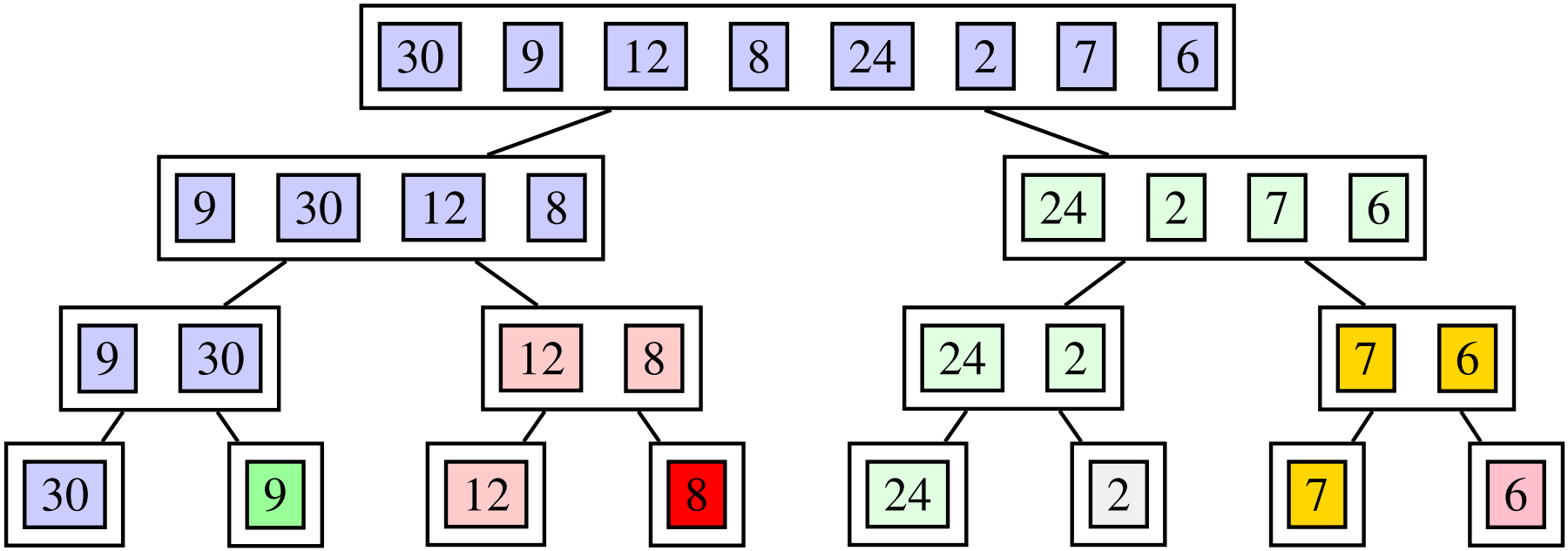
Merge Sort – Example 2

- Current array has 2 elements.
- Invoke merge sort on right sub-array.



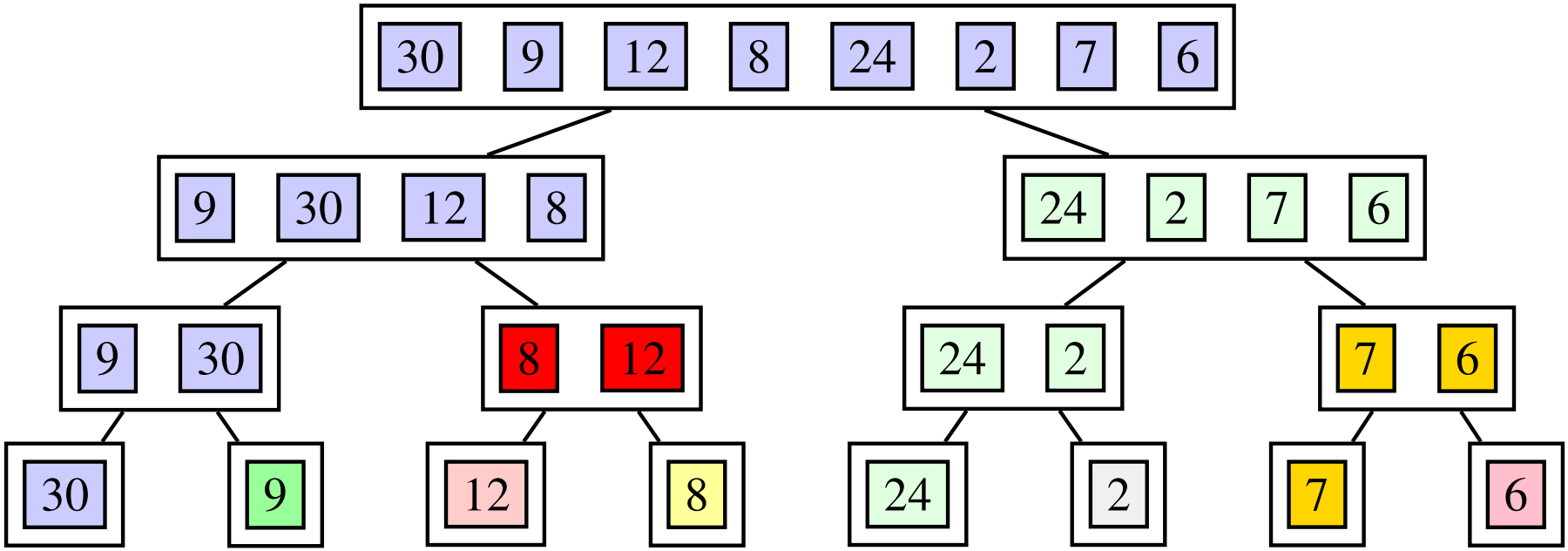
Merge Sort – Example 2

- Current array has 1 element, thus it is sorted.
- Return.



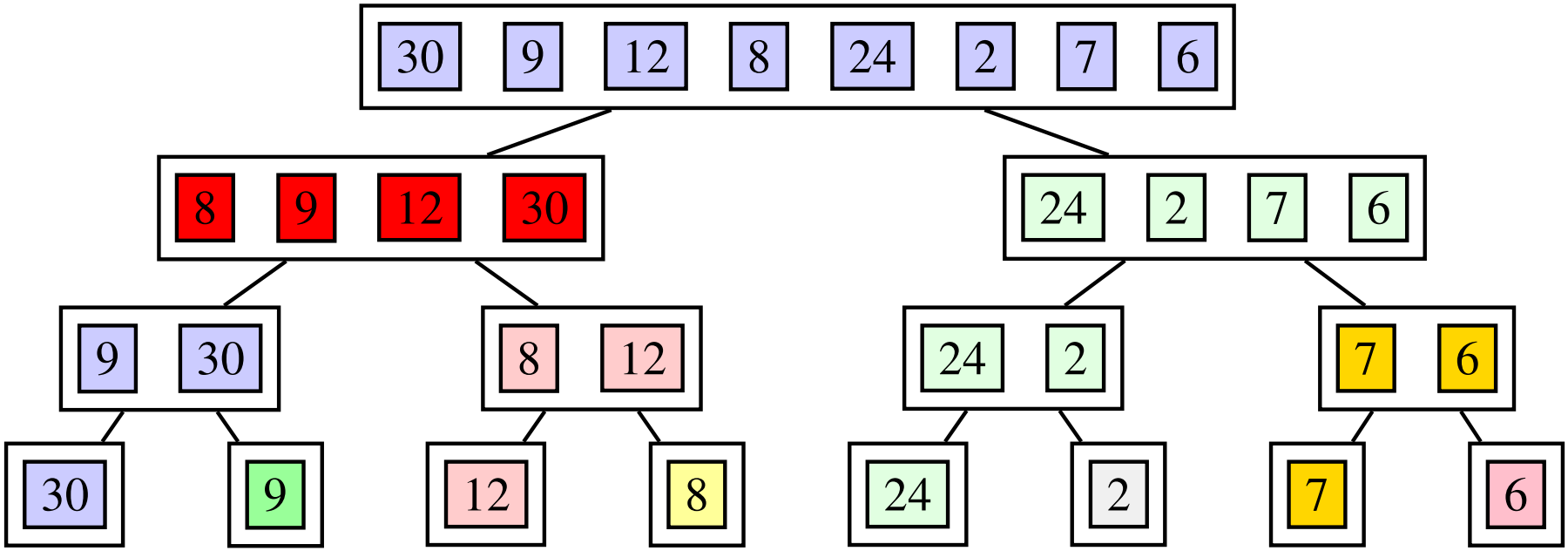
Merge Sort – Example 2

- Merge sorted sub-arrays.
- Return.



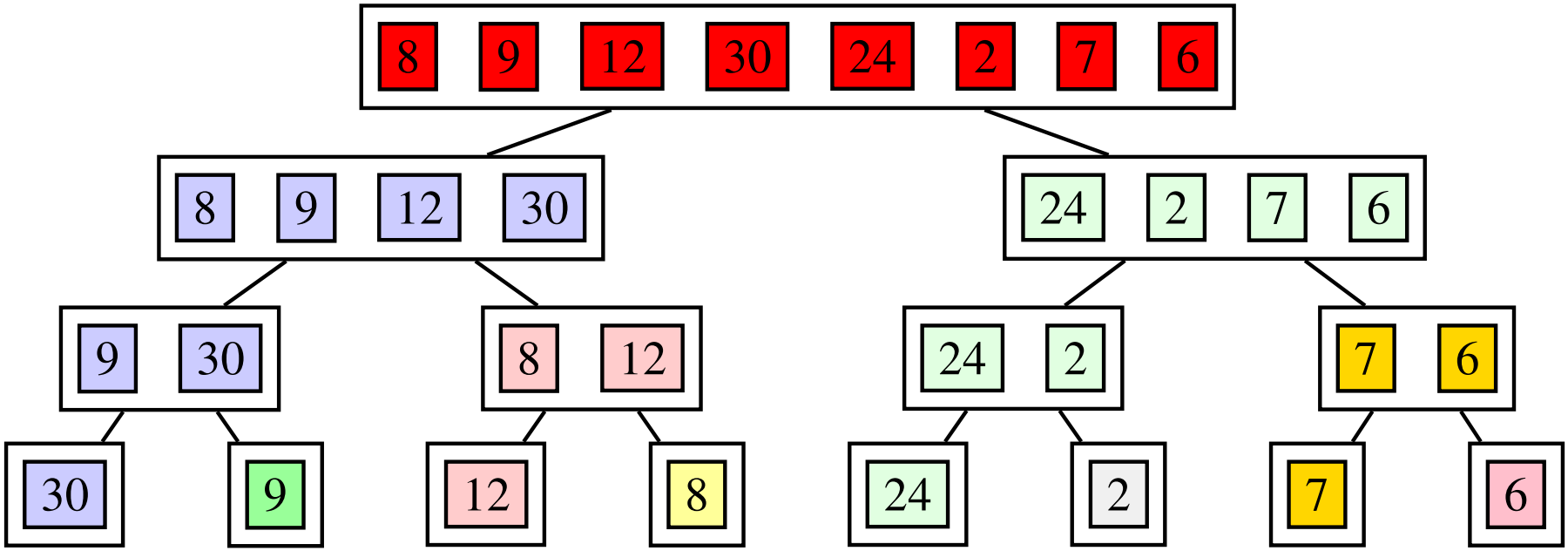
Merge Sort – Example 2

- Merge sorted sub-arrays.
- Return.



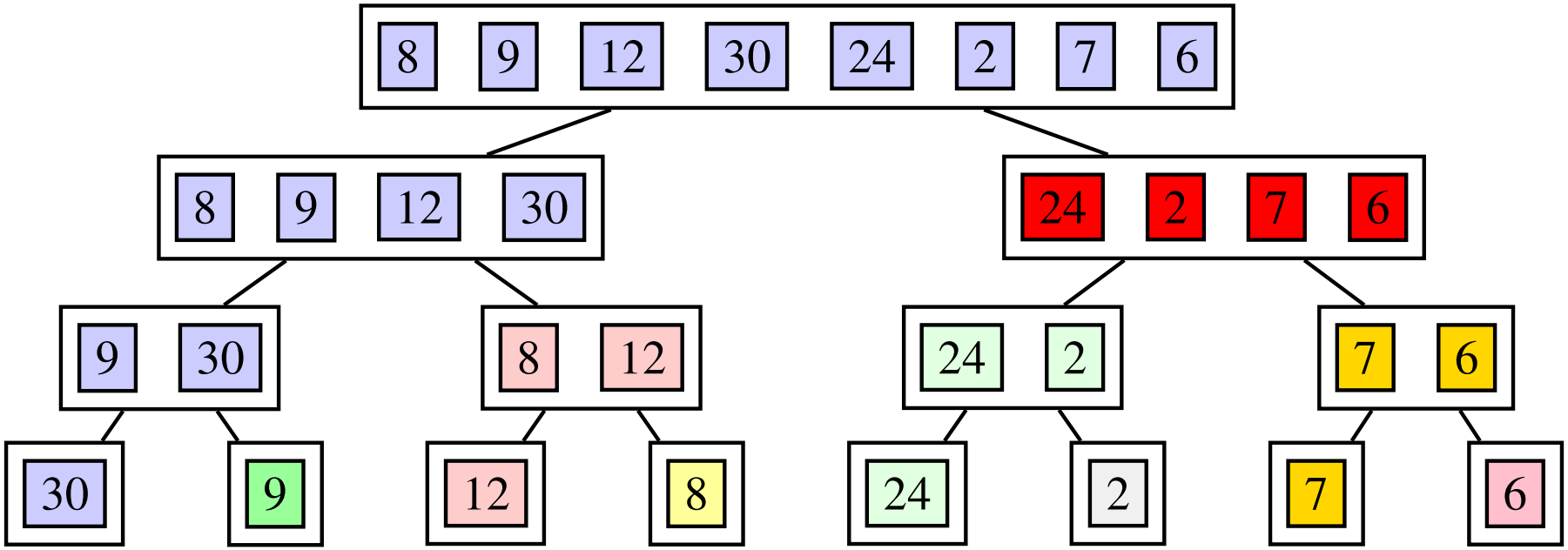
Merge Sort – Example 2

- Current array has 8 elements.
- Invoke merge sort on right sub-array.



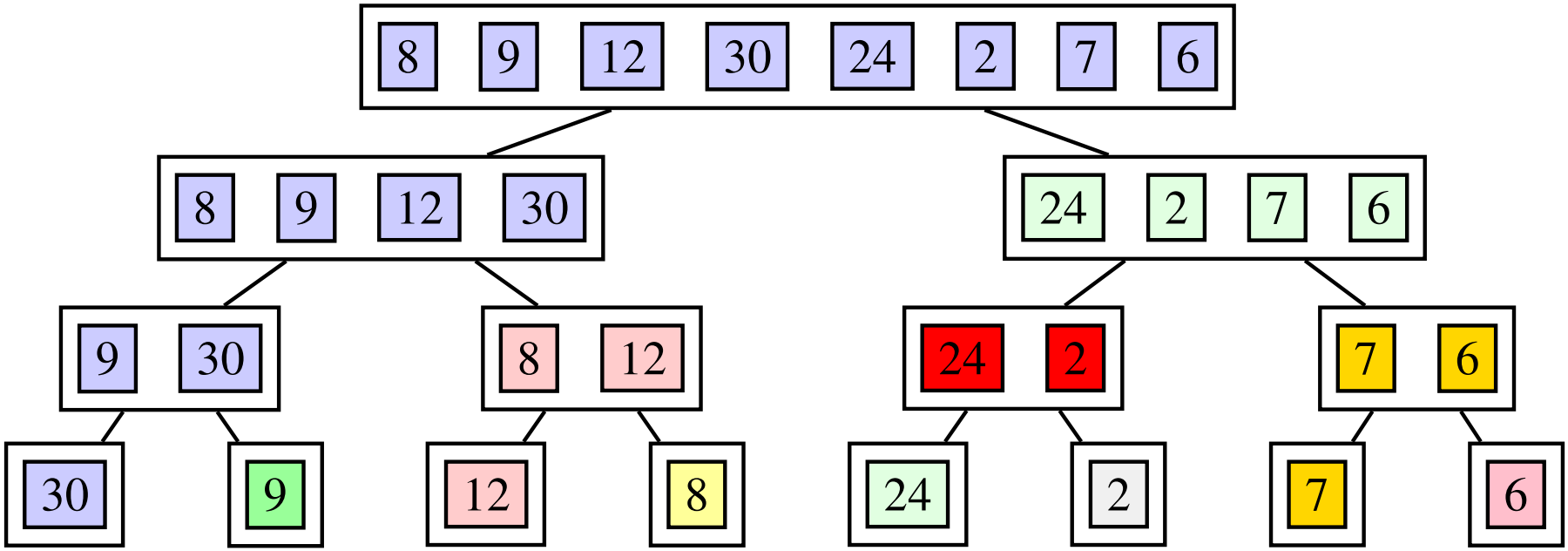
Merge Sort – Example 2

- Current array has 4 elements.
- Invoke merge sort on left sub-array.



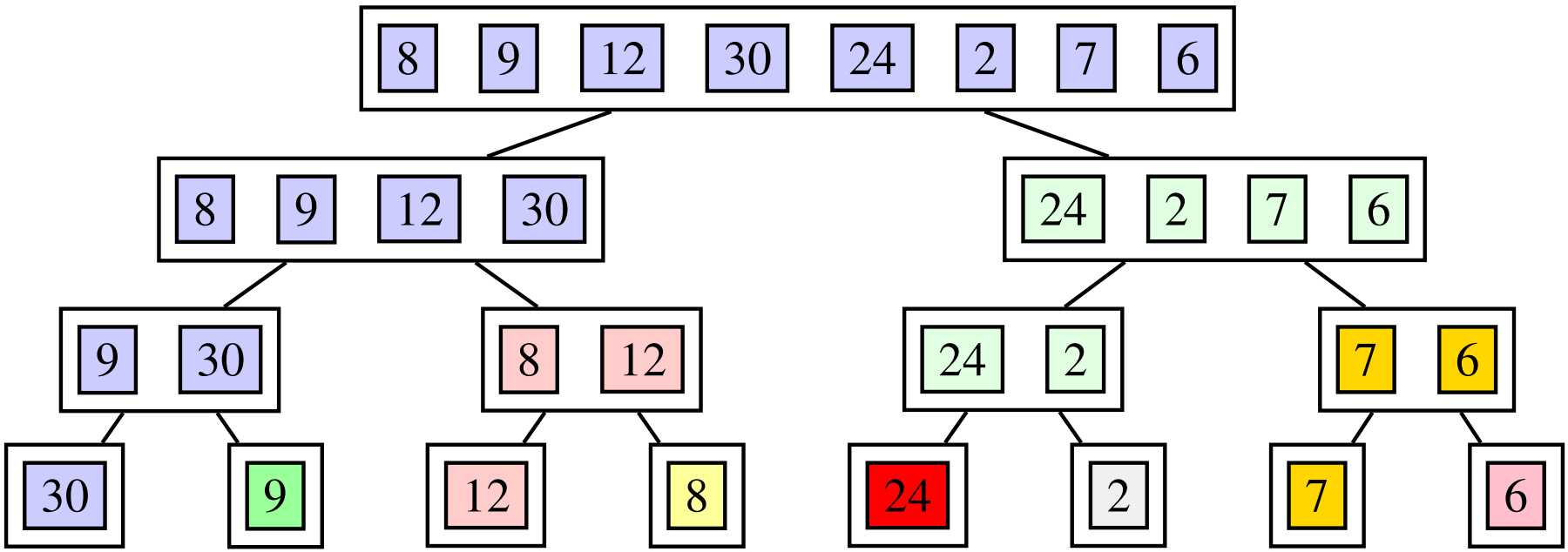
Merge Sort – Example 2

- Current array has 2 elements.
- Invoke merge sort on left sub-array.



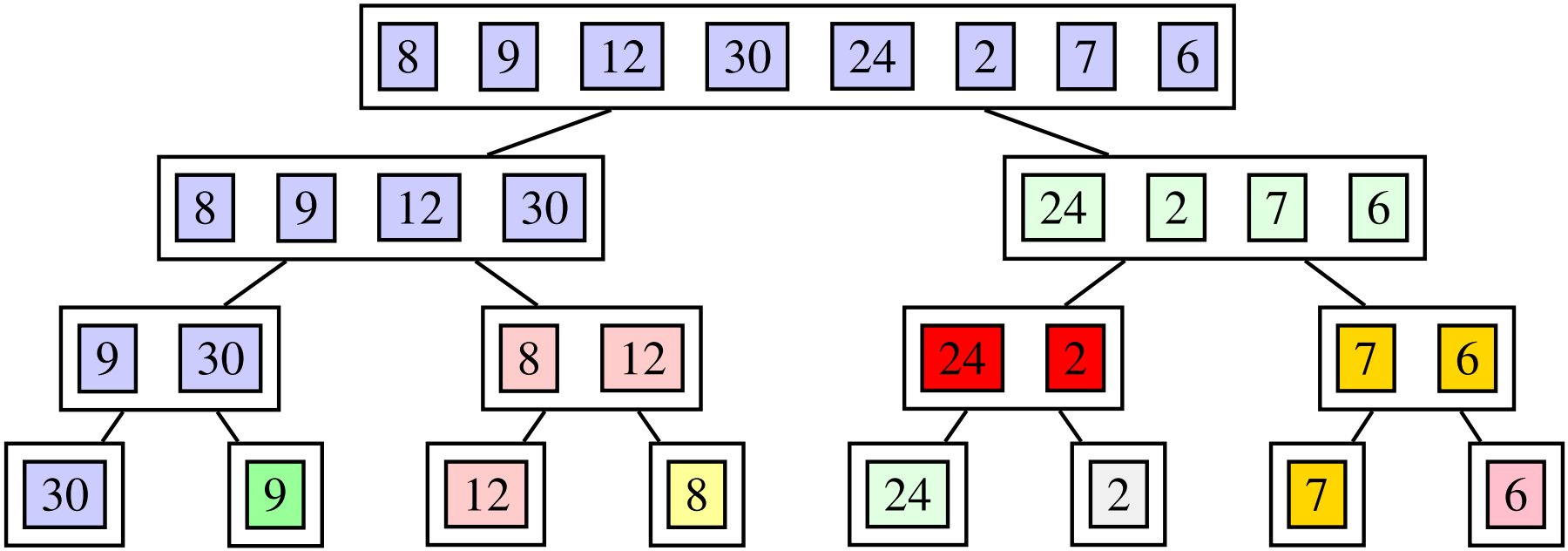
Merge Sort – Example 2

- Current array has 1 element, thus it is sorted.
- Return.



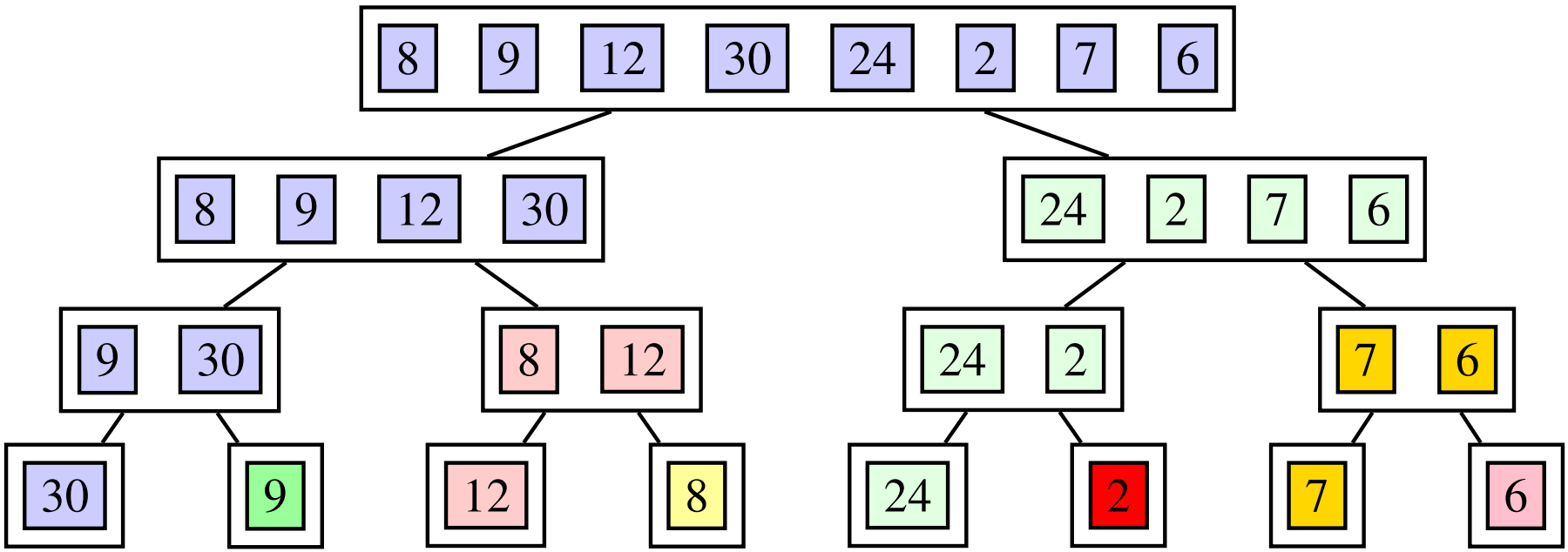
Merge Sort – Example 2

- Current array has 2 elements.
- Invoke merge sort on right sub-array.



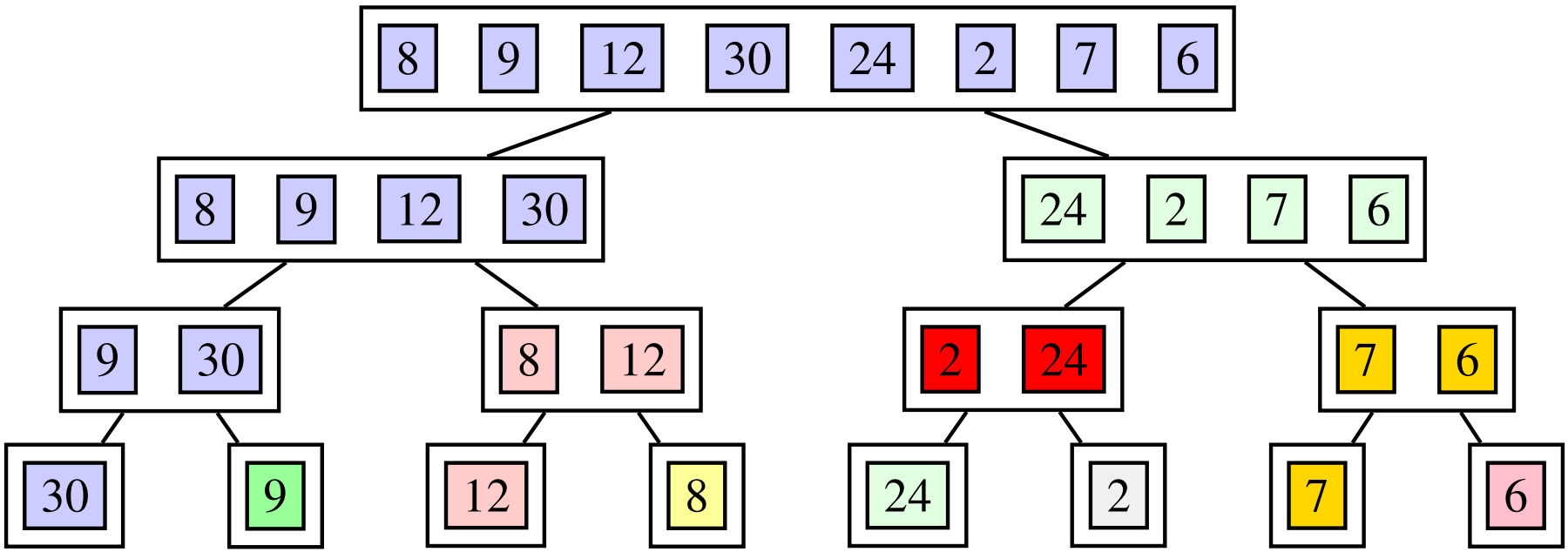
Merge Sort – Example 2

- Current array has 1 element, thus it is sorted.
- Return.



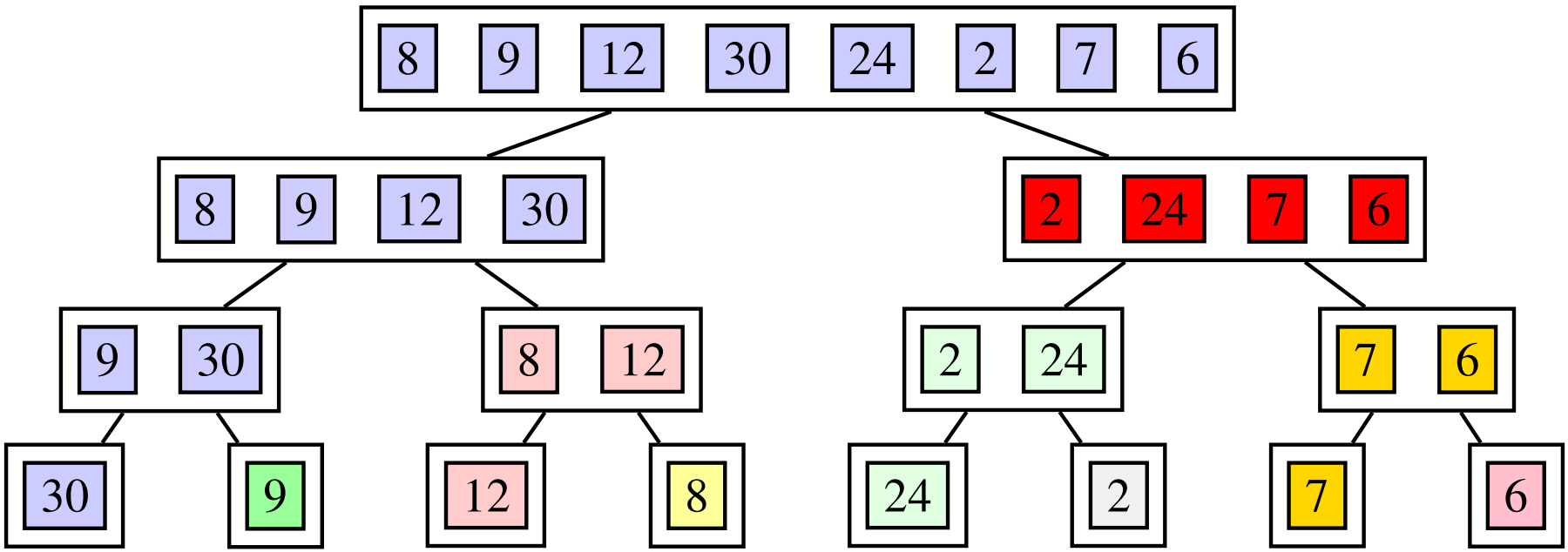
Merge Sort – Example 2

- Merge sorted sub-arrays.
- Return.



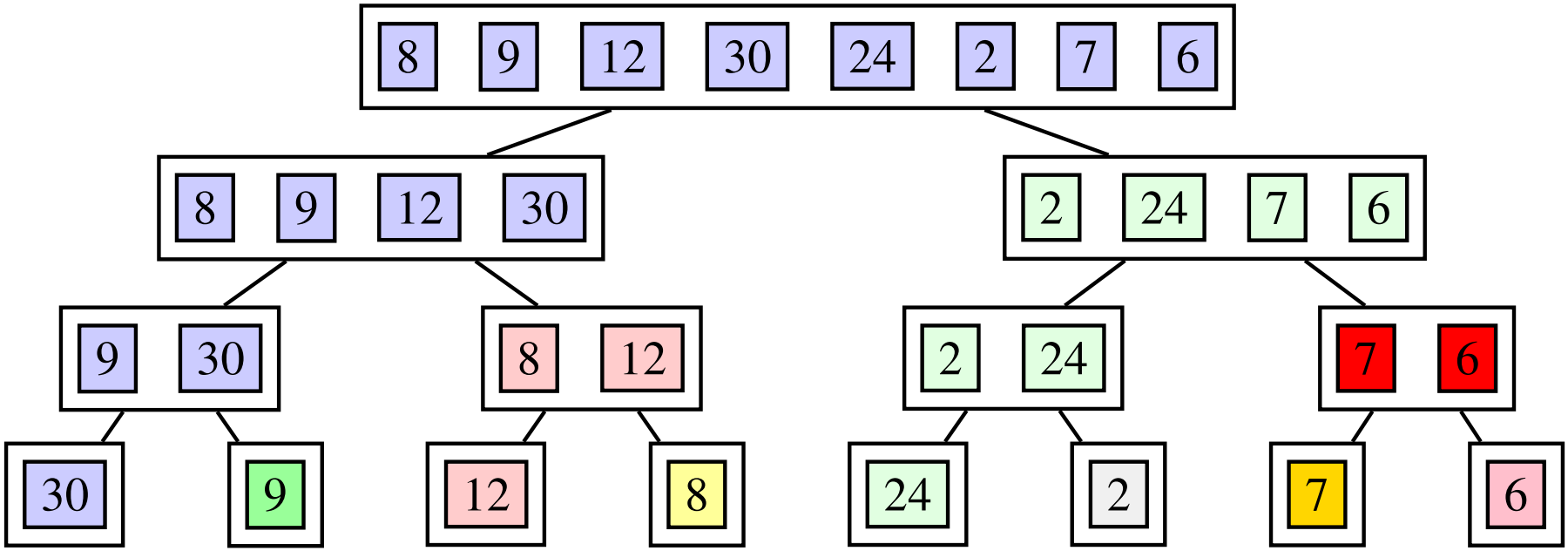
Merge Sort – Example 2

- Current array has 4 elements.
- Invoke merge sort on right sub-array.



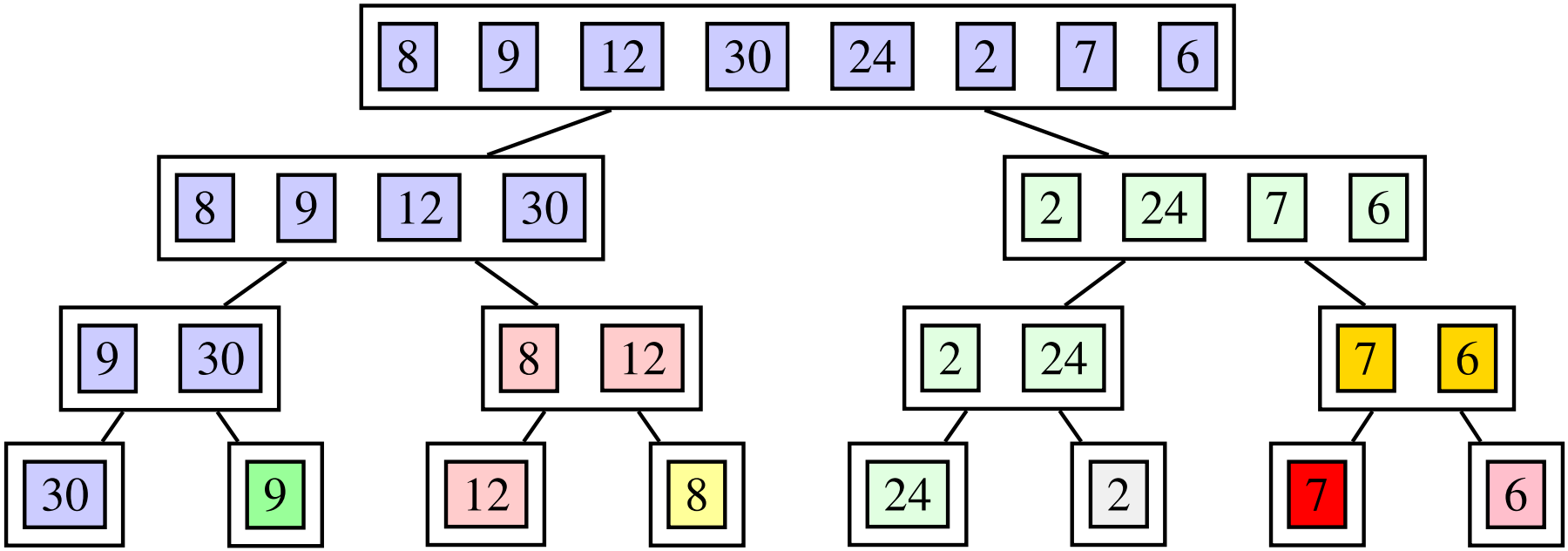
Merge Sort – Example 2

- Current array has 2 elements.
- Invoke merge sort on left sub-array.



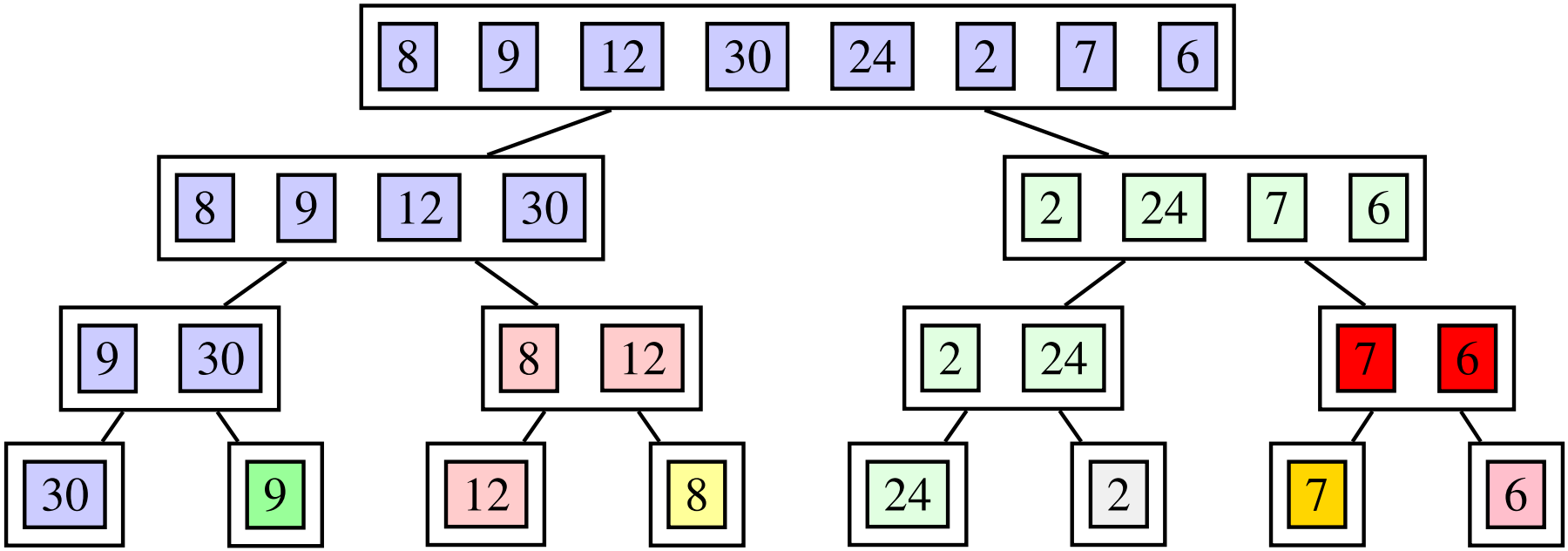
Merge Sort – Example 2

- Current array has 1 element, thus it is sorted.
- Return.



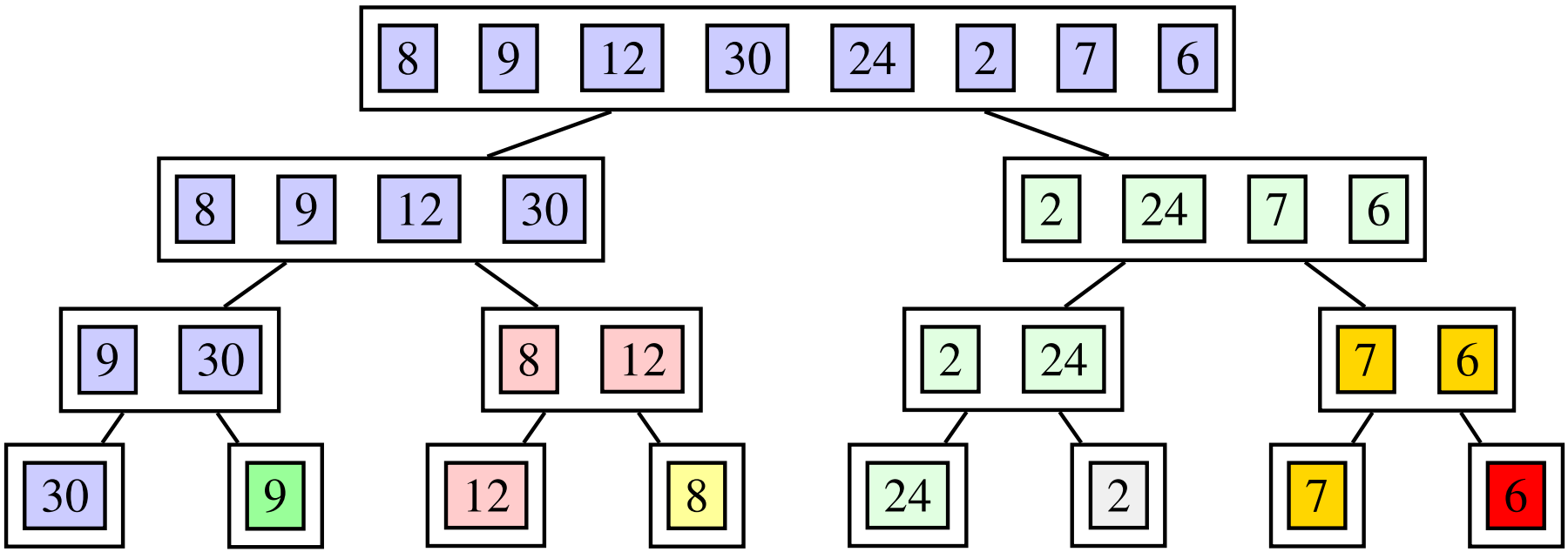
Merge Sort – Example 2

- Current array has 2 elements.
- Invoke merge sort on right sub-array.



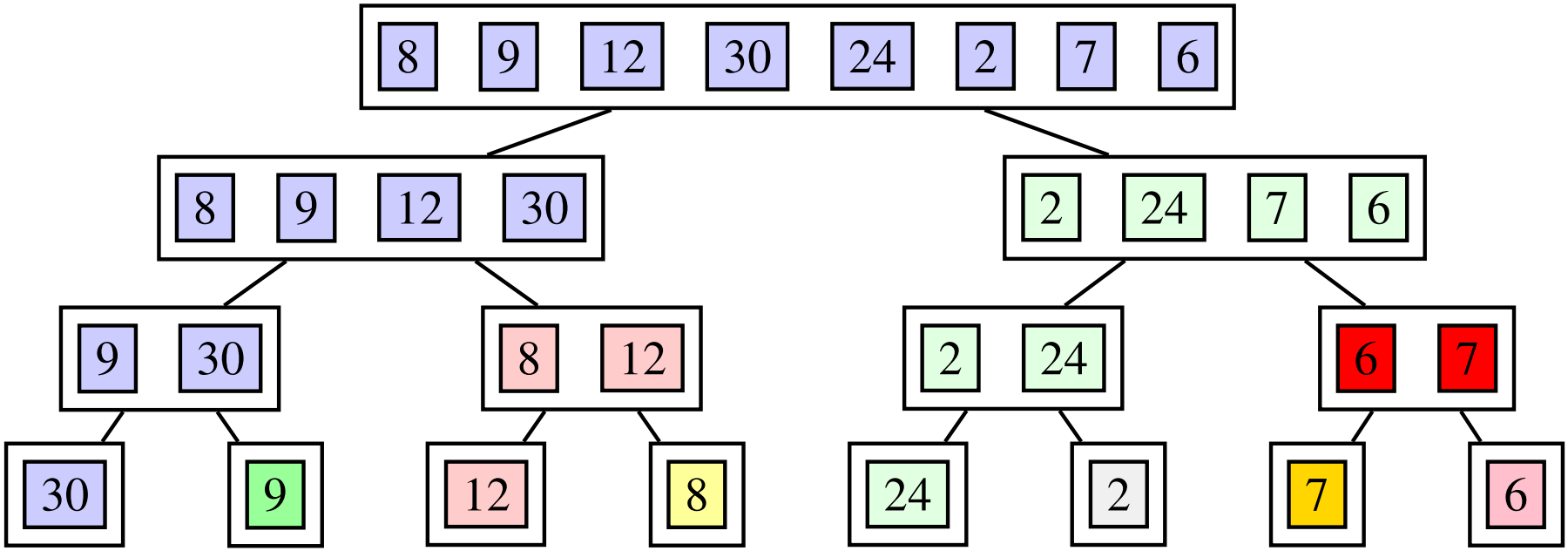
Merge Sort – Example 2

- Current array has 1 element, thus it is sorted.
- Return.



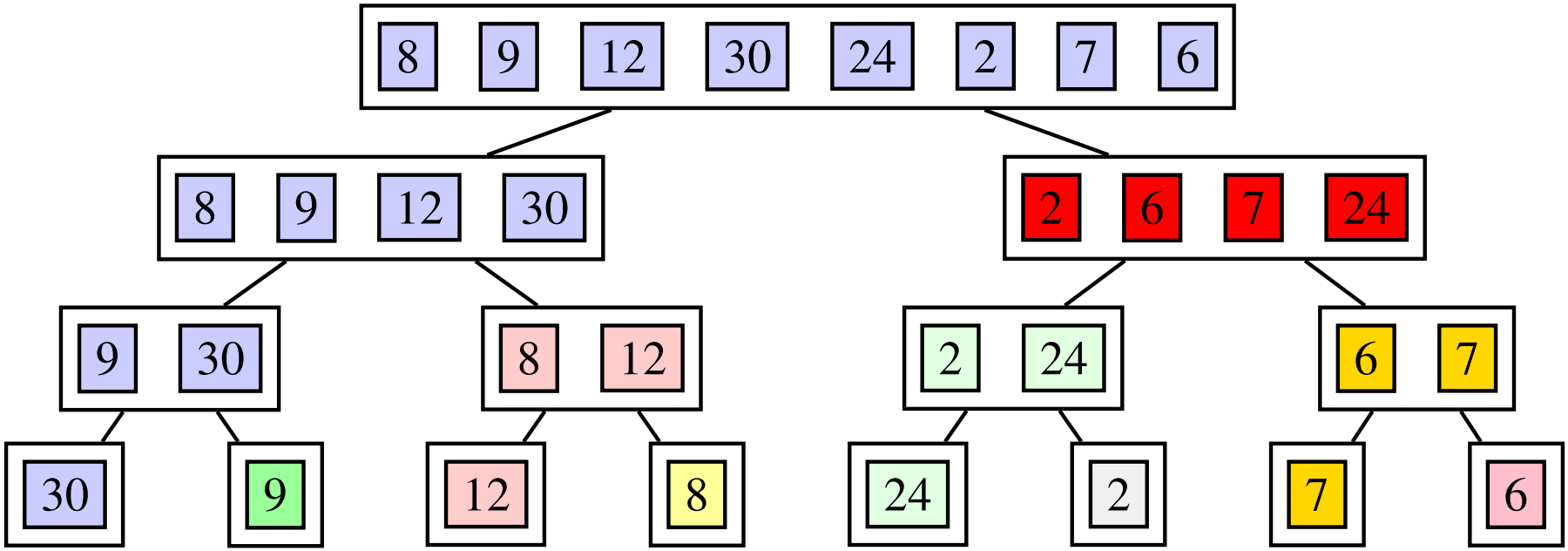
Merge Sort – Example 2

- Merge sorted sub-arrays.
- Return.



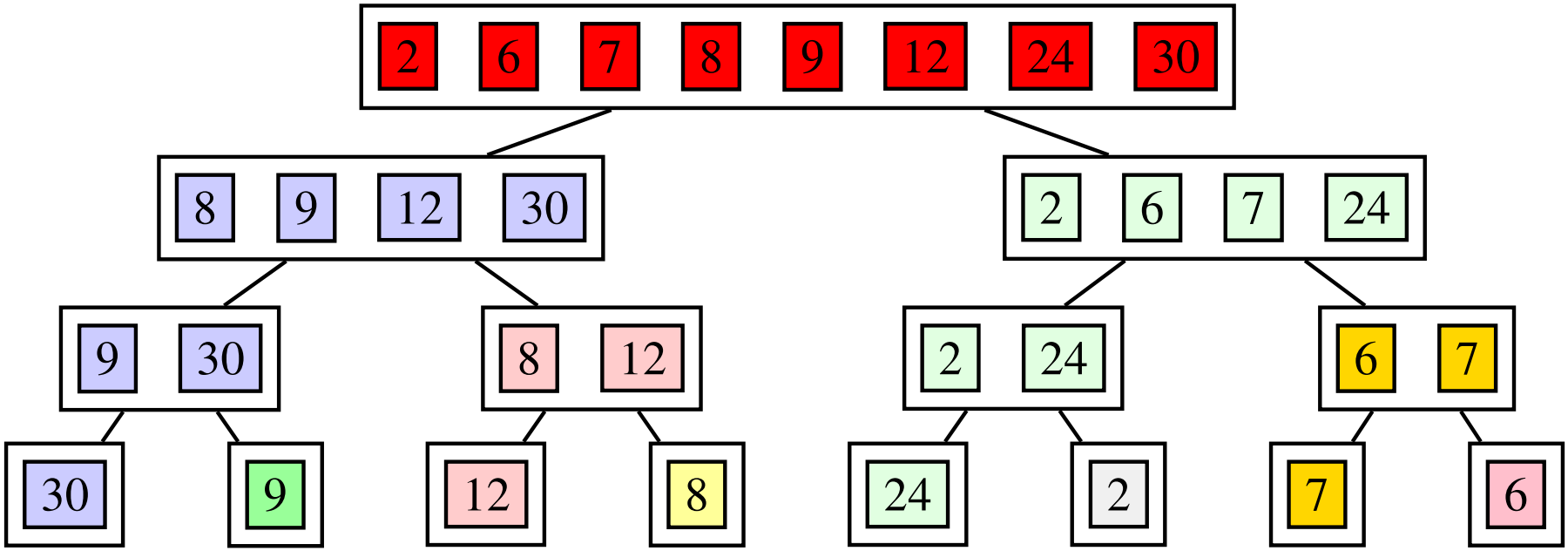
Merge Sort – Example 2

- Merge sorted sub-arrays.
- Return.



Merge Sort – Example 2

- Merge sorted sub-arrays. Return.
- The whole list is now sorted.



Merge Sort – Performance

Let $T(n)$ denote the running time to merge sort an array of n elements. Assume that $n = 2^k$ for some $k > 0$.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= \end{aligned}$$

Merge Sort – Performance

Let $T(n)$ denote the running time to merge sort an array of n elements. Assume that $n = 2^k$ for some $k > 0$.

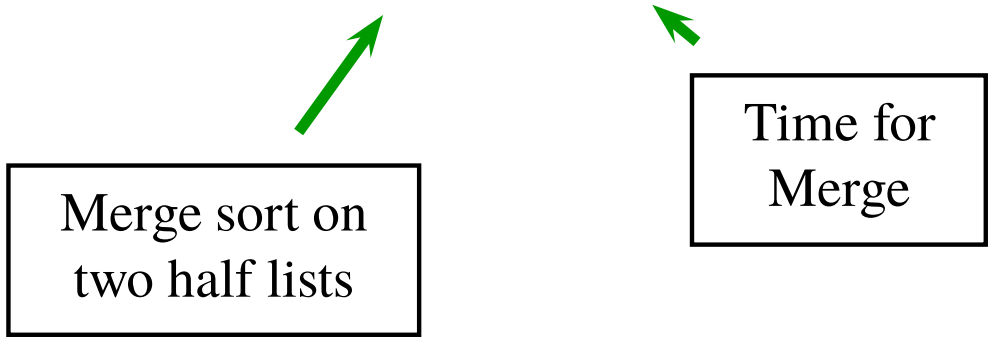
$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n/2) \end{aligned}$$

Merge sort on two half lists

Merge Sort – Performance

Let $T(n)$ denote the running time to merge sort an array of n elements. Assume that $n = 2^k$ for some $k > 0$.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n/2) + cn \end{aligned}$$



Merge Sort – Performance

Recursive formula:

$$T(n) = 2T(n/2) + cn$$

Then

$$T(n/2) = 2T(n/2^2) + cn/2$$

Therefore,

$$T(n) = 2\left(2T(n/2^2) + cn/2\right) + cn = 2^2T(n/2^2) + cn + cn$$

Similarly,

$$\begin{aligned} T(n) &= 2^2\left(2T(n/2^3) + cn/2^2\right) + cn + cn \\ &= 2^3T(n/2^3) + cn + cn + cn \\ &= 2^4T(n/2^4) + cn + cn + cn + cn \\ &= \cdots \\ &= 2^kT(n/2^k) + kcn = nT(1) + cn\log_2 n = \Theta(n\log n). \end{aligned}$$

Recall that $2^k = n$, thus $k = \log_2 n$.

So,

$$T(n) = 2^k T(n/2^k) + kcn$$

for some k. If we let $k = \log n$, then

$$\begin{aligned} T(n) &= nT(1) + cn\log n \\ &= n + cn\log n \\ &= \Theta(n\log n) \end{aligned}$$

So,

$$T(n) = 2^k T(n/2^k) + kcn$$

for some k. If we let $k = \log n$, then

$$\begin{aligned} T(n) &= nT(1) + cn\log n \\ &= n + cn\log n \\ &= \Theta(n\log n) \end{aligned}$$

- Therefore, Merge Sort is a $\Theta(n\log n)$ sorting algorithm (divide and conquer)
- However, it requires $\Theta(n)$ extra space
- It is usually used as the basis of most external sorting techniques where the intermediate results or many **files** are combined.
- Merge Sort uses the smallest number of comparisons among the popular sorting algorithms. Was used in Java to sort arrays of objects (in this case comparisons are expensive).

Quick Sort

- One of the fastest practical sorting algorithms. Developed by C.A.R. Hoare in 1960.
  - → $O(n \log n)$ on average
  - → $O(n^2)$ at worst, which can be made very unlikely by carefully designing algorithm.
- Like many of the other efficient algorithms we have considered, it is a **divide and conquer** algorithm

Quick Sort

- One of the fastest practical sorting algorithms. Developed by C.A.R. Hoare in 1960.
  - → $O(n \log n)$ on average
  - → $O(n^2)$ at worst, which can be made very unlikely by carefully designing algorithm.
- Like many of the other efficient algorithms we have considered, it is a **divide and conquer** algorithm

Basic Algorithm: Sort a list, L, of n integers

1. If $n = 0$ or $n = 1$ then return.
2. Pick an element p in L and call it the **pivot**.
3. Divide L into two groups:
   - → $L_1 = \{x \in L | x \leq p\}$ (i.e., elements less than or equal to pivot)
   - → $L_2 = \{x \in L | x > p\}$ (i.e., elements greater than or equal to pivot)
4. Return: `quicksort(`L_1`)`, p, `quicksort(`L_2`)`

Example: Sort the list

19	28	10	61	42	17	12

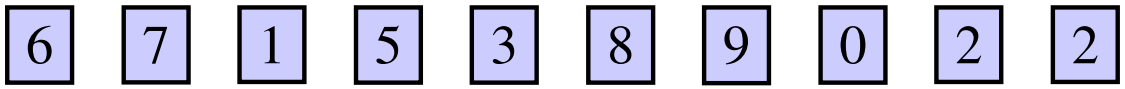
Select Pivot	19	28	10	61	42	17	12
Partition elements	19	28	10	61	42	17	12
Partition into sub-list	10	17	12	19	28	61	42
Sort each sub-list	10	12	17	19	28	42	61
Done	10	12	17	19	28	42	61

Algorithm Pseudocode:

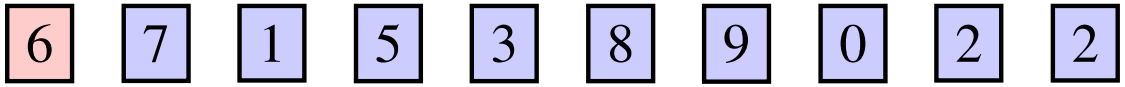
1. Select an element p as the pivot using some algorithm (to be discussed shortly ...) and swap it with the last element in the array. Set $i = 0$, $j = n - 2$.
2. Scan from left (i) until an element, q, is found such that $p < q$.
3. Scan list from the right (j) until an element r is found such that $p \geq r$
4. If $i \geq j$, exchange p with value at i and repeat on sub-lists 0 to j and $i + 1$ to $n - 1$
5. Else, exchange r and q, increment i, decrement j and goto step 2.

Quick Sort – Example

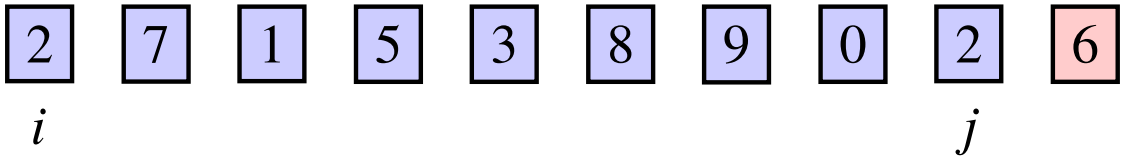
Sort:



Select Pivot:



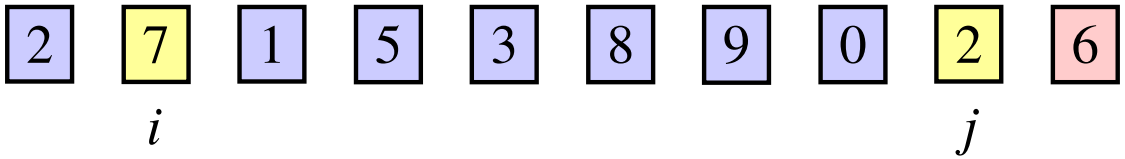
Swap Pivot to the end:



Idea: Push all elements less than pivot the left and all pivot greater to the right.

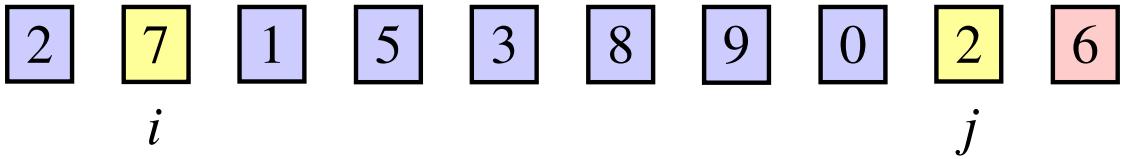
Quick Sort – Example

Push all elements less than pivot the left and all pivot greater to the right.

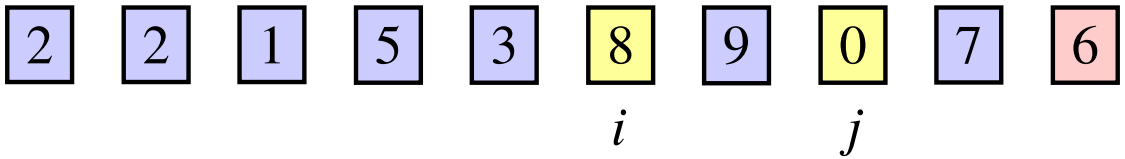


Quick Sort – Example

Push all elements less than pivot the left and all pivot greater to the right.

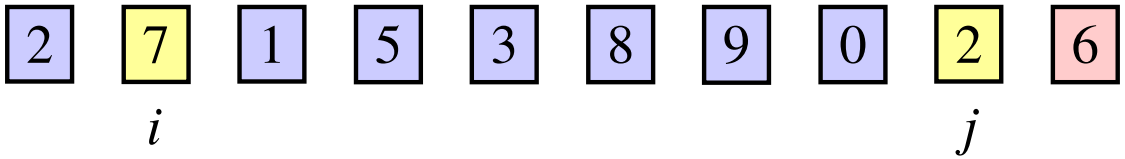


Swap, continue scanning:

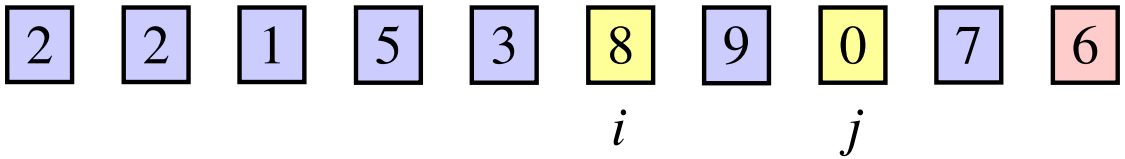


Quick Sort – Example

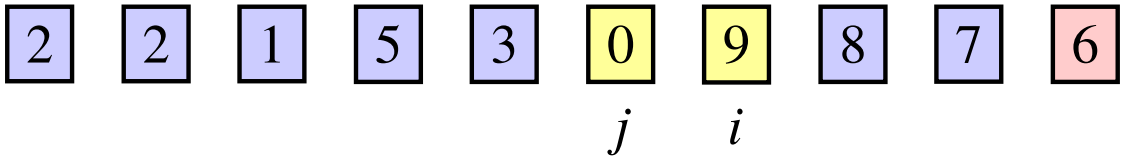
Push all elements less than pivot the left and all pivot greater to the right.



Swap, continue scanning:

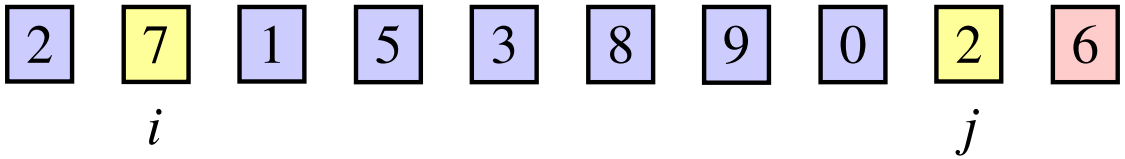


Swap, continue scanning:

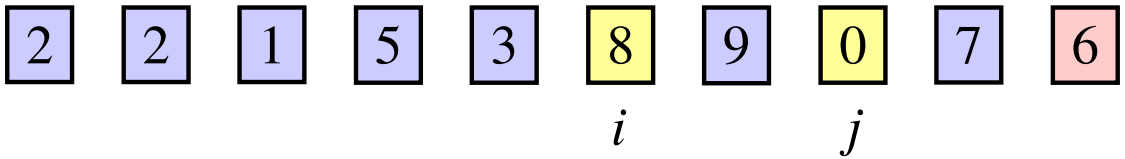


Quick Sort – Example

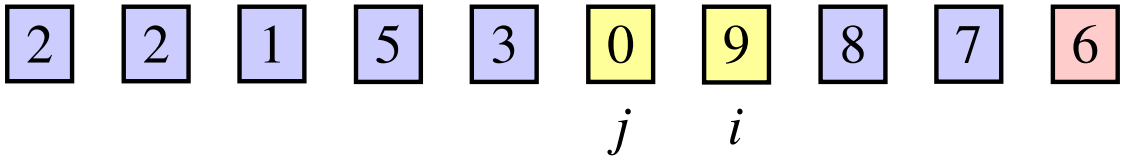
Push all elements less than pivot the left and all pivot greater to the right.



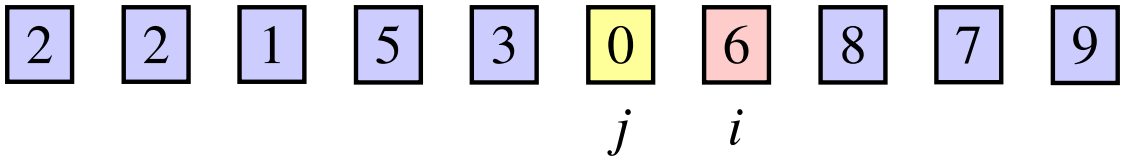
Swap, continue scanning:



Swap, continue scanning:

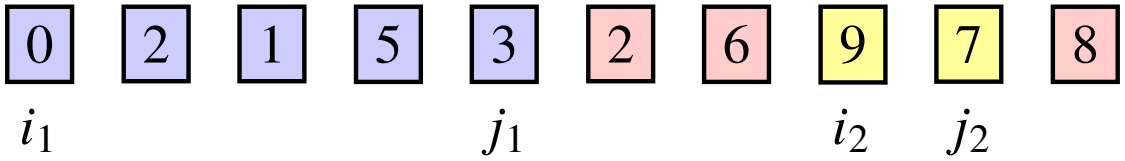


Stop, swap pivot back into list:



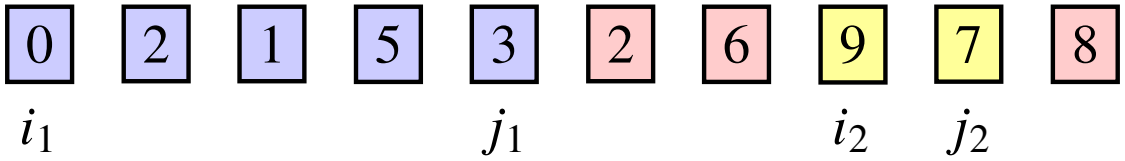
Quick Sort – Example

Repeat Recursively: with list L_1 from index 0 to j and list L_2 from index $i + 1$ to $n - 1$

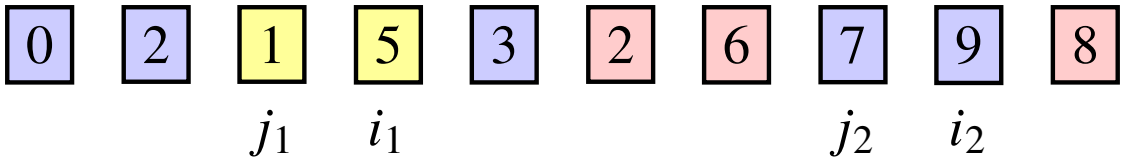


Quick Sort – Example

Repeat Recursively: with list L_1 from index 0 to j and list L_2 from index $i+1$ to $n-1$

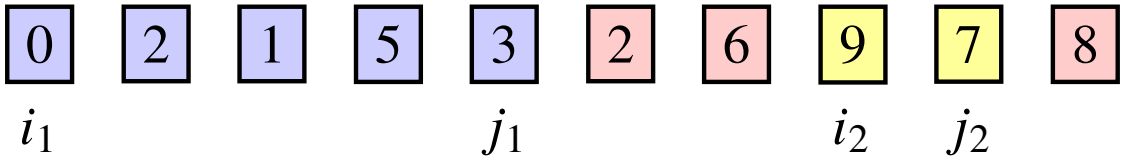


Scan and Swap:

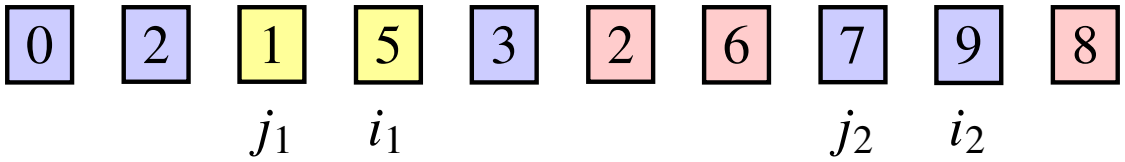


Quick Sort – Example

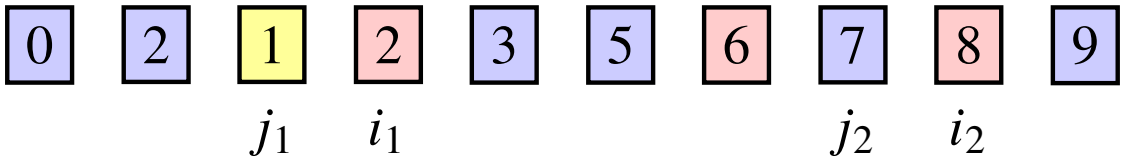
Repeat Recursively: with list L_1 from index 0 to j and list L_2 from index $i+1$ to $n-1$



Scan and Swap:

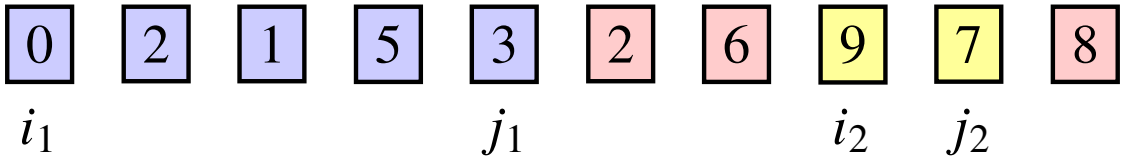


Stop and swap pivots into list:

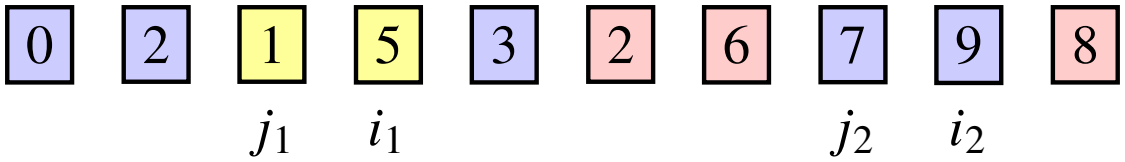


Quick Sort – Example

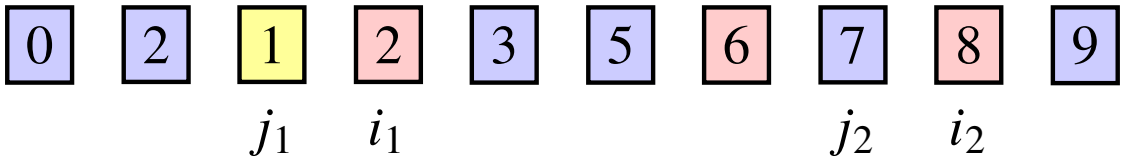
Repeat Recursively: with list L_1 from index 0 to j and list L_2 from index $i+1$ to $n-1$



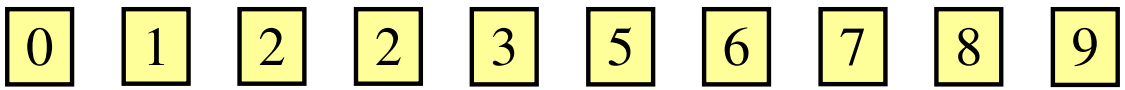
Scan and Swap:



Stop and swap pivots into list:



Continue recursively in each sublist:



Quick Sort – Pivot

- How you pick the pivot has a big impact on the performance of quick sort.

Example: Consider choosing the first element as the pivot in each sub-list and sort the list

Quick Sort – Pivot

⇨ How you pick the pivot has a big impact on the performance of quick sort.

Example: Consider choosing the first element as the pivot in each sub-list and sort the list

Pick pivot, and scan: 1 swap for pivot, 4 comparisons, no element swaps.

Quick Sort – Pivot

McMaster University

- How you pick the pivot has a big impact on the performance of quick sort.

Example: Consider choosing the first element as the pivot in each sub-list and sort the list

Pick pivot, and scan: 1 swap for pivot, 4 comparisons, no element swaps.

Pick pivot, and scan: 1 swap for pivot, 3 comparisons, no element swaps.

Quick Sort – Pivot

McMaster University

⇨ How you pick the pivot has a big impact on the performance of quick sort.

Example: Consider choosing the first element as the pivot in each sub-list and sort the list

Pick pivot, and scan: 1 swap for pivot, 4 comparisons, no element swaps.

Pick pivot, and scan: 1 swap for pivot, 3 comparisons, no element swaps.

Pick pivot, and scan: 1 swap for pivot, 3 comparisons, no element swaps.

Continue: until you get a sorted list again.

Takes $O(n^2)$ operations to do **nothing**!

Quick Sort – Pivot

Question: How do you pick the pivot in an efficient manner?

- **Random Selection**
  - → Pick pivot randomly from the list
  - → Usually avoids the worst case performance
  - → Generating random numbers is expensive

Quick Sort – Pivot

Question: How do you pick the pivot in an efficient manner?

- **Random Selection**
  - → Pick pivot randomly from the list
  - → Usually avoids the worst case performance
  - → Generating random numbers is expensive
- **Median of List**
  - → The median is the element whose value is in the "middle" of all elements in the list.
  - → This procedure can also be expensive and slow down performance

Quick Sort – Pivot

Question: How do you pick the pivot in an efficient manner?

- **Random Selection**
  - → Pick pivot randomly from the list
  - → Usually avoids the worst case performance
  - → Generating random numbers is expensive
- **Median of List**
  - → The median is the element whose value is in the "middle" of all elements in the list.
  - → This procedure can also be expensive and slow down performance
- **Median of Three Partitioning**
  - → Pick the first, last and middle element in the list and use the median of these three as the pivot
  - → Usually gives good performance, not too complex and avoids worst case of sorting a sorted list

Quick Sort – Cutoff

- Like merge sort, quick sort breaks the problem of sorting into a number of smaller problems
  - → We will show that on average quick sort takes $\Theta(n \log n)$ operations
- However, for short lists (say < 20 elements) and almost sorted lists insertion sort has better performance

Quick Sort – Cutoff

- Like merge sort, quick sort breaks the problem of sorting into a number of smaller problems
  - → We will show that on average quick sort takes $\Theta(n \log n)$ operations
- However, for short lists (say < 20 elements) and almost sorted lists insertion sort has better performance

Solution: For large lists, sort with quick sort, but when a list (or sub-list) is below a **cutoff** switch to insertion sort.

Quick Sort – Cutoff

- Like merge sort, quick sort breaks the problem of sorting into a number of smaller problems
  - → We will show that on average quick sort takes $\Theta(n \log n)$ operations
- However, for short lists (say < 20 elements) and almost sorted lists insertion sort has better performance

Solution: For large lists, sort with quick sort, but when a list (or sub-list) is below a **cutoff** switch to insertion sort.

- Leave the list slightly unsorted and finish with insertion sort.
- Using this technique can provide up to 15% improvement over just using quick sort.
- A cutoff value of 10–20 is usually good.
- Using a cutoff removes the problem of trying to do median of three partitioning with a list of less than 3 elements.

Quick Sort – Code

McMaster University

- A simple C routine performing Quicksort

```
1  #define CUTOFF 10
2  void qsort(int *list, int left, int right){
3      int i,j,pivot;
4      if(left+CUTOFF <= right){
5          pivot=findpivot(list,left,right);  /* see Fig. 7.13 for code */
6          i=left; j=right-1;
7          for(;;){
8              while(list(++i)<pivot); while(list(--j)>pivot);
9              if(i<j) swap(&list[i],&list[j]);
10             else break;
11         }
12         swap(&list[i],&list[right-1]);  /* put pivot back */
13         qsort(list,left,i-1);  /* recursively sort each sub-list */
14         qsort(list,i+1,right);
15     } else
16         insertion(list+left,right-left+1);  /* use insertion sort
17                                                on short lists */
18 }
```

Quick Sort – Example

Sort: Median of three partitioning, cutoff=3

55	19	13	61	42	28	75	91	20	68	21

Select Pivot: using Median of Three Partitioning

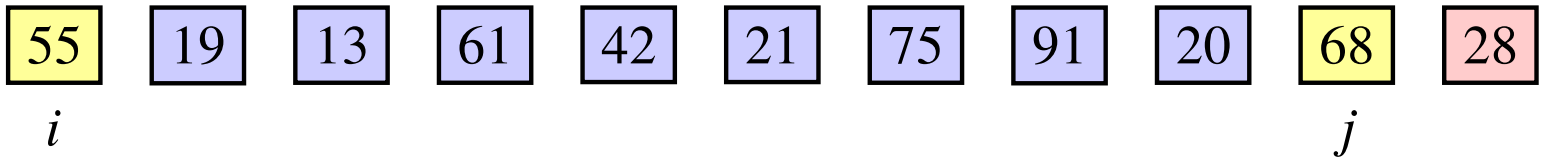
55	19	13	61	42	28	75	91	20	68	21

Swap Pivot to the end:

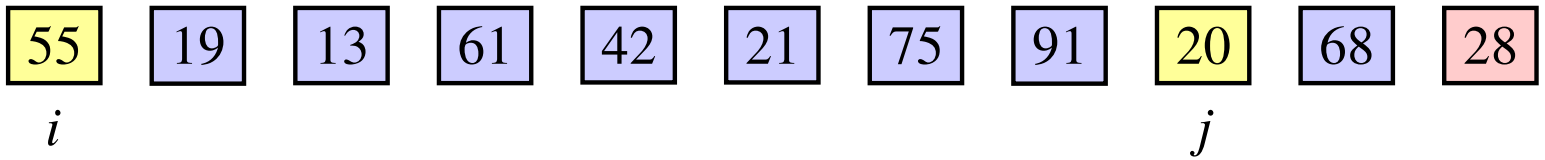
55	19	13	61	42	21	75	91	20	68	28

Quick Sort – Example

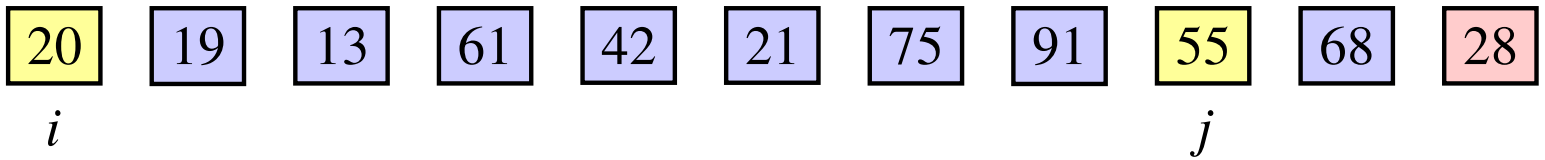
Swap Pivot to the end:



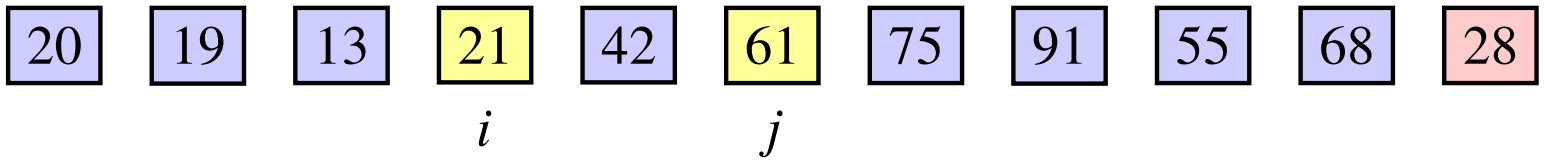
Scan:



Swap:

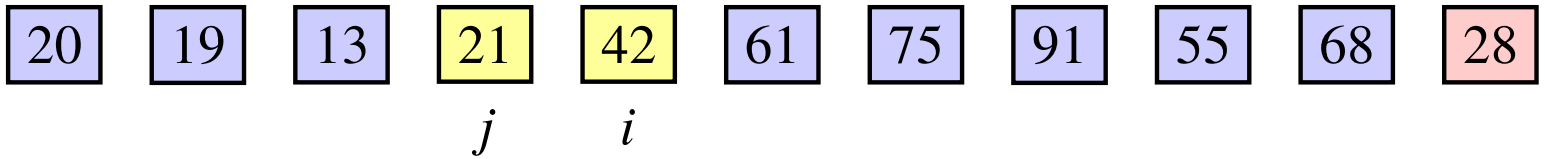


Scan and Swap:



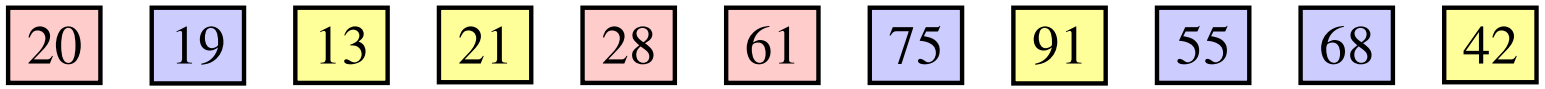
Quick Sort – Example

Scan and Swap:



Re-insert pivot, repeat: **(pass 2)**

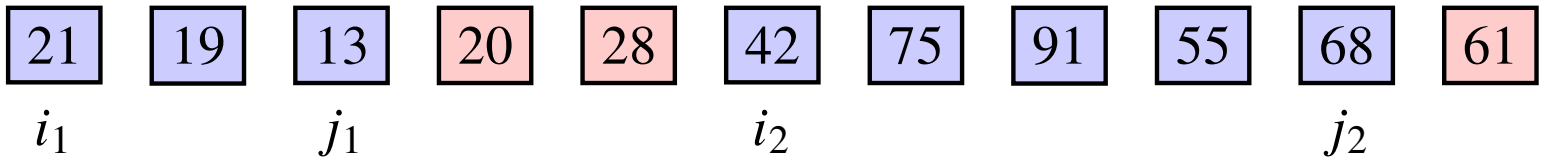
Median of three partitioning in each sublist:



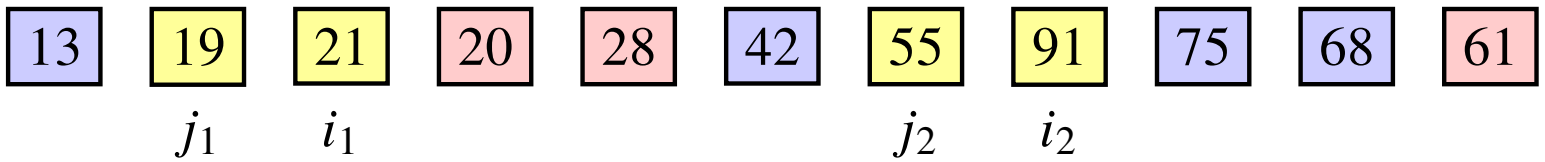
Swap pivots to the end:

Quick Sort – Example

Swap pivots to the end:



Scan and swap:



Re-insert pivots:

Quick Sort – Example

Cut-off Reached!: Perform insertion sort on the four remaining sublists.
Before:

After:

⇨ Notice that only one swap is required for the final insertion sort since each sublist is nearly sorted.

Quick Sort – Performance

- Quick sort seems like an efficient divide-and-conquer sorting algorithm
  - → However, worst case time is still takes $\Theta(n^2)$ on a sorted list.

Question: How can we compute the run-time performance of quick sort
Answer: Like the other recursive algorithms considered, we need to write a recurrence relation for the run-time.
Assumptions:

- $T(0) = T(1) = 1$
- Assume random pivot selection which divides the list into two lists of size i and $n - i - 1$ elements

Quick Sort – Performance

Recurrence Relation:

$$
\begin{aligned}
T(1) &= 1 \\
T(n) &=
\end{aligned}
$$

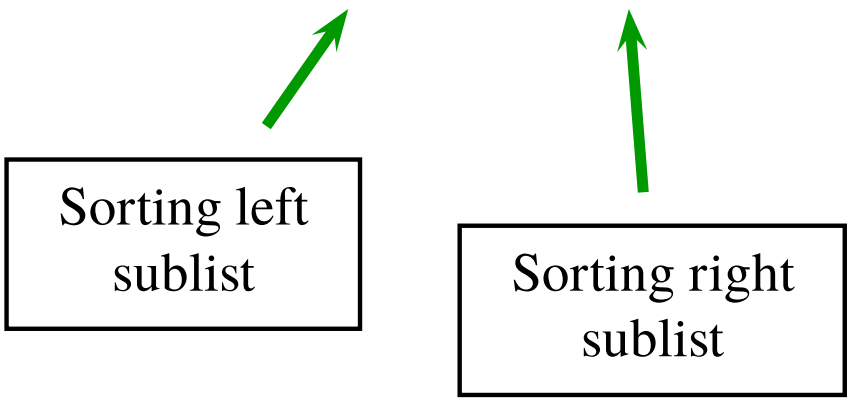
Recurrence Relation:

$$T(1) = 1$$
$$T(n) = T(i)$$

Sorting left sublist

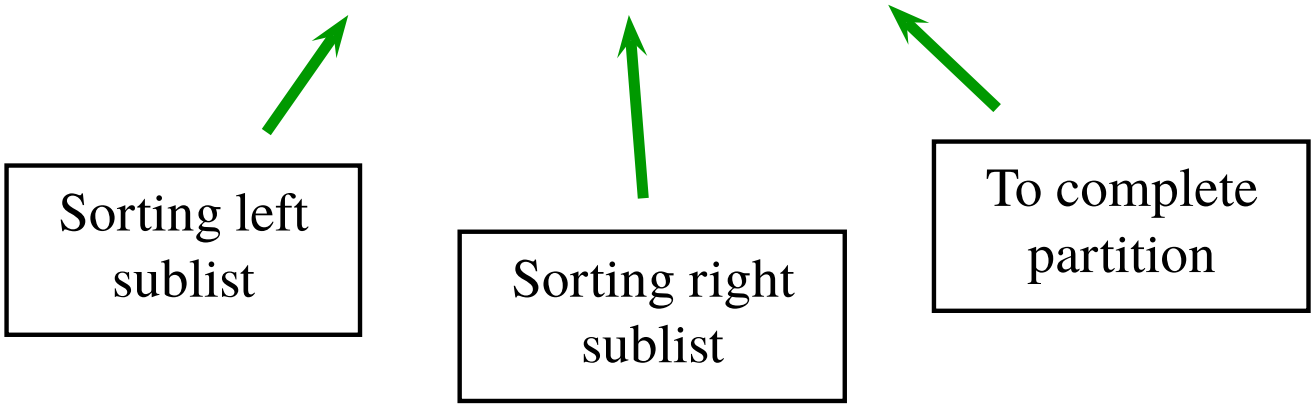
Recurrence Relation:

$$
\begin{aligned}
T(1) &= 1 \\
T(n) &= T(i) + T(n-i-1)
\end{aligned}
$$



Recurrence Relation:

$$
\begin{aligned}
T(1) &= 1 \\
T(n) &= T(i) + T(n-i-1) + cn
\end{aligned}
$$



Worst Case: When $i = 0$ or $i = n - 1$

- ⇨ In this case, the list is partitioned so that one sublist is very small and the other sublist is large
- ⇨ This is precisely what happened when we used quick sort on the initially sorted list

Example: If $i = 0$,

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

This can be simplified to

$$\begin{aligned} T(n) &= T(n-2) + c\left[(n-1) + n\right] + 2 \\ &= T(n-3) + c\left[(n-2) + (n-1) + n\right] + 3 \\ &= \cdots \\ &= T(1) + c\sum_{i=2}^{n} i + (n-1) \\ &= \Theta(n^2) \end{aligned}$$

McMaster University

Worst Case

A different way of solving the recursion, using **telescoping**

$$\begin{aligned} T(n) &= T(n-1) + cn + 1 \\ T(n-1) &= T(n-2) + c(n-1) + 1 \\ T(n-2) &= T(n-3) + c(n-2) + 1 \\ &\vdots \\ T(2) &= T(1) + c \cdot 2 + 1 \end{aligned}$$

After adding the above relations and canceling the like terms, one gets

$$\begin{aligned} T(n) &= T(1) + c\left[n + (n-1) + (n-2) + \cdots + 2\right] + 1 \cdot (n-1) \\ &= 1 + c\left(\frac{n(n+1)}{2} - 1\right) + (n-1) \\ &= \Theta(n^2). \end{aligned}$$

Best Case: When $i = n/2$

- In this case, the list is partitioned evenly: the left and right sublists have nearly the same length.

Example:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + cn \\ &= \Theta(n \log n) \end{aligned}$$

Average Case:

$$T(n) = T(i) + T(n-i-1) + cn$$

→ If we assume that all values are equally likely and that there is no bias in the partitioning.

→ The value of i can vary between 0 and $n-1$.

On average,

$$T_{\text{avg}}(n) = \frac{1}{n}\left[\sum_{j=0}^{n-1} T_{\text{avg}}(j) + T_{\text{avg}}(n-j-1)\right] + cn$$

Notice that each $T_{\text{avg}}(k)$ repeat twice, so

$$T_{\text{avg}}(n) = \frac{2}{n}\left[\sum_{j=0}^{n-1} T_{\text{avg}}(j)\right] + cn$$

Average Case:

$$T_{\text{avg}}(n) = \frac{2}{n}\left[\sum_{j=0}^{n-1} T_{\text{avg}}(j)\right] + cn$$

Multiply both sides by n to give

$$nT_{\text{avg}}(n) = 2\left[\sum_{j=0}^{n-1} T_{\text{avg}}(j)\right] + cn^2$$

To get rid of the summation, we can use a technique called **telescoping** (see proof of closed from for the geometric sum). Consider,

$$(n-1)T_{\text{avg}}(n-1) = 2\left[\sum_{j=0}^{n-2} T_{\text{avg}}(j)\right] + c(n-1)^2$$

Now,

$$nT_{\text{avg}}(n) - (n-1)T_{\text{avg}}(n-1) = 2T_{\text{avg}}(n-1) + 2cn - c$$

We now have $T_{\text{avg}}(n)$ in terms of $T_{\text{avg}}(n-1)$,

$$nT_{\text{avg}}(n) = (n+1)T_{\text{avg}}(n-1) + 2cn - c$$

To find a closed form, we need to telescope again. Consider dividing both sides by $n(n+1)$ to give

$$\frac{T_{\text{avg}}(n)}{n+1} = \frac{T_{\text{avg}}(n-1)}{n} + \frac{2c}{n+1} - \frac{c}{n(n+1)}$$

Telescoping again gives,

$$\begin{aligned}
\frac{T_{\text{avg}}(n-1)}{n} &= \frac{T_{\text{avg}}(n-2)}{n-1} + \frac{2c}{n} - \frac{c}{(n-1)n} \\
\frac{T_{\text{avg}}(n-2)}{n-1} &= \frac{T_{\text{avg}}(n-3)}{n-2} + \frac{2c}{n-1} - \frac{c}{(n-2)(n-1)} \\
&\vdots \\
\frac{T_{\text{avg}}(2)}{3} &= \frac{T_{\text{avg}}(1)}{2} + \frac{2c}{3} - \frac{c}{6}
\end{aligned}$$

Add the equations from the telescoping

$$\begin{aligned}
\frac{T_{\text{avg}}(n)}{n+1} &= \frac{T_{\text{avg}}(n-1)}{n} + \frac{2c}{n+1} - \frac{c}{n(n+1)} \\
\frac{T_{\text{avg}}(n-1)}{n} &= \frac{T_{\text{avg}}(n-2)}{n-1} + \frac{2c}{n} - \frac{c}{(n-1)n} \\
\frac{T_{\text{avg}}(n-2)}{n-1} &= \frac{T_{\text{avg}}(n-3)}{n-2} + \frac{2c}{n-1} - \frac{c}{(n-2)(n-1)} \\
&\vdots \\
\frac{T_{\text{avg}}(2)}{3} &= \frac{T_{\text{avg}}(1)}{2} + \frac{2c}{3} - \frac{c}{6},
\end{aligned}$$

to give the expression

$$\frac{T_{\text{avg}}(n)}{n+1} = \frac{T_{\text{avg}}(1)}{2} + 2c\sum_{i=3}^{n+1}\frac{1}{i} - c\sum_{i=2}^{n}\frac{1}{i(i+1)}$$

We can use these expression to give the bound,

$$c\sum_{i=1}^{n+1}\frac{1}{i} - c_2 \leq \frac{T_{\text{avg}}(n)}{n+1} \leq 2c\sum_{i=1}^{n+1}\frac{1}{i} + c_1.$$

$$c \sum_{i=1}^{n+1} \frac{1}{i} - c_2 \leq \frac{T_{\text{avg}}(n)}{n+1} \leq 2c \sum_{i=1}^{n+1} \frac{1}{i} + c_1.$$

Notice: The resulting summation is the **harmonic sum** that we saw in Math Review:

$$\mathcal{H}_n = \sum_{i=1}^{n} \frac{1}{i}$$
$$\ln(n+1) \leq \mathcal{H}_n \leq \ln n + 1$$

Thus, for n sufficiently large we have

$$\frac{c}{2} \ln n \leq c(\ln(n+2) - c_2) \leq \frac{T_{\text{avg}}(n)}{n+1} \leq 2c(\ln(n+1)+1) + c_1 \leq (6c + c_1) \ln n$$

$$c_2 n \ln n \leq \frac{c}{2}(n+1) \ln n \leq T_{\text{avg}}(n) \leq (6c + c_1)(n+1) \ln n \leq c_3 n \ln n$$

And so,

$$\boxed{T_{\text{avg}}(n) = \Theta(n \log n)}$$

Quick Sort – Performance

- On **average**, quick sort is an efficient sorting algorithm.
- Quick sort is an inherently recursive algorithm
  - → it is difficult to code this as an iterative algorithm (but not impossible - a stack can be used).
- The list is partitioned into variable length sub-lists (unlike merge sort), however, this can be done very efficiently.

The Master Theorem

- Up until now we have been computing runtime of divide and conquer algorithms by unfolding recursive formula. Is there any that can be said in general?

The Master Theorem

- Up until now we have been computing runtime of divide and conquer algorithms by unfolding recursive formula. Is there any that can be said in general?
- Consider a Divide and Conquer Algorithm where $T(n) = AT(n/B) + cn^k$ for $n > 1$
  - the problem is divided into $A \geq 1$ **sub-problems**;
  - each sub-problem has size n/B, for some $B > 1$;
  - The **overhead** takes $\Theta(n^k)$ time, where $k \geq 1$
    - the overhead is the work to split the problem and form the solution to the original problem based on sub-problems solutions).
  - n/B can be interpreted either as $\lfloor n/B \rfloor$ or as $\lceil n/B \rceil$

The Master Theorem

- Up until now we have been computing runtime of divide and conquer algorithms by unfolding recursive formula. Is there any that can be said in general?
- Consider a Divide and Conquer Algorithm where $T(n) = AT(n/B) + cn^k$ for $n > 1$
  - the problem is divided into $A \geq 1$ **sub-problems**;
  - each sub-problem has size n/B, for some $B > 1$;
  - The **overhead** takes $\Theta(n^k)$ time, where $k \geq 1$
    - the overhead is the work to split the problem and form the solution to the original problem based on sub-problems solutions).
  - n/B can be interpreted either as $\lfloor n/B \rfloor$ or as $\lceil n/B \rceil$
- Then:

$$T(n) = \begin{cases} \Theta(n^{\log_B A}) & : \text{ for } A > B^k \\ \Theta(n^k \log n) & : \text{ for } A = B^k \\ \Theta(n^k) & : \text{ for } A < B^k \end{cases}$$

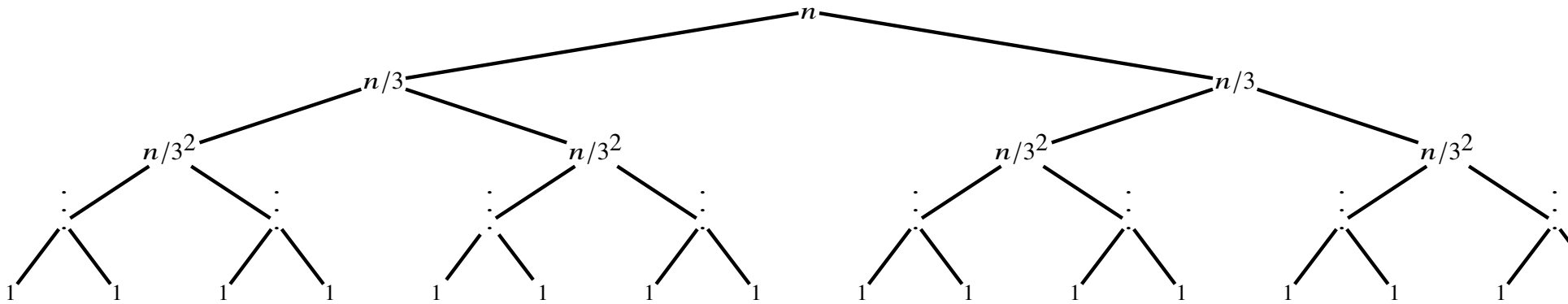
Divide and Conquer with Polynomial Overhead

- For Merge Sort and best case QuickSort: $A = 2$, $B = 2$, $k = 1$. Falls under Case 2.
  - → Complexity: $\Theta(n \log n)$

Divide and Conquer with Polynomial Overhead

- Consider the special case: $k = 1$, $A = 2$ and $B = 3$ (Case 3).
  - → Then $T(n) = 2T(n/3) + n$ for $n > 1$, $T(1) = 1$.
- Every node in the recursion tree (except for the leaves) has two children.
- Assume that $n = 3^\ell$. Then tree levels are: $0, 1, 2, \cdots, \ell$.
- Number of operations per non-leaf node at level i: $n/3^i$. Number of nodes at level i: 2^i.
- Number of operations per leaf: 1. Number of leaves: 2^ℓ.

Divide and Conquer with Polynomial Overhead



$$\begin{aligned} T(n) &= 2^0 \times n/3^0 + 2^1 \times n/3^1 + 2^2 \times n/3^2 + \cdots + 2^{\ell-1} \times n/3^{\ell-1} + 2^\ell \times 1 \\ &= n\left\{\left(\frac{2}{3}\right)^0 + \left(\frac{2}{3}\right)^1 + \left(\frac{2}{3}\right)^2 + \cdots + \left(\frac{2}{3}\right)^{\ell-1}\right\} + 2^\ell \\ &= n\frac{1-\left(\frac{2}{3}\right)^\ell}{1-\frac{2}{3}} + 2^\ell \end{aligned}$$

- Then

$$\lim_{n\to\infty} \frac{T(n)}{n} =$$

$$\lim_{n\to\infty} \frac{n\frac{1-\left(\frac{2}{3}\right)^\ell}{1-\frac{2}{3}} + 2^\ell}{n} =$$

$$\lim_{\ell\to\infty} \left(\frac{1-\left(\frac{2}{3}\right)^\ell}{1-\frac{2}{3}} + \frac{2^\ell}{3^\ell}\right) = 3$$

Thus, $T(n) = \Theta(n)$. Q.E.D.

Quick Select

Problem: Find an efficient algorithm to find the k-th largest element in an unsorted array of n integers.

Quick Select

Problem: Find an efficient algorithm to find the k-th largest element in an unsorted array of n integers.

Solution 1:

- Use Quick Sort to sort the array – $\Theta(n \log n)$ time on average
- Then, pick the k-th element – $\Theta(1)$ time.
- This technique is inefficient since you are sorting the entire list needlessly
- Is there a more efficient way to solve this problem?

Solution 2: **Quick Select**

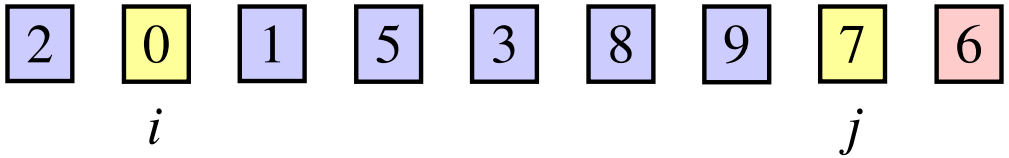
- This technique is based on the quick sort partitioning
- Pick a random pivot and partition the list, L, into two sub-lists, L_1 and L_2, as in quick sort (can use same algorithms as in quick sort to select pivot).
- If $|L_1| \geq k$, then the k-th smallest element is in L_1 and it can be found be recursively performing quick select in L_1. Else perform quick select in L_2 to find the $(k - |L_1| - 1)$-th element.

Quick Select – Example

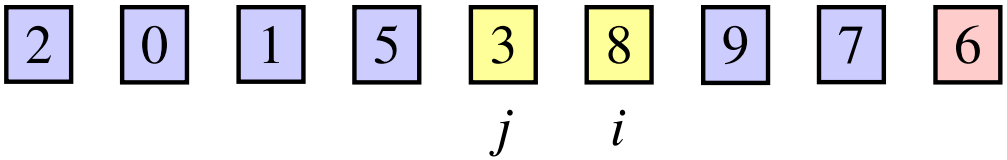
Find: the 8^{th} largest element in the list. Use first element as the pivot.

Select and hide pivot:

Scan and swap:



Scan:



Restore pivot:

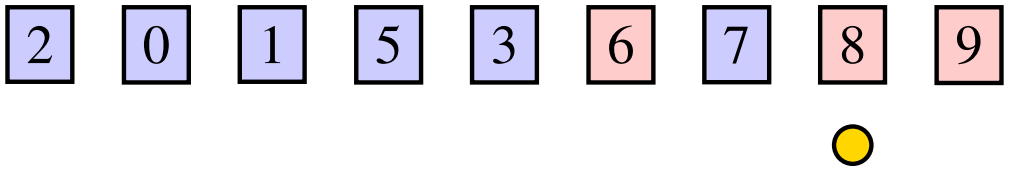
Quick Select – Example

Repeat Quick Select: Look for $8 - 5 - 1 = 2$-nd largest element in L_2.

Scan and swap:

Repeat Quick Select: Look for 2-nd largest element in remaining sub-list

Repeat Quick Select: Look for 2-nd largest element in remaining sub-list



- Found 8-th largest element by doing successive partitioning of the list with respect to the chosen pivot value.

Performance:

- There are on average $\log n$ partitions which are made (if the pivot is selected wisely).
- The number of comparison required is the length of each sub-list. Therefore at best,

$$\begin{aligned} T(n) &= n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots \\ &= n \sum_{i=0}^{k} 2^{-i} \\ &\leq n \sum_{i=0}^{\infty} 2^{-i} \\ &= 2n = O(n) \end{aligned}$$

- So, it is possible to use an identical partitioning strategy to find the k-th largest element in an unsorted list in **linear time** on average.

Bucket Sort

- The best sorting algorithms considered thus far have had run-time on the order of $\Theta(n \log n)$
  - → It can be shown that for general purpose sorting algorithms that use only comparisons require $\Omega(n \log n)$ time.
- However, it is possible to do better if something is known about the input data set

Example: **Bucket Sort** (also known as *Counting Sort*)
Assume all integers in the list are in the range 0 to $k-1$, for some known k.

- Create a table with k entries to hold the counts of integers in the range 0 to $k-1$.
- Initialize all counts to 0.
- Go through the list and if the entry has value i, increment the count of i.

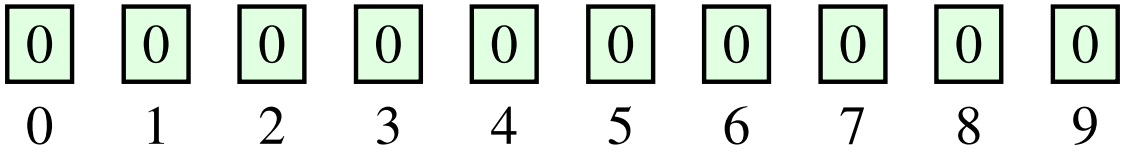
Bucket Sort

Sort: A list with elements from $0, 1, \ldots, 9$.

Bucket Sort

Sort: A list with elements from $0, 1, \ldots, 9$.

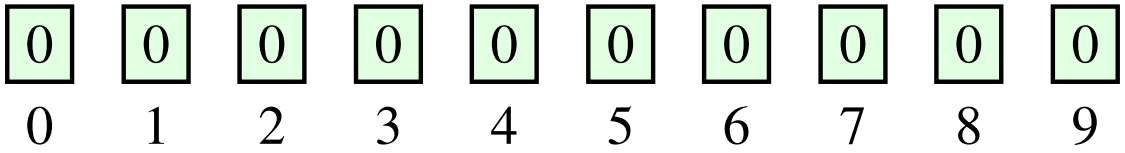
Create an array to hold the counts of each type of element:



Bucket Sort

Sort: A list with elements from $0, 1, \ldots, 9$.

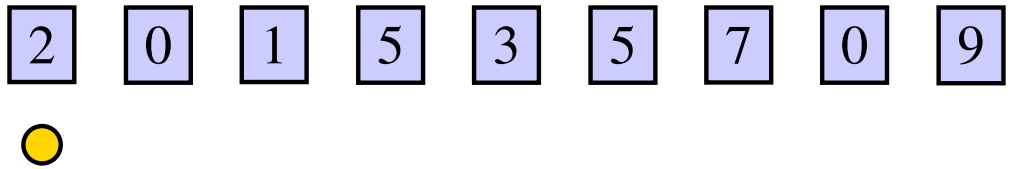
Create an array to hold the counts of each type of element:



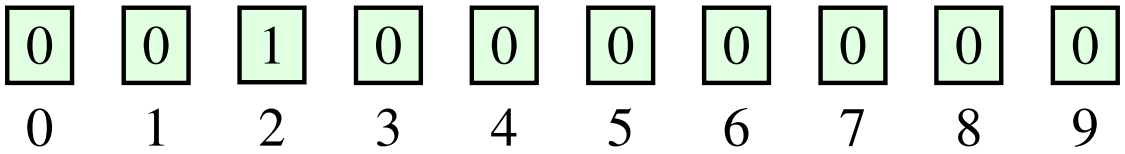
Loop through all elements in the list and increment appropriate index location

Bucket Sort

Sort: A list with elements from $0, 1, \ldots, 9$.



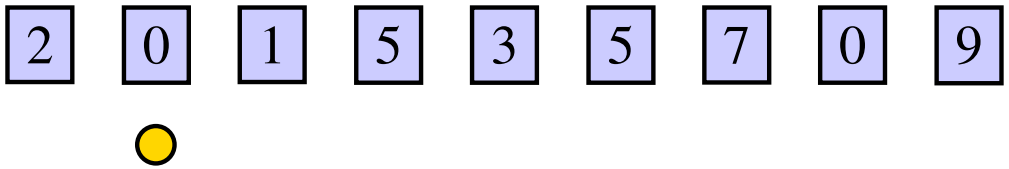
Create an array to hold the counts of each type of element:



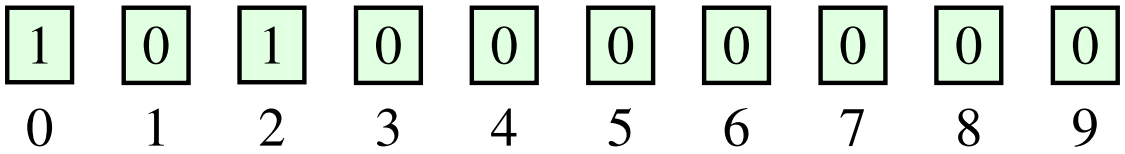
Loop through all elements in the list and increment appropriate index location

Bucket Sort

Sort: A list with elements from $0, 1, \ldots, 9$.



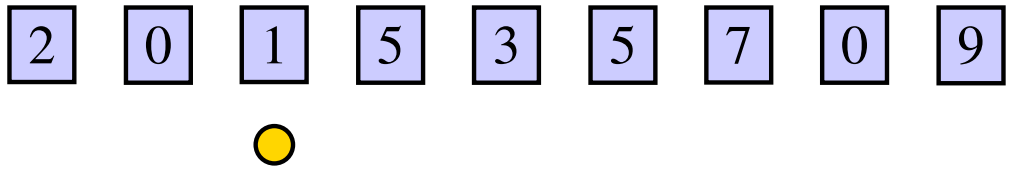
Create an array to hold the counts of each type of element:



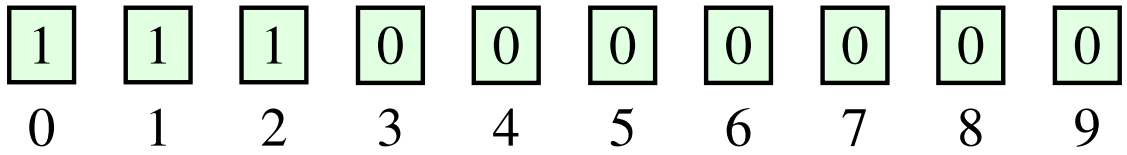
Loop through all elements in the list and increment appropriate index location

Bucket Sort

Sort: A list with elements from $0, 1, \ldots, 9$.



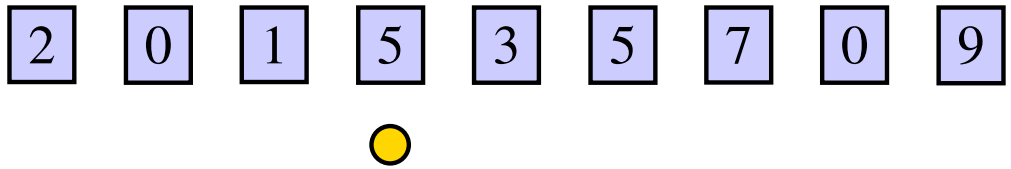
Create an array to hold the counts of each type of element:



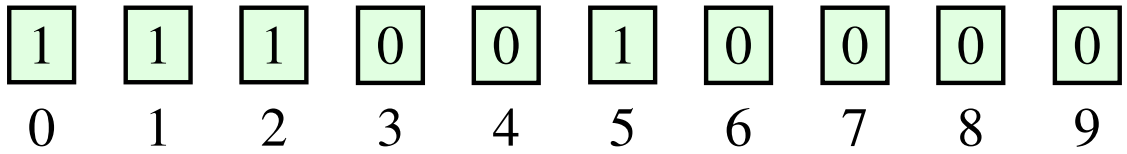
Loop through all elements in the list and increment appropriate index location

Bucket Sort

Sort: A list with elements from $0, 1, \ldots, 9$.



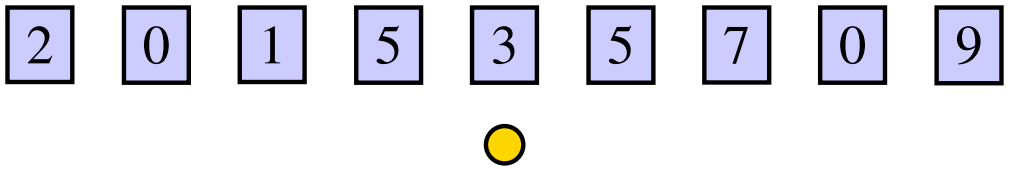
Create an array to hold the counts of each type of element:



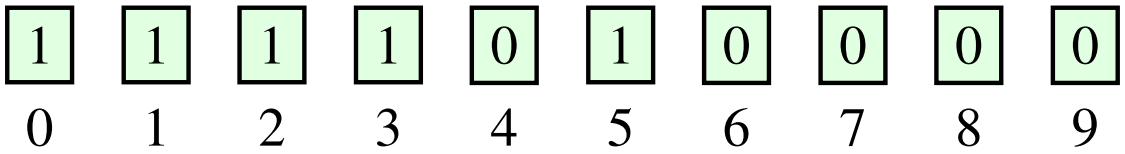
Loop through all elements in the list and increment appropriate index location

Bucket Sort

Sort: A list with elements from $0, 1, \ldots, 9$.



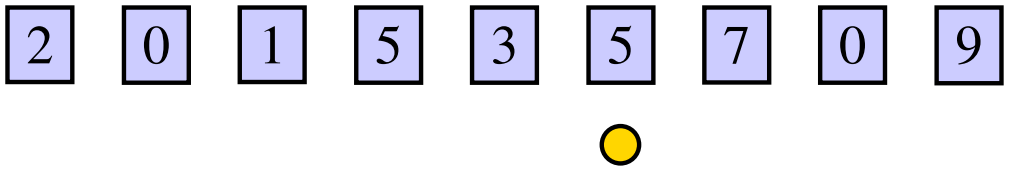
Create an array to hold the counts of each type of element:



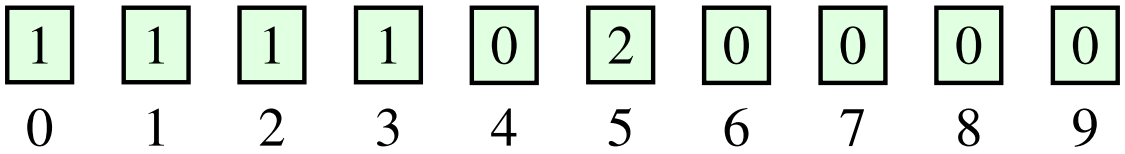
Loop through all elements in the list and increment appropriate index location

Bucket Sort

Sort: A list with elements from $0, 1, \ldots, 9$.



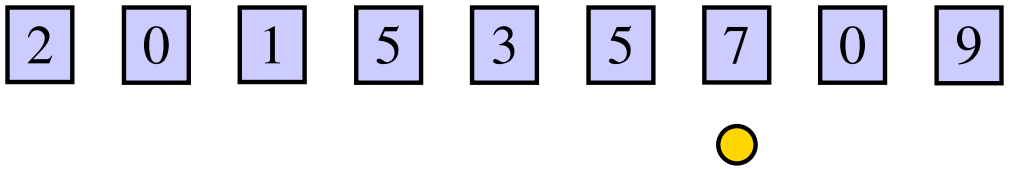
Create an array to hold the counts of each type of element:



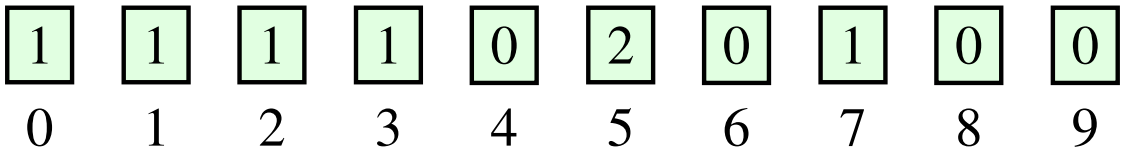
Loop through all elements in the list and increment appropriate index location

Bucket Sort

Sort: A list with elements from $0, 1, \ldots, 9$.



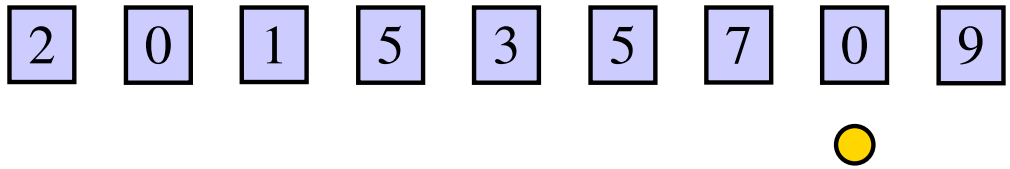
Create an array to hold the counts of each type of element:



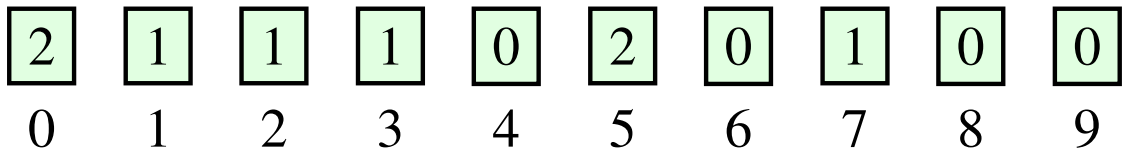
Loop through all elements in the list and increment appropriate index location

Bucket Sort

Sort: A list with elements from $0, 1, \ldots, 9$.



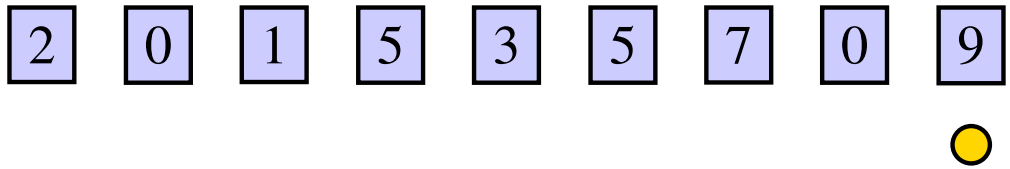
Create an array to hold the counts of each type of element:



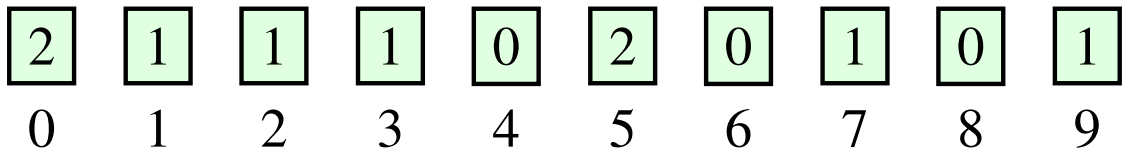
Loop through all elements in the list and increment appropriate index location

Bucket Sort

Sort: A list with elements from $0, 1, \ldots, 9$.



Create an array to hold the counts of each type of element:



Loop through all elements in the list and increment appropriate index location

Bucket Sort

Sort: A list with elements from $0, 1, \ldots, 9$.

2	0	1	5	3	5	7	0	9

Create an array to hold the counts of each type of element:

2	1	1	1	0	2	0	1	0	1
0	1	2	3	4	5	6	7	8	9

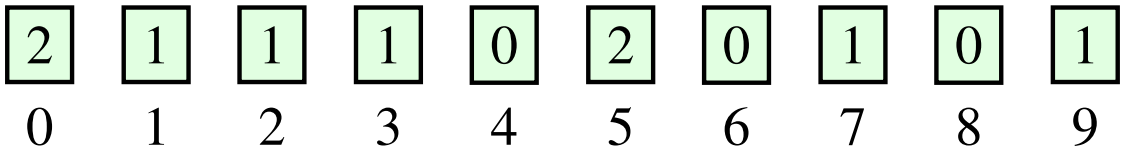
Loop through all elements in the list and increment appropriate index location
Output sorted list.

0	0	1	2	3	5	5	7	9

Bucket Sort

Sort: A list with elements from $0, 1, \ldots, 9$.

Create an array to hold the counts of each type of element:



Loop through all elements in the list and increment appropriate index location
Output sorted list.

☞ Runs in $\Theta(n+k)$ time, $\Theta(k)$ extra memory. If $k = O(n)$, then running time is $\Theta(n)$.

Radix Sort

- A generalization of *Bucket Sort* – instead of having one element per bucket have a group of elements
- Sort numbers of k digits, represented in base b, into b buckets.
- Pass through the list k times looking at digits from least to most significant
  - → Group the elements in bins depending on size of k-th digit
  - → Repeat for each digit

Radix Sort

Example: Sort the list. $k = 3$, $b = 10$.

631	104	481	492	485	135	685	021	405

Example: Sort on the least significant digit

0	1	2	3	4	5	6	7	8	9
	631 481 021	492		104	485 135 685 405				

... to give the list

631	481	021	492	104	485	135	685	405

☞ Notice, the list is now sorted by the 1's value

Example: Sort on the next significant digit.

631	481	021	492	104	485	135	685	405

0	1	2	3	4	5	6	7	8	9
104 405		021	631 135					481 485 685	492

... to give the list

104	405	021	631	135	481	485	685	492

☞ Notice, the list is now sorted by the 10's value

Example: Sort on the most significant digit.

104	405	021	631	135	481	485	685	492

0	1	2	3	4	5	6	7	8	9
021	104 135			405 481 485 492		631 685			

... to give the **sorted** list

021	104	135	405	481	485	492	631	685

- ☞ Notice, the list is now sorted by the 100's value
- ☞ Run-time: $\Theta(k(n+b))$, Extra Memory: $\Theta(n+b)$ extra memory.
- ☞ If $k = \Theta(1)$ and $b = \Theta(1)$, then run-time and extra memory: $\Theta(n)$

Sorting

- In this first part of the topic, we have seen a number of important sorting algorithms.
- Sorting is a fundamental activity in a large number of computer systems.
- The sorting algorithm that is chosen depends on the size of the list.
  - → For Quick Sort, we saw to switching to insertion sort when the list is "almost sorted" can lead to performance gains.
- In most cases, quick sort is a good technique for sorting general lists.
- Sub $O(n \log n)$ performance is possible, as in bucket and radix sort, if the input alphabet is limited.
- ☞ In the next part on *Sorting*, we will consider a new data structure call a **heap** to develop an efficient sorting algorithm.