Sorting and Binary Search

Sorting

You are given an array and want to sort it:

```
[2, 5, 1, 3, 8, 6, 8, 4] -> [1, 2, 3, 4, 6, 8, 8]
```

Bubble Sort

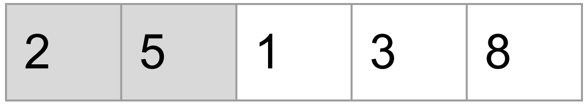
Bubble sort

Loop over the array, if two elements are in the wrong order then swap them. Repeat this process until the array is sorted.

2	5	1	3	8

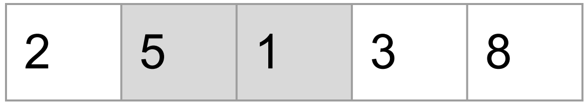
Bubble sort

Loop over the array, if two elements are in the wrong order then swap them.
Repeat this process until the array is sorted.



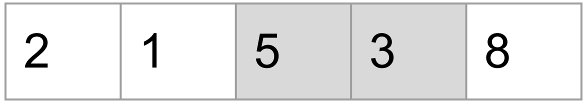
Bubble sort

Loop over the array, if two elements are in the wrong order then swap them.
Repeat this process until the array is sorted.



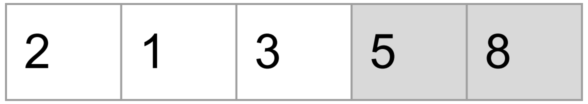
Bubble sort

Loop over the array, if two elements are in the wrong order then swap them.
Repeat this process until the array is sorted.



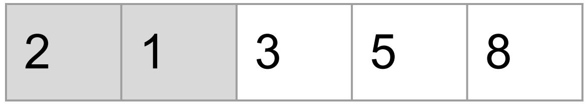
Bubble sort

Loop over the array, if two elements are in the wrong order then swap them.
Repeat this process until the array is sorted.



Bubble sort

Loop over the array, if two elements are in the wrong order then swap them.
Repeat this process until the array is sorted.



Bubble sort

Loop over the array, if two elements are in the wrong order then swap them.
Repeat this process until the array is sorted.

1	2	3	5	8

Bubble Sort: Time Complexity

Every time we sweep over the array, the suffix that is sorted increases by at least 1. Since each sweep over the array takes O(N) time and we have to do at most O(N) sweeps, the algorithm will take at most **O(N^2)** time.

We can show that this worst case complexity is achieved by looking at a reverse sorted array:

7	6	5	4	3	2	1

Bubble Sort: Implementation

C++

```cpp
void bubbleSort(int *arr, int sz){
    while(true){
        int numSwaps = 0;
        for (int i = 1; i < sz; i++) {
            if(arr[i] < arr[i-1]){
                swap(arr[i-1], arr[i]);
                numSwaps++;
            }
        }
        // If no elements were swapped in the
        // last sweep, the array must be sorted
        if(numSwaps == 0) break;
    }
}
```

Python

```python
def bubbleSort(arr):
    while True:
        numSwaps = 0
        for i in range(1, len(arr)):
            if arr[i] < arr[i-1]:
                # This is how you swap two numbers in Python
                arr[i-1], arr[i] = arr[i], arr[i-1]
                numSwaps += 1

        # If no elements were swapped in the
        # last sweep, the array must be sorted
        if numSwaps == 0:
            break
```

Merge Sort

Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

1	2	5	7	11	12

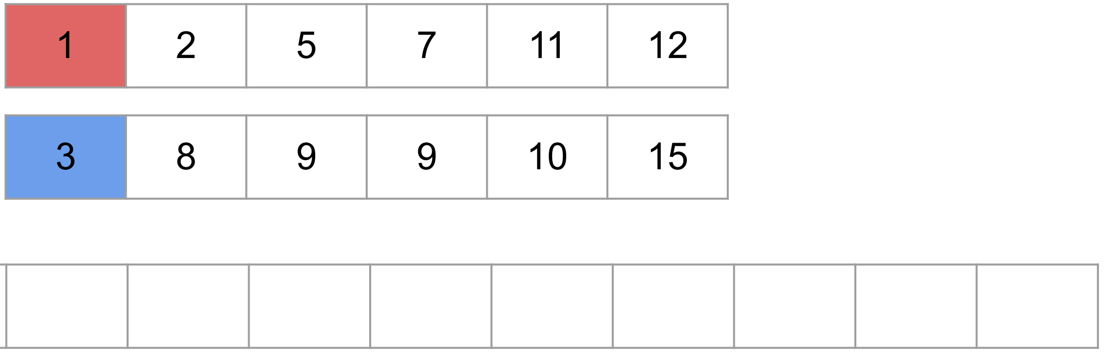
3	8	9	9	10	15

Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

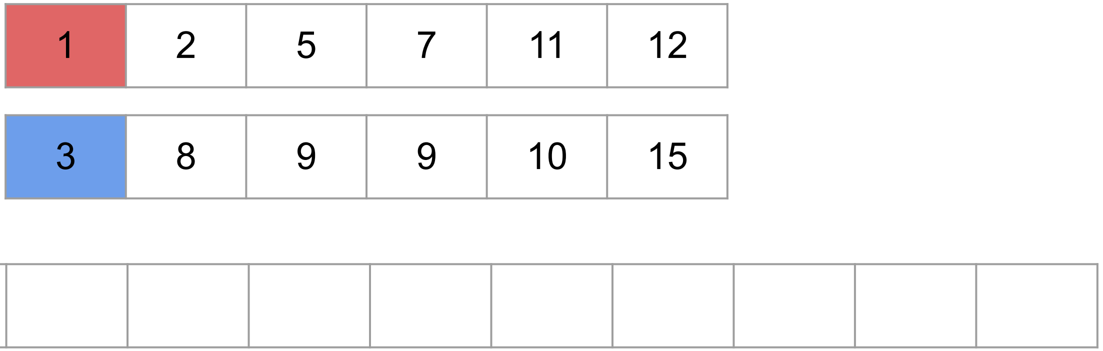


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

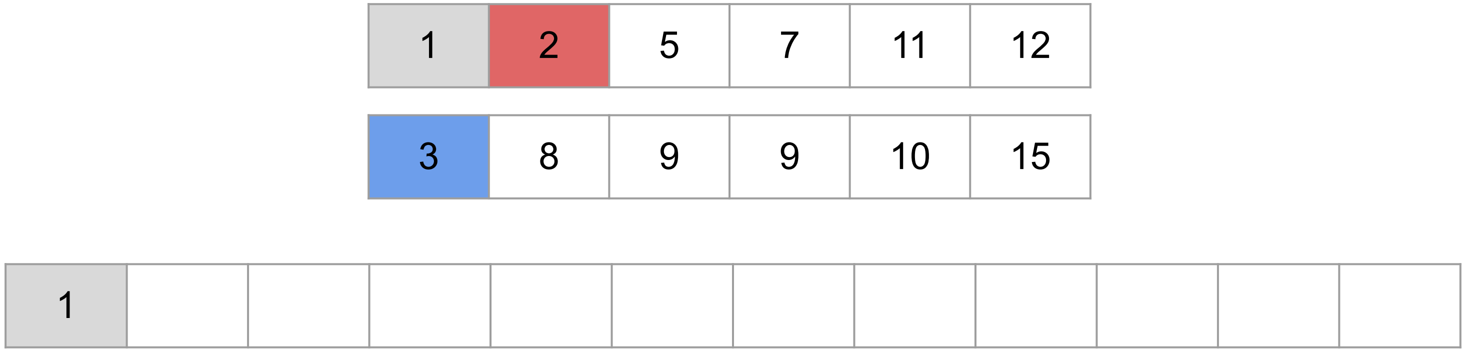


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

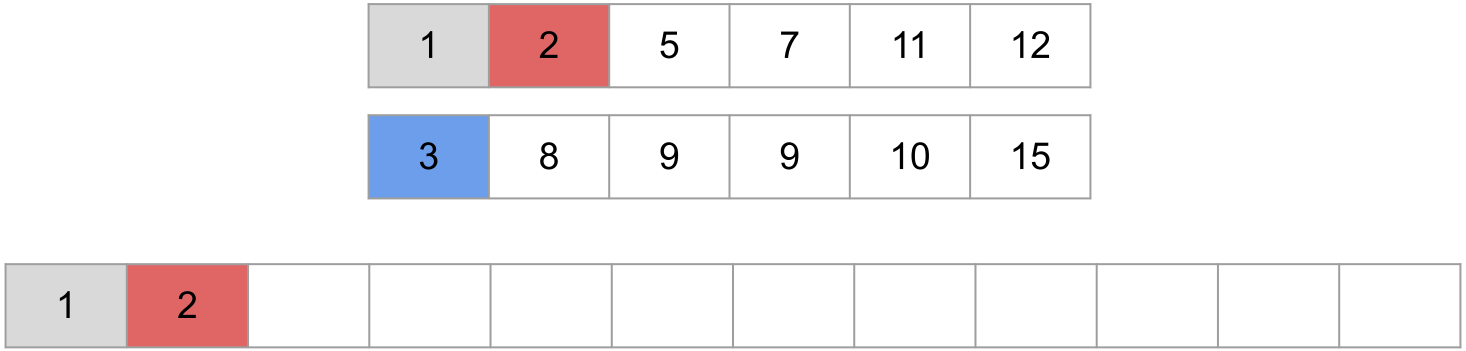


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

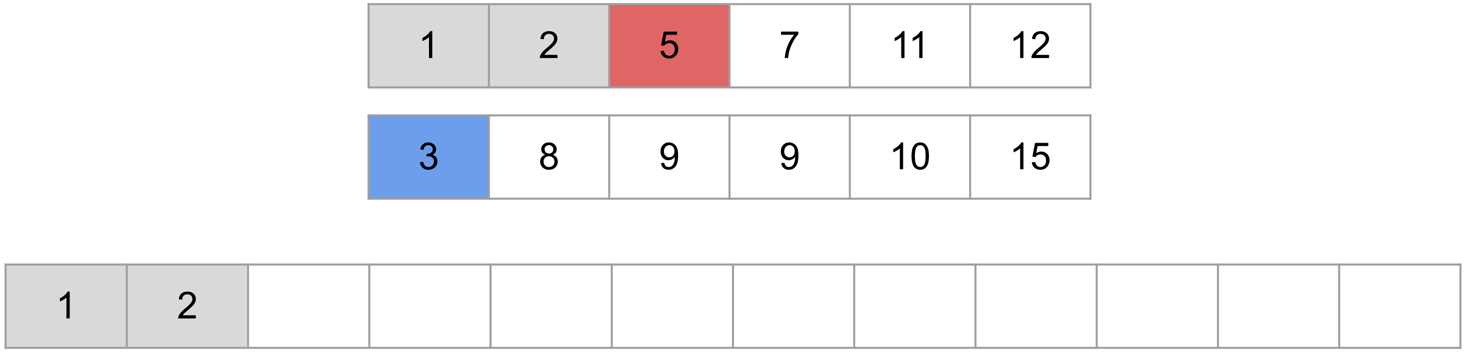


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

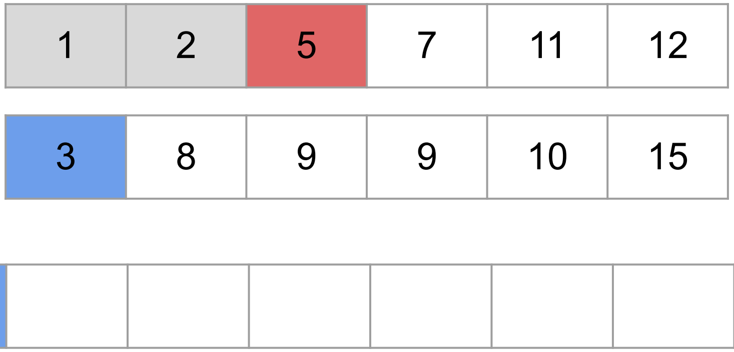


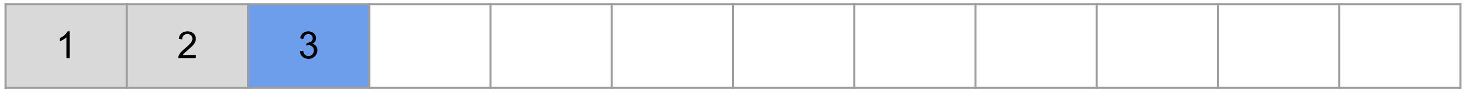
Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.



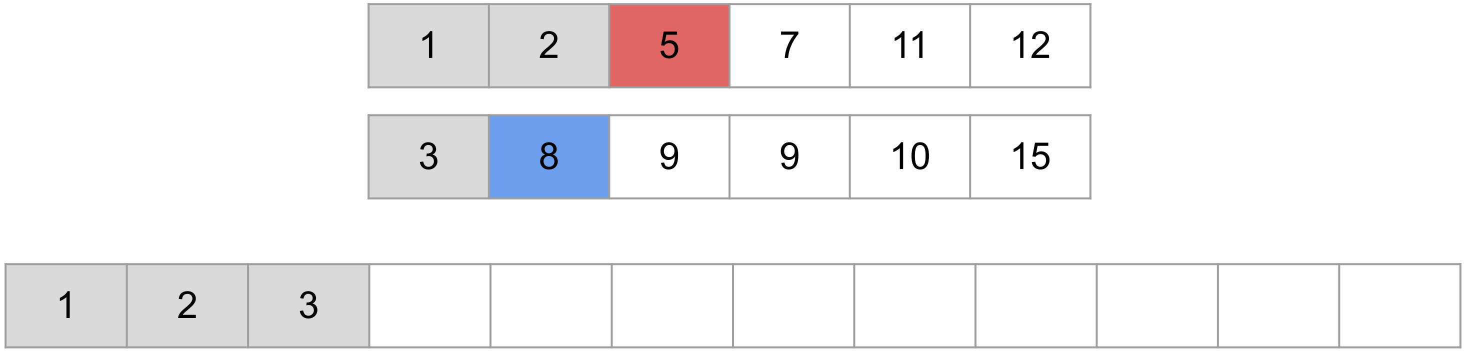


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

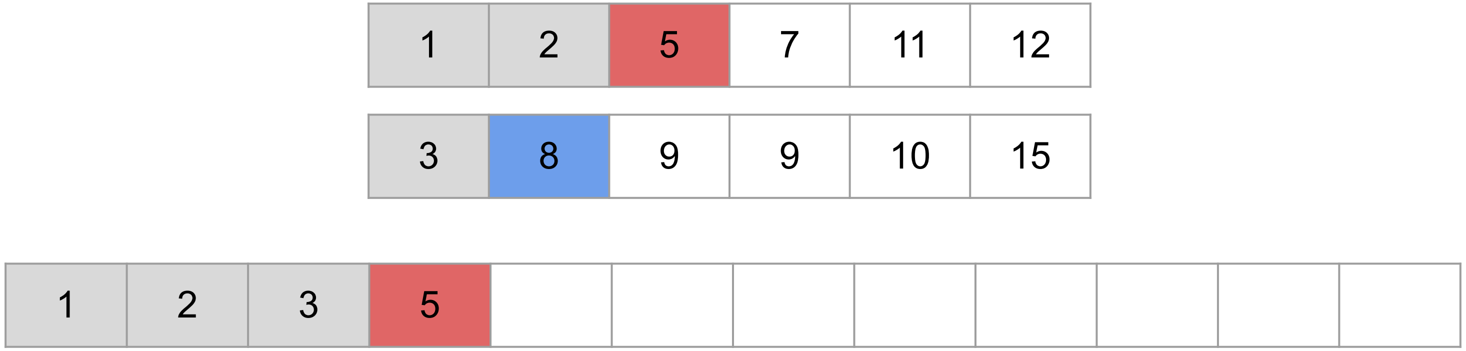


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

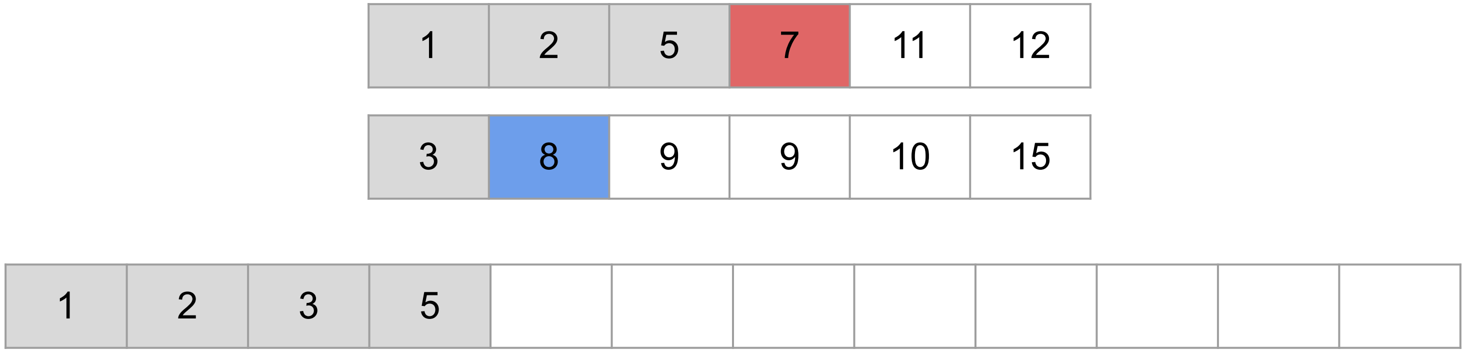


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

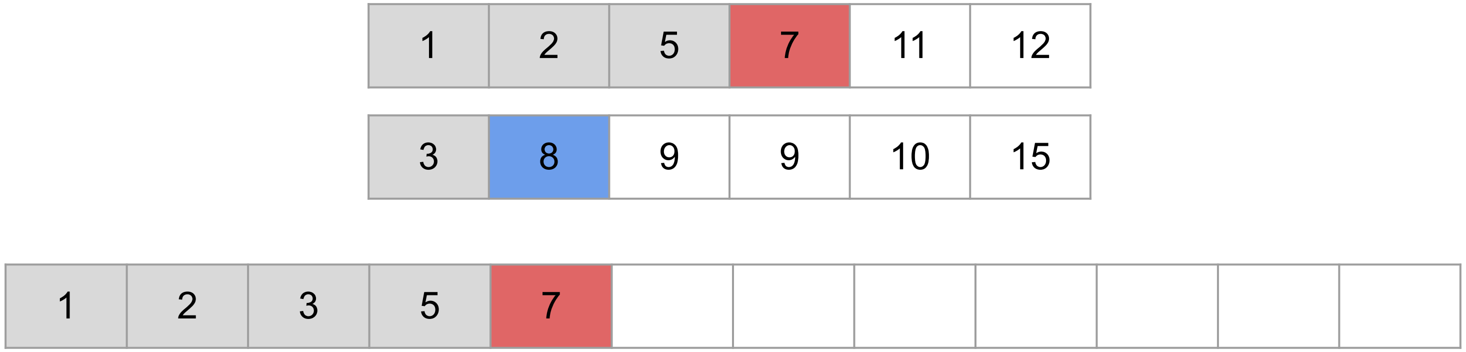


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

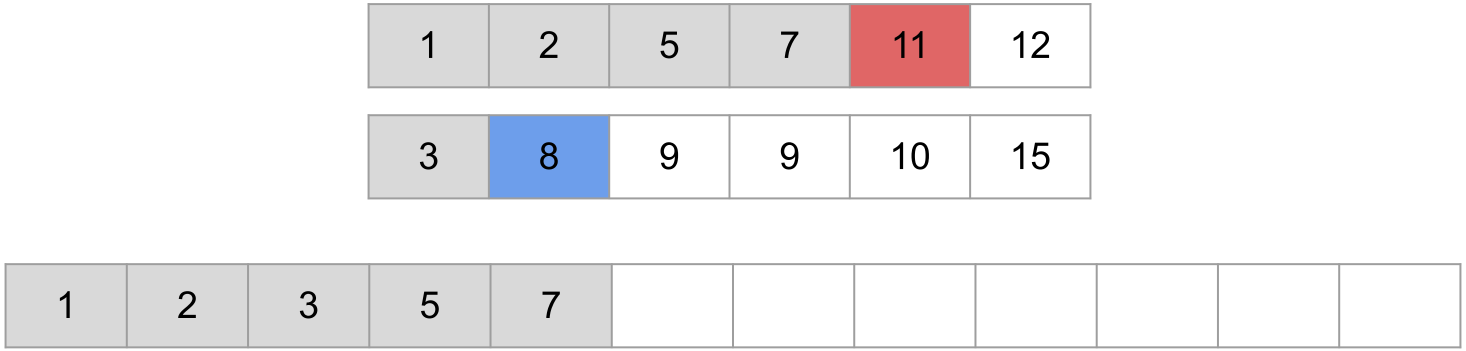


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

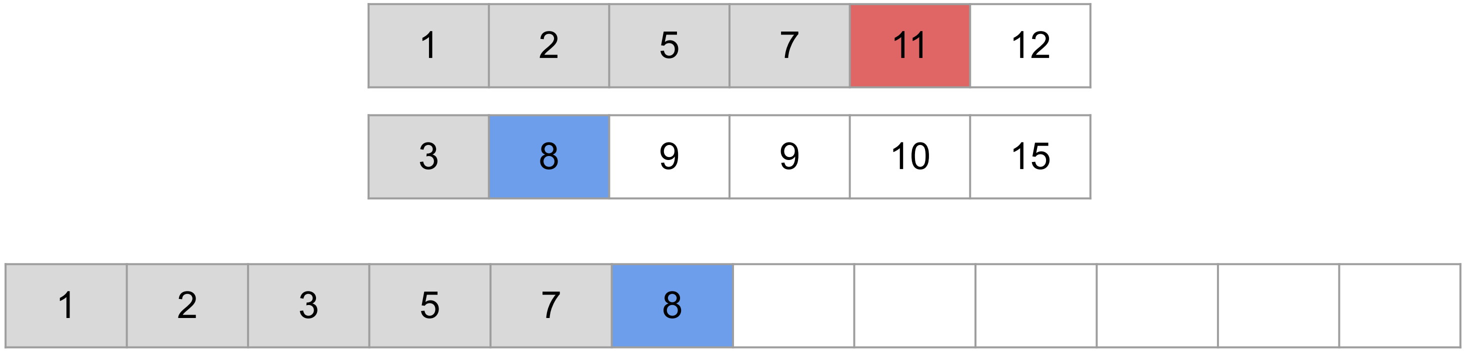


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

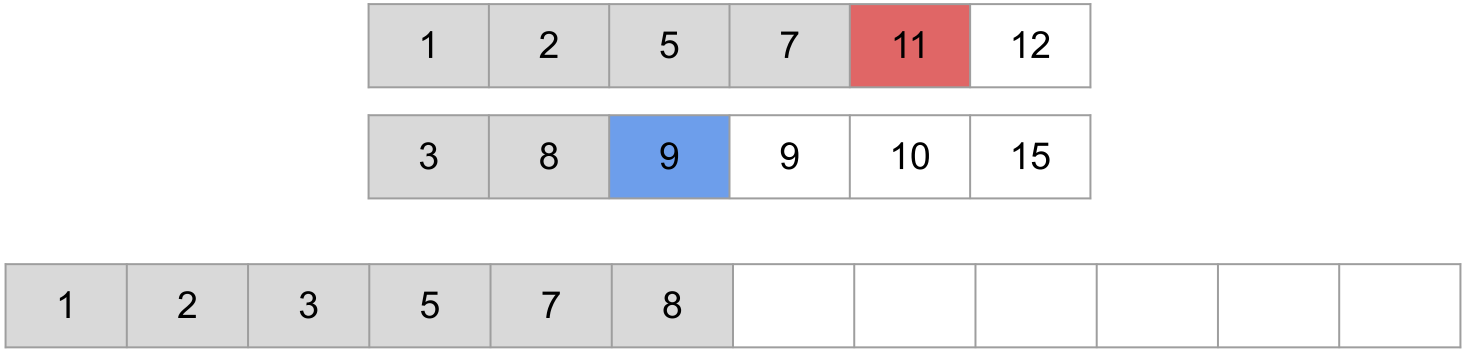


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

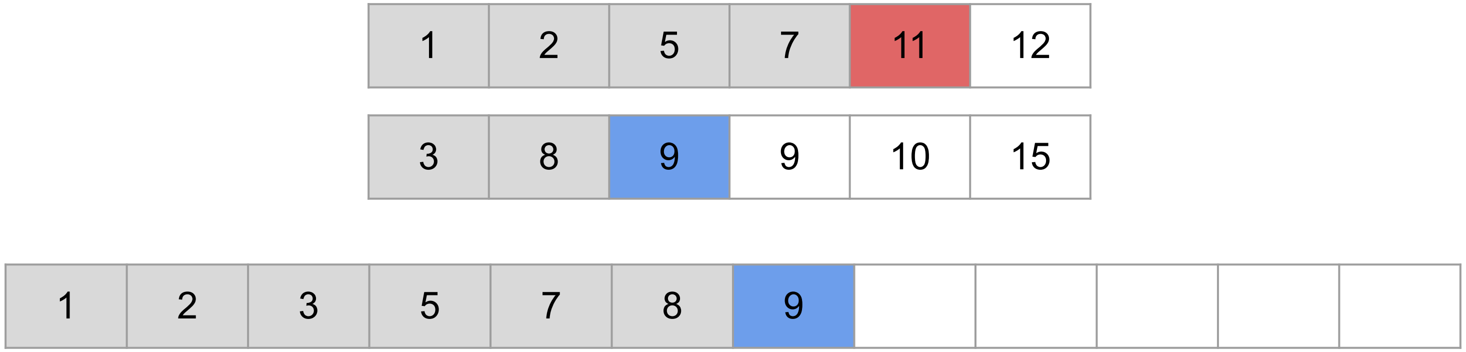


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

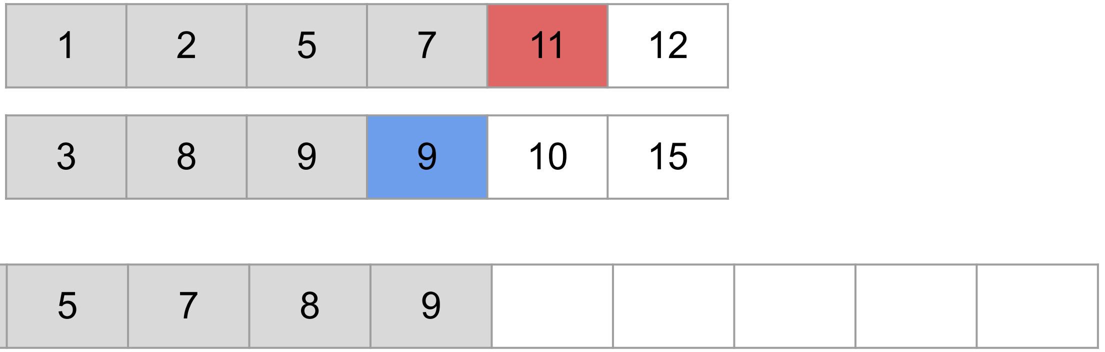


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

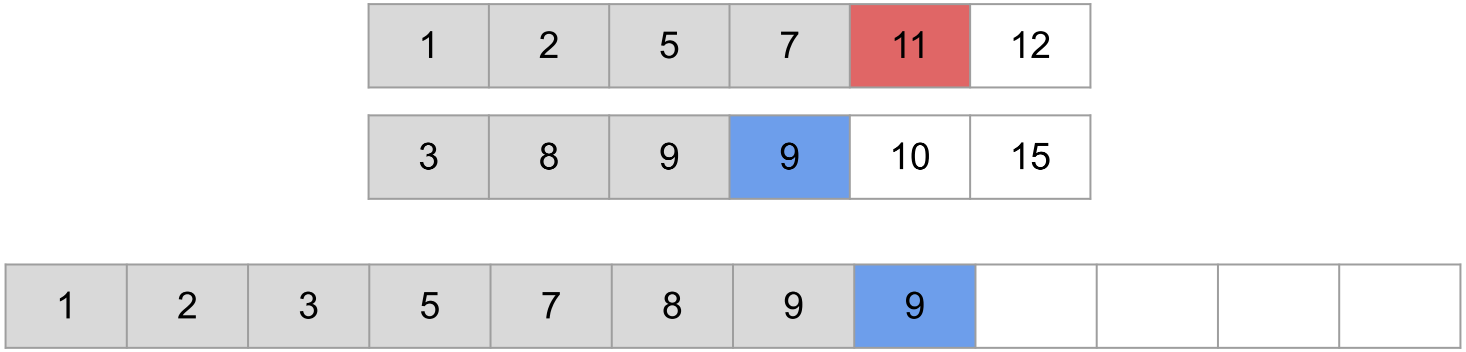


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

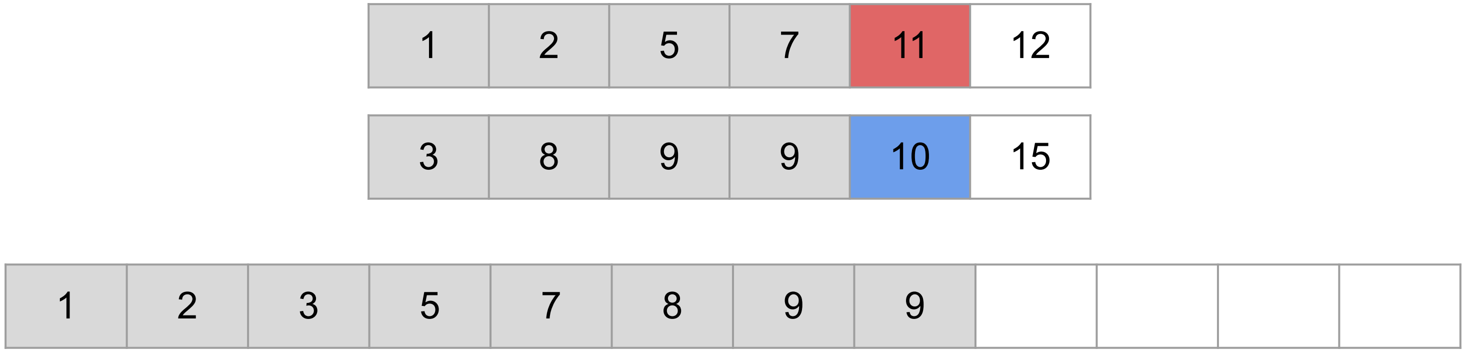


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

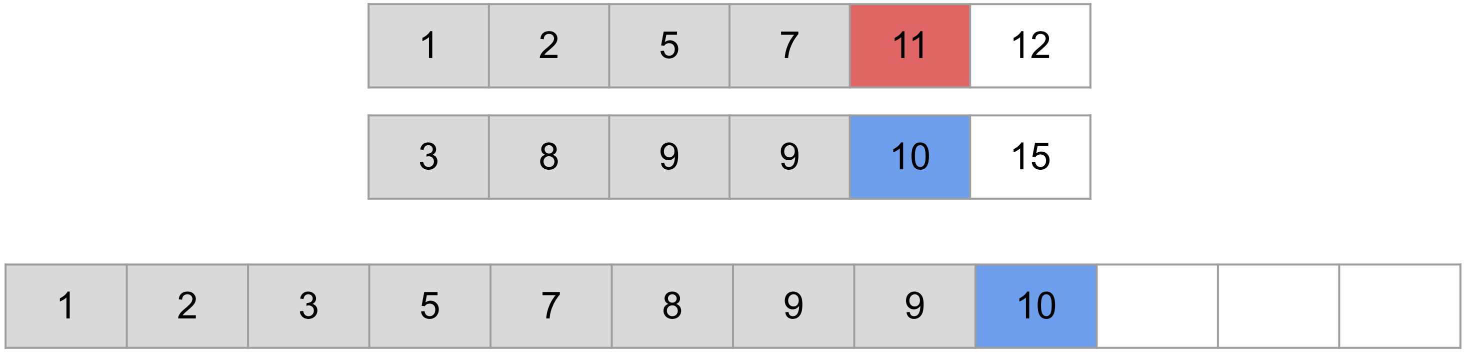


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

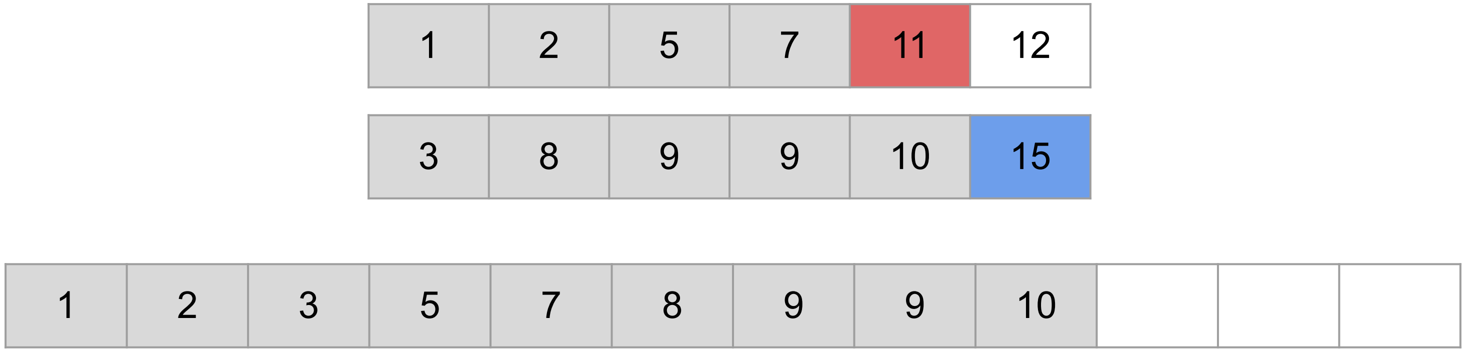


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

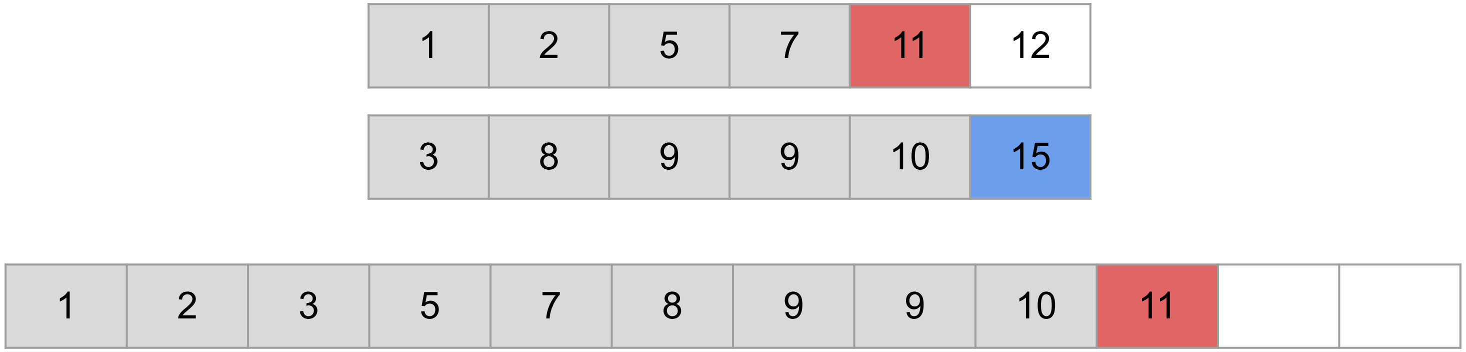


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

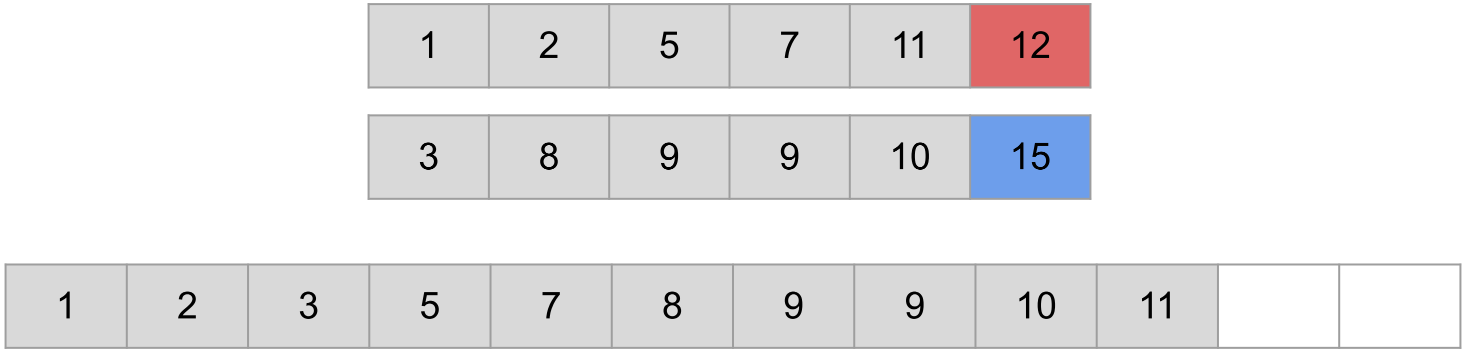


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

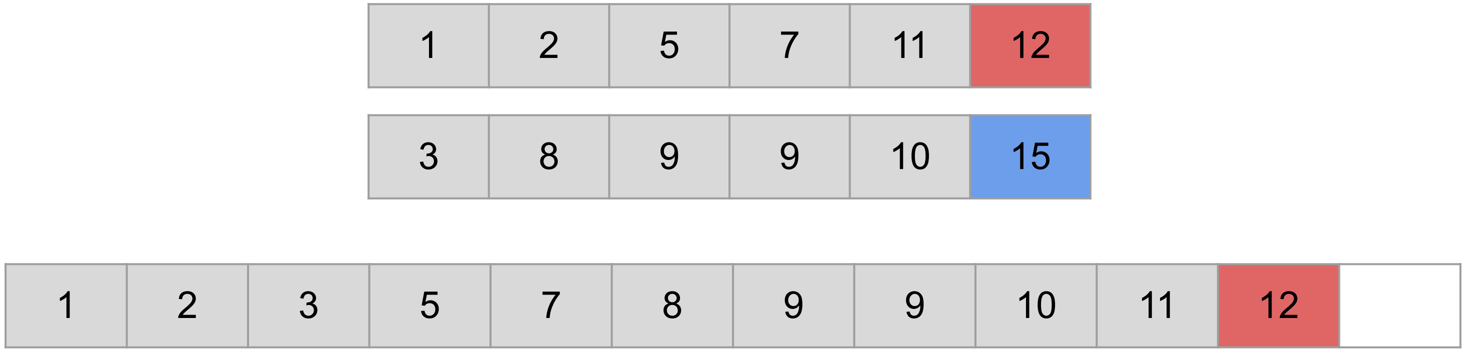


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

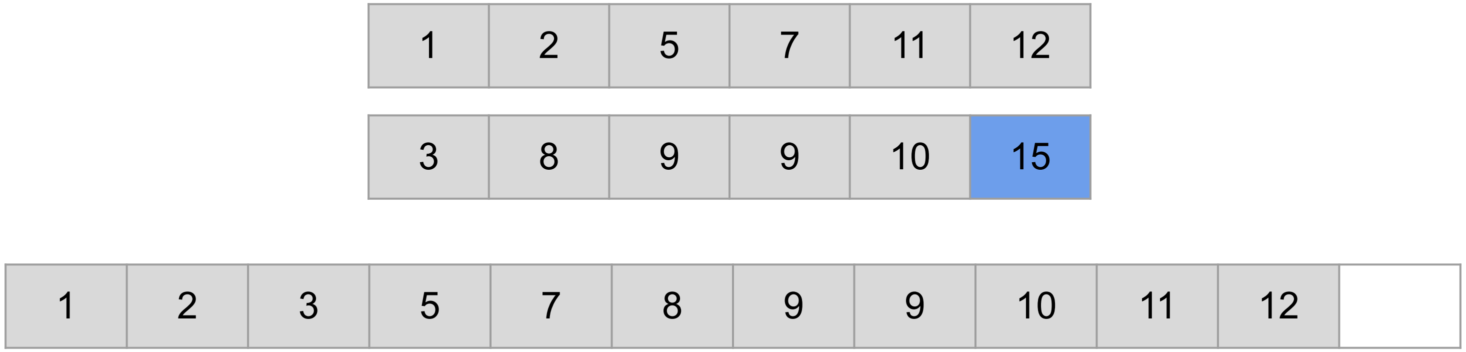


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

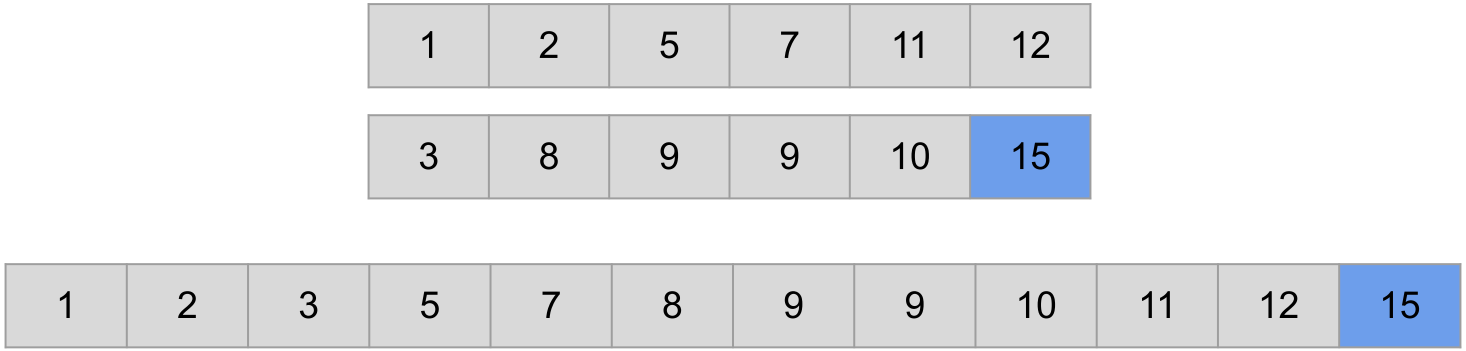


Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.



Merge Sort: Merging Two Sorted Arrays

Merge sort relies on the following concept:

Given two sorted arrays of length N and M, it is possible to “merge” them into a single sorted array of length N + M in O(N + M) time.

We construct the result array in order, starting from the smallest element. To do so, we keep a pointer/index for each of the two input arrays, which point to the next smallest element in that array that have not yet been added to the result array. When deciding what the next element of the result array should be, we can simply take the minimum of the values at the two pointers and advance that pointer by one. This is clearly optimal as each of those two values will be smaller than or equal to all of the other remaining values in their respective arrays.

1	2	5	7	11	12

3	8	9	9	10	15

1	2	3	5	7	8	9	9	10	11	12	15

Merge Sort: Algorithm

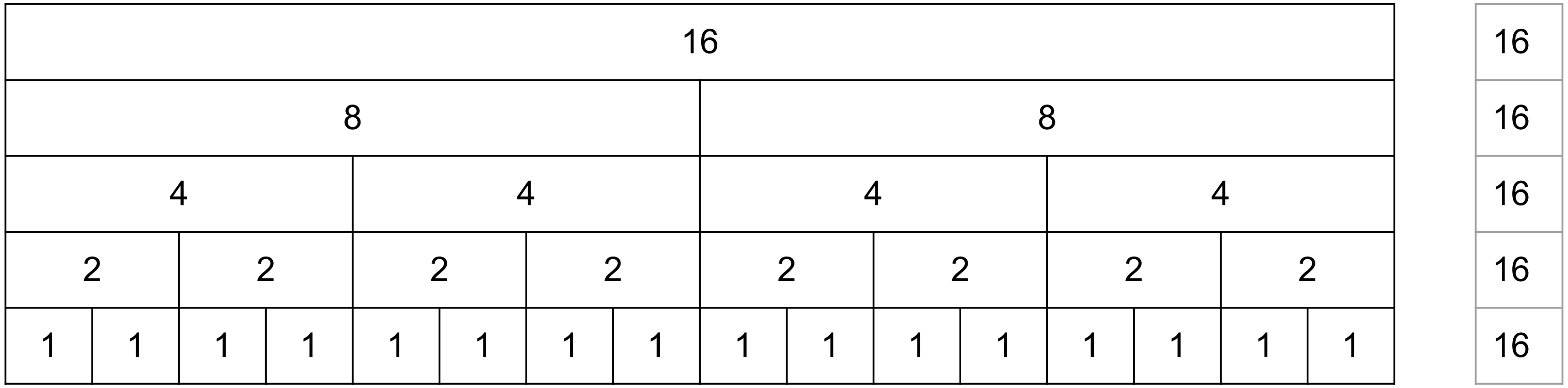
For the actual sorting, we use a recursive function. To sort the array, we split it into two smaller halves, sort those, then merge them using the previously discussed merge function. If the length of the array is already 1, we do nothing.

This has time complexity **O(N log N)**.

We can prove this in several ways.

We will only prove this time complexity for arrays whose length is a power of 2. It can be shown that this implies that arrays of arbitrary lengths can also be sorted in O(N log N) time. Proving this is left as an exercise to the reader.

Merge Sort: Complexity Proof: Visual



Sum of numbers on each layer: N

Number of layers: $\log_2 N + 1$

Overall complexity: O((N + 1) log N) = O(N log N + log N) = **O(N log N)**

Merge Sort: Complexity Proof: Math

Let `T(N)` denote an upper bound for the time required to do merge sort on an array of length `N`.

We get `T(N) >= 2*T(N/2) + N`.

We can show that $T(N) = N*\log_2(N)$ satisfies this by substituting.

$$N*\log_2(N) >= 2*N/2*\log_2(N/2) + N$$

$$N*\log_2(N) >= N*\log_2(N/2) + N$$

$$N*\log_2(N) >= N*(\log_2(N) - 1) + N$$

$$N*\log_2(N) >= N*(\log_2(N) - 1 + 1)$$

$$N*\log_2(N) >= N*\log_2(N)$$

Merge Sort: Implementation

```cpp
template <typename T>
void mergeSort(T *begin, T *end, T *mem){
    int n = end - begin;
    if(n <= 1) return;
    int *mid = begin + n/2;
    mergeSort(begin, mid, mem);
    mergeSort(mid, end, mem);
    T *i = begin;
    T *j = mid;
    T *m = mem;
    while(i != mid && j != end) *m++ = *i < *j ? *i++ : *j++;
    while(i != mid) *m++ = *i++;
    while(j != end) *m++ = *j++;
    for(int k = 0; k < n; ++k) begin[k] = mem[k];
}

template <typename T>
void mergeSort(T *begin, T *end){
    int n = end - begin;
    T *mem = new T[n];
    mergeSort(begin, end, mem);
    delete[] mem;
}
```

```python
def mergeSortRecur(arr, l, r, mem):
    sz = r - l
    if sz <= 1: return
    mid = (l + r)//2
    mergeSortRecur(arr, l, mid, mem)
    mergeSortRecur(arr, mid, r, mem)
    i = l
    j = mid
    m = 0

    while i != mid and j != l + sz:
        if arr[i] < arr[j]: mem[m] = arr[i]; i += 1
        else: mem[m] = mem[j]; j += 1
        m += 1
    while i != mid:
        mem[m] = arr[i]; i += 1; m += 1
    while j != l + sz:
        mem[m] = arr[j]; j += 1; m += 1
    for i in range(sz):
        arr[l + i] = mem[i]

def mergeSort(arr):
    mem = [None] * len(arr)
    mergeSortRecur(arr, 0, len(arr), mem)
```

Quick Sort

Quick Sort

Pick a random element of the array called the pivot. For each element in the array, determine whether it is less than or greater than the value of the pivot. Reorder the elements such that all of the ones with value less than the pivot come before it and all of the ones with value greater than it come after it. Recursively call this function on these two disjoint subarrays.

Some implementations may also handle elements with value equal to the pivot separately.

There are many possible implementations for this reorder step. All reasonable implementations will do it in O(N) time, but may have significantly different constant factors.

Quick Sort: Complexity

The idea behind quicksort is that picking a pivot randomly results in the array getting roughly cut in half, leading to a very similar situation as we had with merge sort.

Quick sort has average time complexity of **O(N log N)** and worst case complexity of O(N^2).

However**,** this is different than the average vs. worst case complexities you might see with some other algorithms (e.g non-balanced binary search trees). The difference is that with quicksort, you’re the one generating the random number. This is important because it means that there is no “worst input” for your program (as long as an attacker can’t guess the random numbers produced by your random number generator).

Quick Sort: Implementation

```cpp
mt19937 rng(865457513);
int randInt(int a, int b){
    return uniform_int_distribution(a, b)(rng);
}

// Not a very efficient implementation
template <typename T>
void quickSort(T *arr, int sz){
    if(sz <= 1) return;
    int pivot = randInt(0, sz-1);
    // Find expected location of pivot
    int numSmaller = 0;
    for(int i = 0; i < sz; i++) numSmaller += arr[i] < arr[pivot];
    swap(arr[pivot], arr[numSmaller]);
    pivot = numSmaller;
    // Rearrange rest of the array
    int l = 0;
    int m = pivot+1;
    int r = sz-1;
    while(l < pivot){
        if(arr[l] < arr[pivot]) l++;
        else if(arr[pivot] < arr[l]) swap(arr[l], arr[r--]);
        else swap(arr[l], arr[m++]);
    }
    quickSort(arr, pivot);
    quickSort(arr + m, sz - m);
}
```

```python
# Not a very efficient implementation
def quickSortRecur(arr, l, r):
    sz = r - l
    if sz <= 1: return
    pivot = random.randint(l, r-1)
    # Find expected position of pivot
    numSmaller = l
    for i in range(l, r):
        if arr[i] < arr[pivot]: numSmaller += 1
    arr[pivot], arr[numSmaller] = arr[numSmaller], arr[pivot]
    pivot = numSmaller
    # Rearrange rest of the array
    lInd = l
    mInd = pivot+1
    rInd = r-1
    while lInd < pivot:
        if arr[lInd] < arr[pivot]:
            lInd += 1
        elif arr[pivot] < arr[lInd]:
            arr[lInd], arr[rInd] = arr[rInd], arr[lInd]; rInd -= 1
        else:
            arr[lInd], arr[mInd] = arr[mInd] = arr[lInd]; mInd += 1

    quickSortRecur(arr, l, pivot)
    quickSortRecur(arr, mInd, r)

def quickSort(arr):
    quickSortRecur(arr, 0, len(arr))
```

Standard Library Sort and More Complicated Types

C++

`sort(begin, end)`
Sorts the range [begin, end)

`pair` is a type that stores a pair of elements. When sorting them, first the first element gets compared and the value of the second element is only used if the values of the first are the same.

Custom comparison functions

Operator overloading:

```
struct CustomType {
    int a, b;

    const bool operator<(const CustomType &other) const {
        return a*other.b < other.a*b;
    }
};
```

Calling sort with a comparison function:
`sort(begin, end, cmp)`
Sorts the range [begin, end) and compares elements according to cmp.

```
// Sort elements based on number of set bits because why not
bool cmp(unsigned int a, unsigned int b){
    return popcount(a) < popcount(b);
}
```

Then later you could have

```
sort(arr, arr + n, cmp);
```

Standard Library Sort and More Complicated Types

Python

`arr.sort()`
Sorts a list in-place.

`sorted(arr)`
Returns a list which is sorted (but doesn't modify the original one).

Python's tuples can also be sorted by default and are compared first by their first value and the next value is only used if the first values were equal.
Example of sorting tuples:

```
>>> arr = [(3, 6, 7), (3, 9, 2), (5, 1, 4), (0, -1, 3), (6, 2, 2)]
>>> print(sorted(arr))
[(0, -1, 3), (3, 6, 7), (3, 9, 2), (5, 1, 4), (6, 2, 2)]
```

Sorting a class in Python:

```
class CustomType:
    def __init__(self, a, b):
        self.a = a
        self.b = b

arr = [CustomType(2, 4), CustomType(7, 2), CustomType(6, 5)
arr.sort(key=lambda ct: ct.b)
```

Further reading with examples
https://docs.python.org/3/howto/sorting.html

Binary Search

Binary Search

Problem:
Given a sorted array, find the position of a given value x and return its index in the array.

Binary search algorithm:
Pick the middle element of the array and compare it to x. If x is smaller than it, binary search on the first half of the array. If x is greater than it, binary search on the second half of the array.

The time complexity of this is **O(logN)** since each iteration takes O(1) and halves the search space.

Fun fact, you have likely done binary search before, for example when looking through a dictionary.

We can easily generalize this algorithm so that instead of checking only for exactly x, it finds the index of the first element greater than or equal to x, which is often more useful.

Binary Search: Implementation

C++

```cpp
// Returns a pointer to the first element greater
than or equal to val
// The range [lo, hi) must be sorted
template <typename T>
T* binarySearch(T *lo, T *hi, T val){
    // The answer is always in [lo, hi]
    while(lo < hi){
        T *mid = lo + (hi - lo)/2;
        if(*mid < val) lo = mid + 1;
        else hi = mid;
    }
    return lo;
}
```

Python

```python
# Returns the first value i such that arr[i] >= val
# arr must be sorted
def binarySearch(arr, val):
    lo = 0
    hi = len(arr)
    # The answer is always in the range [lo, hi]
    # The if statements at the end are made such that
this will always be true
    while lo < hi:
        mid = (lo + hi)//2
        if arr[mid] < val: lo = mid + 1
        else: hi = mid

    return lo
```

Binary Search: Standard Library Functions

If you want to binary search on arrays or lists, these built in functions are what you should actually use. However, learning binary search wasn't useless as sometimes you will still have to implement it yourself as shown in the next few slides.

C++

`lower_bound(begin, end, val)`
Returns an iterator pointing to the first element greater than or equal to val in the range [first, last).

`upper_bound(begin, end, val)`
Returns an iterator pointing to the first element greater than to val in the range [first, last).

binary_search(begin, end, val)
Returns a bool indicating whether [first, last) contains the value val.

https://en.cppreference.com/w/cpp/algorithm/lower_bound
https://en.cppreference.com/w/cpp/algorithm/upper_bound
https://en.cppreference.com/w/cpp/algorithm/binary_search

Python

`bisect_left(a, x)`
Returns the first index in a with value greater or equal to than x.

`bisect_right(a, x)`
Returns the first index in a with value greater than x.

`bisect(a, x)`
Same as bisect_right.

https://docs.python.org/3/library/bisect.html

Binary Search: More Than Just Arrays

Binary search can be used on more than just arrays. If you have any function that you can efficiently evaluate at a single point and is monotonically increasing (`f(i) <= f(j)` for all `i <= j`), then you can use binary search on it.

This is extremely useful for CP as you will often encounter a question that asks for the least/greatest value that satisfies some constraint. It is often much easier to check whether a specific value works or not rather than directly solving the problem.

Practice

- Prove that any algorithm which can sort an array of length N, where N is a power of 2, in O(N log N) time can also sort an array of arbitrary N in O(N log N) time.
- https://dmoj.ca/problem/a4b1 (sorting)
- https://dmoj.ca/problem/ccc20s1 (sorting)
- https://dmoj.ca/problem/seed2 (binary search)
- https://dmoj.ca/problem/ccc21s3 (binary search on slope)
- https://dmoj.ca/problem/cco18p4 (binary search on slope)
- https://dmoj.ca/problem/ccc10s3 (binary search)
- https://dmoj.ca/problem/aac2p3 (binary search + grid bfs + 2D prefix sum array)
- https://dmoj.ca/problem/cco21p1 (inversions)

Note that the last 2 problems are more difficult than the others and include stuff we haven’t really learned yet.