CSE 12: Week 6 Discussion

2-9-21

Focus: Recursion & Sorting

Recursion

Definition: A function that calls itself.

Recursive functions break bigger problems into smaller problems until we reach the base case, which is simple to solve.

Example: Write a program to calculate n!.

```
Recall n! = n * (n-1)!

(n-1)! = (n-1) * (n-2)!

…

2! = 2 * 1! = 2
```

Recursion

Example: Write a program to calculate n!.

Recall:

```
n! = n * (n-1)!                ← bigger problem

(n-1)! = (n-1) * (n-2)!

…

2! = 2 * 1! = 2                        ← smaller problem

0! = 1! = 1                                      ← base case
```

Recursion - Base Case

Example: Write a program to calculate n!.

```
int factorial(int n) {
    if (___(1)____)
        ____(2)_____;
    else
        return ___(3)_____;
}
```

What should go in the first blank?

A. `n >= 1`

B. `n <= 1`

C. `n == (n-1) * (n-2)`

D. `factorial(n) == n`

Recursion - Base Case

Example: Write a program to calculate n!.

```
int factorial(int n) {
      if (n <= 1)
            ____(2)_____;
      else
            return ____(3)____;
}
```

What should go in the first blank?

A. `n >= 1`

B. `n <= 1`

C. `n == (n-1) * (n-2)`

D. `factorial(n) == n`

Recursion - Base Case

Example: Write a program to calculate n!.

```
int factorial(int n) {
      if (n <= 1)
            _____(2)_____;
      else
            return ____(3)_____;
}
```

What should go in the second blank?

A. `n--`

B. `return 0`

C. `return factorial(n)`

D. `return 1`

Recursion - Base Case

Example: Write a program to calculate n!.

```
int factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return ___(3)______;
}
```

What should go in the second blank?

A. `n--`

B. `return 0`

C. `return factorial(n)`

D. `return 1`

Recursion - Base Case

Example: Write a program to calculate n!.

```
int factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return ____(3)_____;
}
```

What should go in the third blank?

A. `n * factorial(n - 1);`
B. `factorial(n);`
C. `factorial(n-1);`
D. `factorial(n-1) * factorial(n-2);`

Recursion - Base Case

Example: Write a program to calculate n!.

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

What should go in the third blank?

A. `n * factorial(n - 1);`
B. `factorial(n);`
C. `factorial(n-1);`
D. `factorial(n-1) * factorial(n-2);`

Recursion QuickSort

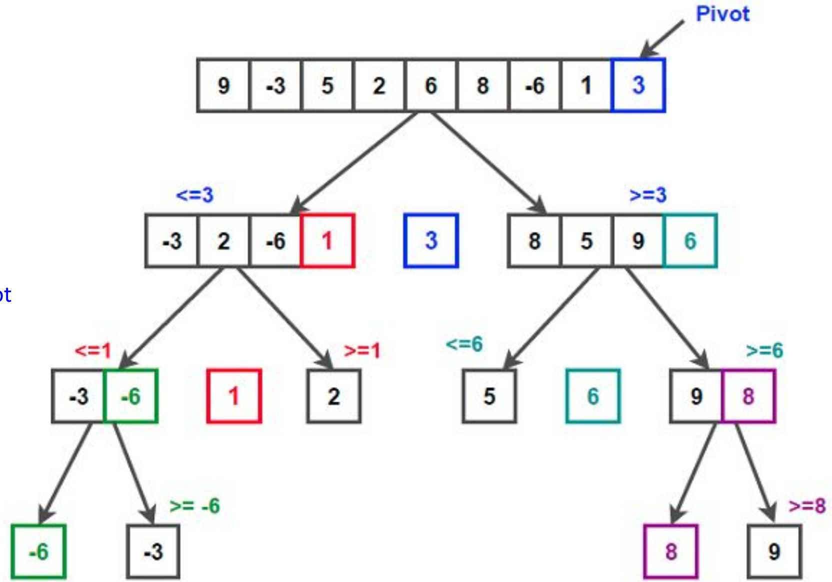
```
static int partition(String[] array, int l, int h) {...}

static void qsort(String[] array, int low, int high) {
  if(high - low < 1) { return; }
  int splitAt = partition(array, low, high);
  qsort(array, low, splitAt);
  qsort(array, splitAt + 1, high);
}

public static void sort(String[] array) {
  qsort(array, 0, array.length); //initial call
}
```

Left side of pivot

Right side of pivot

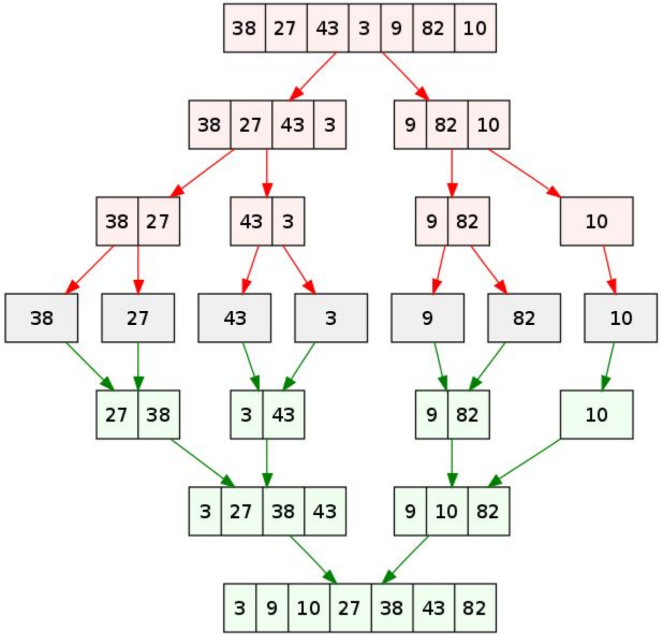


Resultant array: [-6, -3, 1, 2, 3, 5, 6, 8, 9]

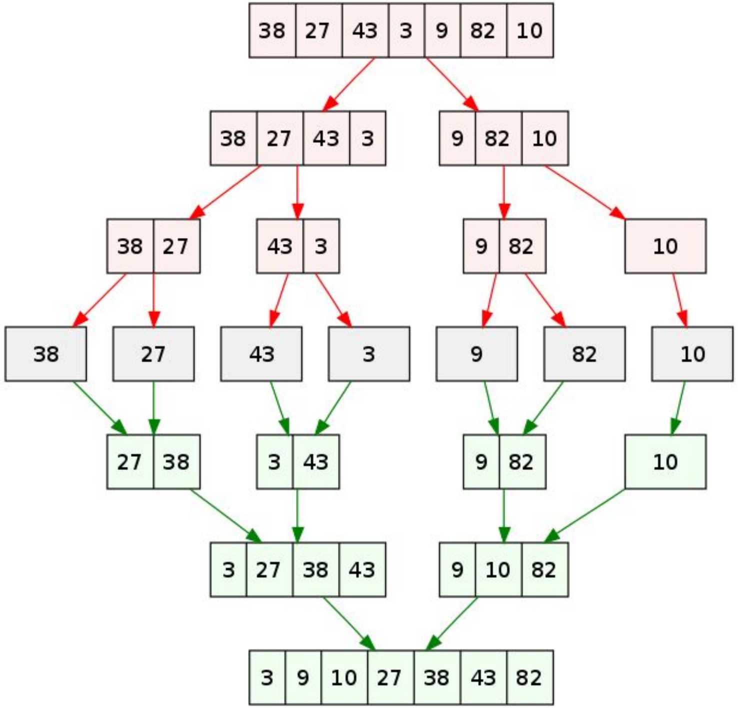
MergeSort

```
static int[] combine(int[] p1, int[] p2) {...}

static int[] mergeSort(int[] arr) {
  int len = arr.length
  if(len <= 1) { return arr; }
  else {
    int[] p1 = Arrays.copyOfRange(arr, 0, len / 2);
    int[] p2= Arrays.copyOfRange(arr, len / 2, len);
    int[] sortedPart1 = mergeSort(p1);
    int[] sortedPart2 = mergeSort(p2);
    int[] sorted = combine(sortedPart1, sortedPart2);
    return sorted;
  }
}
```



Runtime of MergeSort



Given an input size of n, how many split layers will we have until we reach the base case? (Use image to the left as reference)

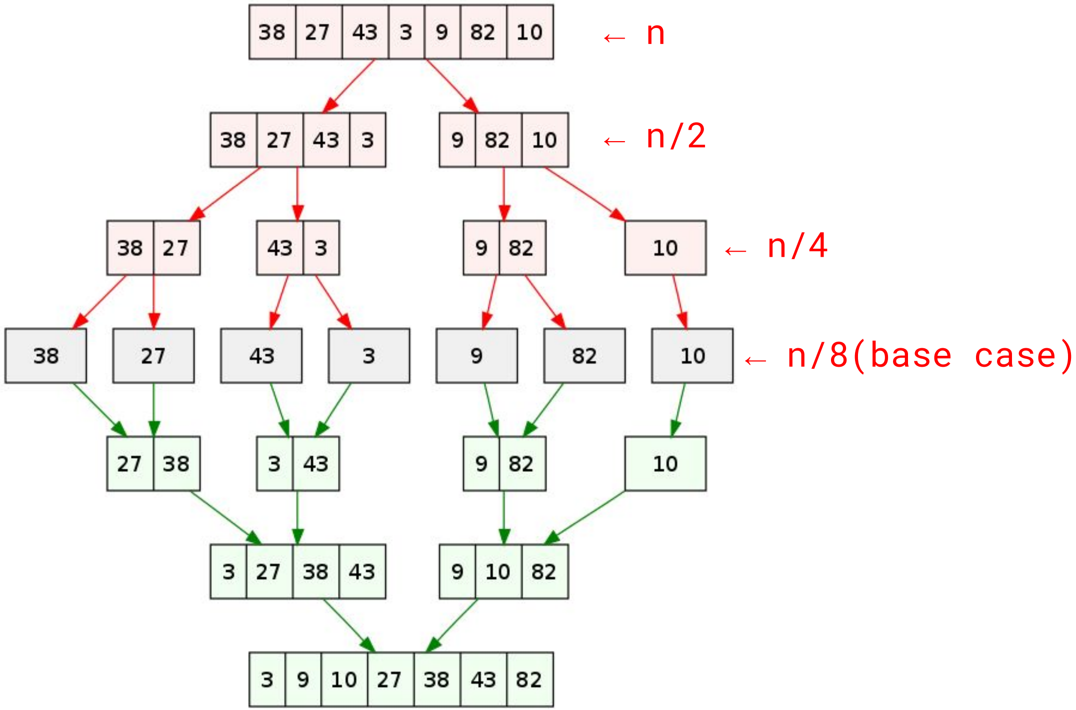
A. n

B. n/2

C. n^2

D. log(n)

Runtime of MergeSort



We are dividing the list in half recursively until we no longer can. This constant halving means we will halve $\log_2(n)$ total layers.

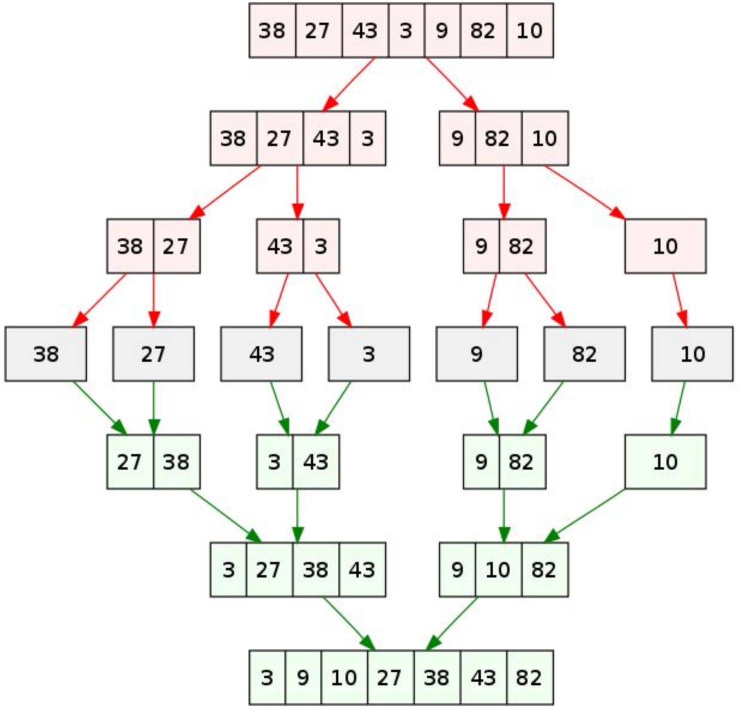
A. n

B. n/2

C. n^2

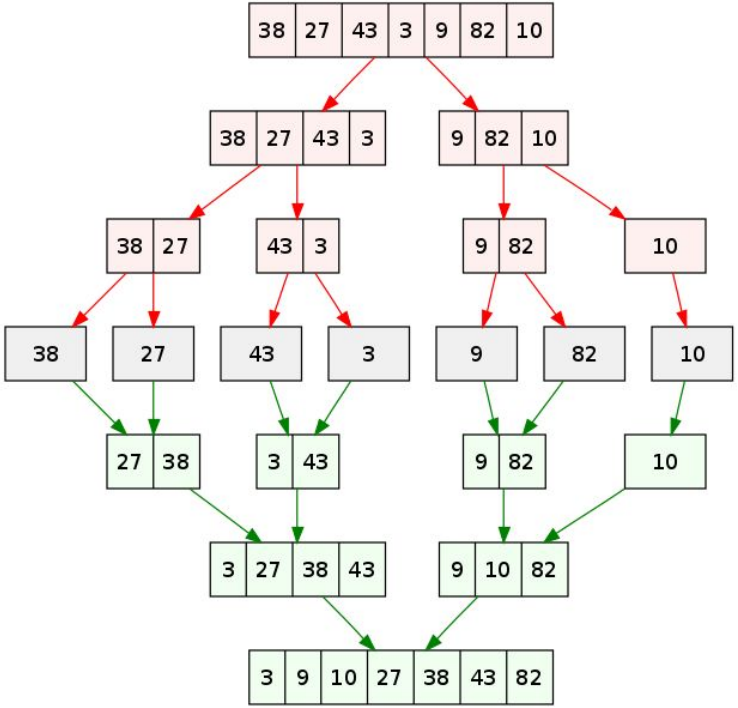
D. log(n)

Runtime of MergeSort



We can count the number of operations per split layer by thinking about how we split the arrays. There will be at most n operations, as we need to iterate through all n elements to create the left and right sub halves.

Runtime of MergeSort



Given an input size of n, how many merge layers will we have until we have the sorted array?

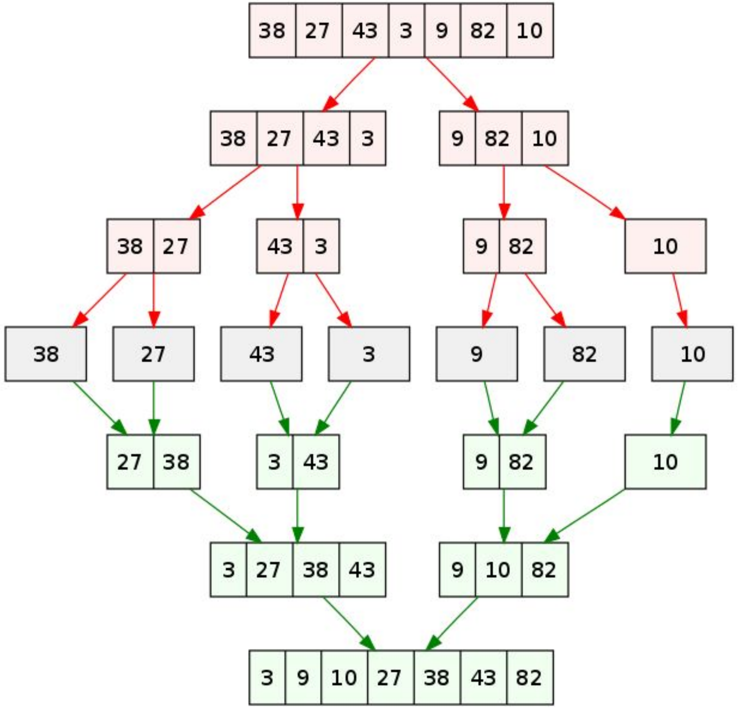
A. n

B. n/2

C. n^2

D. log(n)

Runtime of MergeSort



Since merging is the reverse of splitting, we will have exactly the same amount of merge and split layers.

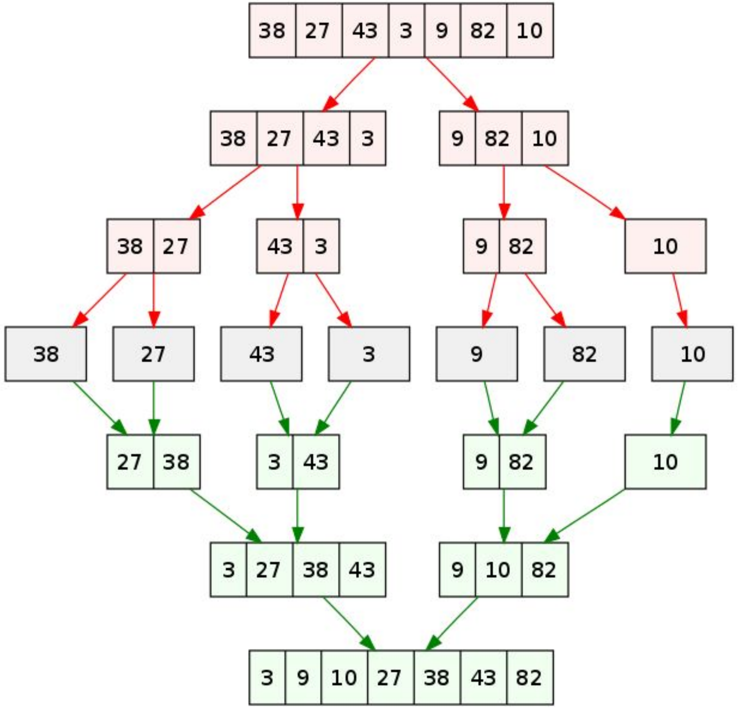
A. n

B. n/2

C. n^2

D. log(n)

Runtime of MergeSort



How many comparisons must be made at each merge layer?

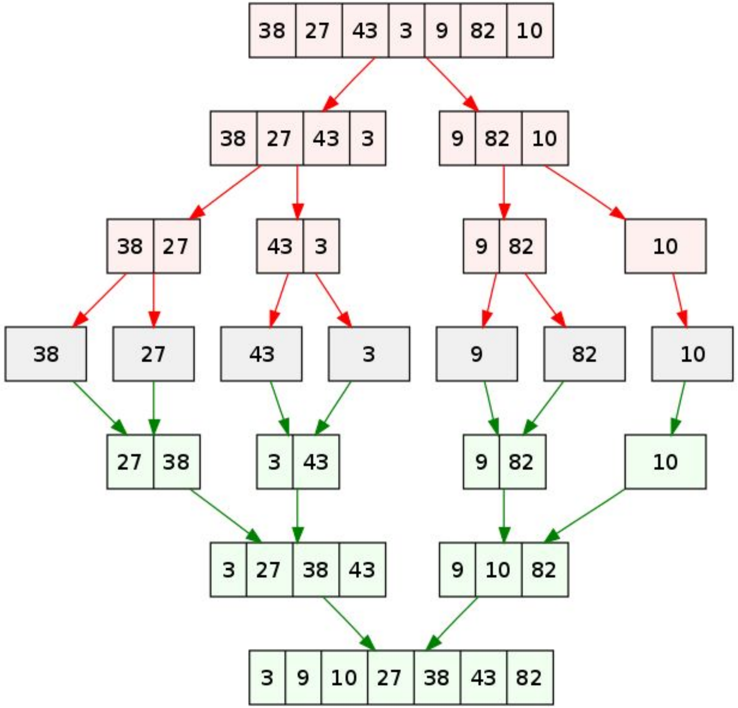
A. n

B. n/2

C. n^2

D. log(n)

Runtime of MergeSort



Every element in each subarray must be copied into a larger array in the layer below it, meaning we must go through every single element of each layer.

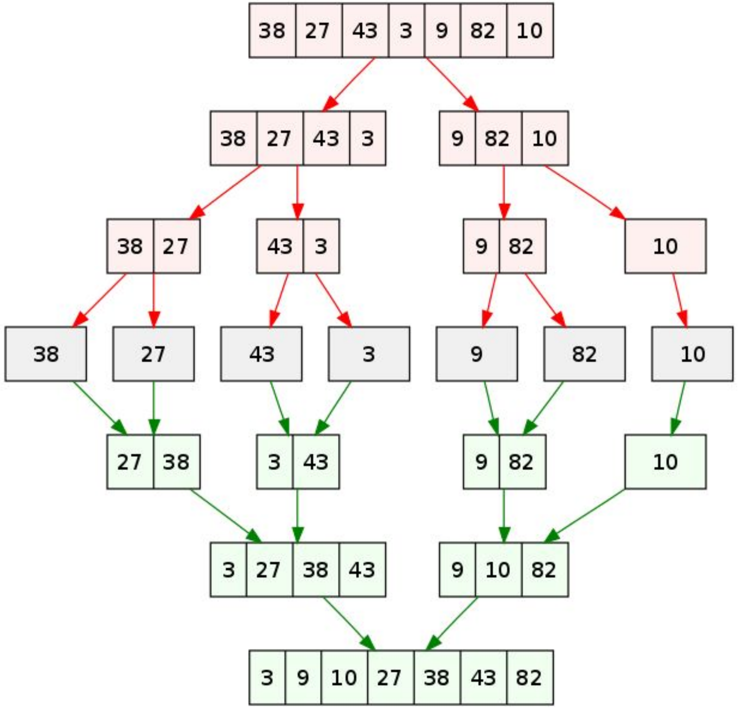
A. n

B. n/2

C. n^2

D. log(n)

Runtime of MergeSort



Putting it all together, what is the overall runtime of MergeSort.

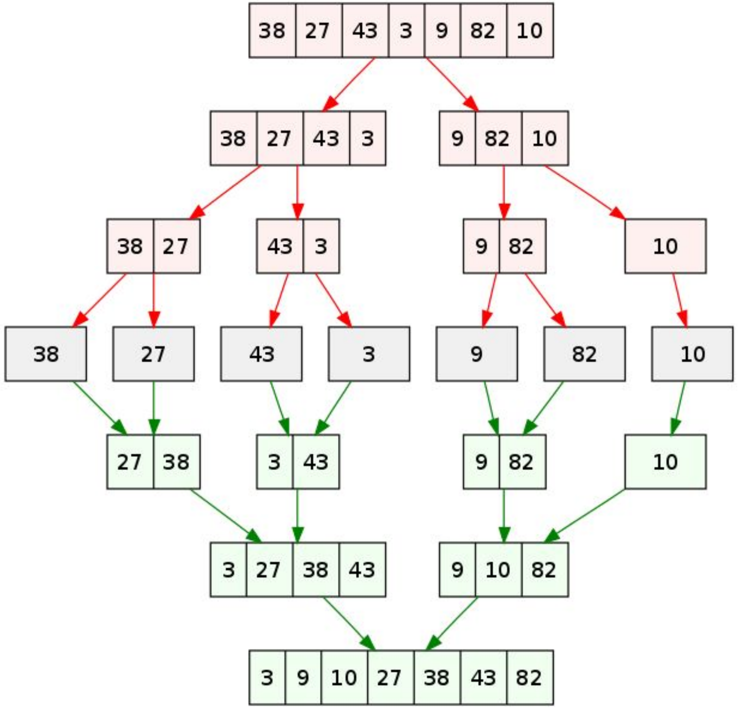
A. log(n)

B. n^3

C. n^2

D. n*log(n)

Runtime of MergeSort



We have a total of 2log(n) layers and on every layer we do n operations. We can generalize this to a runtime of n*log(n).

A. log(n)

B. n^3

C. n^2

D. n*log(n)