Merge Sort

https://cs.pomona.edu/classes/cs140
Outline

Topics and Learning Objectives

• Learn how the merge sort algorithm operates
• Become aware of the “Divide and Conquer” algorithmic paradigm by analyzing merge sort

Assessments

• Recursion tree activity
Extra Resources

• Chapter 4: divide-and-conquer
Divide and Conquer

• This is an algorithm design paradigm
• Most divide and conquer algorithms are recursive in nature
• The basic idea is to break the problem into easier-to-solve subproblems

• What's easier to do:
  • Sort 0, 1, or 2 numbers, or
  • Sort 10 numbers
Merge Sort

• This is a “Divide and Conquer”-style algorithm

• Improves over insertion sort in the worst case

• Unlike insertion sort, the best/average/worst case running times of merge sort are all the same
FUNCTION MergeSort(array)
    n = array.length
    IF n == 1
        RETURN array

    left_sorted = MergeSort(array[0..<n//2])
    right_sorted = MergeSort(array[n//2..<n])

    array_sorted = Merge(left_sorted, right_sorted)

    RETURN array_sorted
FUNCTION MergeSort(array)

n = array.length

IF n == 1
   RETURN array

left_sorted = MergeSort(array[0..<n//2])
right_sorted = MergeSort(array[n//2..<n])

array_sorted = Merge(left_sorted, right_sorted)

RETURN array_sorted

What is the running time of each line?
FUNCTION MergeSort(array)

n = array.length

IF n == 1

RETURN array

left_sorted = MergeSort(array[0..< n//2])
right_sorted = MergeSort(array[n//2..< n])

array_sorted = Merge(left_sorted, right_sorted)

RETURN array_sorted
FUNCTION MergeSort(array)

n = array.length

IF n == 1
RETURN array

left_sorted = MergeSort(array[0..<n//2])
right_sorted = MergeSort(array[n//2..<n])

array_sorted = Merge(left_sorted, right_sorted)

RETURN array_sorted
FUNCTION MergeSort(array)
    n = array.length
    IF n == 1
        RETURN array
    T(n) = 2 T(n/2) + O(?) + 4 O(1)
    = 2 T(n/2) + O(?)

    left_sorted = MergeSort(array[0..<n//2])
    T(n/2)

    right_sorted = MergeSort(array[n//2..<n])
    T(n/2)

    array_sorted = Merge(left_sorted, right_sorted)
    O(?)

    RETURN array_sorted
    O(1)
Recurrence Equation

\[ T(n) = 2 \ T(n/2) + \Theta(\ ?) + 4 \ 0(1) \]

\[ = 2 \ T(n/2) + \Theta(\ ?) \]
Merge Sort

Divide

Recursion/Conquer (not shown)

Merge

5 4 1 8 7 2 6 3
5 4 1 8
7 2 6 3
1 4 5 8
2 3 6 7
1 2 3 4 5 6 7 8
Merge Sort

Write the Merge routine

Divide

Recursion/Conquer (not shown)

Merge
FUNCTION Merge(one, two)
    out[one.length + two.length]  # Declare array
FUNCTION Merge(one, two)
    out[one.length + two.length] =
    i = j = k = 0
    WHILE k < out.length
        IF one[i] < two[j]
            out[k] = one[i]
            i = i + 1
        ELSE
            out[k] = two[j]
            j = j + 1
        k = k + 1

What is the total running time?

Ignoring invalid indices
FUNCTION Merge(one, two)
out[one.length + two.length]
i = j = k = 0
WHILE k < out.length
  IF one[i] < two[j]
    out[k] = one[i]
    i = i + 1
  ELSE
    out[k] = two[j]
    j = j + 1
  k = k + 1

Total Running Time
4
3
2 (m + 1)
3 m
3 m
2 m
0
3 m
2 m
2 m

Ignoring invalid indicies
Ignoring invalid indices

FUNCTION Merge(one, two)
    out[one.length + two.length]
    i = j = k = 0
    WHILE k < out.length
        IF one[i] < two[j]
            out[k] = one[i]
            i = i + 1
        ELSE
            out[k] = two[j]
            j = j + 1
        k = k + 1

Total Running Time

$T_{merge}(m) = 12m + 9$
Simplifying the running time

• We don’t need to be *exactly* correct with the running time of Merge
• We will eventually remove lower order terms anyway
• Let’s simplify the expression a bit:

\[ T_{merge}(m) = 12m + 9 \]

\[ T_{merge}(m) \leq 12m + 9m \]

\[ T_{merge}(m) \leq 21m \]
Merging

We have an idea of the cost of an individual call to merge:

\[ T(m) \leq 21m \]

What else do we need to know to calculate the total time of **MergeSort**?

1. How many times do we merge in total?
2. What is the size of each merge? (In other words: **What is** \( m \)?)
What is the running time of each line?

FUNCTION MergeSort(array)

n = array.length

IF n == 1

RETURN array

left_sorted = MergeSort(array[0..<n//2])

right_sorted = MergeSort(array[n//2..<n])

array_sorted = Merge(left_sorted, right_sorted)

RETURN array_sorted

T(n) = 2 T(n/2) + O(?) + 4 O(1)

= 2 T(n/2) + O(?)

T(n)
FUNCTION MergeSort(array)

\[ n = \text{array.length} \]

IF \( n == 1 \) RETURN array

\[ T(n) = 2 \cdot T(n/2) + O(n) + 4 \cdot O(1) \]
\[ = 2 \cdot T(n/2) + O(n) \]

left_sorted = MergeSort(array[0 ..< n//2])

right_sorted = MergeSort(array[n//2 ..< n])

array_sorted = Merge(left_sorted, right_sorted)

RETURN array_sorted
How many times do we call **Merge**?

```
Level 0

Level 1

Level 2

Level 3

Level ?

Entire input: size n

size n/2

size n/4

size n/8

size 1
```
How many times do we call **Merge**?

Level 0: Entire input: size n

Level 1: size n/2

Level 2: size n/4

Level 3: size n/8

**Total Levels:** \( \log_2(n) + 1 \)
Exercise

How many sub-problems are there at level $L$? The top level is Level 0, the second level is Level 1, and the bottom level is Level $\log_2(n)$

Answer: $2^L$

How many elements are there for a given sub-problem found in level $L$?

Answer: $n/2^L$

How many computations are performed at a given level?
The cost of a Merge was 21m.
Exercise

How many sub-problems are there at level \( L \)? The top level is Level 0, the second level is Level 1, and the bottom level is Level \( \log_2(n) \)

Answer: \( 2^L \)

How many elements are there for a given sub-problem found in level \( L \)?

Answer: \( n/2^L \)

How many computations are performed at a given level? The cost of a \texttt{Merge} was 21m.

Answer: \( 2^L \cdot 21(n/2^L) \Rightarrow 21n \)

What is the total computational cost of merge sort?

Answer: \( 21n \cdot (\log_2(n) + 1) \)
Exercise

How many sub-problems are there at level $L$? The top level is Level 0, the second level is Level 1, and the bottom level is Level $\log_2(n)$

**Answer:** $2^L$

How many elements are there for a given sub-problem found in level $L$?

**Answer:** $n/2^L$

How many computations are performed at a given level? The cost of a Merge was 21m.

**Answer:** $2^L \times 21(n/2^L) \rightarrow 21n$

What is the total computational cost of merge sort?

**Answer:** $21n (\log_2(n) + 1)$
Merge Sort

Divide and Conquer
• constantly halving the problem size and then merging

**Total running time** of roughly $21n \log_2(n) + 21n$

Compared to insertion sort with an average **total running time** of $\frac{1}{2} n^2$
• For small values of $n$, insertion sort is better

Which algorithm is better?
Merge Sort
Verse
Insertion
Sort
Worst-Case

What does this plot tell you?

$5n^2$

$21n(\log_2(n) + 1)$
Merge Sort
Verse
Insertion
Sort
Worst-Case

$5n^2$

$21n(\log_2(n) + 1)$
Constants

0.01n^2

100n \log_2(n)