

MergeSort

- Divide-and-Conquer Paradigm
- MergeSort Overview
- Analysis

Divide-and-Conquer

Solve a problem by recursively breaking it down into smaller problems, solving those problems, and combining the solutions...

- How do you define the “smaller problems”?
- How do you combine the smaller problems?

SZ = 1



2



4



8



16



Visual trace of bottom-up mergesort

MergeSort Overview

Recursively...

- Split the input in half
- Combine by merging the two (sorted) halves

Algorithm *mergeSort*(S, C)

Input sequence S with n elements, comparator C

Output sequence S sorted according to C

if $S.size() > 1$

→ $(S_1, S_2) \leftarrow \text{partition}(S, n/2)$

→ $\text{mergeSort}(S_1, C)$

→ $\text{mergeSort}(S_2, C)$

→ $S \leftarrow \text{merge}(S_1, S_2)$

MergeSort Overview

Recursively...

$$O(N \log N)$$

- Split the input in half
- Combine by merging the two (sorted) halves

$$T(1) = 1$$

Recurrence Relation:

$$T(N) = 2T(N/2) + N$$

Algorithm *mergeSort*(S, C)

Input sequence S with n elements, comparator C

Output sequence S sorted according to C

if $S.size() > 1$

$(S_1, S_2) \leftarrow \text{partition}(S, n/2)$

$\text{mergeSort}(S_1, C)$

$\text{mergeSort}(S_2, C)$

$S \leftarrow \text{merge}(S_1, S_2)$

Algorithm *mergeSort*(Array *A* of size *N*):

temp = an array of size *N*

mergeSortAux(*A*, *temp*, 0, *N*-1)

end *mergeSort*

procedure *mergeSortAux*(Array *A*, Array *temp*, int *i*, int *j*)

if (*j*-*i*<1):

return

m = (*j*+*i*)/2

mergeSortAux(*A*, *temp*, *i*, *m*)

mergeSortAux(*A*, *temp*, *m*+1, *j*)

merge(*A*, *temp*, *i*, *m*, *m*+1, *j*)

end *mergeSortAux*

Algorithm *mergeSort*(Array *A* of size *N*):

temp = an array of size *N*

mergeSortAux(*A*, *temp*, 0, *N*-1)

end *mergeSort*

procedure *mergeSortAux*(Array *A*, Array *temp*, int *i*, int *j*)

if (*j*-*i*<1):

return

m = (*j*+*i*)/2

mergeSortAux(*A*, *temp*, *i*, *m*)

mergeSortAux(*A*, *temp*, *m*+1, *j*)

merge(*A*, *temp*, *i*, *m*, *m*+1, *j*)

end *mergeSortAux*

procedure *merge*(Array *A*, Array *temp*, int *a*, int *b*, int *x*, int *y*)

k = *a*

c = *a*

while (*a* <= *b* || *x* <= *y*)

if (*a* > *b*):

temp[*c*] = *A*[*x*]

c++

x++

else if (*x* > *y*):

temp[*c*] = *A*[*a*]

c++

a++

else if (*A*[*a*] <= *A*[*x*]):

temp[*c*] = *A*[*a*]

c++

a++

else

temp[*c*] = *A*[*x*]

c++

x++

end while

for *i* **from** *k* **to** *y*:

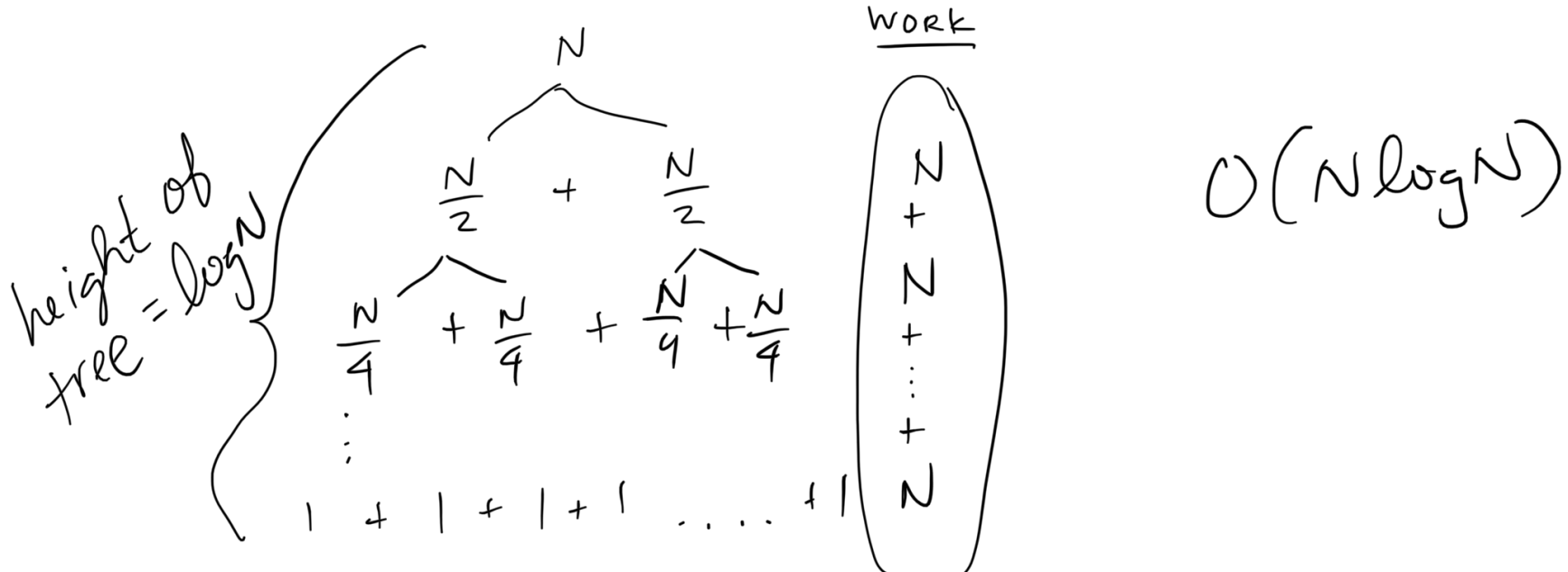
A[*i*] = *temp*[*i*]

end *merge*

Analysis: Recurrence Relation

Recurrence Relation:
 $T(N) = 2T(N/2) + N$

Base Case:
 $T(1) = 1$



Analysis: Tree Representation (N elements)

- How much work is done at each level?

$O(N) \rightarrow$ for merge

- How many levels are there?

$\log N$

- What's the total work required?

$O(N \log N)$

Mergesort Summary

- Guarantees the proven best worst case runtime for comparison-based sorting methods: $O(N\log N)$
- Uses a Divide & Conquer approach where subproblems are smaller sorted arrays that are then combined through merging. Note that the actual sorting is done through the merging because the base case (1 element) is already sorted.
- Not typically done in place. Space usage takes into account the call stack (which is $O(\log N)$) and the extra copy of the array (which is $O(N)$), so the space usage is $O(N)$.
- Sometimes, Mergesort is combined with Insertion Sort since the latter works well on tiny arrays. For example, you might use Mergesort but instead of dividing and combining all the way to 1-element arrays, you stop at an array of size 16 or less and sort those with Insertion Sort before combining them back together. An example of this kind of hybrid sorting is Timsort.

