Computer Science 331

Merge Sort

Instructor: Wayne Eberly

Department of Computer Science
University of Calgary

Lecture #16

Introduction

Merge Sort is another sorting algorithm. 

- It is *much* faster, in the worst case, than any of the classical algorithms, when used to sort *large* arrays.
- Unfortunately, it does not sort in place: Another array is produced as output — so the *storage requirements* of this algorithm are significantly higher than for any of Selection Sort, Insertion Sort, or Bubble Sort.

Goals for Today: A presentation of this algorithm, a sketch of a proof of its correctness, and and and analysis of its worst case running time.

How Might We Recursively Sort an Array?

One Way:

1. Split the input array `A` into two pieces, each approximately half as large: $\texttt{B}_1$ would include the *first* half of the entries of `A`, and $\texttt{B}_2$ would include the *second* half of the entries of `A`.

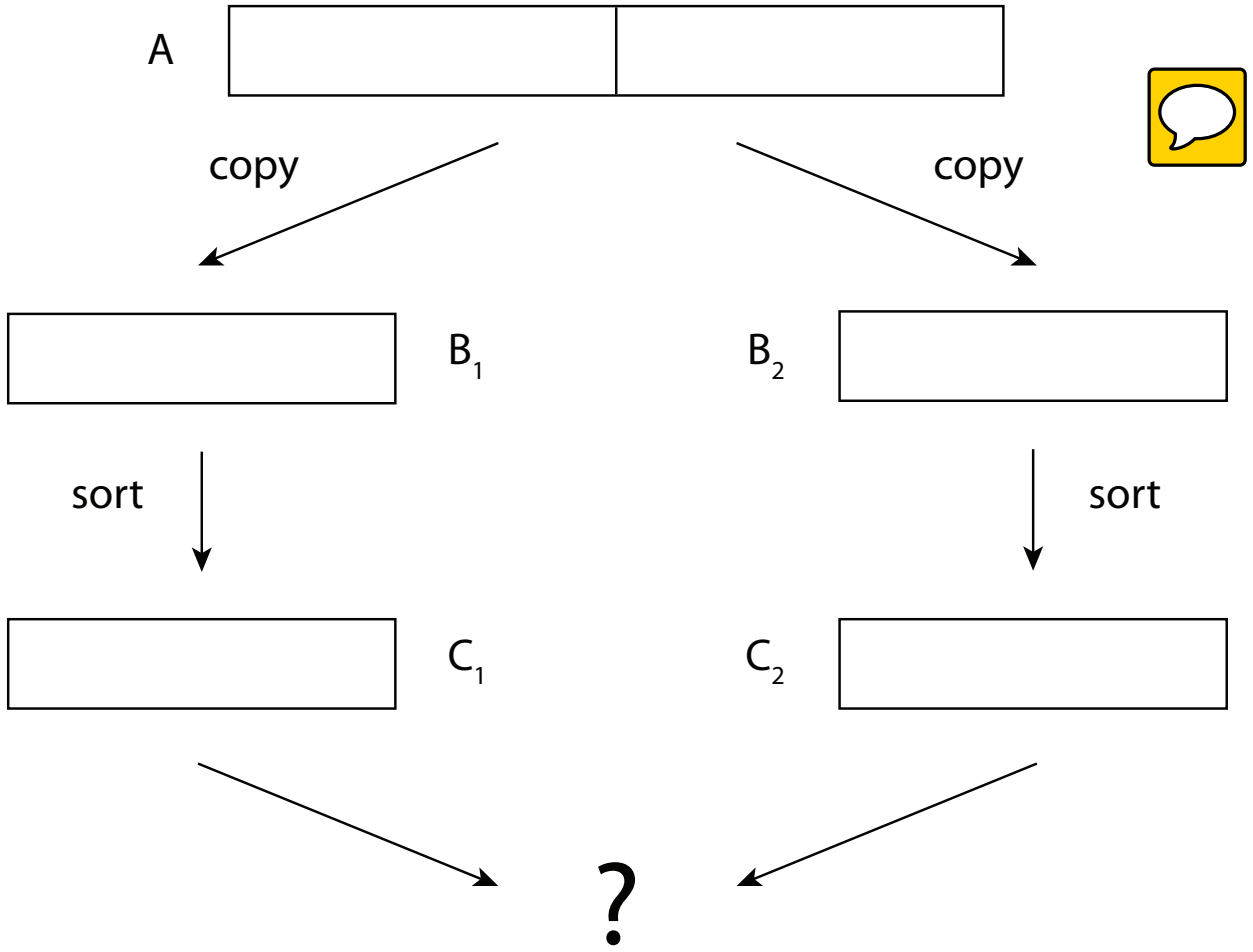
2. Recursively **sort** $\texttt{B}_1$, to produce a sorted array $\texttt{C}_1$, and recursively **sort** $\texttt{B}_2$, to produce a sorted array $\texttt{C}_2$.

How Might We Recursively Sort an Array

Question:

How should we *finish*?

How Might We Recursively Sort an Array?



The "Merging" Problem

The Merging Problem:

Precondition:

1. C_1 is an array with length $n_1 \geq 1$ with elements from some ordered type T that is sorted in nondecreasing order — so that $C_1[h] \leq C_1[h+1]$ for every integer h such that $0 \leq h \leq n_1 - 2$.
2. C_2 is an array with length $n_2 \geq 1$ with elements from the same ordered type T that is also sorted in nondecreasing order — so that $C_2[h] \leq C_2[$ $]$ for every integer h such that $0 \leq h \leq n_2 - 2$.

The "Merging" Problem

Postcondition:

1. An array `D` with length $\mathtt{n}_1 + \mathtt{n}_2$ is returned as output whose entries are the entries are those of $\mathtt{C}_1$ and of $\mathtt{C}_2$ — reordered, but otherwise unchanged — such that `D[h]` $\leq$ `D[h+1]` for every integer `h` such that $0 \leq \mathtt{h} \leq \mathtt{n}_1 + \mathtt{n}_2 - 2$.

Answer for the Previous Question:

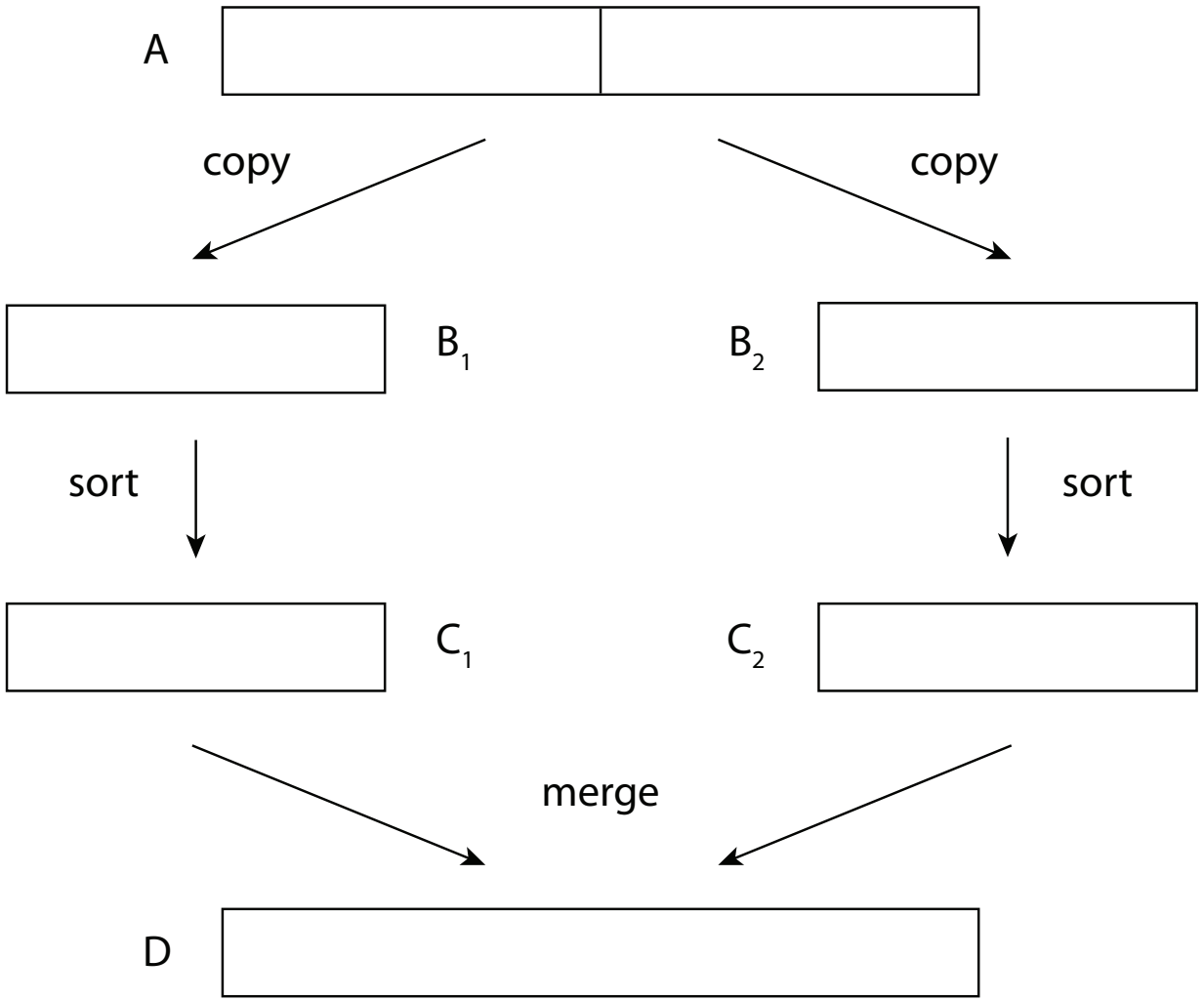
Answer for the Previous Question:

An algorithm that solves

the “Merging” problem

could be used to complete our recursive algorithm for sorting!

The "Merging Problem"



The "Merging Problem:" Making Initial Progress

To begin suppose that might have started to copy elements from $\texttt{C}_1$ and $\texttt{C}_2$ into $\texttt{D}$ — filling in $\texttt{D}$ by increasing position — but there are still some elements from both arrays that remain to be copied.

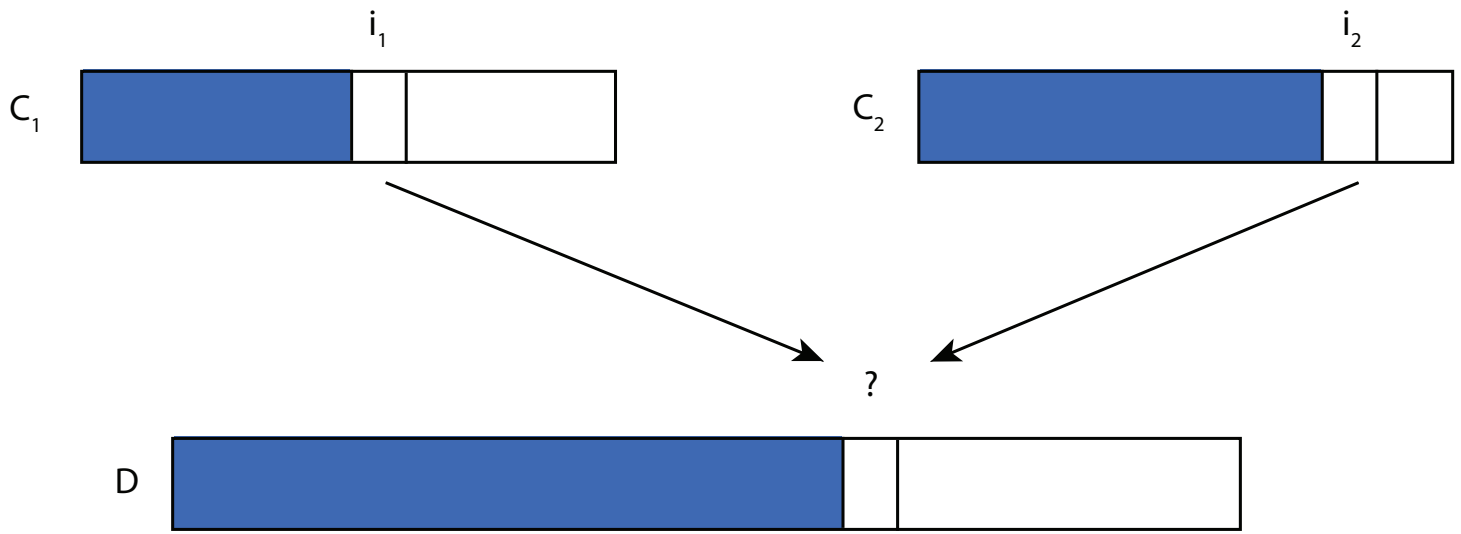
- Suppose $\texttt{i}_1$ elements of $\texttt{C}_1$ have filled in, so that $0 \leq \texttt{i}_1 < \texttt{n}_1$. Since $\texttt{D}$ will be a *sorted* array when this is finished, the elements of $\texttt{C}_1$ that have already been copied must be $\texttt{C}_1\texttt{[0]}, \texttt{C}_1\texttt{[1]}, \ldots, \texttt{C}_1\texttt{[i}_1 - \texttt{1]}$.
- Suppose $\texttt{i}_2$ elements of $\texttt{C}_2$ have filled in, so that $0 \leq \texttt{i}_2 < \texttt{n}_2$. Since $\texttt{D}$ will be a *sorted* array when this is finished, the elements of $\texttt{C}_1$ that have already been copied must be $\texttt{C}_2\texttt{[0]}, \texttt{C}_2\texttt{[1]}, \ldots, \texttt{C}_2\texttt{[i}_2 - \texttt{1]}$.
- The *next* element to be copied into $\texttt{D}$ will be *either* $\texttt{C}_1\texttt{[i}_1\texttt{]}$ *or* $\texttt{C}_2\texttt{[i}_2\texttt{]}$.

The “Merging Problem:” Making Initial Progress

Question:

Which of These Should be Next?

The "Merging Problem:" Making Initial Progress



- ***Answer:*** Whichever of $C_1[i_1]$ or $C_2[i_2]$ is *smaller* — to make sure that `D` will be sorted, when we are done.

  Ties can be broken either way.

The "Merging Problem:" Making Initial Progress

A `merge` algorithm should therefore *begin* as follows — if `j` will be used as the next position of `D` to be used:

```
T[] merge(T[] C1, T[] C2) {
 1.  integer n1 = C1.length;
 2.  integer n2 = C2.length;
 3.  T[] D = new T[n1 + n2];
 4.  integer i1 = 0;
 5.  integer i2 = 0;
 6.  integer j = 0;
```

The "Merging Problem:" Making Initial Progress

```
 7.  while ((i1 < n1) && (i2 < n2)) {
 8.    if (C1[i1] ≤ C2[i2]) {
 9.      D[j] = C1[i1];
10.      i1 = i1 + 1;
       } else {
11.      D[j] = C2[i2];
12.      i2 = i2 + 1;
       };
13.    j = j + 1;
     };
```

The "Merging Problem:" Making Initial Progress

The `while` loop has a rather complicated (but important) ***loop invariant:***

Loop Invariant for Loop #1:

1. $\texttt{C}_1$ is an input array, with length $\texttt{n}_1 \geq 1$, storing elements from some ordered type `T`, whose entries are sorted in nondecreasing order.
2. $\texttt{C}_2$ is an input array, with length $\texttt{n}_2 \geq 1$, storing elements from the same ordered type `T`, whose entries are sorted in nondecreasing order.
3. `D` is an array with length $\texttt{n}_1 + \texttt{n}_2$ storing elements from type `T`.

The "Merging Problem:" Making Initial Progress

4. $\texttt{i}_1$ is an integer variable such that $0 \leq \texttt{i}_1 \leq \texttt{n}_1$.
5. $\texttt{i}_2$ is an integer variable such that $0 \leq \texttt{i}_2 \leq \texttt{n}_2$.
6. $\texttt{j}$ is an integer variable such that $\texttt{j} = \texttt{i}_1 + \texttt{i}_2$.
7. The first $\texttt{j}$ entries of $\texttt{D}$ include the first $\texttt{i}_1$ entries of $\texttt{C}_1$ and the first $\texttt{i}_2$ entries of $\texttt{C}_2$, in some order.
8. $\texttt{D[h]} \leq \texttt{D[h+1]}$ for every integer h such that $0 \leq \texttt{h} \leq \texttt{j} - 2$.

The "Merging Problem:" Making Initial Progress

The following *additional assertions* can also be established:

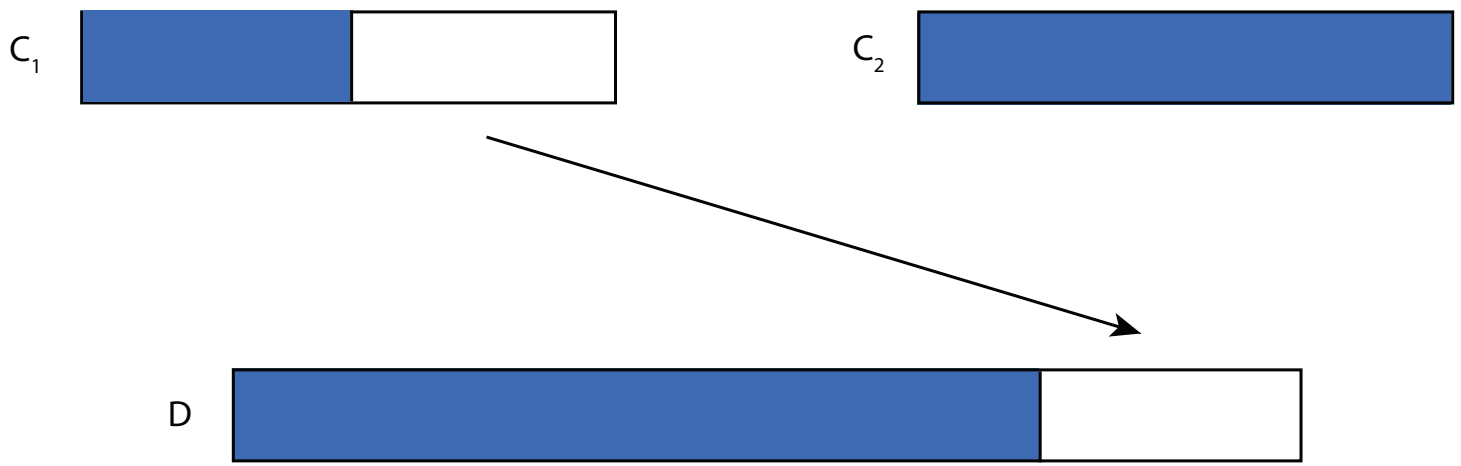
- *Before the Loop:* $\mathtt{i}_1 = 0$ and $\mathtt{i}_2 = 0$.
- *At the Beginning of the Body of the Loop:* $\mathtt{i}_1 < \mathtt{n}_1$ and $\mathtt{i}_2 < \mathtt{n}_2$.
- *At the End of the Body of the Loop: Nothing more that is useful.*
- *After the Loop:* Either $\mathtt{i}_1 = \mathtt{n}_1$ or $\mathtt{i}_2 = \mathtt{n}_2$.

Exercise: Confirm that the above really *is* a loop invariant for this first `while` loop, and confirm that the additional assertions are all satisfied when they are supposed to be, as well.

Confirm, as well, that the function $(\mathtt{n}_1 + \mathtt{n}_2) - (\mathtt{i}_1 + \mathtt{i}_2) - 1$ is a ***bound function*** for this `while` loop.

The "Merging Problem:" Continuing

A second phase of this algorithm should include any remaining elements of C_1 in D.



The "Merging Problem:" Continuing

Pseudocode for this part of the algorithm is as follows.

```
14.  while (i1 < n1) {
15.    D[j] = C1[i1];
16.    i1 = i1 + 1;
17.    j = j + 1;
     };
```

The "Merging Problem:" Continuing

Loop Invariant for Loop #2: Same as for Loop #1!

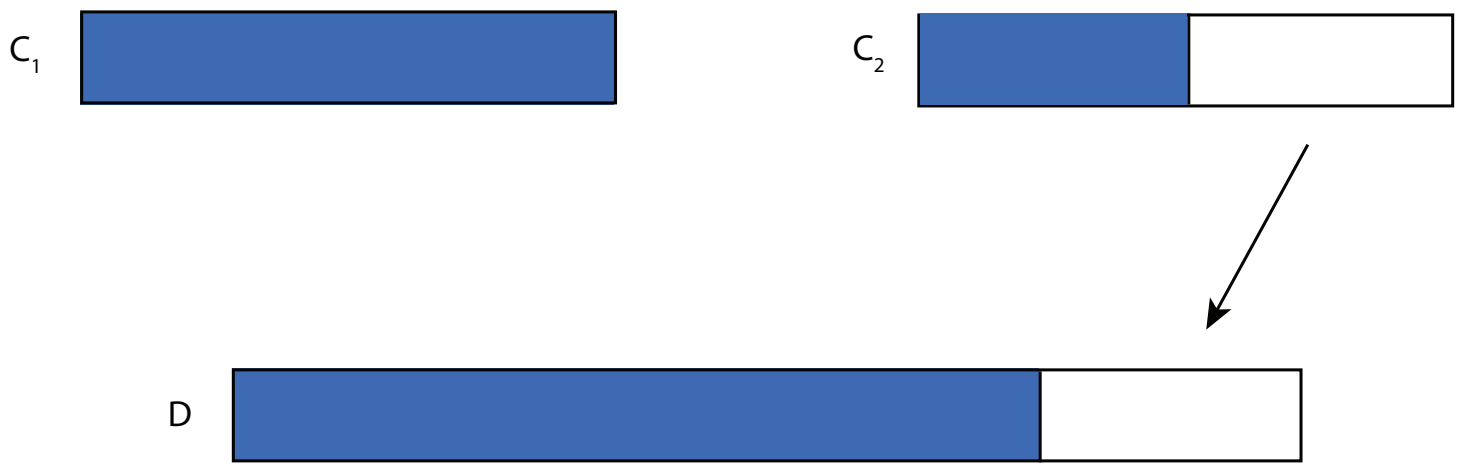
Additional Assertions:

- *Before the Loop:* Either $\mathtt{i}_1 = \mathtt{n}_1$ or $\mathtt{i}_2 = \mathtt{n}_2$.
- *At the Beginning of the Body of the Loop:* $0 \leq \mathtt{i}_1 < \mathtt{n}_1$ and $\mathtt{i}_2 = \mathtt{n}_2$.
- *At the End of the Body of the Loop:* $1 \leq \mathtt{i} \leq \mathtt{n}_1$ and $\mathtt{i}_2 = \mathtt{n}_2$.
- *After the Loop:* $\mathtt{i}_1 = \mathtt{n}_1$.

The function $\mathtt{n}_1 - \mathtt{i}_1$ is a ***bound function*** for Loop #2.

The "Merging Problem:" Ending

A final phase of this algorithm should include any remaining elements of C_2 in D.



The "Merging Problem:" Ending

Pseudocode for this part of the algorithm is as follows.

```
18.  while (i2 < n2) {
19.    D[j] = C2[i2];
20.    i2 = i2 + 1;
21.    j = j + 1;
     };
22.  return D;
}      // End of Algorithm!
```

The "Merging Problem:" Ending

Loop Invariant for Loop #3:

- Parts 1–3 and 5–8 are the same as for the previous loop invariants.
- Part 4 (which asserts that $\mathtt{i}_1$ is an integer such that $0 \leq \mathtt{i}_1 \leq \mathtt{n}_1$ in the other loop invariants) should now state that $\mathtt{i}_1 = \mathtt{n}_1$.

Additional Assertions:

- *Before the Loop: Nothing useful.*
- *At the Beginning of the Body of the Loop:* $0 \leq \mathtt{i}_2 < \mathtt{n}_2$.
- *At the End of the Body of the Loop:* $1 \leq \mathtt{i}_2 \leq \mathtt{n}_2$.
- *After the Loop:* $\mathtt{i}_2 = \mathtt{n}_2$.

The function $\mathtt{n}_2 - \mathtt{i}_2$ is a ***bound function*** for Loop #3.

The "Merging" Problem: Correctness of This Solution

Establishing Partial Correctness:

- Notice that, taken together, the
  - *Loop Invariant* for Loop #3, and
  - additional assertion that holds after this loop

  establish that the entries of `D` are the entries of $\texttt{C}_1$ and of $\texttt{C}_2$ (reordered, but otherwise unchanged) — and that $\texttt{D[h]} \leq \texttt{D[h+1]}$ for every integer `h` such that $0 \leq \texttt{h} \leq \texttt{n}_1 + \texttt{n}_2 - 2$.

  In other words, `D` has all the properties described (for the output) in the postcondition for the "Merging" problem.

- Since this array is *returned* as output, immediately after this (at line 22) — and one can see by inspection of the code that it has no undocumented side-effects — this establishes the *partial correctness* of this algorithm.

The “Merging” Problem: Correctness of This Solution

Establishing Termination:

- The bodies of Loops #1–#3 all consist of sequences of a fixed number of statements (including tests) that do not call other methods — so every execution of the body of any of these loops must terminate.
- The existence of **bound functions** for all three of these loops imply that — when included in an execution of the algorithm when the the precondition for the “Merging” property is satisfied — all executions of these *loops* terminate too.

The “Merging” Problem: Correctness of the Solution

- There is only a fixed number of statements (which do not call other methods) in the rest of the algorithm. It follows that if the precondition for the “Merging” problem is satisfied and the algorithm is executed then it terminates.

Thus this algorithm is also **correct**.

The “Merging” Problem: Efficiency of This Solution

Bounding Running Time:

Notice that — when the uniform cost criterion is used to define running time, and numbered steps in the code are counted —

- every execution of the body of Loop #1 includes the execution of exactly 4 steps. (either at lines 8, 9, 10 and 13, or at lines 8, 11, 12 and 13).
- every execution of the body of Loop #2 includes the execution of exactly 3 steps (at lines 15, 16, and 17).
- every execution of the body of Loop #3 includes the execution of exactly 3 steps (at lines 19, 20, and 21).

The "Merging Problem:" Efficiency of This Solution

Claim: If the `merge` algorithm is executed when the precondition for the "Merging" problem is satisfied, with arrays of length n_1 and n_2 as input, then the following properties are satisfied.

(a) There are at most $n_1 + n_2 - 1$ executions of the body of Loop #1.

(b) The sum of the number of executions of the bodies of Loops #1, #2 and #3 is equal to $n_1 + n_2$.

The "Merging Problem:" Efficiency of This Solution

Sketch of Proof: As previously noted, the function $(\mathtt{n}_1 + \mathtt{n}_2) - (\mathtt{i}_1 + \mathtt{i}_2) - 1$ is a bound function for Loop #1, and the initial value of this function is $\mathtt{n}_1 + \mathtt{n}_2 - 1$ — implying part (a).

To prove (b), one should notice that

- the initial value of $\mathtt{j}$ is 0,
- the value of $\mathtt{j}$ is increased by *exactly* one very time the body of any of these loops is executed, and
- the final value of $\mathtt{j}$ is $\mathtt{n}_1 + \mathtt{n}_2$.

The "Merging Problem:" Efficiency of This Solution

- **Consequence:** The total of number of steps included in the executions of the bodies of *all* these loops is at least $3(n_1 + n_2)$ and at most $4(n_1 + n_2 - 1) + 3 = 4(n_1 + n_2) - 1$.
- The *test* for a loop is executed one more time than the *body* of the loop. Each test has unit cost so it follows by part (b) that the total number of steps included in the executions of the loop tests (at lines 7, 14 and 18) is equal to $n_1 + n_2 + 3$.
- An execution of the algorithm includes the execution of 7 more steps — at lines 1–6 and 22.

Adding at all together: An execution of the algorithm includes an execution of *at least* $4(n_1 + n_2) + 10$ steps, and *at most* $5(n_1 + n_2) + 9$ steps.

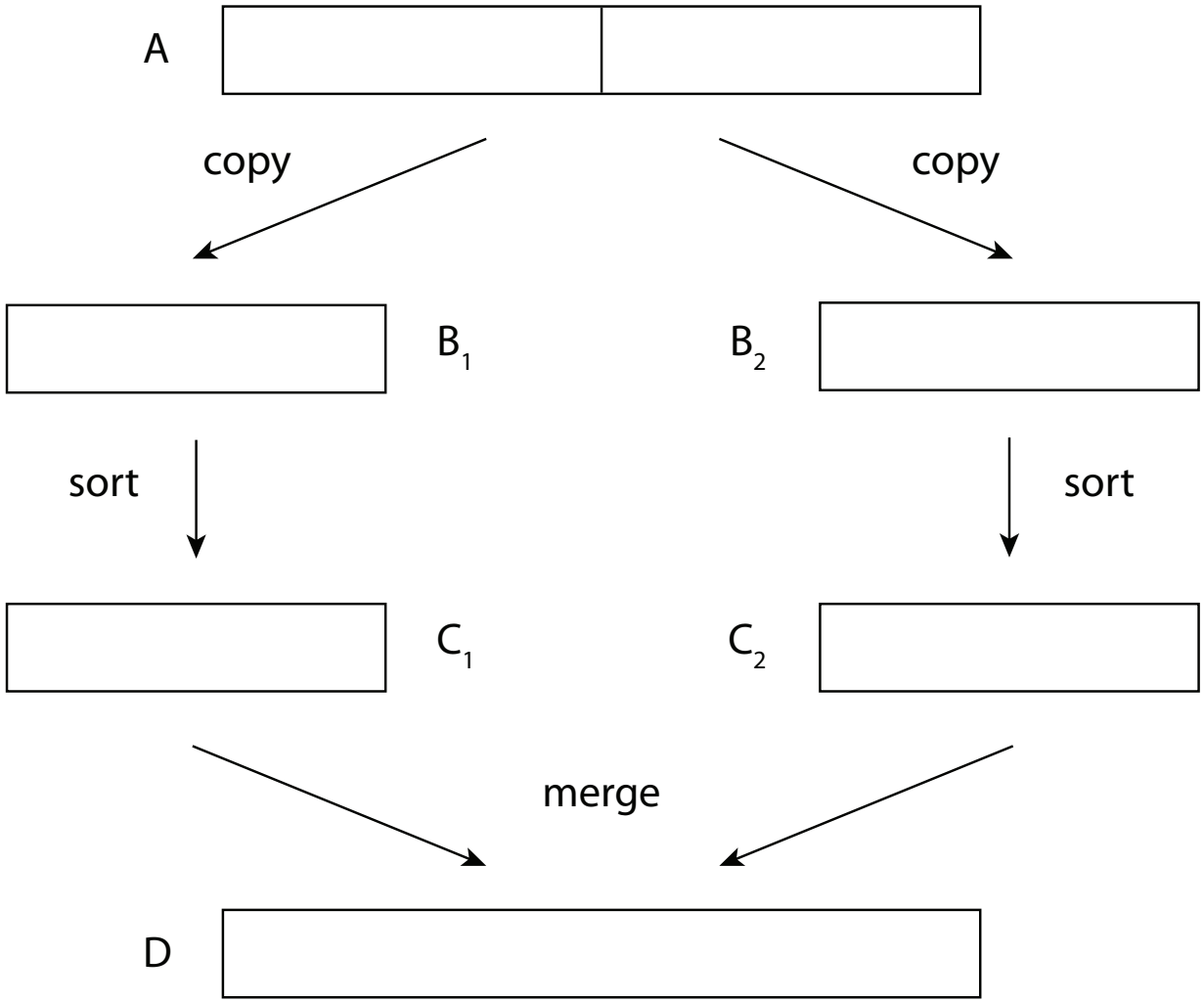
So the *best case running time* and the *worst case running time* of this algorithm are both in $\Theta(n_1 + n_2)$.

Merge Sort: The Algorithm

Consider how to solve the "Sorting Problem" when the input array $\mathtt{A}$ has length $\mathtt{n} \geq 1$.

- If $\mathtt{n} = 1$ then $\mathtt{A}$ is already sorted — so a copy of $\mathtt{A}$ should be produced and returned.
- Otherwise, $\lceil \mathtt{n}/2 \rceil \leq \mathtt{n} - 1$ and $\lfloor \mathtt{n}/2 \rfloor \leq \mathtt{n} - 1$ — so that, choosing the lengths of arrays $\mathtt{B}_1$ and $\mathtt{B}_2$ to be $\mathtt{n}_1 = \lceil \mathtt{n}/2 \rceil$ and $\mathtt{n}_2 = \lfloor \mathtt{n}/2 \rfloor$, respectively, one can proceed as previously described:

Merge Sort: The Algorithm



Merge Sort

```
T[] MergeSort(T[] A) {
 1.  if (A.length == 1) {
 2.    T[] D = new T[1];
 3.    D[0] = A[0];
 4.    return D;
     } else {
     // Copy:
 5.    integer n_1 = ⌈n/2⌉;
 6.    T[] B_1 = new T[n_1];
 7.    integer n_2 = ⌊n/2⌋;
 8.    T[] B_2 = new T[n_2];
 9.    integer i = 0;
```

Merge Sort: The Algorithm

```
10.    while (i < n1) {
11.      B1[i] = A[i];
12.      i = i + 1;
       };
13.    while (i < n) {
14.      B2[i − n1] = A[i];
15.      i = i + 1;
       };
       // Recursively Sort and Merge
16.    T[] C1 = MergeSort(B1);
17.    T[] C2 = MergeSort(B2);
18.    return merge(C1, C2);
     };
}
```

Merge Sort: Correctness

Claim: The `MergeSort` algorithm correctly solves the sorting problem.

Method of Proof: Induction on the length $\texttt{n} = \texttt{A.length}$ of the input array — using the strong form of mathematical induction, and considering the case $\texttt{n} = 1$ in the basis.

- An examination of the steps at lines 2–4 is sufficient to complete the basis.
- An application of the inductive hypothesis (to establish what happens when the steps at lines 16 and 17 are executed) and the correctness of the `merge` algorithm (applied at line 18) are sufficient to complete the inductive step.

Merge Sort: Efficiency

Let $T_{\texttt{MergeSort}}(\texttt{n})$ be the number of steps executed, in the worst case, when the precondition for the "Sorting" problem is satisfied and the algorithm is executed with an array of length $\texttt{n} \geq 1$ as input.

Counting steps — and using the upper bound for the worst case running time of the `merge` algorithm — one can confirm that

- $T_{\texttt{MergeSort}}(\texttt{n}) \leq 4$, if $\texttt{n} = 1$, and
- $T_{\texttt{MergeSort}}(\texttt{n}) \leq$

$$T_{\texttt{MergeSort}}(\lceil \texttt{n}/2 \rceil) + T_{\texttt{MergeSort}}(\lfloor \texttt{n}/2 \rfloor) + 8\texttt{n} + 20, \quad \text{if } \texttt{n} \geq 2.$$

Merge Sort: Efficiency

Please recall that students in this course are **not** expected to be able to discover solutions for recurrences with this one without help... but they *are* expected to be able to use mathematical induction to *verify* solutions if these are given.

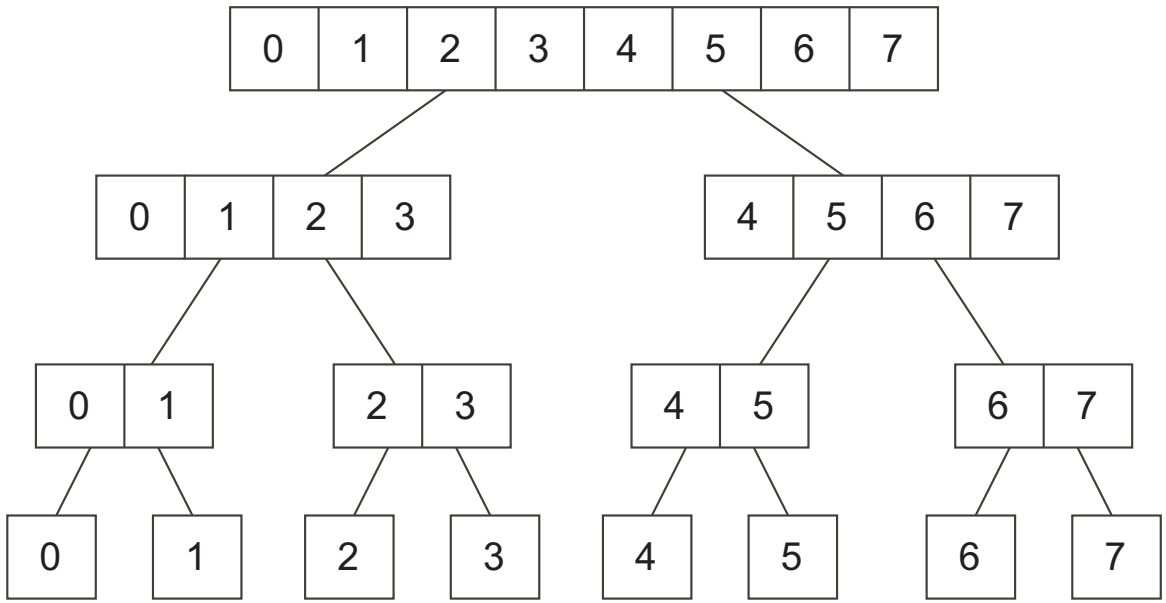
Exercise: *(Pulling a Rabbit out of My Hat):* Use the recurrence that has now been given to confirm that

$$\begin{aligned} T_{\texttt{MergeSort}}(\texttt{n}) &\leq 8\texttt{n} \times \lceil \log_2 \texttt{n} \rceil + 24\texttt{n} - 20 \\ &\leq 8\texttt{n} \log_2 \texttt{n} + 32\texttt{n} - 20 \in O(\texttt{n} \log_2 \texttt{n}) \end{aligned}$$

for every integer $\texttt{n} \geq 1$.

Analyzing the Running Time Another Way

Consider the various arrays that are formed, recursively sorted, and then merged together as this algorithm is used.



Analyzing the Running Time Another Way

- With a bit of work — and a consideration of the relationship between the lengths of input arrays at two consecutive levels, one can argue that there are always exactly $\lceil \log_2 \mathtt{n} \rceil + 1$ levels in this tree, if the input array has length $\mathtt{n} \geq 1$.
- With the possible exception of the bottom level, the sum of the lengths of all input arrays at any level is $\mathtt{n}$.

  The sum of the lengths of all input arrays at the *bottom* level might be *less than n*.
- Examining the code once again, one can use this to argue that the *best case running time* and the *worst case running time* of this algorithm are both in $\Theta(\mathtt{n} \log_2 \mathtt{n})$.

Additional Reference

- Section 2.3 of *Introduction to Algorithms* gives another — rather different — introduction to this algorithm and its analysis.

A Disadvantage of Merge Sort

Unfortunately, `MergeSort` algorithm does not ***sort in place*** — and its ***storage requirements*** are considerably higher than those of any of the "classical" sorting algorithms.

Wouldn't It Be Nice...

... if we had a sorting algorithm that combined the best of both worlds:

- Its running time was in $O(n \log n)$ (for an input array with like n), like `MergeSort`, and
- it sorts in place, with low storage requirements, like the classical sorting algorithms?

An algorithm with these properties will be developed and analyzed over the next two lectures.