CSE 373: Analysis of Algorithms

Lectures 5 – 8
(Correctness of Algorithms)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Fall 2014

Insertion Sort

Input: An array $A[\,1 : n\,]$ of n numbers.

Output: Elements of $A[\,1 : n\,]$ rearranged in non-decreasing order of value.

```
INSERTION-SORT ( A )

for $j = 2$ to $A.length$
    $key = A[j]$
    // insert $A[j]$ into the sorted sequence $A[1..j-1]$
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$
```

Loop Invariants

We use *loop invariants* to prove correctness of iterative algorithms

A loop invariant is associated with a given loop of an algorithm, and it is a formal statement about the relationship among variables of the algorithm such that

- **[Initialization]** It is true prior to the first iteration of the loop
- **[Maintenance]** If it is true before an iteration of the loop, it remains true before the next iteration
- **[Termination]** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct

Loop Invariants for Insertion Sort

```
INSERTION-SORT ( $A$ )

for $j = 2$ to $A.length$
    $key = A[j]$
    // insert $A[j]$ into the sorted sequence $A[1..j-1]$
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$
```

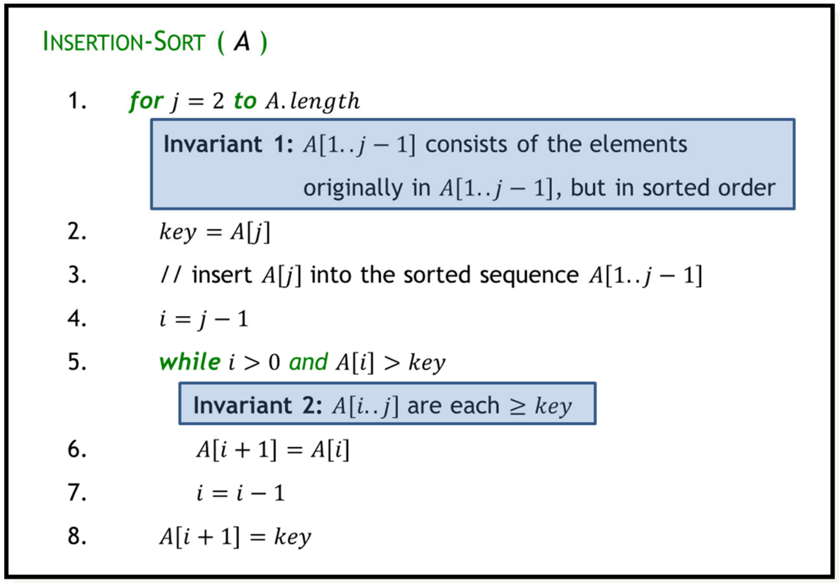
Loop Invariants for Insertion Sort

```
INSERTION-SORT ( A )

for $j = 2$ to $A.length$
    Invariant 1: $A[1..j-1]$ consists of the elements
                 originally in $A[1..j-1]$, but in sorted order
    $key = A[j]$
    // insert $A[j]$ into the sorted sequence $A[1..j-1]$
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$
```

Loop Invariants for Insertion Sort

```
INSERTION-SORT ( A )
    for $j = 2$ to $A.length$
        Invariant 1: $A[1..j-1]$ consists of the elements
                     originally in $A[1..j-1]$, but in sorted order
        $key = A[j]$
        // insert $A[j]$ into the sorted sequence $A[1..j-1]$
        $i = j - 1$
        while $i > 0$ and $A[i] > key$
            Invariant 2: $A[i..j]$ are each $\geq key$
            $A[i+1] = A[i]$
            $i = i - 1$
        $A[i+1] = key$
```

Loop Invariant 1: Initialization



```
INSERTION-SORT ( A )

1.  for $j = 2$ to $A.length$
        Invariant 1: $A[1..j-1]$ consists of the elements
                     originally in $A[1..j-1]$, but in sorted order
2.      $key = A[j]$
3.      // insert $A[j]$ into the sorted sequence $A[1..j-1]$
4.      $i = j - 1$
5.      while $i > 0$ and $A[i] > key$
            Invariant 2: $A[i..j]$ are each $\geq key$
6.          $A[i+1] = A[i]$
7.          $i = i - 1$
8.      $A[i+1] = key$
```

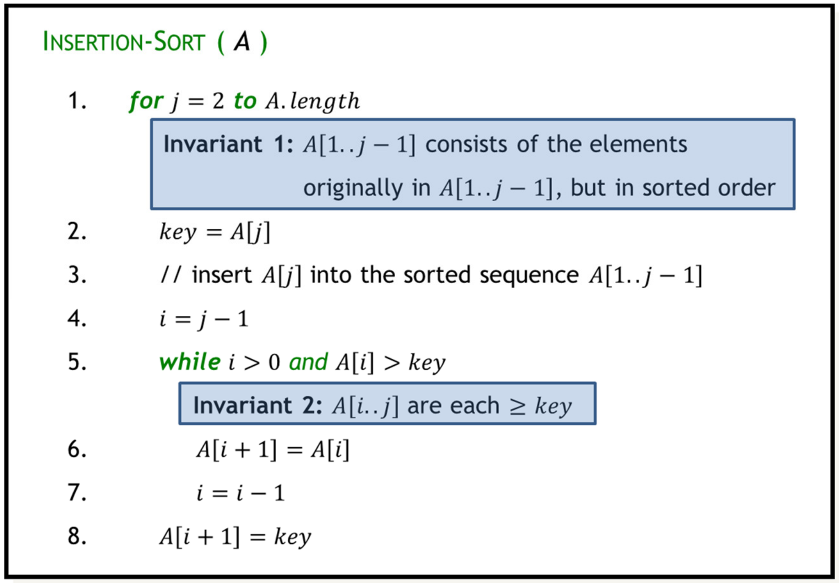
At the start of the first iteration of the loop (in lines $1 - 8$): $j = 2$

Hence, subarray $A[1..j-1]$ consists of a single element $A[1]$, which is in fact the original element in $A[1]$.

The subarray consisting of a single element is trivially sorted.

Hence, the invariant holds initially.

Loop Invariant 1: Maintenance



```
INSERTION-SORT ( A )
for $j = 2$ to $A.length$
    Invariant 1: $A[1..j-1]$ consists of the elements
                 originally in $A[1..j-1]$, but in sorted order
    $key = A[j]$
    // insert $A[j]$ into the sorted sequence $A[1..j-1]$
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        Invariant 2: $A[i..j]$ are each $\geq key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$
```

We assume that invariant 1 holds before the start of the current iteration.

Hence, the following holds: $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$, but in sorted order.

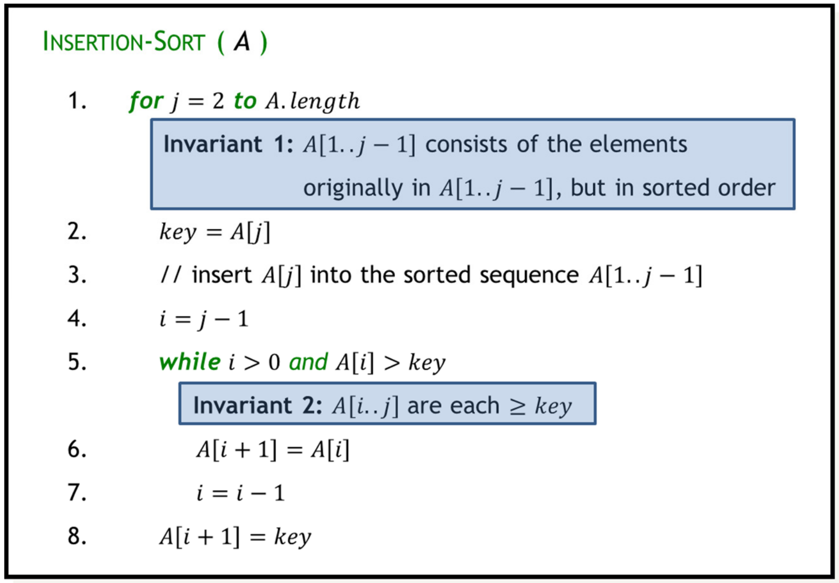
For invariant 1 to hold before the start of the next iteration, the following must hold at the end of the current iteration:

$A[1..j]$ consists of the elements originally in $A[1..j]$, but in sorted order.

We use invariant 2 to prove this.

Loop Invariant 1: Maintenance

Loop Invariant 2: Initialization



```
INSERTION-SORT ( A )

1.  for $j = 2$ to $A.length$
        Invariant 1: $A[1..j-1]$ consists of the elements
                     originally in $A[1..j-1]$, but in sorted order
2.      $key = A[j]$
3.      // insert $A[j]$ into the sorted sequence $A[1..j-1]$
4.      $i = j - 1$
5.      while $i > 0$ and $A[i] > key$
            Invariant 2: $A[i..j]$ are each $\geq key$
6.          $A[i+1] = A[i]$
7.          $i = i - 1$
8.      $A[i+1] = key$
```

At the start of the first iteration of the loop (in lines 5 − 7): $i = j - 1$

Hence, subarray $A[i..j]$ consists of only two entries: $A[i]$ and $A[j]$.

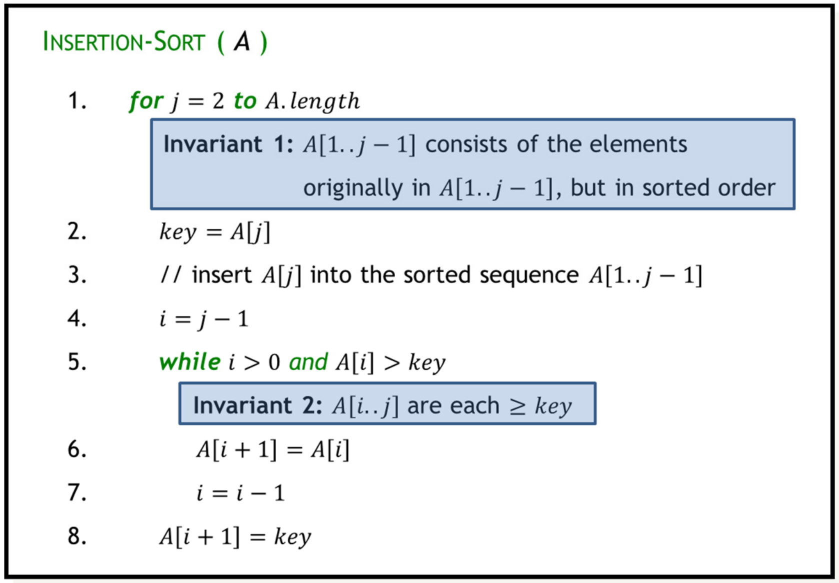
We know the following:

- $A[i] > key$ (explicitly tested in line 5)
- $A[j] = key$ (from line 2)

Hence, invariant 2 holds initially.

Loop Invariant 1: Maintenance

Loop Invariant 2: Maintenance



```
INSERTION-SORT ( A )
1.  for $j = 2$ to $A.length$
        Invariant 1: $A[1..j-1]$ consists of the elements
                     originally in $A[1..j-1]$, but in sorted order
2.      $key = A[j]$
3.      // insert $A[j]$ into the sorted sequence $A[1..j-1]$
4.      $i = j - 1$
5.      while $i > 0$ and $A[i] > key$
            Invariant 2: $A[i..j]$ are each $\geq key$
6.          $A[i+1] = A[i]$
7.          $i = i - 1$
8.      $A[i+1] = key$
```

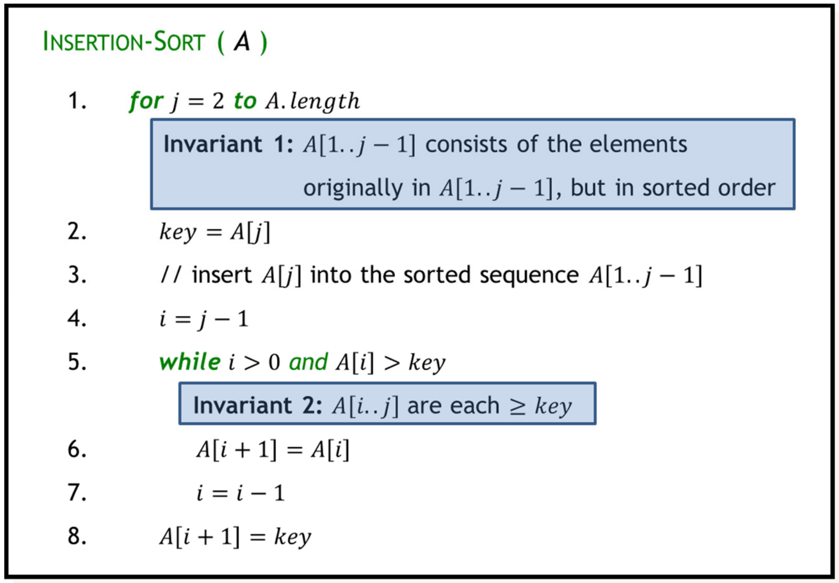
We assume that invariant 2 holds before the start of the current iteration.

Hence, the following holds: $A[i..j]$ are each $\geq key$.

Since line 6 copies $A[i]$ which is known to be $> key$ to $A[i+1]$ which also held a value $\geq key$, the following holds at the end of the current iteration: $A[i+1..j]$ are each $\geq key$.

Before the start of the next iteration the check $A[i] > key$ in line 5 ensures that invariant 2 continues to hold.

Loop Invariant 1: Maintenance
Loop Invariant 2: Maintenance



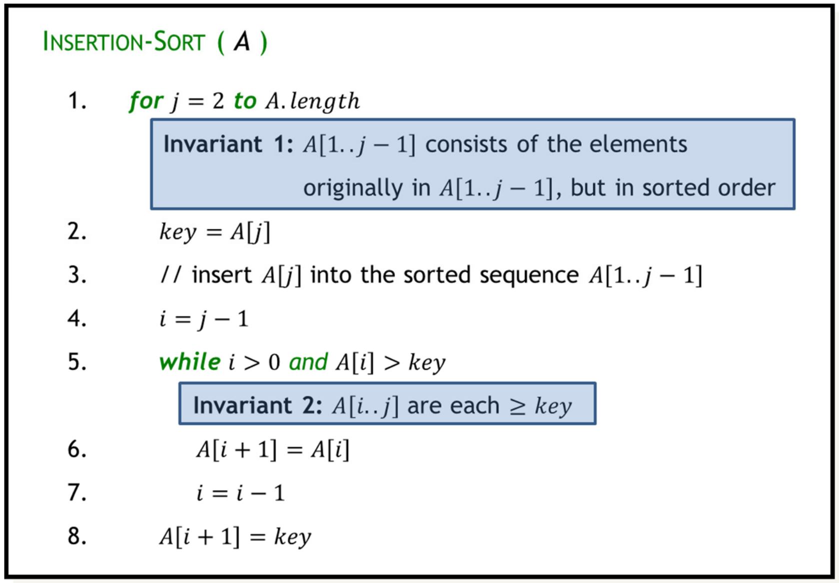
```
INSERTION-SORT ( A )

1.  for $j = 2$ to $A.length$
        Invariant 1: $A[1..j-1]$ consists of the elements
                     originally in $A[1..j-1]$, but in sorted order
2.      $key = A[j]$
3.      // insert $A[j]$ into the sorted sequence $A[1..j-1]$
4.      $i = j - 1$
5.      while $i > 0$ and $A[i] > key$
            Invariant 2: $A[i..j]$ are each $\geq key$
6.          $A[i+1] = A[i]$
7.          $i = i - 1$
8.      $A[i+1] = key$
```

Observe that the inner loop (in lines $5-7$) does not destroy any data because though the first iteration overwrites $A[j]$, that $A[j]$ has already been saved in key in line 2.

As long as key is copied back into a location in $A[1..j]$ without destroying any other element in that subarray, we maintain the invariant that $A[1..j]$ contains the first j elements of the original list.

Loop Invariant 1: Maintenance

Loop Invariant 2: Termination

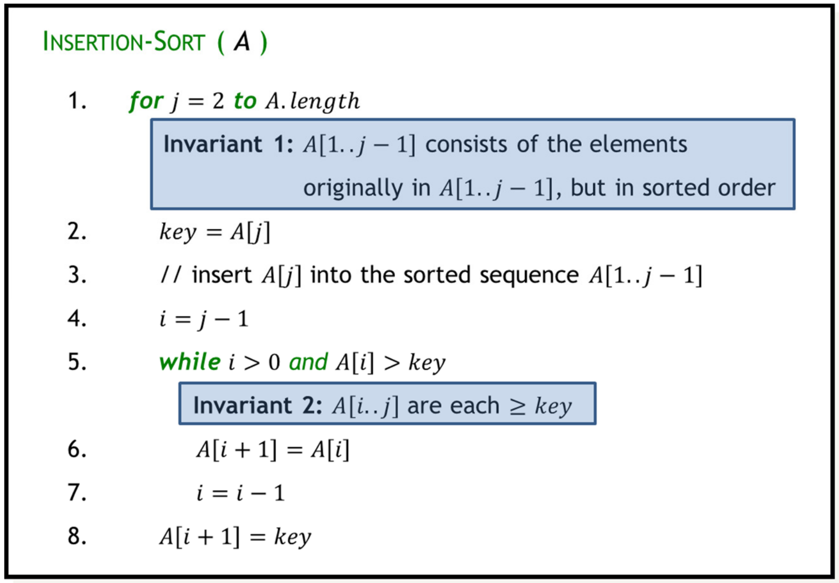


```
INSERTION-SORT ( A )
for $j = 2$ to $A.length$
    Invariant 1: $A[1..j-1]$ consists of the elements
        originally in $A[1..j-1]$, but in sorted order
    $key = A[j]$
    // insert $A[j]$ into the sorted sequence $A[1..j-1]$
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        Invariant 2: $A[i..j]$ are each $\geq key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$
```

When the inner loop terminates we know the following.

- $A[1..i]$ is sorted with each element $\leq key$
    - if $i = 0$, true by default
    - if $i > 0$, true because $A[1..i]$ is sorted and $A[i] \leq key$
- $A[i+1..j]$ is sorted with each element $\geq key$ because the following held before i was decremented: $A[i..j]$ is sorted with each item $\geq key$
- $A[i+1] = A[i+2]$ if the loop was executed at least once, and $A[i+1] = key$ otherwise

Loop Invariant 1: Maintenance
Loop Invariant 2: Termination



```
INSERTION-SORT ( A )

1.  for $j = 2$ to $A.length$
        Invariant 1: $A[1..j-1]$ consists of the elements
                     originally in $A[1..j-1]$, but in sorted order
2.      $key = A[j]$
3.      // insert $A[j]$ into the sorted sequence $A[1..j-1]$
4.      $i = j - 1$
5.      while $i > 0$ and $A[i] > key$
            Invariant 2: $A[i..j]$ are each $\geq key$
6.          $A[i+1] = A[i]$
7.          $i = i - 1$
8.      $A[i+1] = key$
```

When the inner loop terminates we know the following.

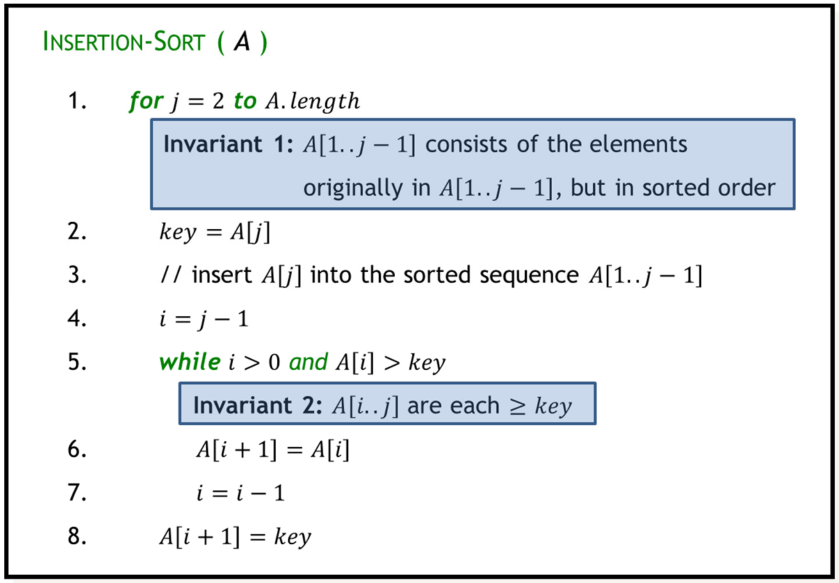
— $A[1..i]$ is sorted with each element $\leq key$

— $A[i+1..j]$ is sorted with each element $\geq key$

— $A[i+1] = A[i+2]$ or $A[i+1] = key$

Given the facts above, line 8 does not destroy any data, and gives us $A[1..j]$ as the sorted permutation of the original data in $A[1..j]$.

Loop Invariant 1: Termination



```
INSERTION-SORT ( A )
for $j = 2$ to $A.length$
    Invariant 1: $A[1..j-1]$ consists of the elements
                 originally in $A[1..j-1]$, but in sorted order
    $key = A[j]$
    // insert $A[j]$ into the sorted sequence $A[1..j-1]$
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        Invariant 2: $A[i..j]$ are each $\geq key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$
```

When the outer loop terminates we know that $j = A.length + 1$.

Hence, $A[1..j-1]$ is the entire array $A[1..A.length]$, which is sorted and contains the original elements of $A[1..A.length]$.

Worst Case Runtime of Insertion Sort (Upper Bound)

```
INSERTION-SORT ( A )

1.  for $j = 2$ to $A.length$
2.      $key = A[j]$
3.      // insert $A[j]$ into the sorted sequence $A[1..j-1]$
4.      $i = j - 1$
5.      while $i > 0$ and $A[i] > key$
6.          $A[i+1] = A[i]$
7.          $i = i - 1$
8.      $A[i+1] = key$
```

	cost	times
1.	c_1	n
2.	c_2	$n-1$
3.	0	
4.	c_4	
5.	c_5	$\sum_{2 \le j \le n} j$
6.	c_6	$\sum_{2 \le j \le n} (j-1)$
7.	c_7	
8.	c_8	$n-1$

Running time, $T(n) \le c_1 n + c_2(n-1) + c_4(n-1)$

$$+c_5 \sum_{j=2}^{n} j + c_6 \sum_{j=2}^{n} (j-1) + c_7 \sum_{j=2}^{n} (j-1) + c_8(n-1)$$

$$= 0.5(c_5 + c_6 + c_7)n^2 + 0.5(2c_1 + 2c_2 + 2c_4 + c_5 - c_6 - c_7 + 2c_8)n$$
$$-(c_2 + c_4 + c_5 + c_8)$$

$$\Rightarrow T(n) = O(n^2)$$

Best Case Runtime of Insertion Sort (Lower Bound)

```
INSERTION-SORT ( $A$ )
for $j = 2$ to $A.length$
    $key = A[j]$
    // insert $A[j]$ into the sorted sequence $A[1..j-1]$
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$
```

	cost	times
1.	c_1	n
2.	c_2	$n-1$
3.	0	
4.	c_4	
5.	c_5	
6.	c_6	0
7.	c_7	
8.	c_8	$n-1$

Running time, $T(n) \geq c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$

$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

$$\Rightarrow T(n) = \Omega(n)$$

Selection Sort

Input: An array $A[\,1 : n\,]$ of n numbers.

Output: Elements of $A[\,1 : n\,]$ rearranged in non-decreasing order of value.

```
SELECTION-SORT ( $A$ )

for $j = 1$ to $A.length$
    // find the index of an entry with the smallest value in $A[j\,..\,A.length]$
    $min = j$
    for $i = j + 1$ to $A.length$
        if $A[i] < A[min]$
            $min = i$
    // swap $A[j]$ and $A[min]$
    $A[j] \leftrightarrow A[min]$
```

Selection Sort

Input: An array $A[\,1 : n\,]$ of n numbers.

Output: Elements of $A[\,1 : n\,]$ rearranged in non-decreasing order of value.

```
SELECTION-SORT ( A )

for $j = 1$ to $A.length$
    Invariant 1: ?
    // find the index of an entry with the smallest value in $A[j..A.length]$
    $min = j$
    for $i = j + 1$ to $A.length$
        Invariant 2: ?
        if $A[i] < A[min]$
            $min = i$
    // swap $A[j]$ and $A[min]$
    $A[j] \leftrightarrow A[min]$
```

Merging Two Sorted Subarrays

Input: Two subarrays $A[\,p : q\,]$ and $A[q+1 : r]$ in sorted order ($p \leq q < r$).

Output: A single sorted subarray $A[\,p : r\,]$ by merging the input subarrays.

```
MERGE ( A, p, q, r )

$n_1 = q - p + 1$
$n_2 = r - q$
Let $L[1 : n_1 + 1]$ and $R[1 : n_2 + 1]$ be new arrays
for $i = 1$ to $n_1$
    $L[i] = A[p + i - 1]$
for $j = 1$ to $n_2$
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ to $r$
    if $L[i] \leq R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$
```

Merging Two Sorted Subarrays

Input: Two subarrays $A[\,p : q\,]$ and $A[q + 1 : r]$ in sorted order ($p \le q < r$).

Output: A single sorted subarray $A[\,p : r\,]$ by merging the input subarrays.

```
MERGE ( A, p, q, r )

1.  $n_1 = q - p + 1$
2.  $n_2 = r - q$
3.  Let $L[1: n_1 + 1]$ and $R[1: n_2 + 1]$ be new arrays
4.  for $i = 1$ to $n_1$
5.      $L[i] = A[p + i - 1]$
6.  for $j = 1$ to $n_2$
7.      $R[j] = A[q + j]$
8.  $L[n_1 + 1] = \infty$
9.  $R[n_2 + 1] = \infty$
10. $i = 1$
11. $j = 1$
12. for $k = p$ to $r$
13.     if $L[i] \le R[j]$
14.         $A[k] = L[i]$
15.         $i = i + 1$
16.     else $A[k] = R[j]$
17.         $j = j + 1$
```

Loop Invariant

At the start of each iteration of the ***for*** loop of lines 12–17 the following invariant holds:

The subarray $A[p: k - 1]$ contains the $k - p$ smallest elements of $L[1: n_1 + 1]$ and $R[1: n_2 + 1]$, in sorted order.

Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Merging Two Sorted Subarrays

Input: Two subarrays $A[\, p : q \,]$ and $A[q + 1 : r]$ in sorted order ($p \leq q < r$).

Output: A single sorted subarray $A[\, p : r \,]$ by merging the input subarrays.

```
MERGE ( A, p, q, r )

1.   $n_1 = q - p + 1$
2.   $n_2 = r - q$
3.   Let $L[1{:}\, n_1 + 1]$ and $R[1{:}\, n_2 + 1]$ be new arrays
4.   for $i = 1$ to $n_1$
5.       $L[i] = A[p + i - 1]$
6.   for $j = 1$ to $n_2$
7.       $R[j] = A[q + j]$
8.   $L[n_1 + 1] = \infty$
9.   $R[n_2 + 1] = \infty$
10.  $i = 1$
11.  $j = 1$
12.  for $k = p$ to $r$
13.      if $L[i] \leq R[j]$
14.          $A[k] = L[i]$
15.          $i = i + 1$
16.      else $A[k] = R[j]$
17.          $j = j + 1$
```

Running Time

Let $n = r - p + 1$.
Then $n = n_1 + n_2$.

The loop in lines 4–5 takes $\Theta(n_1)$ time.

The loop in lines 6–7 takes $\Theta(n_2)$ time.

The loop in lines 12–17 takes $\Theta(n)$ time.

Lines 1–3 and 8–11 take $\Theta(1)$ time.

Overall running time

$$= \Theta(n_1) + \Theta(n_2) + \Theta(n) + \Theta(1)$$
$$= \Theta(n)$$

Divide-and-Conquer

1. **Divide:** divide the original problem into smaller subproblems that are easier to solve
2. **Conquer:** solve the smaller subproblems (perhaps recursively)
3. **Merge:** combine the solutions to the smaller subproblems to obtain a solution for the original problem

Intuition Behind Merge Sort

1. **Base case:** We know how to correctly sort an array containing only a single element.

   Indeed, an array of one number is already trivially sorted!

2. **Reduction to base case (recursive divide-and-conquer):**

   At each level of recursion we split the current subarray at the midpoint (approx) to obtain two subsubarrays of equal or almost equal lengths, and sort them recursively.

   We are guaranteed to reach subproblems of size 1 (i.e., the base case size) eventually which are trivially sorted.

3. **Merge:** We know how to merge two (recursively) sorted subarrays to obtain a longer sorted subarray.

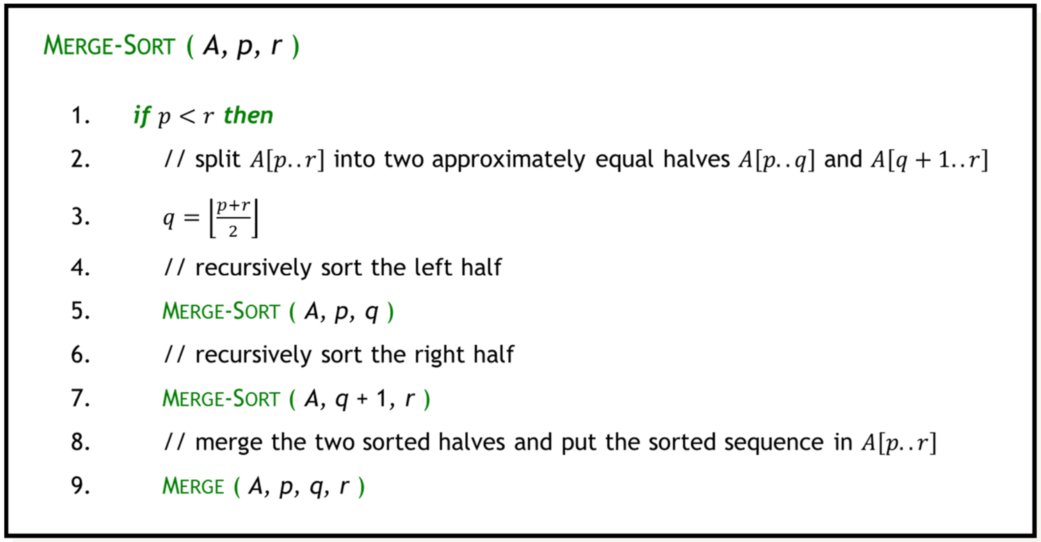
Merge Sort

Input: A subarray $A[\,p : r\,]$ of $r - p + 1$ numbers, where $p \leq r$.

Output: Elements of $A[\,p : r\,]$ rearranged in non-decreasing order of value.

```
MERGE-SORT ( A, p, r )

    if $p < r$ then
        // split $A[p..r]$ into two approximately equal halves $A[p..q]$ and $A[q + 1..r]$
        $q = \left\lfloor \frac{p+r}{2} \right\rfloor$
        // recursively sort the left half
        MERGE-SORT ( A, p, q )
        // recursively sort the right half
        MERGE-SORT ( A, q + 1, r )
        // merge the two sorted halves and put the sorted sequence in $A[p..r]$
        MERGE ( A, p, q, r )
```

Correctness of Merge Sort

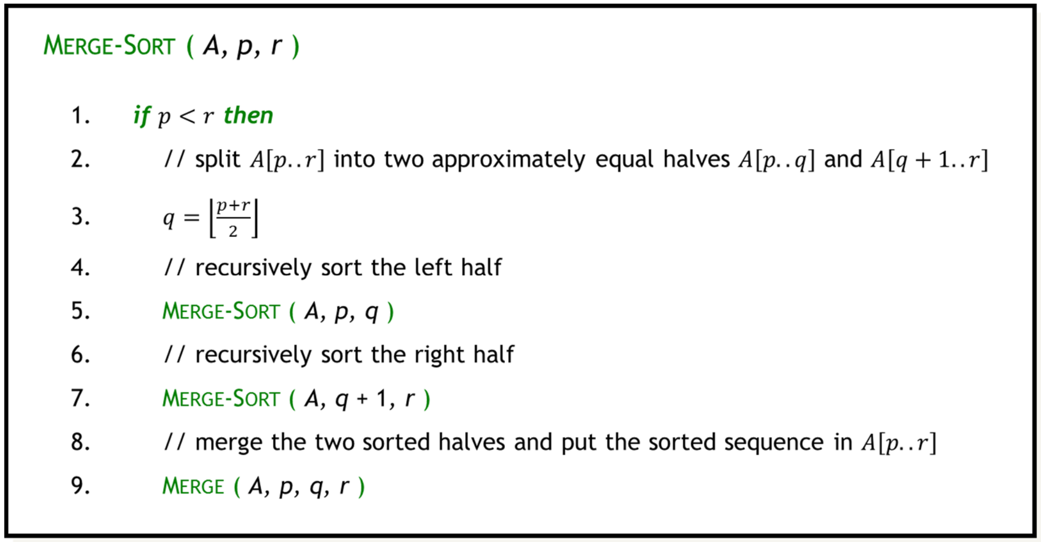


```
MERGE-SORT ( A, p, r )

if $p < r$ then
    // split $A[p..r]$ into two approximately equal halves $A[p..q]$ and $A[q+1..r]$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    // recursively sort the left half
    MERGE-SORT ( A, p, q )
    // recursively sort the right half
    MERGE-SORT ( A, q + 1, r )
    // merge the two sorted halves and put the sorted sequence in $A[p..r]$
    MERGE ( A, p, q, r )
```

The proof has two parts.

- First we will show that the algorithm terminates.
- Then we will show that the algorithm produces correct results (assuming the algorithm terminates).

Termination Guarantee



```
MERGE-SORT ( A, p, r )

if $p < r$ then
    // split $A[p..r]$ into two approximately equal halves $A[p..q]$ and $A[q+1..r]$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    // recursively sort the left half
    MERGE-SORT ( A, p, q )
    // recursively sort the right half
    MERGE-SORT ( A, q + 1, r )
    // merge the two sorted halves and put the sorted sequence in $A[p..r]$
    MERGE ( A, p, q, r )
```

Size of the input subarray, $n = r - p + 1$

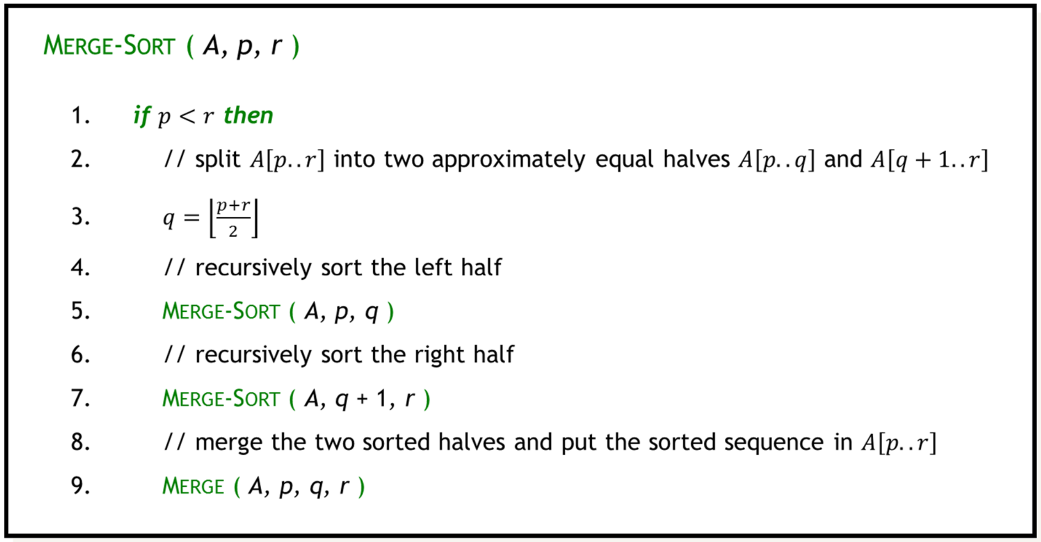
Size of the left half, $n_1 = q - p + 1$

Size of the right half, $n_2 = r - (q + 1) + 1 = r - q$

We will show the following: $\boxed{n_1 < n \text{ and } n_2 < n}$

Meaning: Sizes of subproblems decrease by at least 1 in each recursive call, and so there cannot be more than $n - 1$ levels of recursion. So MERGE-SORT will terminate in finite time.

Termination Guarantee



```
MERGE-SORT ( A, p, r )

1.  if $p < r$ then
2.      // split $A[p..r]$ into two approximately equal halves $A[p..q]$ and $A[q+1..r]$
3.      $q = \lfloor \frac{p+r}{2} \rfloor$
4.      // recursively sort the left half
5.      MERGE-SORT ( A, p, q )
6.      // recursively sort the right half
7.      MERGE-SORT ( A, q + 1, r )
8.      // merge the two sorted halves and put the sorted sequence in $A[p..r]$
9.      MERGE ( A, p, q, r )
```

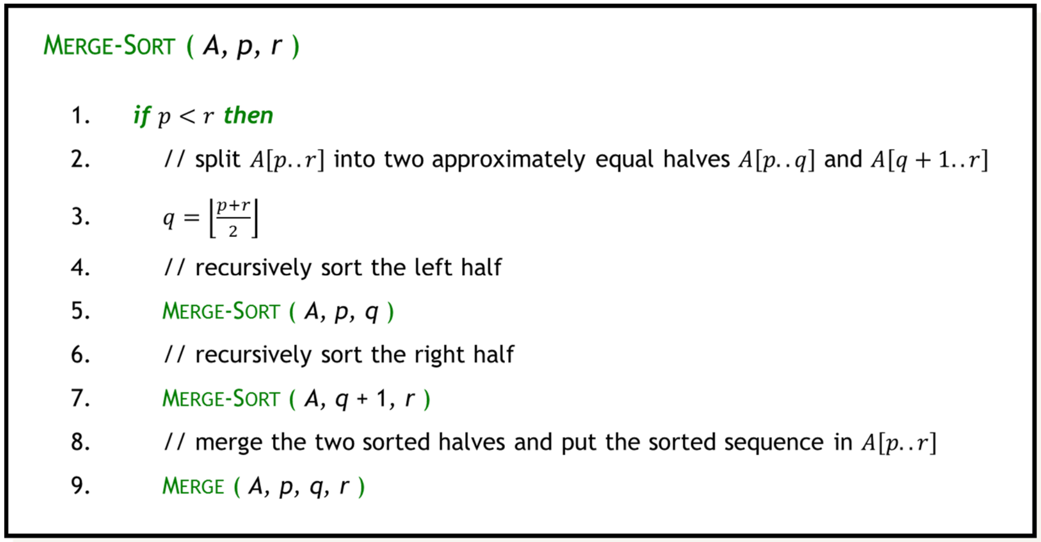
A problem will be recursively subdivided (i.e., lines 5 and 7 will be executed) provided the following holds in line 1: $p < r$

But $p < r$ implies:

$$p + r < 2r \Rightarrow \frac{p+r}{2} < r \Rightarrow \left\lfloor \frac{p+r}{2} \right\rfloor < r$$

$$\Rightarrow q < r \Rightarrow q - p + 1 < r - p + 1 \Rightarrow n_1 < n$$

Termination Guarantee



```
MERGE-SORT ( A, p, r )

1.  if $p < r$ then
2.      // split $A[p..r]$ into two approximately equal halves $A[p..q]$ and $A[q+1..r]$
3.      $q = \left\lfloor \frac{p+r}{2} \right\rfloor$
4.      // recursively sort the left half
5.      MERGE-SORT ( A, p, q )
6.      // recursively sort the right half
7.      MERGE-SORT ( A, q + 1, r )
8.      // merge the two sorted halves and put the sorted sequence in $A[p..r]$
9.      MERGE ( A, p, q, r )
```

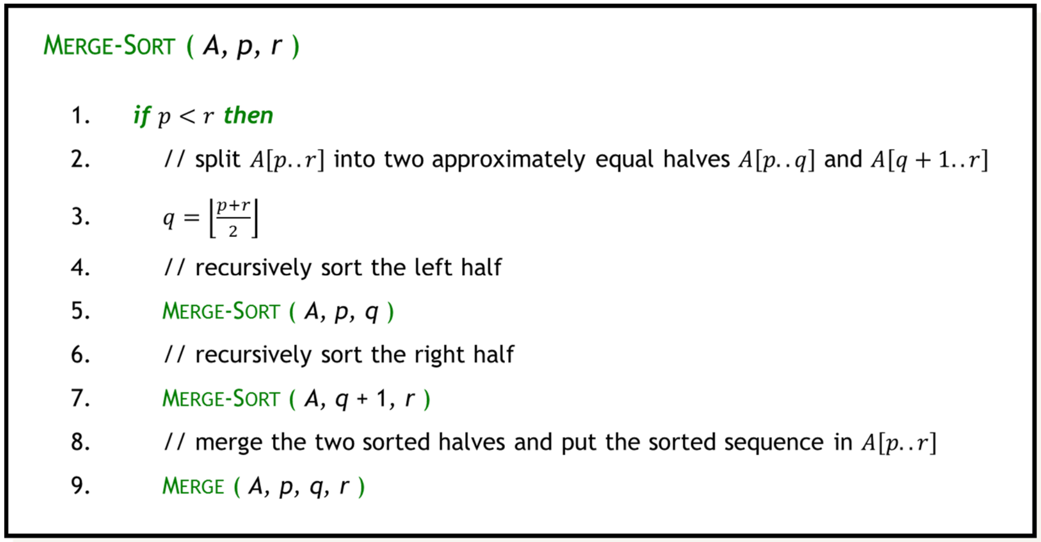
A problem will be recursively subdivided (i.e., lines 5 and 7 will be executed) provided the following holds in line 1: $p < r$

$p < r$ also implies:

$$2p < p + r \Rightarrow p < \frac{p+r}{2} \Rightarrow p \leq \left\lfloor \frac{p+r}{2} \right\rfloor \Rightarrow p \leq q$$

$$\Rightarrow -q \leq -p \Rightarrow r - q \leq r - p \Rightarrow r - q < r - p + 1 \Rightarrow n_2 < n$$

Inductive Proof of Correctness



```
MERGE-SORT ( A, p, r )

if $p < r$ then
    // split $A[p..r]$ into two approximately equal halves $A[p..q]$ and $A[q+1..r]$
    $q = \left\lfloor \frac{p+r}{2} \right\rfloor$
    // recursively sort the left half
    MERGE-SORT ( A, p, q )
    // recursively sort the right half
    MERGE-SORT ( A, q + 1, r )
    // merge the two sorted halves and put the sorted sequence in $A[p..r]$
    MERGE ( A, p, q, r )
```

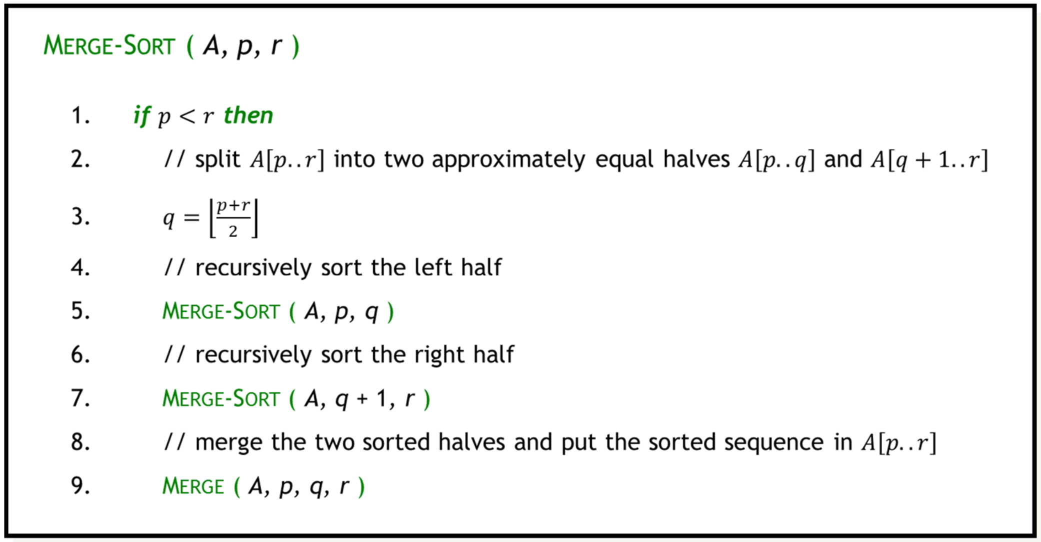
Let $n = r - p + 1$.

Base Case: The algorithm is trivially correct when $r \geq p$, i.e., $n \leq 1$.

Inductive Hypothesis: Suppose the algorithm works correctly for all integral values of n not larger than k, where $k \geq 1$ is an integer.

Inductive Step: We will prove that the algorithm works correctly for $n = k + 1$.

Inductive Proof of Correctness



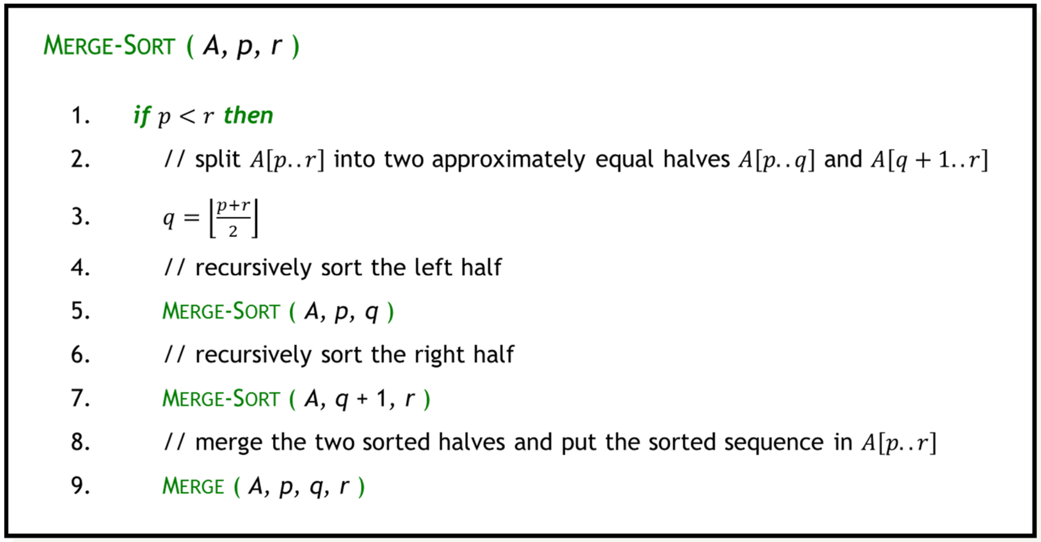
```
MERGE-SORT ( A, p, r )

1.  if $p < r$ then
2.      // split $A[p..r]$ into two approximately equal halves $A[p..q]$ and $A[q+1..r]$
3.      $q = \left\lfloor \frac{p+r}{2} \right\rfloor$
4.      // recursively sort the left half
5.      MERGE-SORT ( A, p, q )
6.      // recursively sort the right half
7.      MERGE-SORT ( A, q + 1, r )
8.      // merge the two sorted halves and put the sorted sequence in $A[p..r]$
9.      MERGE ( A, p, q, r )
```

When $n = k + 1$, lines 2–9 of the algorithm will be executed because $k \geq 1 \Rightarrow n > 1 \Rightarrow r - p + 1 > 1 \Rightarrow p < r$ holds in line 1.

The algorithm splits the input subarray $A[p{:}r]$ into two parts: $A[p{:}q]$ and $A[q + 1{:}r]$, where $q = \left\lfloor \frac{p+r}{2} \right\rfloor$.

The recursive call in line 5 sorts the left part $A[p{:}q]$. Since $A[p{:}q]$ containis $n_1 = q - p + 1 < n \Rightarrow n_1 \leq k$ numbers, it is sorted correctly (using inductive hypothesis).

Inductive Proof of Correctness



```
MERGE-SORT ( A, p, r )

1.  if $p < r$ then
2.      // split $A[p..r]$ into two approximately equal halves $A[p..q]$ and $A[q+1..r]$
3.      $q = \lfloor \frac{p+r}{2} \rfloor$
4.      // recursively sort the left half
5.      MERGE-SORT ( A, p, q )
6.      // recursively sort the right half
7.      MERGE-SORT ( A, q + 1, r )
8.      // merge the two sorted halves and put the sorted sequence in $A[p..r]$
9.      MERGE ( A, p, q, r )
```

The recursive call in line 7 sorts the right part $A[q+1:r]$. Since $A[q+1:r]$ containis $n_2 = r - q < n \Rightarrow n_2 \leq k$ numbers, it is sorted correctly (using inductive hypothesis).

We know that the MERGE algorithm can merge two sorted arrays correctly. So, line 9 correctly merges the sorted left and right parts of the input subarray into a single sorted sequence in $A[p:q]$.

Therefore, the algorithm works correctly for $n = k + 1$, and consequently for all integral values of n.

Analyzing Divide-and-Conquer Algorithms

Let $T(n)$ be the running time of the algorithm on a problem of size n.

- If the problem size is small enough, say $n \leq c$ for some constant c, the straightforward solution takes $\Theta(1)$ time.
- Suppose our division of the problem yields a subproblems, each of which is $1/b$ the size of the original.
- Let $D(n) =$ time needed to divide the problem into subproblems.
- Let $C(n) =$ time needed to combine the solutions to the subproblems into the solution to the original problem.

$$\text{Then } T(n) = \begin{cases} \Theta(1) & if\ n \leq c, \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & otherwise. \end{cases}$$

Analysis of Merge Sort

Let $T(n)$ be the worst-case running time of MERGE-SORT on n numbers.

We reason as follows to set up the recurrence for $T(n)$.

- When $n = 1$, MERGE-SORT takes $\Theta(1)$ time.
- When $n > 1$, we break down the running time as follows.
    - **Divide:** This step simply computes the middle of the subarray, which takes constant time. Hence, $D(n) = \Theta(1)$.
    - **Conquer:** We recursively solve 2 subproblems of size $n/2$ each, which adds $2T(n/2)$ to the running time.
    - **Combine:** The MERGE procedure takes $\Theta(n)$ time on an n-element subarray. Hence, $C(n) = \Theta(n)$.

$$\text{Then } T(n) = \begin{cases} \Theta(1) & if\ n = 1, \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & if\ n > 1. \end{cases}$$

Analysis of Merge Sort

Let us assume for simplicity that $n = 2^k$ for some integer $k \geq 0$, and for constants c_1 and c_2:

$$T(n) = \begin{cases} c_1 & if\ n = 1, \\ 2T\left(\frac{n}{2}\right) + c_2 n & if\ n > 1; \end{cases}$$

where, c_1 is the time needed to solve a problem of size 1, and c_2 is the time per array element of the divide and combine steps.

Let’s see how the recursion unfolds.

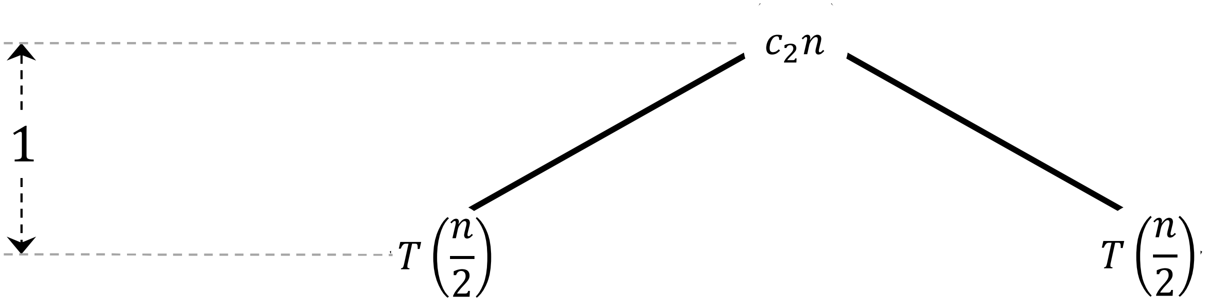
Analysis of Merge Sort

Running time on an input of size $n = 2^k$ for some integer $k \geq 0$:

$$T(n)$$

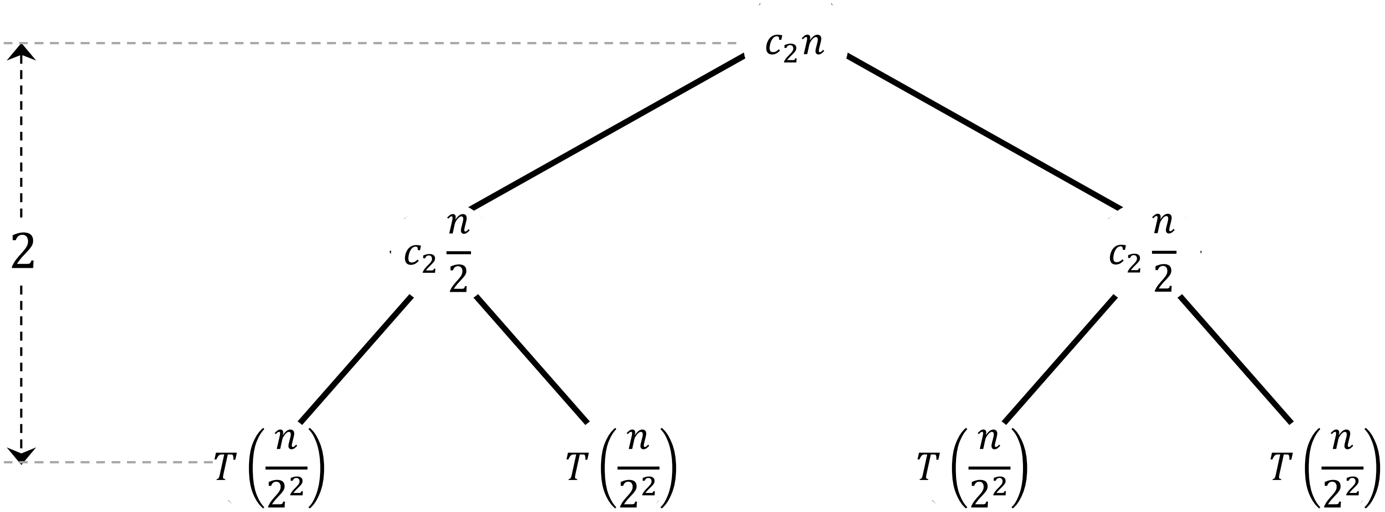
Analysis of Merge Sort

Unfolding the recurrence up to level 1:



Analysis of Merge Sort

Unfolding the recurrence up to level 2:



$c_2 n$

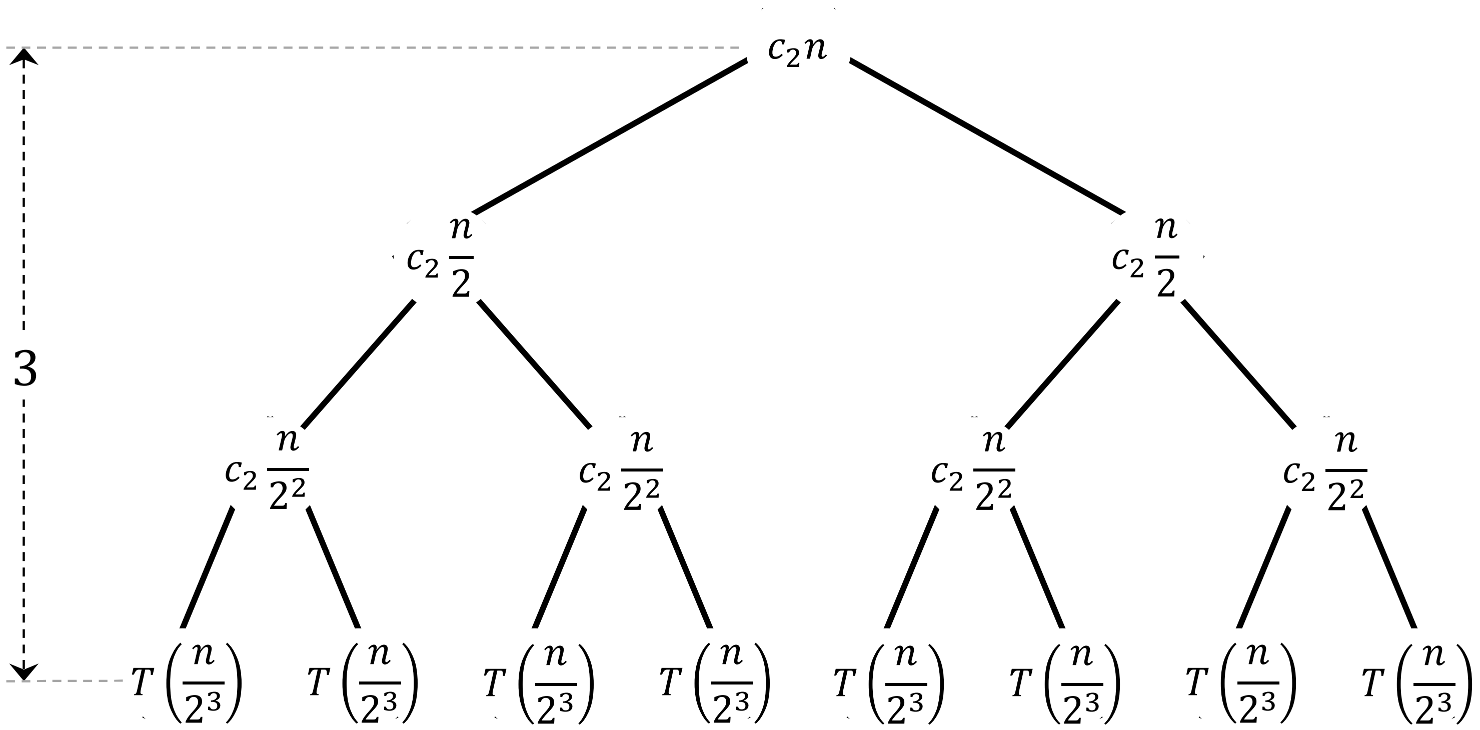
$c_2 \frac{n}{2}$ $c_2 \frac{n}{2}$

$T\left(\frac{n}{2^2}\right)$ $T\left(\frac{n}{2^2}\right)$ $T\left(\frac{n}{2^2}\right)$ $T\left(\frac{n}{2^2}\right)$

2

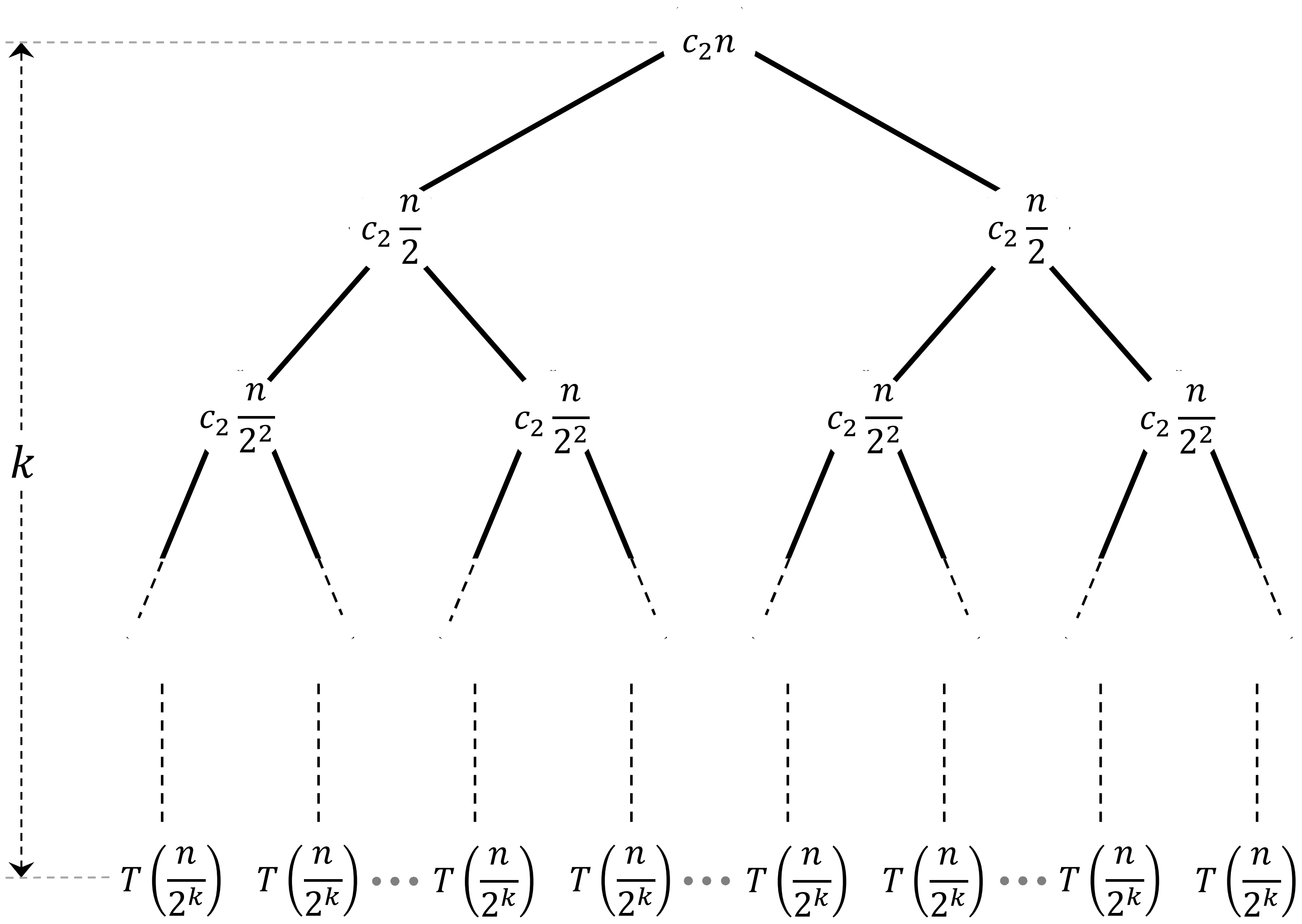
Analysis of Merge Sort

Unfolding the recurrence up to level 3:



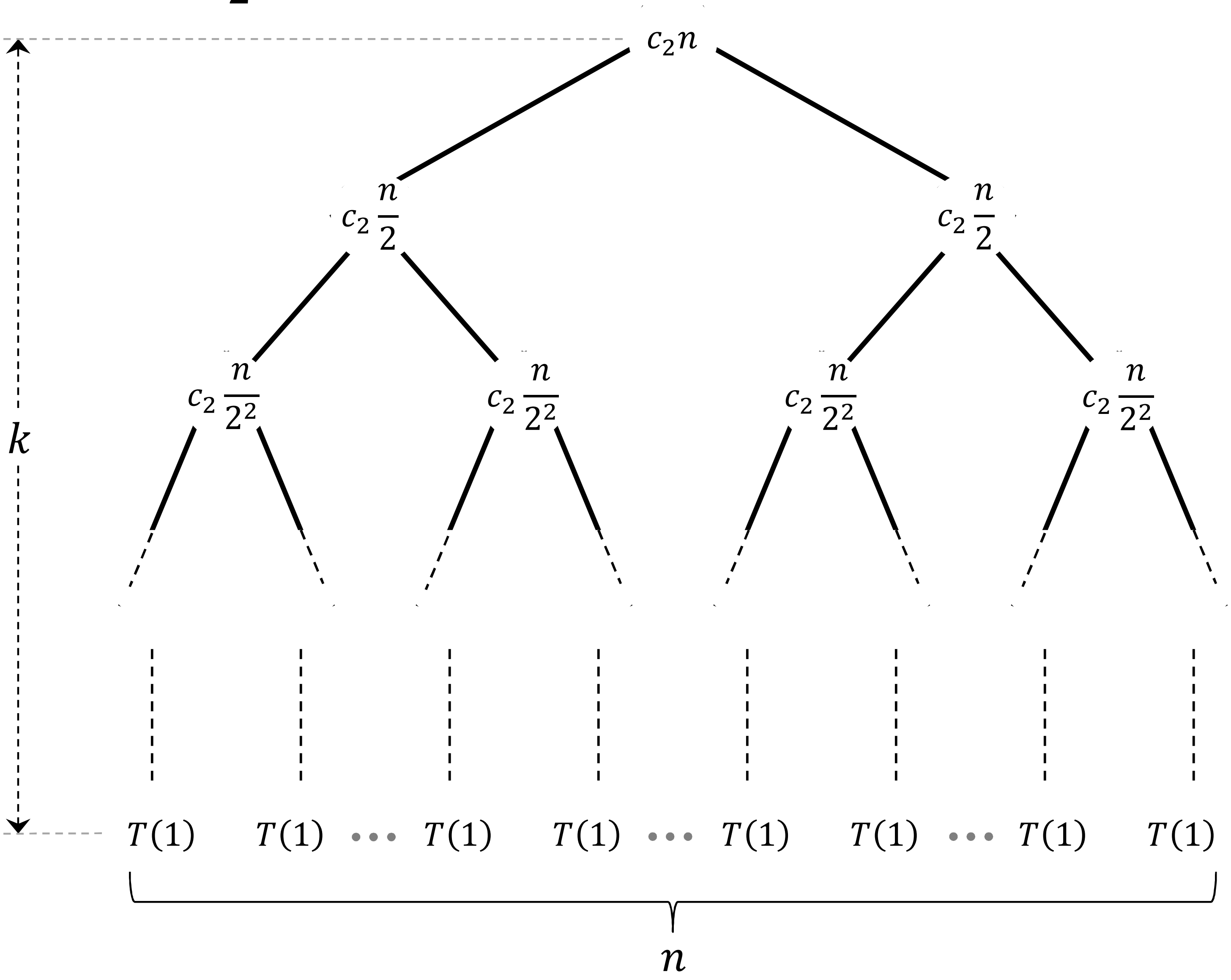
Analysis of Merge Sort

Unfolding the recurrence up to level k:



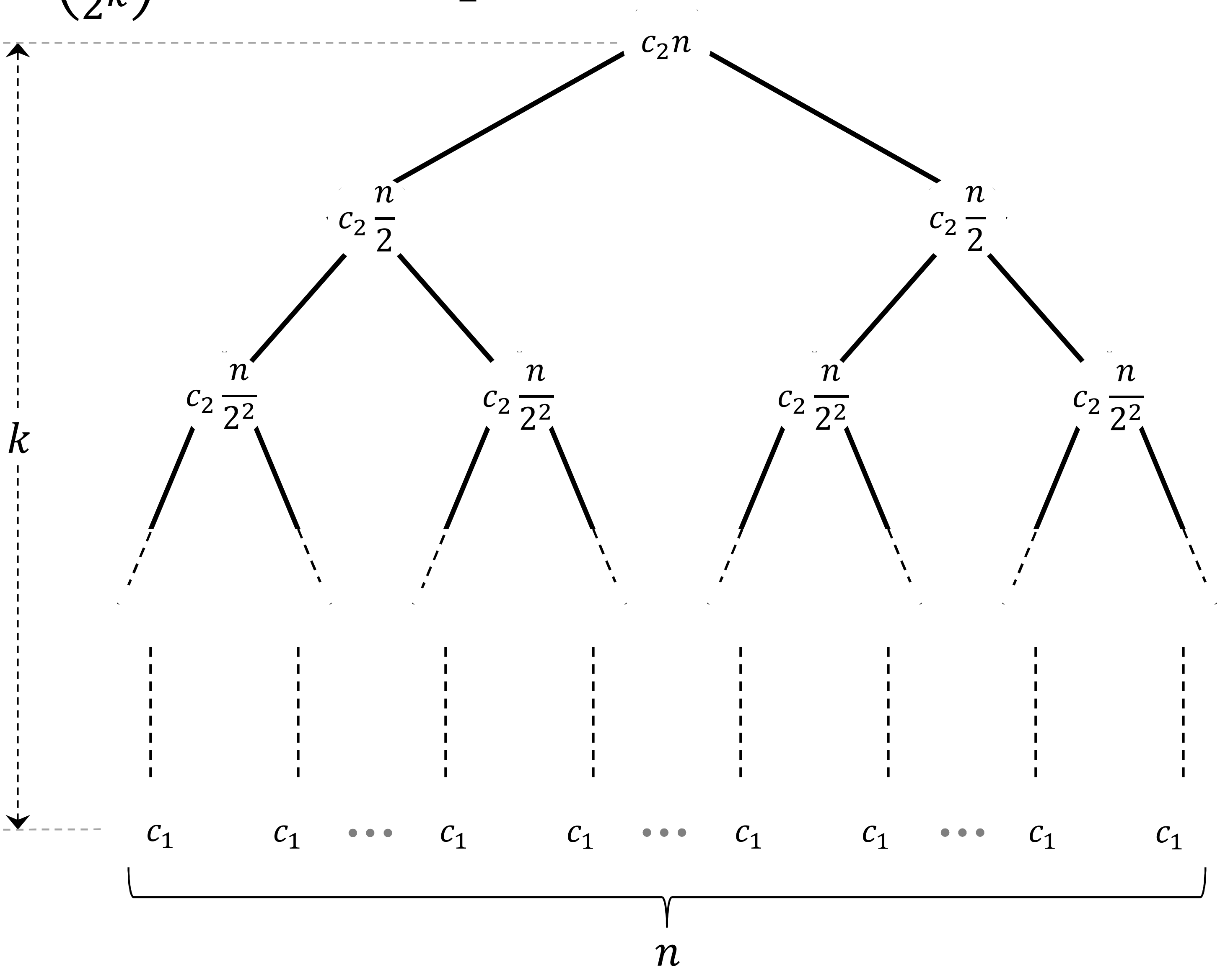
Analysis of Merge Sort

But $n = 2^k \Rightarrow \frac{n}{2^k} = 1$, and there will be n nodes (leaves) at level k:



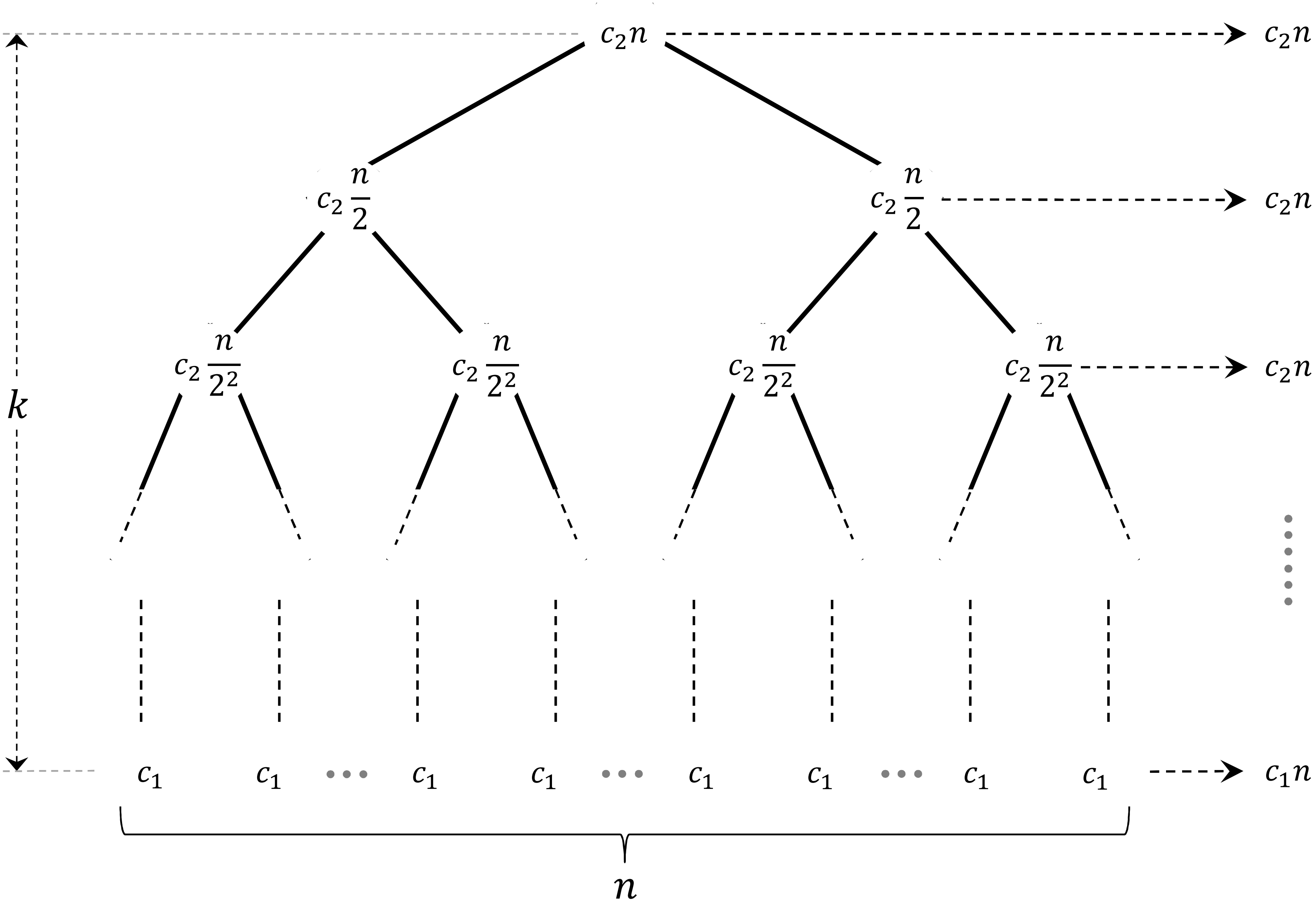
Analysis of Merge Sort

Then $T\left(\frac{n}{2^k}\right) = T(1) = c_1$:



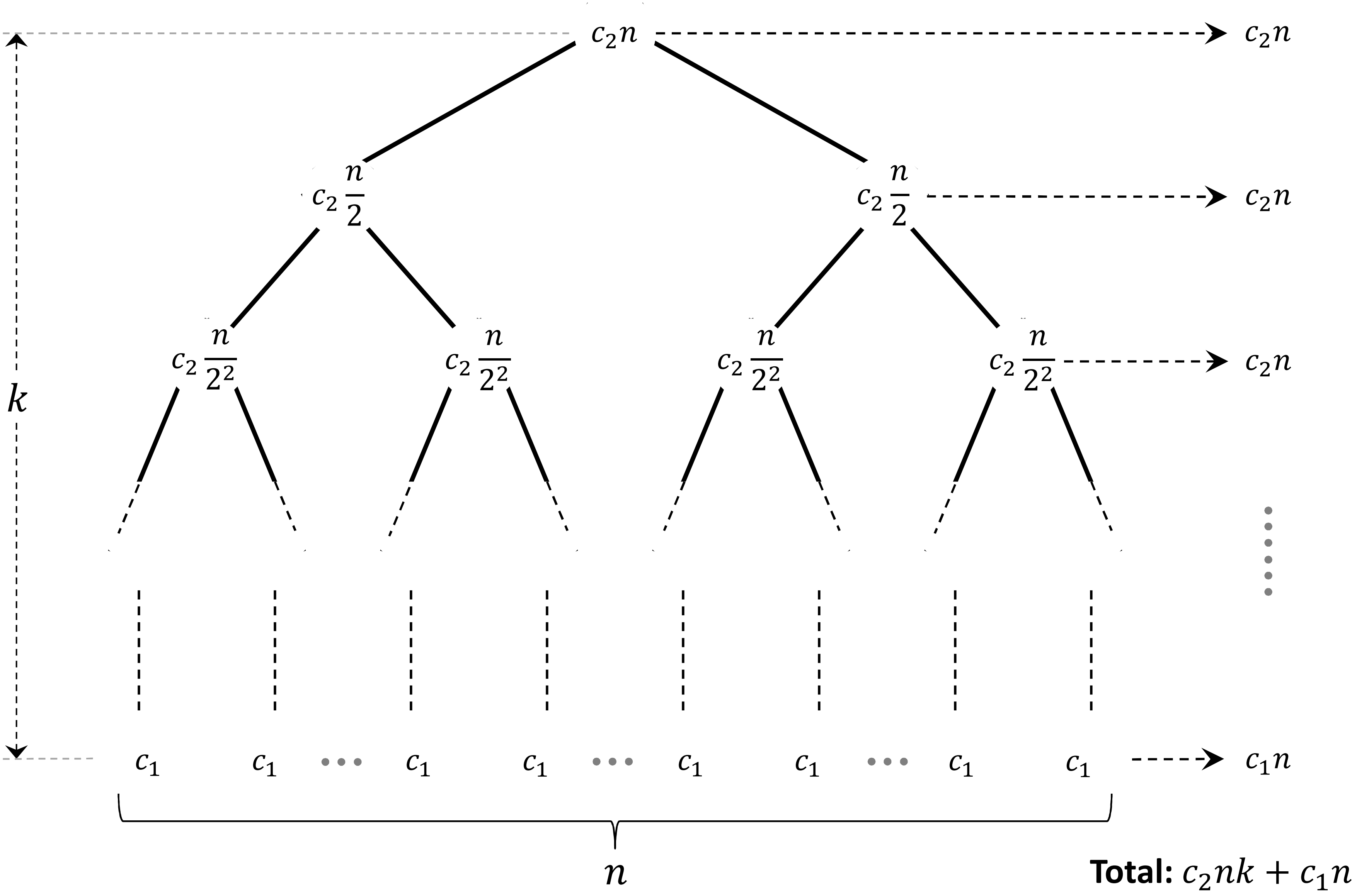
Analysis of Merge Sort

Total work at each level:



Analysis of Merge Sort

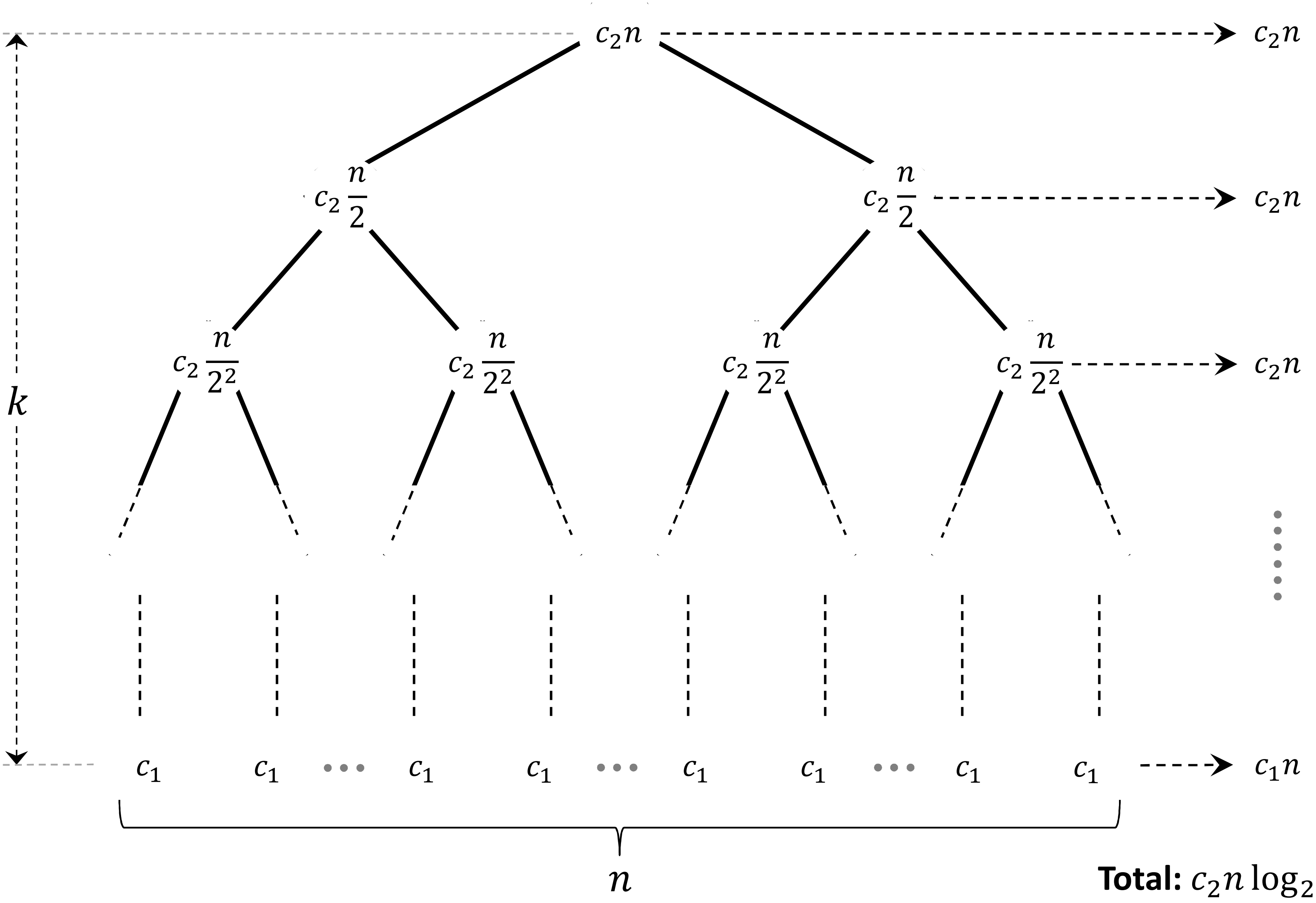
Total work across all levels:



Total: $c_2 nk + c_1 n$

Analysis of Merge Sort

But $n = 2^k \Rightarrow k = \log_2 n$:



Total: $c_2 n \log_2 n + c_1 n$

$= \Theta(n \log n)$