CSE 548: Analysis of Algorithms

Prerequisites Review 1
(Divide-and-Conquer Algorithms: Merge Sort)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Fall 2019

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.  n1 = q − p + 1
2.  n2 = r − q
3.  Let L[1: n1 + 1] and R[1: n2 + 1] be new arrays
4.  for i = 1 to n1
5.      L[i] = A[p + i − 1]
6.  for j = 1 to n2
7.      R[j] = A[q + j]
8.  L[n1 + 1] = ∞
9.  R[n2 + 1] = ∞
10. i = 1
11. j = 1
12. for k = p to r
13.     if L[i] ≤ R[j]
14.         A[k] = L[i]
15.         i = i + 1
16.     else A[k] = R[j]
17.         j = j + 1
```


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

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.  n1 = q − p + 1
2.  n2 = r − q
3.  Let L[1: n1 + 1] and R[1: n2 + 1] be new arrays
4.  for i = 1 to n1
5.      L[i] = A[p + i − 1]
6.  for j = 1 to n2
7.      R[j] = A[q + j]
8.  L[n1 + 1] = ∞
9.  R[n2 + 1] = ∞
10. i = 1
11. j = 1
12. for k = p to r
13.     if L[i] ≤ 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.

(Prerequisites Review 1) Divide-and-Conquer Algorithms: Merge Sort

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] = ∞
9.   R[n_2 + 1] = ∞
10.  i = 1
11.  j = 1
12.  for k = p to r
13.      if L[i] ≤ 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 $p \le r$.

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 = ⌊(p+r)/2⌋
    // 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 )
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 = ⌊(p+r)/2⌋
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 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 )
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 = ⌊(p+r)/2⌋
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 )
```


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 = ⌊(p+r)/2⌋
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 = ⌊(p+r)/2⌋
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:

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

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

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 = ⌊(p+r)/2⌋
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 )
```

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 $n = k + 1$.

14

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 = ⌊(p+r)/2⌋
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).

15

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 = ⌊(p+r)/2⌋
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.

16

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.

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

17

(Prerequisites Review 1) Divide-and-Conquer Algorithms: Merge Sort

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, $C(n) = \Theta(n)$.

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 (Upper Bound)

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

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

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

Let's see how the recursion unfolds.

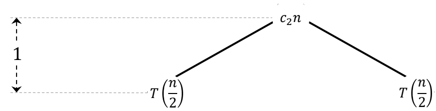
Analysis of Merge Sort (Upper Bound)

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

$T(n)$

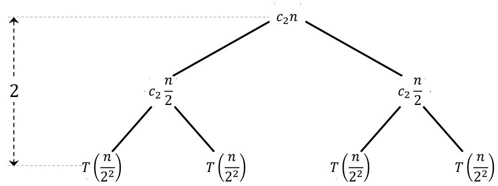
Analysis of Merge Sort (Upper Bound)

Unfolding the recurrence up to level 1:



Analysis of Merge Sort (Upper Bound)

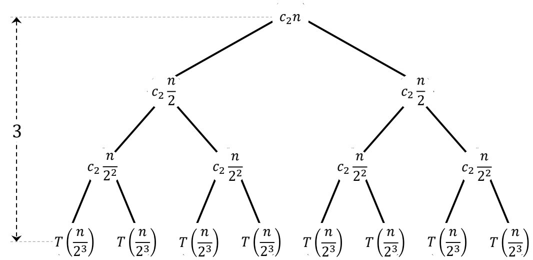
Unfolding the recurrence up to level 2:



22

Analysis of Merge Sort (Upper Bound)

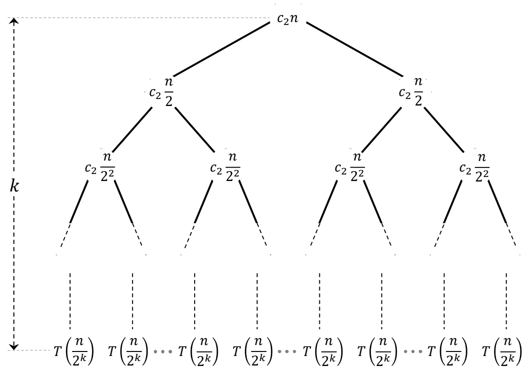
Unfolding the recurrence up to level 3:



23

Analysis of Merge Sort (Upper Bound)

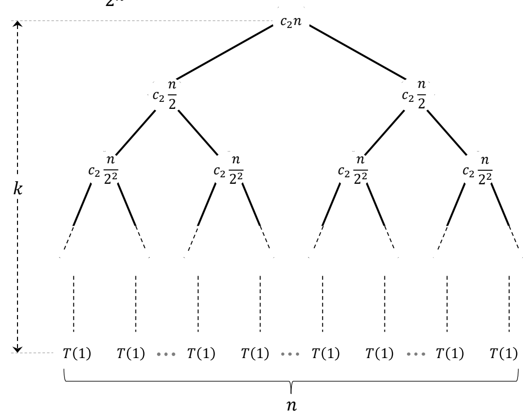
Unfolding the recurrence up to level k:



24

Analysis of Merge Sort (Upper Bound)

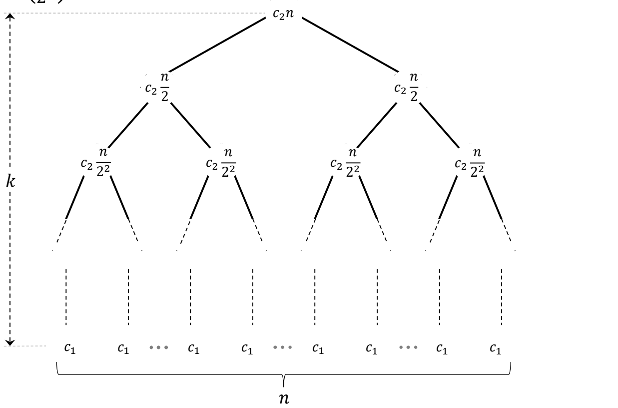
But $n = 2^k \Rightarrow \frac{n}{2^k} = 1$, and there will be n nodes (leaves) at level k:



25

Analysis of Merge Sort (Upper Bound)

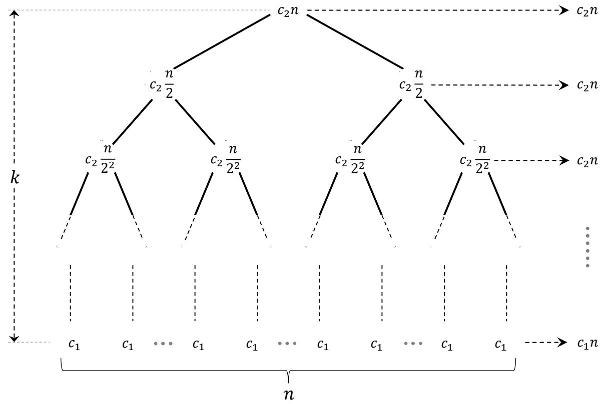
Then $T\left(\frac{n}{2^k}\right) = T(1) = c_1$:



26

Analysis of Merge Sort (Upper Bound)

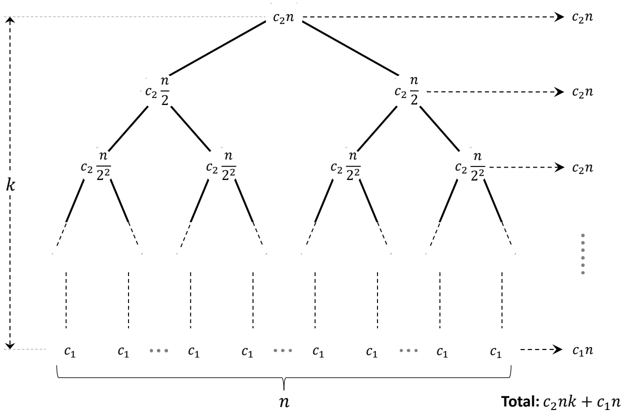
Total work at each level:



27

Analysis of Merge Sort (Upper Bound)

Total work across all levels:

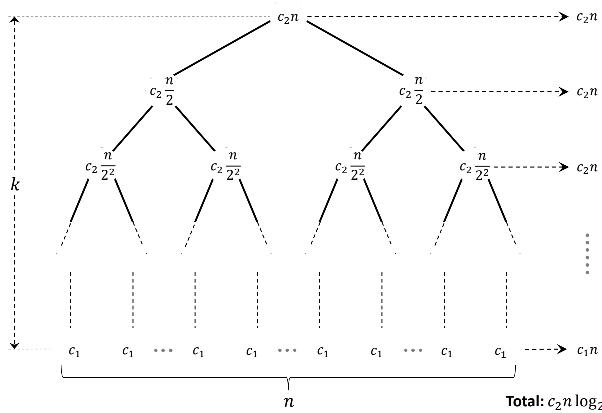


Total: $c_2 nk + c_1 n$

28

Analysis of Merge Sort (Upper Bound)

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



Total: $c_2 n \log_2 n + c_1 n$
$= \Theta(n \log n)$ 29

(Prerequisites Review 1) Divide-and-Conquer Algorithms: Merge Sort

Analysis of Merge Sort (Upper Bound)

Hence, we have:

$$T(n) \leq \Theta(n \log n)$$

Implying:

$$T(n) = O(n \log n)$$

30

Analysis of Merge Sort (Lower Bound)

Assuming $n = 2^k$ for some integer $k \geq 0$, for some constants c_1' and c_2', we have:

$$T(n) \geq \begin{cases} c_1' & if\ n = 1, \\ 2T\left(\frac{n}{2}\right) + c_2' n & if\ n > 1; \end{cases}$$

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

Using the approach we used for proving the upper bound, we have:

$$T(n) \geq \Theta(n \log n)$$

Implying:

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

31

Analysis of Merge Sort (Tight Bound)

We have proved, upper bound: $T(n) = O(n \log n)$

and lower bound: $T(n) = \Omega(n \log n)$

Combining we get the tight bound:

$$\boxed{T(n) = \Theta(n \log n)}$$

32

33