Introduction to Algorithms

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

1 Merge Sort
- The Divide-and-Conquer Approach
- Analyzing Merge Sort

Subsection 1

The Divide-and-Conquer Approach

Divide and Conquer

- To solve a given problem, many algorithms call themselves **recursively** one or more times to deal with closely related subproblems.
- These algorithms typically follow a **divide-and-conquer approach** involving three steps:
  - **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
  - **Conquer** the subproblems by solving them recursively.
  - **Combine** the solutions into the solution for the original problem.
- The **merge sort** algorithm closely follows the divide-and-conquer paradigm operating as follows:
  - **Divide**: Divide the n-element sequence to be sorted into two subsequences of $\frac{n}{2}$ elements each.
  - **Conquer**: Sort the two subsequences recursively using merge sort.
  - **Combine**: Merge the two sorted subsequences to obtain the answer.

  The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done.

The Combine Step of the Algorithm

- The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step.
- To perform the merging, we use an auxiliary procedure $\textsc{Merge}(\mathsf{A}, p, q, r)$, where A is an array and p, q and r are indices numbering elements of the array such that $p \leq q < r$.
    - The procedure takes sorted subarrays $\mathsf{A}[p \ldots q]$, $\mathsf{A}[q+1 \ldots r]$.
    - It merges them to a sorted subarray replacing $\mathsf{A}[p \ldots r]$.
- The $\textsc{Merge}$ procedure takes time $\Theta(n)$, where $n = r - p + 1$ is the number of elements being merged.
- To merge the two arrays into a single sorted array, it chooses the smaller of the two smallest numbers, removing it from its array and placing this card to the first position in the combined array. The step is repeated until one input array is empty. At that time the remaining input array is merged at the end of the output array.
- Each basic step takes constant time checking just the two smallest elements. With at most n basic steps, merging takes $\Theta(n)$ time.

Checking for an Empty Subarray

- The following pseudocode implements the above idea.
- In addition, it avoids having to check whether either subarray is empty in each basic step by placing at the "bottom" of each a special value ∞.
- Once all arrays show ∞, all the other elements have already been placed onto the output array.
- Since we know in advance that exactly $r - p + 1$ cards will be placed onto the output array, we can stop once we have performed that many basic steps.

The Merge Procedure

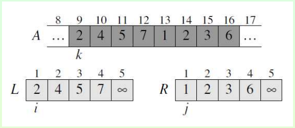
Merge(A, p, q, r)

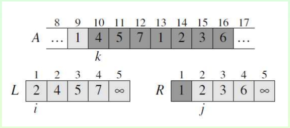
```
1.  $n_1 = q - p + 1$
2.  $n_2 = r - q$
3.  let L$[1 \ldots n_1 + 1]$ and R$[1 \ldots n_2 + 1]$ be new arrays
4.  for $i = 1$ to $n_1$
5.      L$[i]$ = A$[p + i - 1]$
6.  for $j = 1$ to $n_2$
7.      R$[j]$ = A$[q + j]$
8.  L$[n_1 + 1] = \infty$
9.  R$[n_2 + 1] = \infty$
10. $i = 1$
11. $j = 1$
12. 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.         $j = j + 1$
```

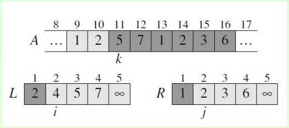
How the MERGE Procedure Works

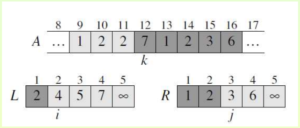
- MERGE procedure works as follows:
  - Line 1 computes the length n_1 of the subarray $A[p \dots q]$, and Line 2 computes the length n_2 of the subarray $A[q+1 \dots r]$.
  - In Line 3 we create arrays L and R of lengths $n_1 + 1$ and $n_2 + 1$, respectively.
  - The for loop of Lines 4-5 copies the subarray $A[p \dots q]$ into $L[1 \dots n_1]$.
  - The for loop of Lines 6-7 copies the subarray $A[q+1 \dots r]$ into $R[1 \dots n_2]$.
  - Lines 8-9 put the ∞'s at the ends of the arrays L and R.
  - Lines 10-17 perform the $r - p + 1$ basic steps by maintaining the following loop invariant:
    - At the start of each iteration of the for loop of Lines 12-17, the subarray $A[p \dots k-1]$ contains the $k - p$ smallest elements of $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$, in sorted order.
    - Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

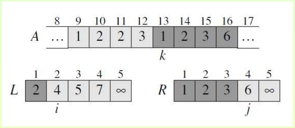
Illustration of MERGE-SORT

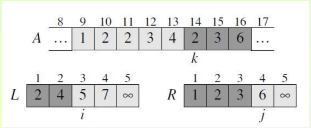


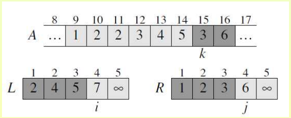


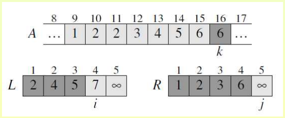


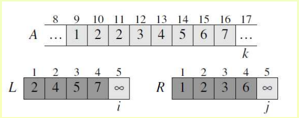












Maintaining the Loop Invariant

- We must show that:
  - The loop invariant holds prior to the first iteration of the for loop of Lines 12-17.
  - Each iteration of the loop maintains the invariant.
  - The invariant provides a useful property to show correctness when the loop terminates.
- Initialization: Prior to the first iteration of the loop, we have $k = p$. So the subarray A$[p \ldots k-1]$ is empty. This empty subarray contains the $k - p = 0$ smallest elements of L and R. Since $i = j = 1$, both L$[i]$ and R$[j]$ are the smallest elements of their arrays that have not been copied back into A.

Maintaining the Loop Invariant (Cont'd)

- Maintenance: To see that each iteration maintains the loop invariant, suppose, first, that $\mathrm{L}[i] \leq \mathrm{R}[j]$. Then $\mathrm{L}[i]$ is the smallest element not yet copied back into A. Because $\mathrm{A}[p \ldots k-1]$ contains the $k-p$ smallest elements, after Line 14 copies $\mathrm{L}[i]$ into $\mathrm{A}[k]$, the subarray $\mathrm{A}[p \ldots k]$ will contain the $k-p+1$ smallest elements. Incrementing k (in the for loop update) and i (in Line 15) reestablishes the loop invariant for the next iteration. If, instead, $\mathrm{L}[i] > \mathrm{R}[j]$, then Lines 16-17 perform the appropriate action to maintain the loop invariant.
- Termination: At termination, $k = r+1$. By the loop invariant, the subarray $\mathrm{A}[p \ldots k-1]$, which is $\mathrm{A}[p \ldots r]$, contains the $k-p = r-p+1$ smallest elements of $\mathrm{L}[1 \ldots n_1+1]$ and $\mathrm{R}[1 \ldots n_2+1]$, in sorted order. The arrays L and R together contain $n_1+n_2+2 = r-p+3$ elements. All but the two largest have been copied back into A, and these two largest elements are the ∞'s.

Performance of MERGE

- To see that the MERGE procedure runs in $\Theta(n)$ time, where $n = r - p + 1$, observe that:
    - Each of Lines 1-3 and 8-11 takes constant time;
    - The for loops of Lines 4-7 take $\Theta(n_1 + n_2) = \Theta(n)$ time;
    - There are n iterations of the for loop of Lines 12-.17, each of which takes constant time.

Introducing MERGE-SORT

- The procedure MERGE-SORT(A, p, r), which uses MERGE as a subroutine, sorts the elements in the subarray $A[p \dots r]$.
  - If $p \geq r$, the subarray has at most one element and is therefore already sorted.
  - Otherwise, the divide step simply computes an index q that partitions $A[p \dots r]$ into two subarrays:
    - $A[p \dots q]$, containing $\lceil \frac{n}{2} \rceil$ elements;
    - $A[q + 1 \dots r]$, containing $\lfloor \frac{n}{2} \rfloor$ elements.

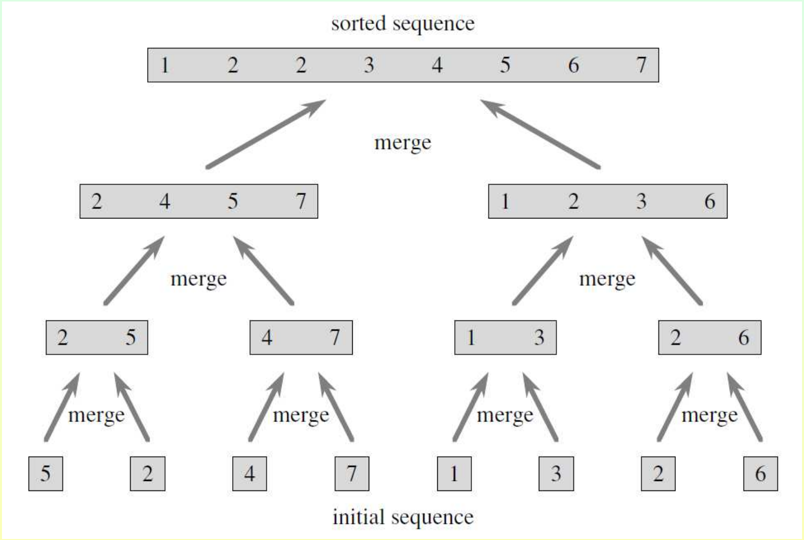
The MERGE-SORT Algorithm and its Performance

MERGE-SORT(A, p, r)

```
1. if $p < r$
2.     $q = \lfloor (p + r)/2 \rfloor$
3.     MERGE-SORT(A, $p$, $q$)
4.     MERGE-SORT(A, $q + 1$, $r$)
5.     MERGE(A, $p$, $q$, $r$)
```

- To sort the entire sequence $A = \langle A[1], A[2], \ldots, A[n] \rangle$, we make the initial call MERGE-SORT(A, 1, length[A]), where $\text{length}[A] = n$.

Illustration of MERGE-SORT



Subsection 2

Analyzing Merge Sort

Analysis of Merge Sort

- Assume that the size of the original problem is a power of 2.
- The recurrence for $T(n)$, the worst-case running time of merge sort on n numbers, is obtained by the following reasoning:
  - Merge sort on just one element takes constant time.
  - When we have $n > 1$ elements, we break down the running time as follows:

    Divide: The divide step just computes the middle of the subarray, which takes constant time, whence $D(n) = \Theta(1)$.

    Conquer: We recursively solve two subproblems, each of size $\frac{n}{2}$, which contributes $2T(\frac{n}{2})$ to the running time.

    Combine: We have already noted that the MERGE procedure on an n-element subarray takes time $\Theta(n)$, so $C(n) = \Theta(n)$.

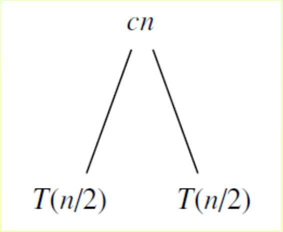
  Adding $D(n)$ and $C(n)$, we get a linear function of n, that is, $\Theta(n)$. Adding it to the $2T(\frac{n}{2})$ term from the "conquer" step gives the recurrence for the worst-case running time $T(n)$ of merge sort.

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

Analysis of Merge Sort

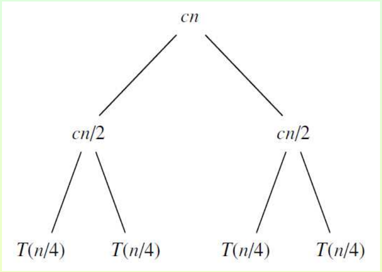
- We rewrite the recurrence $T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 2T(\frac{n}{2}) + \Theta(n), & \text{if } n > 1 \end{cases}$ as $T(n) = \begin{cases} c, & \text{if } n = 1 \\ 2T(\frac{n}{2}) + cn, & \text{if } n > 1 \end{cases}$, where the constant c represents the time required to solve problems of size 1 as well as the time per array element of the divide and combine steps.

  $T(n)$ is expanded into an equivalent tree representing the recurrence. The cn term is the root (the cost at the top level of recursion), and the two subtrees of the root are the two smaller recurrences $T(\frac{n}{2})$.



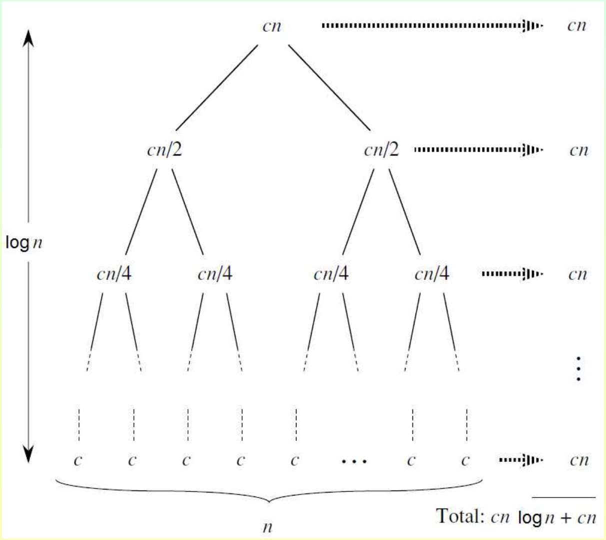
Analysis of Merge Sort (Cont'd)

- This process carried one step further by expanding $T(\frac{n}{2})$.



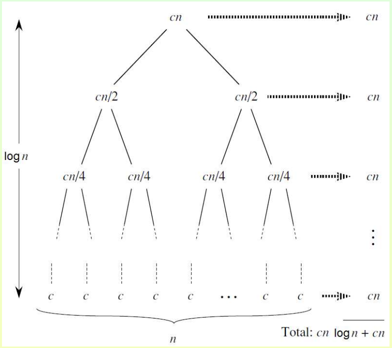
  The cost for each of the two subnodes at the second level of recursion is $c\frac{n}{2}$.
- We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence, until the problem sizes get down to 1, each with a cost of c.

Illustration of the Entire Tree



Cost at Each Level

- We now add the costs across each level of the tree.
  - The top level has total cost cn;
  - The next level down has total cost $c\frac{n}{2} + c\frac{n}{2} = cn$;
  - The level after that has total cost $c\frac{n}{4} + c\frac{n}{4} + c\frac{n}{4} + c\frac{n}{4} = cn$;
  - Level i from the top has 2^i nodes, each contributing a cost of $c\frac{n}{2^i}$, so that the i-th level below the top has total cost $2^i \cdot c\frac{n}{2^i} = cn$.



  - At the bottom level, there are n nodes, each contributing a cost of c, for a total cost of cn.

Number of Levels and Total Cost

- The total number of levels of the "recursion tree" is $\log n + 1$.
  This fact is easily seen by an informal inductive argument.
  - The base case occurs when $n = 1$, in which case there is only one level. Since $\log 1 = 0$, we have that $\log n + 1$ gives the correct number of levels.
  - Now assume as an inductive hypothesis that the number of levels of a recursion tree for 2^i nodes is $\log 2^i + 1 = i + 1$.
  - Because we are assuming that the original input size is a power of 2, the next input size to consider is 2^{i+1}. A tree with 2^{i+1} nodes has one more level than a tree of 2^i nodes, and so the total number of levels is $(i + 1) + 1 = \log 2^{i+1} + 1$.
- To compute the total cost, we simply add up the costs of all the levels: There are $\log n + 1$ levels, each costing cn, for a total cost of $cn(\log n + 1) = cn \log n + cn$. Ignoring the low-order term and the constant c gives the desired result of $\Theta(n \log n)$.