Mergesort

Richard Hua

Semester 2, 2021

Introduction

Mergesort exploits a recursive **divide-and-conquer** approach resulting in a worst-case running time of $\Theta(n \log n)$.
Basic idea:

- **Divide** a big problem into smaller subproblems.
- **Conquer** each subproblem.
- **Combine** the subproblem solutions.

Far better than what we have seen so far.

Question

Can we do better?

Continued

John von Neumann (1903 - 1957), ‘The Father of Modern Computers’, a founding figure in computing.

Invented by John von Neumann, in 1945, in his report on EDVAC (successor to the ENIAC).

Where the idea of storing both program instructions and data on the same medium was first proposed.

The first general purpose sorting algorithm with $O(n \log n)$ time complexity.

Some indication that the algorithm was considered to be classified as ‘TOP SECRET’ at the time of its invention.

Some rumour says John von Neumann was a card-addict, and he came up with the idea of Mergesort when playing cards.

Used by some modern programming language as the built-in sort function.

Mergesort

Mergesort is based on the following basic idea.

- If the size of the list is 0 or 1, return.
- Otherwise, separate the list into two lists of equal or nearly equal size and recursively sort the first and second halves separately.
- Finally, merge the two sorted halves into one sorted list.

Pseudo-code

Require: $0 \leqslant i \leqslant j \leqslant n-1$

function MERGESORT(list $a[0 \cdots n-1]$, integer i, integer j)

  if $i < j$ **then**

    $m \leftarrow \lfloor (i+j)/2 \rfloor$

    $l \leftarrow$ MERGESORT(a, i, m)

    $r \leftarrow$ MERGESORT(a,m+1,j)

    $a \leftarrow$ MERGE(l, r)

  return a

Clearly, all work is done in the MERGE step.
MERGE has to be as efficient as possible.

Efficient MERGE

Require: $0 \leqslant i \leqslant j \leqslant n-1$

function MERGESORT(list $a[0 \cdots n-1]$, integer i, integer j)

  if $i < j$ **then**

    $m \leftarrow \lfloor (i+j)/2 \rfloor$

    $l \leftarrow$ MERGESORT(a, i, m)

    $r \leftarrow$ MERGESORT(a,m+1,j)

    $a \leftarrow$ MERGE(l, r)

  return a

Question

What is the time complexity of MERGE when l and r are two lists of size $n/2$?

Any approximations on the upper- or lower-bounds?

Efficient MERGE

Require: $0 \leqslant i \leqslant j \leqslant n-1$

function MERGESORT(list $a[0 \cdots n-1]$, integer i, integer j)

  if $i < j$ **then**

    $m \leftarrow \lfloor (i+j)/2 \rfloor$

    $l \leftarrow$ MERGESORT(a, i, m)

    $r \leftarrow$ MERGESORT(a,m+1,j)

    $a \leftarrow$ MERGE(l, r)

  return a

MERGE has to be at least linear, i.e. $\Omega(n)$, in the worst case. Since each element needs to be checked once to determine the correct ordering.

Efficient MERGE

function MERGE(list $l[0 \cdots n_l - 1]$, list $r[0 \cdots n_r - 1]$)

  $i, j, k \leftarrow 0$

  $t \leftarrow \text{array}[0 \cdots n_l + n_r - 1]$

  while $i < n_l$ and $j < n_r$ **do**

    if $l[i] \leqslant r[j]$ **then** $t[k] \leftarrow l[i];\ k \leftarrow k + 1;\ i \leftarrow i + 1$

    else $t[k] \leftarrow r[j];\ k \leftarrow k + 1;\ j \leftarrow j + 1$

  while $i < n_l$ **do**

    $t[k] \leftarrow a[i];\ k \leftarrow k + 1;\ i \leftarrow i + 1$

  while $j < n_r$ **do**

    $t[k] \leftarrow a[j];\ k \leftarrow k + 1;\ j \leftarrow j + 1$

  return t

MERGE example

Run MERGE on the following two lists.

2	8	25	70	91

15	20	31	50	65

Linear-time MERGE

Theorem

Two input sorted lists A and B of size n_A and n_B, respectively, can be merged into an output sorted list C of size $n_C = n_A + n_B$ in linear time.

Linear-time MERGE

Proof.

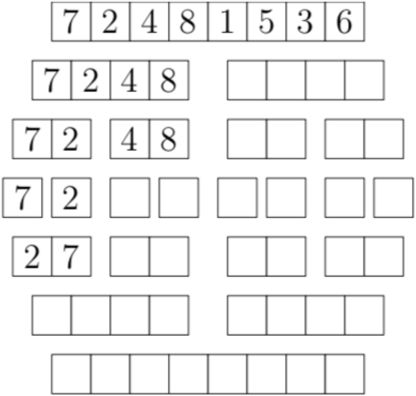
We first show that the number of comparisons needed is linear in n. Let i, j and k be the pointers to current positions in the list A, B and C, respectively. Initially, the pointers are at the first positions, i.e. $i = j = k = 0$. Each time the smaller of the two elements $A[i]$ and $B[j]$ is copied to the current entry $C[k]$ and the corresponding pointers k and either i or j are incremented by 1.
After one of the input lists is exhausted, the rest of the other list is directly copied to list C. Each comparison advances the pointer k by 1 so the max number of comparisons is $n_A + n_B - 1$.
We also make $n_A + n_B$ data movements.
Overall time is $\Theta(n_A + n_B)$. □

MERGESORT in action

https://www.youtube.com/watch?v=XaqR3G_NVoo&list=UUIqiLefbVHsOAXDAxQJH7Xw&index=7

Exercise

Run MERGESORT on the following input.



Time complexity

Question

What is the time complexity of MERGESORT?

Let's derive a recurrence relation first.
A precise recurrence relation would be $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$. $T(1) = 1$ or $T(1) = 0$ doesn't make much difference.
If we use $T(n) = 2T(n/2) + cn$ to approximate the precise formula, then we get $\Theta(n \log n)$.
The precise formula is also $\Theta(n \log n)$. See textbook for a more detailed proof.

Best/Worst case

MERGESORT is very insensitive to input.
Best/worst average case are all $\Theta(n \log n)$.

Example

Show that MERGESORT does the exact same number of operations on already sorted input list and reversed list.

Best/Worst case

Question

What does the worst-case input look like?

Maximum number of comparisons at every level.

Exercise

Show that $5, 1, 7, 3, 6, 2, 8, 4$ *is a worst-case input.*

In-place

Question

Is MERGESORT in-place?

In-place

function MERGE(list $l[0 \cdots n_l - 1]$, list $r[0 \cdots n_r - 1]$)
  $i, j, k \leftarrow 0$
  $t \leftarrow \text{array}[0 \cdots n_l + n_r - 1]$
  while $i < n_l$ and $j < n_r$ **do**
    if $l[i] \leqslant r[j]$ **then** $t[k] \leftarrow l[i]$; $k \leftarrow k + 1$; $i \leftarrow i + 1$
    else $t[k] \leftarrow r[j]$; $k \leftarrow k + 1$; $j \leftarrow j + 1$
  while $i < n_l$ **do**
    $t[k] \leftarrow a[i]$; $k \leftarrow k + 1$; $i \leftarrow i + 1$
  while $j < n_r$ **do**
    $t[k] \leftarrow a[j]$; $k \leftarrow k + 1$; $j \leftarrow j + 1$
  return t

Not in-place if we use an array-based implementation.

Stable

Question

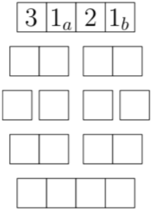
Is MERGESORT stable?

Obviously nothing gets moved around during the 'split' phase. What about MERGE?

Stable

function MERGE(list $l[0 \cdots n_l - 1]$, list $r[0 \cdots n_r - 1]$)
  $i, j, k \leftarrow 0$
  $t \leftarrow \text{array}[0 \cdots n_l + n_r - 1]$
  while $i < n_l$ and $j < n_r$ **do**
    if $l[i] \leqslant r[j]$ **then** $t[k] \leftarrow l[i]$; $k \leftarrow k + 1$; $i \leftarrow i + 1$
    else $t[k] \leftarrow r[j]$; $k \leftarrow k + 1$; $j \leftarrow j + 1$
  while $i < n_l$ **do**
    $t[k] \leftarrow a[i]$; $k \leftarrow k + 1$; $i \leftarrow i + 1$
  while $j < n_r$ **do**
    $t[k] \leftarrow a[j]$; $k \leftarrow k + 1$; $j \leftarrow j + 1$
  return t

Stable example



Additional notes

What we have seen is the standard MERGESORT implementation.
Often called **straight mergesort**.
Often implemented slightly differently in practice.
For example, when n is quite small (e.g. $n < 10$), sorting the list recursively may take more time than using, say, insertion sort.

Exercise

If we implement mergesort using linked-list. How does it affect the performance of the algorithm? Is it in-place, stable?