Discrete Mathematics and its application (CS147)

Lecture 5: Merge Sort

Fanghui Liu

Department of Computer Science, University of Warwick, UK

Target

- Recall that Bubble-sort runs in $\Theta(n^2)$ time.

How to do it efficiently?

Target

- Recall that Bubble-sort runs in $\Theta(n^2)$ time.

How to do it efficiently?

- Divide and conquer algorithm
- How to use recurrence relations to analyse runtimes of algorithms
- Runtime of Merge-sort is $\Theta(n \log n)$

Divide and conquer

- Basic algorithm design paradigm
- consists of 3 steps:
  - Divide: Divide the given problem into smaller subproblems
  - Conquer: Recursively solve each subproblem
  - Combine: Combine the solutions of these subproblems to get a solution for the original problem

Key idea in Merge-sort[1]

- divide: break the array into two parts
- recursive calls: recursively call Merge-sort to sort the two halves of the array
- merge: after the recursive call, the sub-problems are sorted, and then we merge them.

[1] Check the implementation details https://www.geeksforgeeks.org/merge-sort/ if you're interested in.

Key idea in Merge-sort[1]

- divide: break the array into two parts
- recursive calls: recursively call Merge-sort to sort the two halves of the array
- merge: after the recursive call, the sub-problems are sorted, and then we merge them.

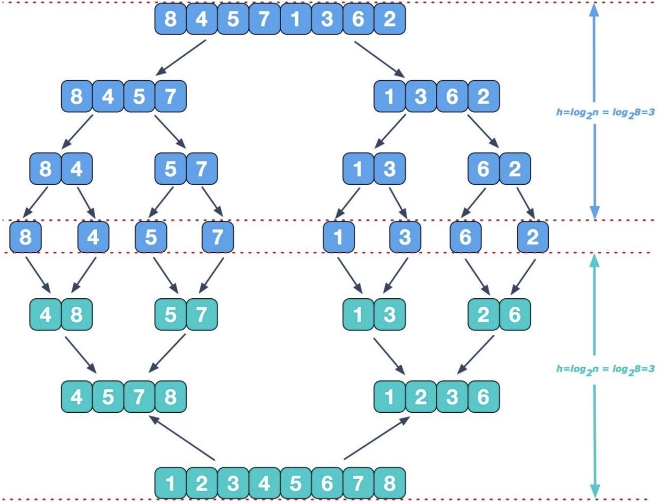
Algorithm 1: Merge-sort (Pseudo-code)

Input: An array $A[1, 2, \ldots, n]$
Output: An sorted array $D[1, 2, \ldots, n]$
`Divide:` split $A[]$ into two parts $B[]$ and $C[]$;
`Recursive calls:` $B[] =$ Merge-sort$(B[])$, $C[] =$ Merge-sort$(C[])$;
Return Merge$(B[], C[])$;

---

[1] Check the implementation details https://www.geeksforgeeks.org/merge-sort/ if you're interested in.

Illustration of Merge-sort



Runtime analysis of Merge-sort

Algorithm 2: MERGE-SORT

Input: An array $A[1, 2, \ldots, n]$
Output: An sorted array $D[1, 2, \ldots, n]$

$\texttt{MERGE-SORT}(A[1, \ldots, \lfloor n/2 \rfloor])$;
$\texttt{MERGE-SORT}(A[\lfloor n/2 \rfloor + 1, \ldots, n])$;
$D[1, \ldots, n] \leftarrow$ Merge $(A[1, \ldots, \lfloor n/2 \rfloor], A[\lfloor n/2 \rfloor + 1, \ldots, n])$;

Runtime analysis of Merge-sort

Algorithm 2: MERGE-SORT

Input: An array $A[1, 2, \ldots, n]$
Output: An sorted array $D[1, 2, \ldots, n]$

```
MERGE-SORT(A[1, ..., ⌊n/2⌋]) #runtime = T(⌊n/2⌋);
MERGE-SORT(A[⌊n/2⌋ + 1, ..., n]) ;
D[1, ..., n] ← Merge (A[1, ..., ⌊n/2⌋], A[⌊n/2⌋ + 1, ..., n]);
```

Runtime analysis of Merge-sort

Algorithm 2: MERGE-SORT

Input: An array $A[1, 2, \ldots, n]$

Output: An sorted array $D[1, 2, \ldots, n]$

$\texttt{MERGE-SORT}(A[1, \ldots, \lfloor n/2 \rfloor])$ #runtime $= T(\lfloor n/2 \rfloor)$;

$\texttt{MERGE-SORT}(A[\lfloor n/2 \rfloor + 1, \ldots, n])$ #runtime $= T(n - \lfloor n/2 \rfloor)$;

$D[1, \ldots, n] \leftarrow$ Merge $(A[1, \ldots, \lfloor n/2 \rfloor], A[\lfloor n/2 \rfloor + 1, \ldots, n])$;

Runtime analysis of Merge-sort

Algorithm 2: MERGE-SORT

Input: An array $A[1, 2, \ldots, n]$
Output: An sorted array $D[1, 2, \ldots, n]$

$\texttt{MERGE-SORT}(A[1, \ldots, \lfloor n/2 \rfloor])$ #runtime $= T(\lfloor n/2 \rfloor)$;
$\texttt{MERGE-SORT}(A[\lfloor n/2 \rfloor + 1, \ldots, n])$ #runtime $= T(n - \lfloor n/2 \rfloor)$;
$D[1, \ldots, n] \leftarrow$ Merge $(A[1, \ldots, \lfloor n/2 \rfloor], A[\lfloor n/2 \rfloor + 1, \ldots, n])$#runtime $= \Theta(n)$;

Runtime analysis of Merge-sort

Algorithm 2: MERGE-SORT

Input: An array $A[1, 2, \ldots, n]$
Output: An sorted array $D[1, 2, \ldots, n]$

```
MERGE-SORT(A[1, ..., ⌊n/2⌋]) #runtime = T(⌊n/2⌋);
MERGE-SORT(A[⌊n/2⌋ + 1, ..., n]) #runtime = T(n − ⌊n/2⌋);
D[1, ..., n] ← Merge (A[1, ..., ⌊n/2⌋], A[⌊n/2⌋ + 1, ..., n])#runtime = Θ(n);
```

Theorem

`Merge` *takes* $\Theta(n)$ *time.*

Runtime analysis of Merge-sort

Algorithm 2: MERGE-SORT

Input: An array $A[1, 2, \ldots, n]$
Output: An sorted array $D[1, 2, \ldots, n]$

$\texttt{MERGE-SORT}(A[1, \ldots, \lfloor n/2 \rfloor])$ #runtime $= T(\lfloor n/2 \rfloor)$;
$\texttt{MERGE-SORT}(A[\lfloor n/2 \rfloor + 1, \ldots, n])$ #runtime $= T(n - \lfloor n/2 \rfloor)$;
$D[1, \ldots, n] \leftarrow$ Merge $(A[1, \ldots, \lfloor n/2 \rfloor], A[\lfloor n/2 \rfloor + 1, \ldots, n])$#runtime $= \Theta(n)$;

Theorem

$\texttt{Merge}$ *takes* $\Theta(n)$ *time.*

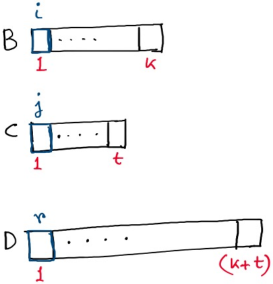
Recurrence relation: $T(n) = T(\lfloor n/2 \rfloor) + T(n - \lfloor n/2 \rfloor) + \Theta(n)$

Merge subroutine

- Input: sorted arrays $B[1,2,\cdots,k]$ and $C[1,2,\cdots,t]$
- Output: Array $D[1,2,\cdots,k+t]$ that contains the enties B and C in an increasing order.

Example

Input: B: [3, 7, 9, 10] ; C: [1,4,5]
Output: D: [1,3,4,5,7,9,10]



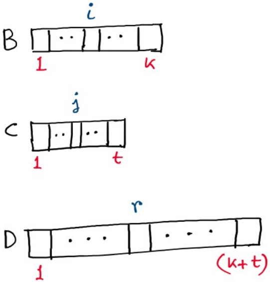
Case $(B(i) \leq C(j))$

$D(r) \leftarrow B(i)$
$r \leftarrow r+1$
$i \leftarrow i+1$

Case $(B(i) > C(j))$

$D(r) \leftarrow C(j)$
$r \leftarrow r+1$
$j \leftarrow j+1$

Merge



Case $(i > k)$

$D(r) \leftarrow C(j)$
$r \leftarrow r + 1$
$j \leftarrow j + 1$

Case $(j > t)$

$D(r) \leftarrow B(i)$
$r \leftarrow r + 1$
$i \leftarrow i + 1$

Persudocode for Merge

Algorithm 3: MERGE

Input: sorted arrays $B[1, 2, \cdots, k]$ and $C[1, 2, \cdots, t]$

Output: An sorted array $D[1, 2, \cdots, k+t]$ that contains the entries B and C.

`Initialization:` $i = 1$ and $j = 1$;
for $r = 1$ *to* $(k + t)$ **do**
    if $i \leq k$ *and* $j \leq t$ **then**
        if $B(i) \leq C(j)$ **then**
            $D(r) \leftarrow B(i)$, $i \leftarrow i + 1$;
        end
        else
            $D(r) \leftarrow C(j)$, $j \leftarrow j + 1$;
        end
    end
    else if $i > k$, **then** $D[r] \leftarrow C[j]$, $j \leftarrow j + 1$;
    else if $j > t$, **then** $D[r] \leftarrow B[i]$, $i \leftarrow i + 1$;
end

Runtime analysis of Merge-sort

Recurrence relation:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(n - \lfloor n/2 \rfloor) + \Theta(n) & \text{otherwise.} \end{cases}$$

Ignore floors and ceilings (and the base case)

Statement (concise form)

$$T(n) = 2T(n/2) + \Theta(n)\,.$$

Next lecture: Will show $T(n) = \Theta(n \log n)$.

*Guess about time complexity

Statement

We guess that $T(n) := 2T(n/2) + \Theta(n)$ *has the time complexity of* $\mathcal{O}(n \log n)$ *if* $n > 1$.

*Guess about time complexity

Statement

We guess that $T(n) := 2T(n/2) + \Theta(n)$ *has the time complexity of* $\mathcal{O}(n \log n)$ *if* $n > 1$.

Proof by induction.

- base case: $T(2) \leq 2c \log 2 = 2c$ for some constant c.

*Guess about time complexity

Statement

We guess that $T(n) := 2T(n/2) + \Theta(n)$ *has the time complexity of* $\mathcal{O}(n \log n)$ *if* $n > 1$.

Proof by induction.

- base case: $T(2) \le 2c \log 2 = 2c$ for some constant c.
- Assume $T(n-1) \le c(n-1)\log(n-1)$, then

*Guess about time complexity

Statement

We guess that $T(n) := 2T(n/2) + \Theta(n)$ *has the time complexity of* $\mathcal{O}(n \log n)$ *if* $n > 1$.

Proof by induction.

- base case: $T(2) \leq 2c \log 2 = 2c$ for some constant c.
- Assume $T(n-1) \leq c(n-1)\log(n-1)$, then

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \leq 2c(n/2)\log(n/2) + cn \\ &= cn(\log n - \log 2) + cn \\ &= cn \log n\,. \end{aligned}$$

$\square$

*Guess about time complexity

Statement

We guess that $T(n) := 2T(n/2) + \Theta(n)$ *has the time complexity of* $\mathcal{O}(n \log n)$ *if* $n > 1$.

Proof by induction.

- base case: $T(2) \leq 2c \log 2 = 2c$ for some constant c.
- Assume $T(n-1) \leq c(n-1)\log(n-1)$, then

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \leq 2c(n/2)\log(n/2) + cn \\ &= cn(\log n - \log 2) + cn \\ &= cn \log n \,. \end{aligned}$$

□

- the best/average/worst time complexity is $\Theta(n \log n)$
- space complexity is $\Theta(n)$

*Insertion sort

Algorithm 4: Insertion-sort (Pseudo-code)

Input: An array $A[1, 2, \ldots, n]$

Output: An sorted array $A[1, 2, \ldots, n]$

```
for j = 2 : n do
    insert A[j] into (sorted) A[1, 2, ··· , j − 1];
end
```