COMP20007 Design of Algorithms

Sorting - Part 2

Daniel Beck

Lecture 12

Semester 1, 2020

Mergesort

function $\textsc{Mergesort}(A[0..n-1])$

  if $n > 1$ **then**

SAME ARRAY

    $B[0..\lfloor n/2 \rfloor - 1] \leftarrow A[0..\lfloor n/2 \rfloor - 1]$

    $C[0..\lfloor n/2 \rfloor - 1] \leftarrow A[\lfloor n/2 \rfloor ..n-1]$

    $\textsc{Mergesort}(B[0..\lfloor n/2 \rfloor - 1])$

    $\textsc{Mergesort}(C[0..\lfloor n/2 \rfloor - 1])$

    $\textsc{Merge}(B, C, A)$

Mergesort - Merge function

function $\textsc{Merge}(B[0..p-1], C[0..q-1], A[0..p+q-1])$

$\quad i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$

$\quad$ **while** $i < p$ **and** $j < q$ **do**

$\quad\quad$ **if** $\boxed{B[i] \leq C[j]}$ **then**

$\quad\quad\quad A[k] \leftarrow B[i]; i \leftarrow i + 1$

$\quad\quad$ **else**

$\quad\quad\quad A[k] \leftarrow C[j]; j \leftarrow j + 1$

$\quad\quad k \leftarrow k + 1$

$\quad$ **if** $i == p$ **then**

$\quad\quad A[k..p+q-1] \leftarrow C[j..q-1]$

$\quad$ **else**

$\quad\quad A[k..p+q-1] \leftarrow B[i..p-1]$

Mergesort - Properties

Divide-and-Conquer algorithm

Mergesort - Properties

Divide-and-Conquer algorithm

- In contrast with decrease-and-conquer algorithms, such as Insertion Sort.

Mergesort - Properties

Divide-and-Conquer algorithm

- In contrast with decrease-and-conquer algorithms, such as Insertion Sort.

Questions!

Mergesort - Properties

Divide-and-Conquer algorithm

- In contrast with decrease-and-conquer algorithms, such as Insertion Sort.

Questions!

- In-place? (or does it require extra memory?)
- Stable?

Mergesort - Properties

- In-place?

Mergesort - Properties

ITERATIVE

- In-place? **No.** (requires $O(n)$ auxiliary array + $\cancel{O(\log n)}$ stack space for recursion)
- Stable?

Mergesort - Properties

344

35

44 → SORTED

- In-place? **No.** (requires $O(n)$ auxiliary array + $O(\log n)$ stack space for recursion)
- Stable? **Yes!** (Merge keeps relative order with additional bookkeeping)

$B[i] \boxed{\leq} C[j]$

34 45

3445

Mergesort - Complexity

- Worst case?
- Best case?
- Average case?

Mergesort

function $\textsc{Mergesort}(A[0..n-1])$ $= C(n)$

if $n > 1$ **then**

$B[0..\lfloor n/2\rfloor - 1] \leftarrow A[0..\lfloor n/2\rfloor - 1]$

$C[0..\lfloor n/2\rfloor - 1] \leftarrow A[\lfloor n/2\rfloor..n-1]$

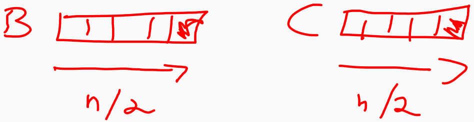
$\textsc{Mergesort}(B[0..\lfloor n/2\rfloor - 1])$

$\textsc{Mergesort}(C[0..\lfloor n/2\rfloor - 1])$

$\textsc{Merge}(B, C, A)$

$$C(n) = 2C(n/2) + \underline{C_{MERGE}(n)}$$

Mergesort - Complexity

Merge: 

$= n-1$

$C^{\text{merge}}_{\text{worst}} = n-1$

$$C(n) = 2C(n/2) + \overset{\Theta(n)}{\boxed{n-1}}$$

$$T(n) = aT(n/b) + F(n)$$

$a = 2 \quad d = 1$

$b = 2$

$$2 = 2^1 \Rightarrow a = b^d \Rightarrow \Theta(n^d \log n)$$

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

$$C^{\text{merge}}_{\text{best}} = \frac{n}{2} - 1 = \Theta(n)$$

$$\hookrightarrow \boxed{\Theta(n \log n)}$$

Mergesort - In Practice

DATA CENTER

HARD DRIVE

- Guaranteed $\boxed{\Theta(n \log n)}$ complexity
- Highly parallelisable
- Multiway Mergesort: excellent for secondary memory
- Used in JavaScript (Mozilla)
- Basis for hybrid algorithms (TimSort: Python, Android)

Take-home message: Mergesort is an excellent choice if **stability** is required and the extra memory cost is low.

O(n)

Quicksort

function $\textsc{Quicksort}\boxed{(A[l..r])}$ $\triangleright$ Starts with $A[0..n-1]$

  if $l < r$ **then**

PIVOT → $s \leftarrow \textsc{Partition}(A[l..r])$

    $\textsc{Quicksort}(A[l..s-1])$

    $\textsc{Quicksort}(A[s+1..r])$

Quicksort - Lomuto partitioning

function $\text{LomutoPartition}(A[l..r])$

  $p \leftarrow A[l]$

  $s \leftarrow l$

  for $i \leftarrow l + 1$ **to** r **do**

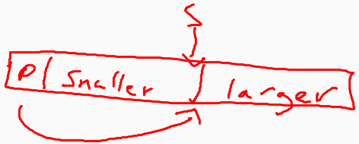
    if $A[i] < p$ **then**

      $s \leftarrow s + 1$

      $\text{Swap}(A[s], A[i])$

  $\text{Swap}(A[l], A[s])$

  return s



Quicksort - Properties

Divide-and-Conquer algorithm

Quicksort - Properties

Divide-and-Conquer algorithm

Questions!

Quicksort - Properties

Divide-and-Conquer algorithm

Questions!

- In-place?
- Stable?

Quicksort - Properties

- In-place?

Quicksort - Properties

- In-place? **Yes**, buy still requires $O(\log n)$ memory for the stack.

Quicksort - Properties

- In-place? **Yes**, buy still requires $O(\log n)$ memory for the stack.
- Stable?

Quicksort - Properties

- In-place? **Yes**, buy still requires $O(\log n)$ memory for the stack.
- Stable? **No** (non-local swaps)

Quicksort - Partitioning

- Lomuto partitioning can be used but not the best in pratice.

Quicksort - Partitioning

- Lomuto partitioning can be used but not the best in pratice.
- Instead, practical implementations use Hoare partitioning (proposed by the inventor of Quicksort).

Quicksort - Partitioning

- Lomuto partitioning can be used but not the best in pratice.
- Instead, practical implementations use Hoare partitioning (proposed by the inventor of Quicksort).
- How does it work? Let's go back to my games first...

Quicksort - Hoare partitioning

function $\textsc{HoarePartition}(A[l..r])$

$p \leftarrow A[l]$

$i \leftarrow l; j \leftarrow r + 1$

repeat

repeat $i \leftarrow i + 1$ **until** $A[i] \geq p$

repeat $j \leftarrow j - 1$ **until** $A[j] \leq p$

$\textsc{Swap}(A[i], A[j])$

until $i \geq j$ → CROSS

$\textsc{Swap}(A[i], A[j])$ → UNDO SWAP

$\textsc{Swap}(\boxed{A[l]}, A[j])$ → PIVOT INTO FINAL POSITION

return j

Quicksort - Complexity

- Worst case?
- Best case?
- Average case?

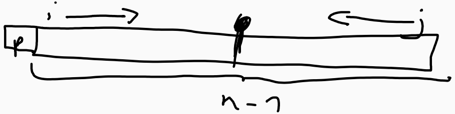
Quicksort - Complexity

- Worst case?
- Best case?
- Average case?

Warning: not trivial, but give it a go. ;)

Quicksort - Complexity

PARTITION



$n-1+1+1 = n+1$

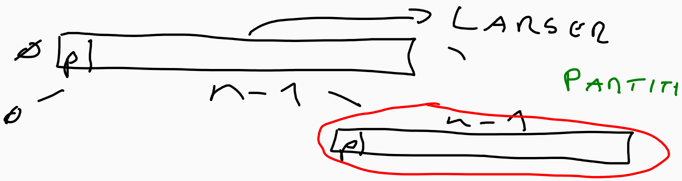
$\boxed{\Theta(n)}$

QUICKSORT

$$C_{BEST} = 2\,C_{BEST}(n/2) + \Theta n = \Theta(n \log n$$

Quicksort - Complexity

QUICKSORT – WORST



PARTITION = $n+1$

$C_{WORST} = (n+1) + n + (n-1) + \dots + 3 = \frac{(n+1)(n+2)}{2} - 3$

$\in \Theta(n^2)$

$C_{BEST} \in \Theta(n \log n)$

C_{AVG}?

S → ~~PIVOT~~ SIZE

$\boxed{\frac{1}{n} \sum_{s=0}^{n-1} n+1 + C_{AVG}(s) + C_{AVG}(n-1-s)} \Rightarrow \Theta(n \log n)$

Quicksort - In practice

- Used in C (qsort)
- Basis for C++ sort (Introsort)
- **Fastest** sorting algorithm in most cases

Quicksort - In practice

- Used in C (qsort)
- Basis for C++ sort (Introsort)
- **Fastest** sorting algorithm in most cases

Take-home message: Quicksort is the algorithm of choice when **speed** matters and stability is not required.

Summary so far

Selection Sort: Slow, but only $O(n)$ key exchanges.

Summary so far

Selection Sort: Slow, but only $O(n)$ key exchanges.

Insertion Sort: Very good for small arrays and when data is expected to be "almost sorted".

Summary so far

Selection Sort: Slow, but only $O(n)$ key exchanges.

Insertion Sort: Very good for small arrays and when data is expected to be "almost sorted".

Mergesort: Better for mid-size arrays and when stability is required.

Summary so far

Selection Sort: Slow, but only $O(n)$ key exchanges.

Insertion Sort: Very good for small arrays and when data is expected to be "almost sorted".

Mergesort: Better for mid-size arrays and when stability is required.

Quicksort: Usually the best choice for large arrays, with excellent **empirical** performance and only $O(\log n)$ memory cost.

Summary so far

Selection Sort: Slow, but only $O(n)$ key exchanges.

Insertion Sort: Very good for small arrays and when data is expected to be "almost sorted".

Mergesort: Better for mid-size arrays and when stability is required.

Quicksort: Usually the best choice for large arrays, with excellent **empirical** performance and only $O(\log n)$ memory cost.

Next lecture: Heapsort