3 Efficient Sorting – The Power of Divide & Conquer

17 October 2022

Sebastian Wild

COMP526 (Fall 2022)
University of Liverpool
version 2022-10-11 21:57

Learning Outcomes

1. Know principles and implementation of *mergesort* and *quicksort*.
2. Know properties and *performance characteristics* of mergesort and quicksort.
3. Know the comparison model and understand the corresponding *lower bound*.
4. Understand *counting sort* and how it circumvents the comparison lower bound.
5. Know ways how to exploit *presorted* inputs.

Unit 3: ***Efficient Sorting***

Outline

3 Efficient Sorting

Why study sorting?

- fundamental problem of computer science that is still not solved
  - Algorithm with optimal #comparisons in worst case?
- building brick of many more advanced algorithms
  - for preprocessing
  - as subroutine
- playground of manageable complexity
  to practice algorithmic techniques

Here:

- "classic" fast sorting method
- exploit **partially sorted** inputs
- ~~**parallel sorting**~~ → Unit 5

Part I

The Basics

Rules of the game

- **Given:**
  - array $A[0..n) = A[0..n-1]$ of n objects
  - a total order relation $\leq$ among $A[0], \ldots, A[n-1]$
    (a comparison function)
    Python: elements support `<=` operator (`__le__()`)
    Java: `Comparable` class (`x.compareTo(y) <= 0`)
- **Goal:** rearrange (i. e., permute) elements within A,
  so that A is *sorted*, i. e., $A[0] \leq A[1] \leq \cdots \leq A[n-1]$

- for now: A stored in main memory (*internal sorting*)
  single processor (*sequential sorting*)

Clicker Question

What is the complexity of sorting? Type you answer, e. g., as "`Theta(sqrt(n))`"

→ *sli.do/comp526*

3.1 Mergesort

Clicker Question

How does mergesort work?

- **A** Split elements around median, then recurse on small / large elements.
- **B** Recurse on left / right half, then combine sorted halves.
- **C** Grow sorted part on left, repeatedly add next element to sorted range.
- **D** Repeatedly choose 2 elements and swap them if they are out of order.
- **E** Don't know.

→ *sli.do/comp526*

Clicker Question

How does mergesort work?

A ~~Split elements around median, then recurse on small / large elements.~~

B Recurse on left / right half, then combine sorted halves. ✓

C ~~Grow sorted part on left, repeatedly add next element to sorted range.~~

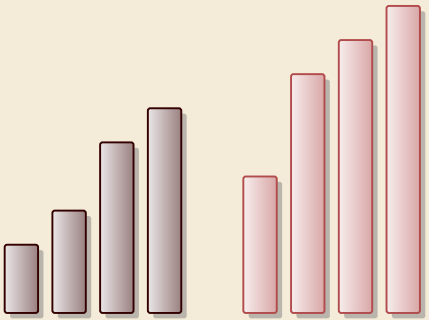
D ~~Repeatedly choose 2 elements and swap them if they are out of order.~~

E ~~Don't know.~~

→ *sli.do/comp526*

Merging sorted lists

Merging sorted lists

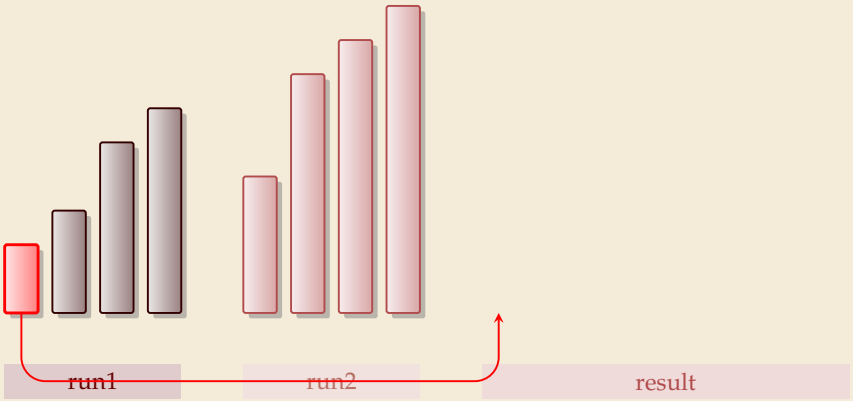


run1 run2 result

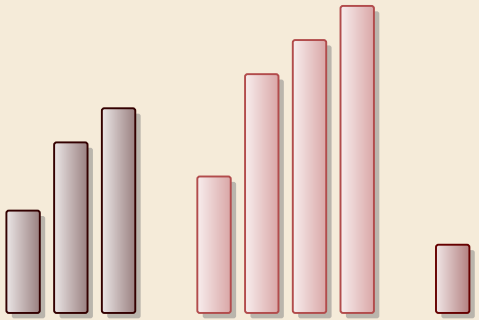
Merging sorted lists

Merging sorted lists

Merging sorted lists



Merging sorted lists

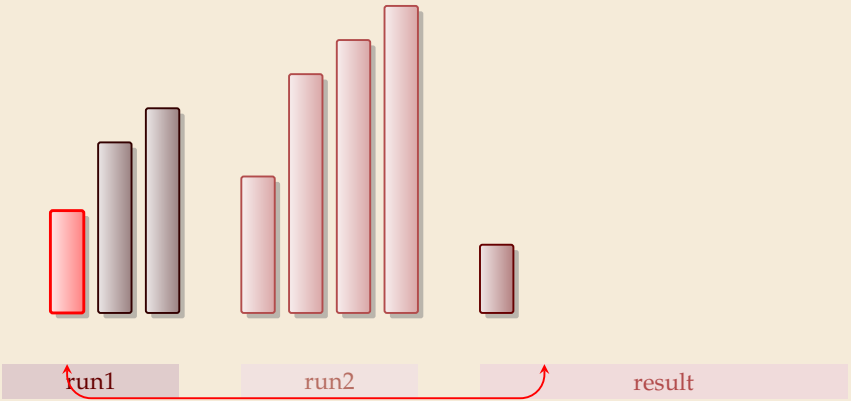


run1 run2 result

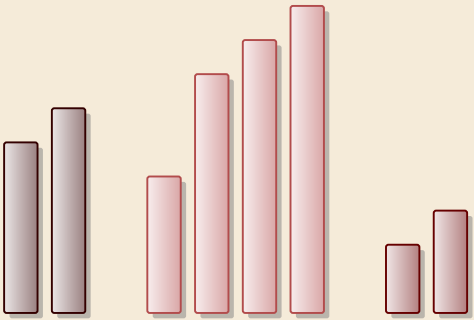
Merging sorted lists

Merging sorted lists

Merging sorted lists

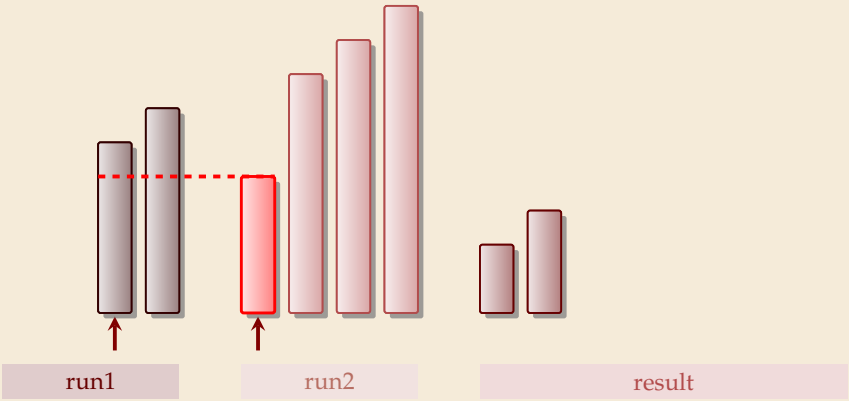


Merging sorted lists



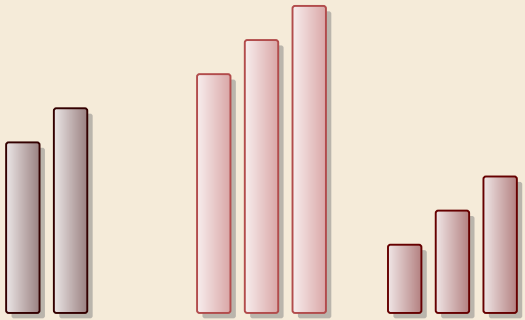
run1 run2 result

Merging sorted lists



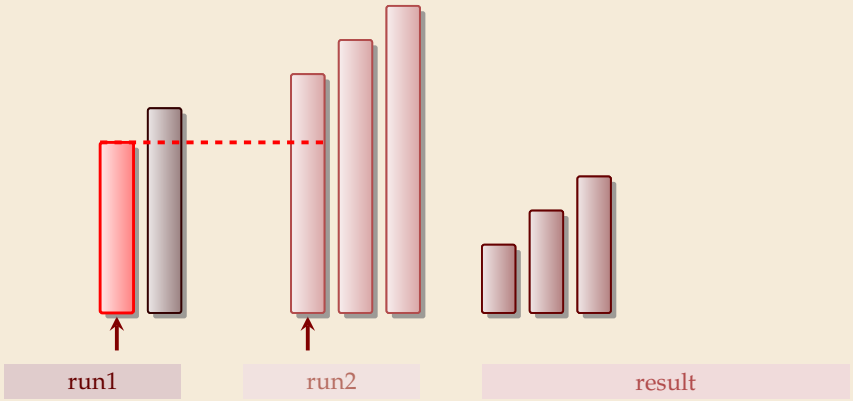
Merging sorted lists

Merging sorted lists

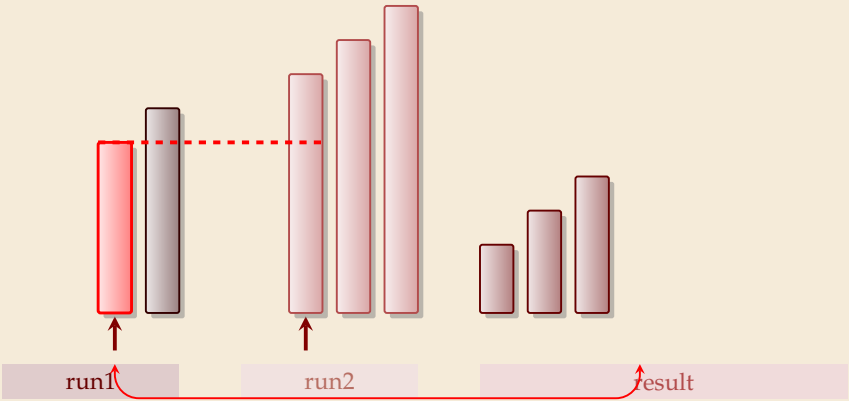


run1 run2 result

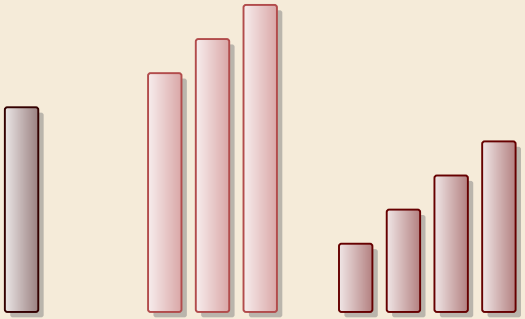
Merging sorted lists



Merging sorted lists

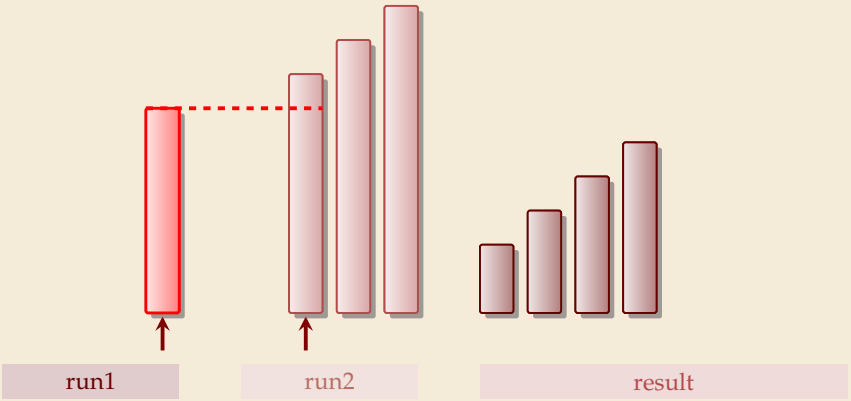


Merging sorted lists

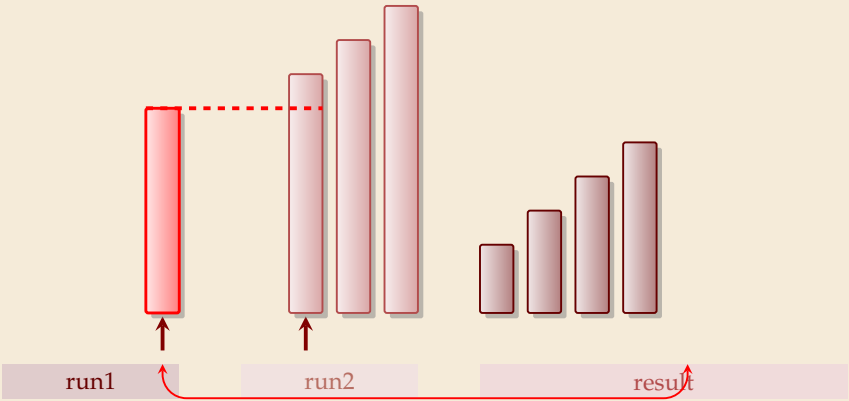


run1 run2 result

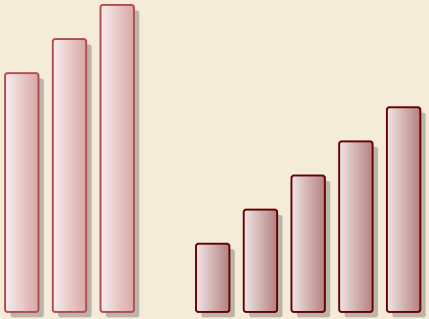
Merging sorted lists



Merging sorted lists

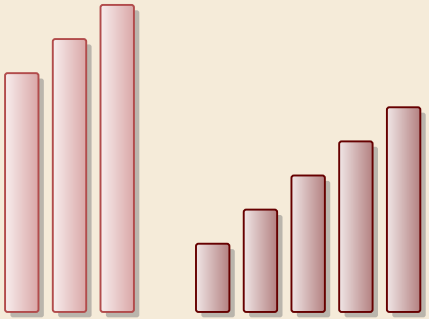


Merging sorted lists



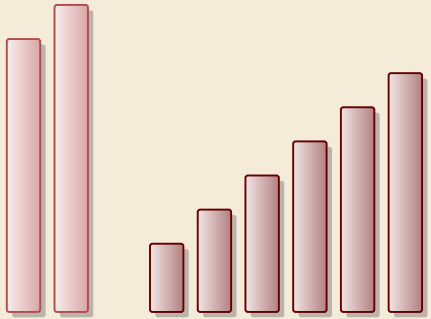
run1 run2 result

Merging sorted lists



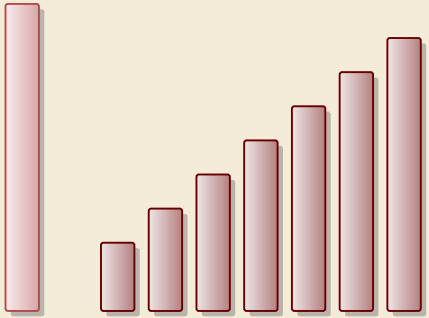
run1 run2 result

Merging sorted lists



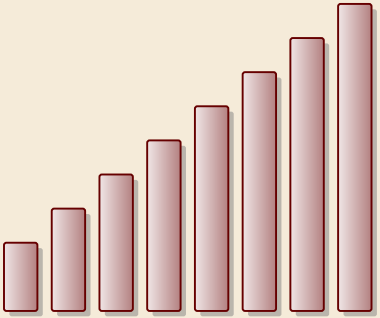
run1 run2 result

Merging sorted lists



run1 run2 result

Merging sorted lists



run1 run2 result

Clicker Question

What is the worst-case running time of mergesort?

- **A** $\Theta(1)$
- **B** $\Theta(\log n)$
- **C** $\Theta(\log \log n)$
- **D** $\Theta(\sqrt{n})$
- **E** $\Theta(n)$
- **F** $\Theta(n \log \log n)$
- **G** $\Theta(n \log n)$
- **H** $\Theta(n \log^2 n)$
- **I** $\Theta(n^{1+\epsilon})$
- **J** $\Theta(n^2)$
- **K** $\Theta(n^3)$
- **L** $\Theta(2^n)$

→ *sli.do/comp526*

Clicker Question

What is the worst-case running time of mergesort?

A ~~$\Theta(1)$~~

B ~~$\Theta(\log n)$~~

C ~~$\Theta(\log \log n)$~~

D ~~$\Theta(\sqrt{n})$~~

E ~~$\Theta(n)$~~

F ~~$\Theta(n \log \log n)$~~

G $\Theta(n \log n)$ ✓

H ~~$\Theta(n \log^2 n)$~~

I ~~$\Theta(n^{1+\epsilon})$~~

J ~~$\Theta(n^2)$~~

K ~~$\Theta(n^3)$~~

L ~~$\Theta(2^n)$~~

→ *sli.do/comp526*

Mergesort

```
1  procedure mergesort(A[l..r))
2      n := r − l
3      if n ≤ 1 return
4      m := l + ⌊n/2⌋
5      mergesort(A[l..m))
6      mergesort(A[m..r))
7      merge(A[l..m), A[m..r), buf)
8      copy buf to A[l..r)
```

proc sort (A[0..n))

mergesort (A[0..n))

- recursive procedure
- merging needs
  - temporary storage *buf* for result (of same size as merged runs)
  - to read and write each element twice (once for merging, once for copying back)

Mergesort

```
1 procedure mergesort(A[l..r))
2     n := r − l
3     if n ≤ 1 return
4     m := l + ⌊n/2⌋
5     mergesort(A[l..m))
6     mergesort(A[m..r))
7     merge(A[l..m), A[m..r), buf)
8     copy buf to A[l..r)
```

- recursive procedure
- merging needs
  - temporary storage *buf* for result (of same size as merged runs)
  - to read and write each element twice (once for merging, once for copying back)

array access

Analysis: count *"element visits"* (read and/or write)

$$C(n) = \begin{cases} 0 & n \le 1 \\ C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + 2n & n \ge 2 \end{cases}$$

same for best and worst case!

Simplification $n = 2^k$

$$C(2^k) = \begin{cases} 0 & k \le 0 \\ 2 \cdot C(2^{k-1}) + 2 \cdot 2^k & k \ge 1 \end{cases} = 2 \cdot 2^k + 2^2 \cdot 2^{k-1} + 2^3 \cdot 2^{k-2} + \cdots + 2^k \cdot 2^1 = 2k \cdot 2^k$$

$$C(n) = 2n \lg(n) = \Theta(n \log n)$$