Divide-and-Conquer: Sorting Problem

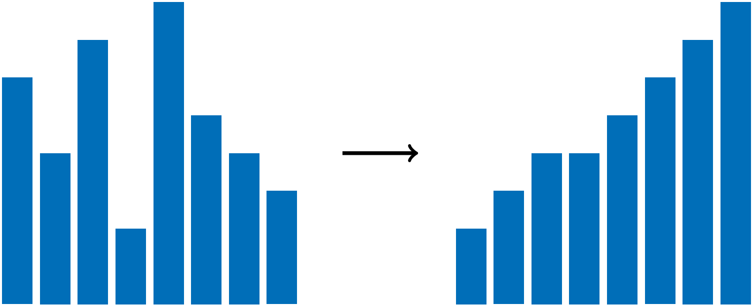
Alexander S. Kulikov

Steklov Institute of Mathematics at St. Petersburg
Russian Academy of Sciences

Algorithmic Design and Techniques
Algorithms and Data Structures at edX

Outline

Sorting Problem



Sorting

Input: Sequence $A[1 \dots n]$.

Output: Permutation $A'[1 \dots n]$ of $A[1 \dots n]$ in non-decreasing order.

Why Sorting?

- Sorting data is an important step of many efficient algorithms.

Why Sorting?

- Sorting data is an important step of many efficient algorithms.
- Sorted data allows for more efficient queries.

Outline

Selection sort: example

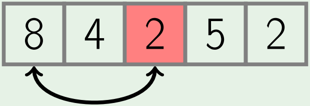
8	4	2	5	2

Selection sort: example

8	4	2	5	2

- Find a minimum by scanning the array

Selection sort: example



- Find a minimum by scanning the array
- Swap it with the first element

Selection sort: example

2	4	8	5	2

- Find a minimum by scanning the array
- Swap it with the first element

Selection sort: example

2	4	8	5	2

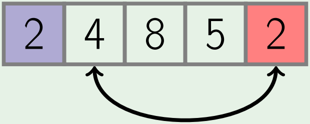
- Find a minimum by scanning the array
- Swap it with the first element
- Repeat with the remaining part of the array

Selection sort: example

2	4	8	5	2

- Find a minimum by scanning the array
- Swap it with the first element
- Repeat with the remaining part of the array

Selection sort: example



- Find a minimum by scanning the array
- Swap it with the first element
- Repeat with the remaining part of the array

Selection sort: example

2	2	8	5	4

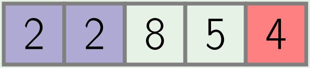
- Find a minimum by scanning the array
- Swap it with the first element
- Repeat with the remaining part of the array

Selection sort: example

2	2	8	5	4

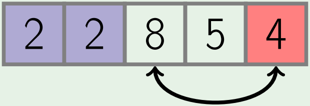
- Find a minimum by scanning the array
- Swap it with the first element
- Repeat with the remaining part of the array

Selection sort: example



- Find a minimum by scanning the array
- Swap it with the first element
- Repeat with the remaining part of the array

Selection sort: example



- Find a minimum by scanning the array
- Swap it with the first element
- Repeat with the remaining part of the array

Selection sort: example

2	2	4	5	8

- Find a minimum by scanning the array
- Swap it with the first element
- Repeat with the remaining part of the array

Selection sort: example

2	2	4	5	8

- Find a minimum by scanning the array
- Swap it with the first element
- Repeat with the remaining part of the array

Selection sort: example

2	2	4	5	8

- Find a minimum by scanning the array
- Swap it with the first element
- Repeat with the remaining part of the array

Selection sort: example

2	2	4	5	8

- Find a minimum by scanning the array
- Swap it with the first element
- Repeat with the remaining part of the array

Selection sort: example

2	2	4	5	8

- Find a minimum by scanning the array
- Swap it with the first element
- Repeat with the remaining part of the array

SelectionSort($A[1 \ldots n]$)

```
for i from 1 to n:
    minIndex ← i
    for j from i + 1 to n:
        if A[j] < A[minIndex]:
            minIndex ← j
    {A[minIndex] = min A[i ... n]}
    swap(A[i], A[minIndex])
    {A[1 ... i] is in final position}
```

SelectionSort($A[1 \ldots n]$)

```
for i from 1 to n:
    minIndex ← i
    for j from i + 1 to n:
        if A[j] < A[minIndex]:
            minIndex ← j
    {A[minIndex] = min A[i ... n]}
    swap(A[i], A[minIndex])
    {A[1 ... i] is in final position}
```

Online visualization: selection sort

Lemma

The running time of `SelectionSort`$(A[1 \ldots n])$ is $O(n^2)$.

Lemma

The running time of `SelectionSort`$(A[1 \ldots n])$ is $O(n^2)$.

Proof

n iterations of outer loop, at most n iterations of inner loop. $\square$

Too Pessimistic Estimate?

- As i grows, the number of iterations of the inner loop decreases: j iterates from $i + 1$ to n.

Too Pessimistic Estimate?

- As i grows, the number of iterations of the inner loop decreases: j iterates from $i+1$ to n.
- A more accurate estimate for the total number of iterations of the inner loop is $(n-1)+(n-2)+\cdots+1$.

Too Pessimistic Estimate?

- As i grows, the number of iterations of the inner loop decreases: j iterates from $i+1$ to n.
- A more accurate estimate for the total number of iterations of the inner loop is $(n-1)+(n-2)+\cdots+1$.
- We will show that this sum is $\Theta(n^2)$ implying that our initial estimate is actually tight.

Arithmetic Series

Lemma

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Arithmetic Series

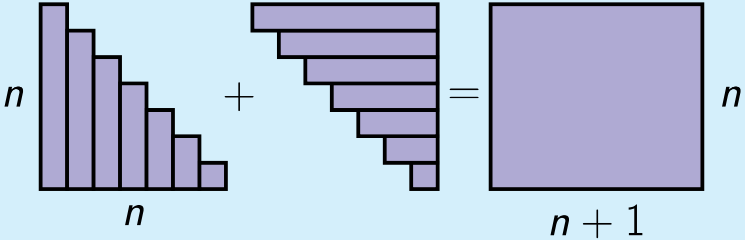
Lemma

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Proof

$$\begin{array}{cccc} 1 & 2 & \cdots & n \\ n & n-1 & \cdots & 1 \\ \hline n+1 & n+1 & \cdots & n+1 \end{array} = n(n+1) \qquad \square$$

Alternative proof



$\square$

Selection Sort: Summary

- Selection sort is an easy to implement algorithm with running time $O(n^2)$.

Selection Sort: Summary

- Selection sort is an easy to implement algorithm with running time $O(n^2)$.
- Sorts in place: requires a constant amount of extra memory.

Selection Sort: Summary

- Selection sort is an easy to implement algorithm with running time $O(n^2)$.
- Sorts in place: requires a constant amount of extra memory.
- There are many other quadratic time sorting algorithms: e.g., insertion sort, bubble sort.

Outline

1. Problem Overview
2. Selection Sort
3. Merge Sort
4. Lower Bound for Comparison Based Sorting
5. Non-Comparison Based Sorting Algorithms

Example: merge sort

7	2	5	3	7	13	1	6

Example: merge sort

7	2	5	3	7	13	1	6

split the array into two halves

7	2	5	3

7	13	1	6

Example: merge sort

7	2	5	3	7	13	1	6

split the array into two halves

7	2	5	3

7	13	1	6

sort the halves recursively

2	3	5	7

1	6	7	13

Example: merge sort

7	2	5	3	7	13	1	6

split the array into two halves

7	2	5	3

7	13	1	6

sort the halves recursively

2	3	5	7

1	6	7	13

merge the sorted halves into one array

1	2	3	5	6	7	7	13

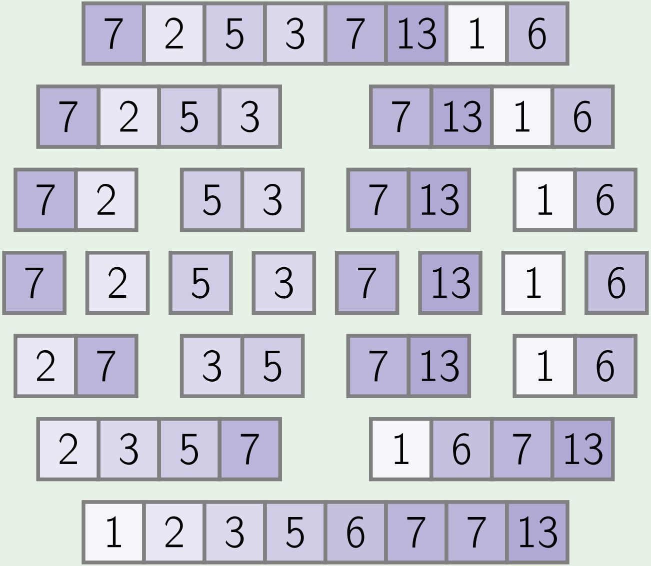
```
MergeSort(A[1 ... n])

if n = 1:
    return A
m ← ⌊n/2⌋
B ← MergeSort(A[1 ... m])
C ← MergeSort(A[m + 1 ... n])
A' ← Merge(B, C)
return A'
```

Merging Two Sorted Arrays

Merge($B[1 \ldots p], C[1 \ldots q]$)

```
{B and C are sorted}
D ← empty array of size p + q
while B and C are both non-empty:
  b ← the first element of B
  c ← the first element of C
  if b ≤ c:
    move b from B to the end of D
  else:
    move c from C to the end of D
move the rest of B and C to the end of D
return D
```

Merge sort: example



Lemma

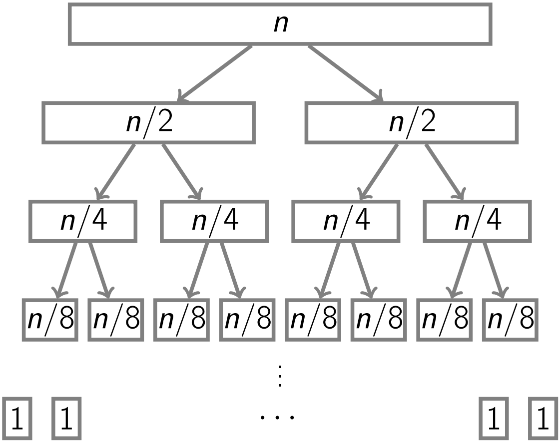
The running time of `MergeSort`$(A[1 \ldots n])$ is $O(n \log n)$.

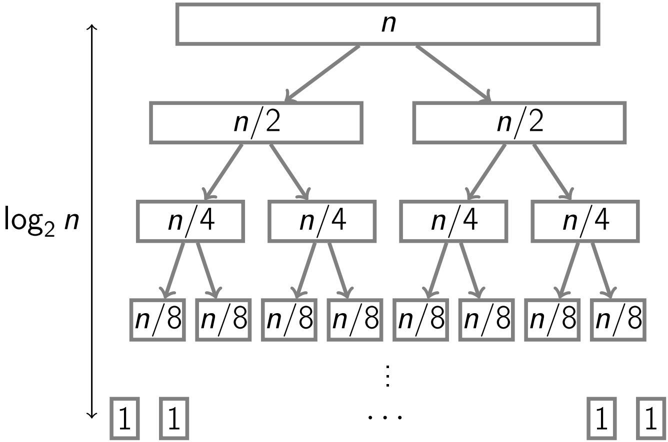
Lemma

The running time of `MergeSort`$(A[1 \ldots n])$ is $O(n \log n)$.

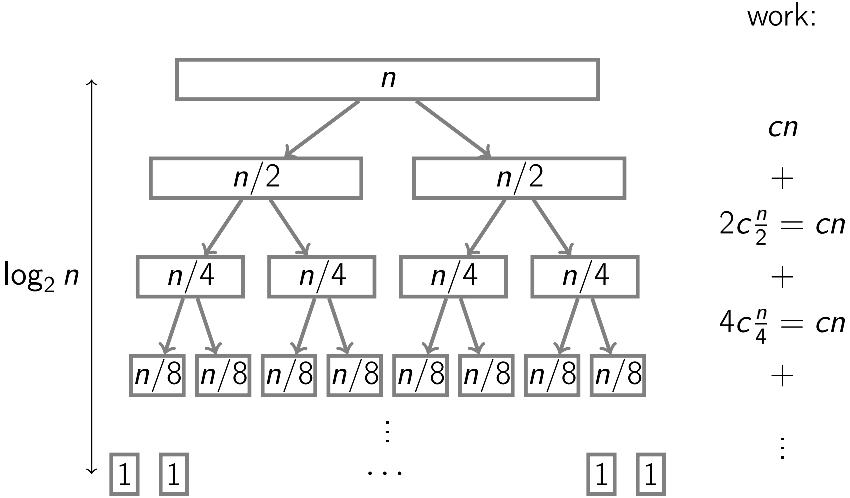
Proof

- The running time of merging B and C is $O(n)$.
- Hence the running time of `MergeSort`$(A[1 \ldots n])$ satisfies a recurrence $T(n) \leq 2T(n/2) + O(n)$.

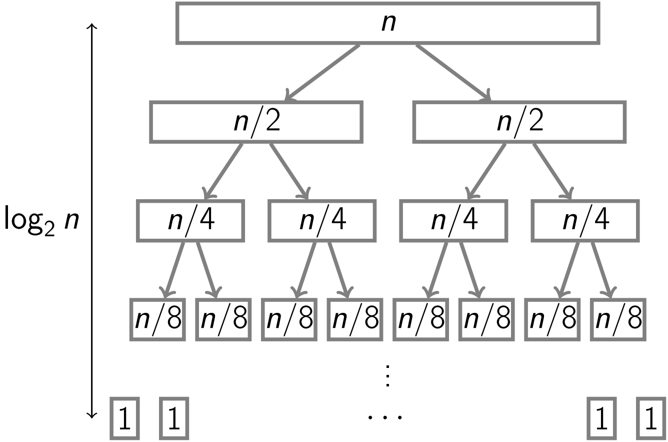

n
$n/2$
$n/2$
$n/4$
$n/4$
$n/4$
$n/4$
$n/8$
$n/8$
$n/8$
$n/8$
$n/8$
$n/8$
$n/8$
$n/8$
1
1
1
1



n
n/2
n/2
n/4
n/4
n/4
n/4
n/8
n/8
n/8
n/8
n/8
n/8
n/8
n/8
1
1
1
1
log2 n



work:
n
cn
$n/2$
$n/2$
$+$
$2c\frac{n}{2} = cn$
$\log_2 n$
$n/4$
$n/4$
$n/4$
$n/4$
$+$
$4c\frac{n}{4} = cn$
$n/8$
$n/8$
$n/8$
$n/8$
$n/8$
$n/8$
$n/8$
$n/8$
$+$
1
1
1
1



work:

cn

$+$

$2c\frac{n}{2} = cn$

$+$

$4c\frac{n}{4} = cn$

$+$

$\vdots$

Total: $cn \log_2 n$

Outline

Definition

A comparison based sorting algorithm sorts objects by comparing pairs of them.

Definition

A comparison based sorting algorithm sorts objects by comparing pairs of them.

Example

Selection sort and merge sort are comparison based.

Lemma

Any comparison based sorting algorithm performs $\Omega(n \log n)$ comparisons in the worst case to sort n objects.

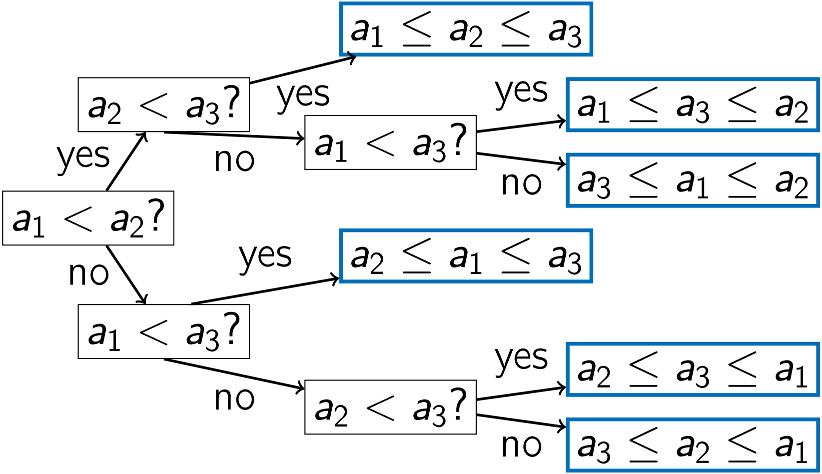
Lemma

Any comparison based sorting algorithm performs $\Omega(n \log n)$ comparisons in the worst case to sort n objects.

In other words

For any comparison based sorting algorithm, there exists an array $A[1 \ldots n]$ such that the algorithm performs at least $\Omega(n \log n)$ comparisons to sort A.

Decision Tree



Estimating Tree Depth

- the number of leaves ℓ in the tree must be at least $n!$ (the total number of permutations)

Estimating Tree Depth

- the number of leaves ℓ in the tree must be at least $n!$ (the total number of permutations)
- the worst-case running time of the algorithm (the number of comparisons made) is at least the depth d

Estimating Tree Depth

- the number of leaves ℓ in the tree must be at least $n!$ (the total number of permutations)
- the worst-case running time of the algorithm (the number of comparisons made) is at least the depth d
- $d \geq \log_2 \ell$ (or, equivalently, $2^d \geq \ell$)

Estimating Tree Depth

- the number of leaves ℓ in the tree must be at least $n!$ (the total number of permutations)
- the worst-case running time of the algorithm (the number of comparisons made) is at least the depth d
- $d \geq \log_2 \ell$ (or, equivalently, $2^d \geq \ell$)
- thus, the running time is at least

$$\log_2(n!) = \Omega(n \log n)$$

Lemma

$\log_2(n!) = \Omega(n \log n)$

Proof

$$\begin{aligned}
\log_2(n!) &= \log_2(1 \cdot 2 \cdot \cdots \cdot n) \\
&= \log_2 1 + \log_2 2 + \cdots + \log_2 n \\
&\geq \log_2 \frac{n}{2} + \cdots + \log_2 n \\
&\geq \frac{n}{2} \log_2 \frac{n}{2} = \Omega(n \log n)
\end{aligned}$$

$\square$

Outline

Example: sorting small integers

	1	2	3	4	5	6	7	8	9	10	11	12
A	2	3	2	1	3	2	2	3	2	2	2	1

Example: sorting small integers

	1	2	3	4	5	6	7	8	9	10	11	12
A	2	3	2	1	3	2	2	3	2	2	2	1

$\downarrow$

	1	2	3
$Count$	2	7	3

Example: sorting small integers

	1	2	3	4	5	6	7	8	9	10	11	12
A	2	3	2	1	3	2	2	3	2	2	2	1

$$\downarrow$$

	1	2	3
$Count$	2	7	3

$$\downarrow$$

	1	2	3	4	5	6	7	8	9	10	11	12
A'	1	1	2	2	2	2	2	2	2	3	3	3

Example: sorting small integers

	1	2	3	4	5	6	7	8	9	10	11	12
A	2	3	2	1	3	2	2	3	2	2	2	1

$\downarrow$

we have sorted these numbers
without actually comparing them!

$\downarrow$

	1	2	3	4	5	6	7	8	9	10	11	12
A'	1	1	2	2	2	2	2	2	2	3	3	3

Counting Sort: Ideas

- Assume that all elements of $A[1 \dots n]$ are integers from 1 to M.

Counting Sort: Ideas

- Assume that all elements of $A[1 \dots n]$ are integers from 1 to M.
- By a single scan of the array A, count the number of occurrences of each $1 \leq k \leq M$ in the array A and store it in $Count[k]$.

Counting Sort: Ideas

- Assume that all elements of $A[1 \ldots n]$ are integers from 1 to M.
- By a single scan of the array A, count the number of occurrences of each $1 \leq k \leq M$ in the array A and store it in $Count[k]$.
- Using this information, fill in the sorted array A'.

CountSort($A[1 \dots n]$)

```
Count[1...M] ← [0,...,0]
for i from 1 to n:
   Count[A[i]] ← Count[A[i]] + 1
{k appears Count[k] times in A}
Pos[1...M] ← [0,...,0]
Pos[1] ← 1
for j from 2 to M:
   Pos[j] ← Pos[j − 1] + Count[j − 1]
{k will occupy range [Pos[k]...Pos[k + 1] − 1]}
for i from 1 to n:
   A′[Pos[A[i]]] ← A[i]
   Pos[A[i]] ← Pos[A[i]] + 1
```

Lemma

Provided that all elements of $A[1 \ldots n]$ are integers from 1 to M, `CountSort`(A) sorts A in time $O(n + M)$.

Lemma

Provided that all elements of $A[1 \ldots n]$ are integers from 1 to M, $\texttt{CountSort}(A)$ sorts A in time $O(n+M)$.

Remark

If $M = O(n)$, then the running time is $O(n)$.

Summary

- Merge sort uses the divide-and-conquer strategy to sort an n-element array in time $O(n \log n)$.
- No comparison based algorithm can do this (asymptotically) faster.
- One can do faster if something is known about the input array in advance (e.g., it contains small integers).