Lectures 6 and 7: Merge-sort and Maximum Subarray Problem

COMS10007 - Algorithms

Dr. Christian Konrad

18.01.2019

Definition of the Sorting Problem

Sorting Problem

- **Input:** An array A of n numbers
- **Output:** A reordering of A s.t. $A[0] \leq A[1] \leq \cdots \leq A[n-1]$

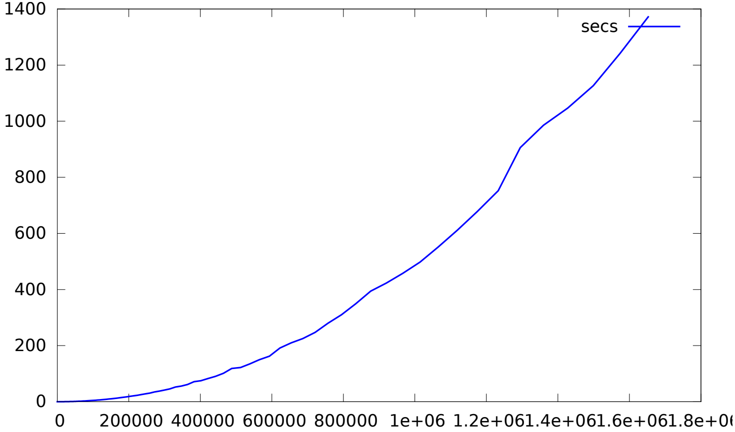
Why is it important?

- Practical relevance: Appears almost everywhere
- Fundamental algorithmic problem, rich set of techniques
- There is a non-trivial lower bound for sorting (rare!)

Insertion Sort

- Worst-case and average-case runtime $O(n^2)$
- Surely we can do better?!

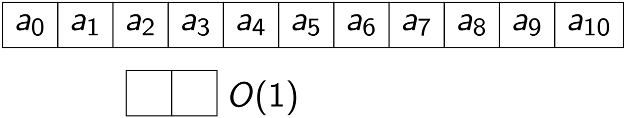
Insertion sort in Practice on Worst-case Instances



n	46929	102428	364178	1014570
secs	1.03084	4.81622	61.2737	497.879

Properties of a Sorting Algorithm

Definition (in place)
A sorting algorithm is *in place* if at any moment at most $O(1)$ array elements are stored outside the array



Example: Insertion-sort is in place

Definition (stability)
A sorting algorithm is *stable* if any pair of equal numbers in the input array appear in the same order in the sorted array

Example: Insertion-sort is stable

Records, Keys, and Satellite Data

Sorting Complex Data

- In reality, data that is to be sorted is rarely entirely numerical (e.g. sort people in a database according to their last name)
- A data item is often also called a **record**
- The **key** is the part of the record according to which the data is to be sorted
- Data different to the key is also referred to as **satellite data**

family name	first name	data of birth	role
Smith	Peter	02.10.1982	lecturer
Hills	Emma	05.05.1975	reader
Jones	Tom	03.02.1977	senior lecturer
...			

Observe: Stability makes more sense when sorting complex data as opposed to numbers

Merge Sort

Key Idea:

- Suppose that left half and right half of array is sorted
- Then we can merge the two sorted halves to a sorted array in $O(n)$ time:

Merge Operation

- Copy left half of A to new array B
- Copy right half of A to new array C
- Traverse B and C simultaneously from left to right and write the smallest element at the current positions to A

Example: Merge Operation

A	1	4	9	10	3	5	7	11

Example: Merge Operation

A	1	4	9	10	3	5	7	11

B	1	4	9	10

C	3	5	7	11

Example: Merge Operation

A								
B			1	4	9	10		
C			3	5	7	11		

Example: Merge Operation

A								

B	1	4	9	10
C	3	5	7	11

Example: Merge Operation

A	1							

B	1	4	9	10
C	3	5	7	11

Example: Merge Operation

A								
	1	3						

B				
	1	4	9	10

C				
	3	5	7	11

Example: Merge Operation

A	1	3	4					
B			1	4	9	10		
C			3	5	7	11		

Example: Merge Operation

A	1	3	4	5				
B			1	4	9	10		
C			3	5	7	11		

Example: Merge Operation

A	1	3	4	5	7			
B			1	4	9	10		
C			3	5	7	11		

Example: Merge Operation

A	1	3	4	5	7	9	10	11

B	1	4	9	10
C	3	5	7	11

Analysis: Merge Operation

Merge Operation

- **Input:** An array A of integers of length n (n even) such that $A[0, \frac{n}{2} - 1]$ and $A[\frac{n}{2}, n - 1]$ are sorted
- **Output:** Sorted array A

Runtime Analysis:

1. Copy left half of A to B: $O(n)$ operations
2. Copy right half of A to C: $O(n)$ operations
3. Merge B and C and store in A: $O(n)$ operations

Overall: $O(n)$ time in worst case

How can we establish that left and right halves are sorted?

Divide and Conquer!

Merge Sort: A Divide and Conquer Algorithm

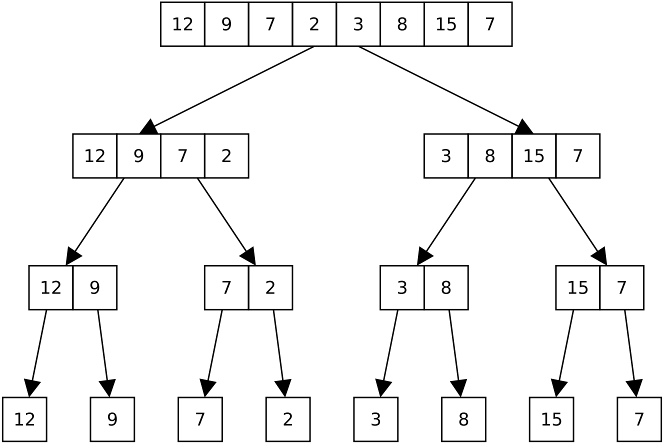
Require: Array A of n numbers
  if $n = 1$ **then**
    return A
  $A[0, \lfloor \frac{n}{2} \rfloor] \leftarrow \textsc{MergeSort}(A[0, \lfloor \frac{n}{2} \rfloor])$
  $A[\lfloor \frac{n}{2} \rfloor + 1, n-1] \leftarrow \textsc{MergeSort}(A[\lfloor \frac{n}{2} \rfloor + 1, n-1])$
  $A \leftarrow \textsc{Merge}(A)$
  return A

MergeSort

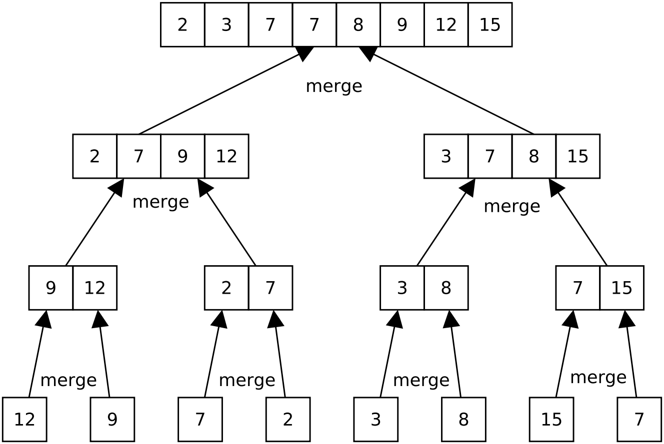
Structure of a Divide and Conquer Algorithm

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively. If the subproblems are small enough, just solve them in a straightforward manner.
- **Combine** the solutions to the subproblems into the solution for the original problem.

Analyzing MergeSort: An Example



Analyzing MergeSort: An Example



Analyzing Merge Sort

Analysis Idea:

- We need to sum up the work spent in each node of the *recursion tree*
- The recursion tree in the example is a *complete binary tree*

  Definition: A tree is a *complete binary tree* if every node has either 2 or 0 children.

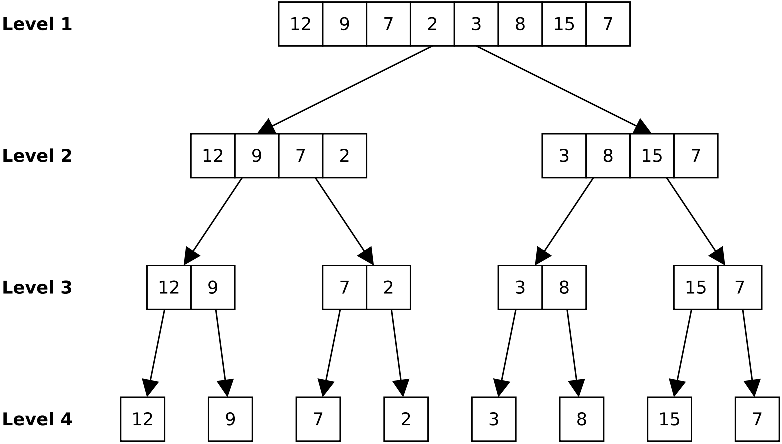
  Definition: A tree is a *binary tree* if every node has at most 2 children.

(we will talk about trees in much more detail later in this unit)

Questions:

- How many levels?
- How many nodes per level?
- Time spent per node?

Number of Levels



Number of Levels (2)

Level i:

- 2^{i-1} nodes (at most)
- Array length in level i is $\lceil \frac{n}{2^{i-1}} \rceil$ (at most)
- Runtime of merge operation for each node in level i: $O(\frac{n}{2^{i-1}})$

Number of Levels:

- Array length in last level l is 1: $\lceil \frac{n}{2^{l-1}} \rceil = 1$

$$\frac{n}{2^{l-1}} \le 1 \Rightarrow n \le 2^{l-1} \Rightarrow \log(n) + 1 \le l$$

- Array length in last but one level $l - 1$ is 2: $\lceil \frac{n}{2^{l-2}} \rceil = 2$

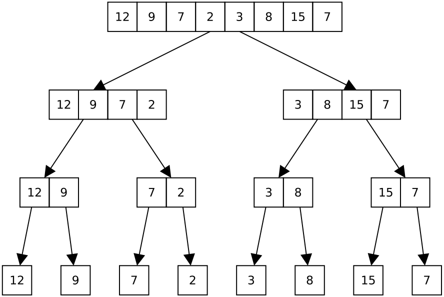
$$\frac{n}{2^{l-2}} > 1 \Rightarrow n > 2^{l-2} \Rightarrow \log(n) + 2 > l$$
$$\log(n) + 1 \le l < \log(n) + 2$$

Hence, $l = \lceil \log n \rceil + 1$.

Runtime of Merge Sort

Sum up Work:

- Levels:
  $l = \lceil \log n \rceil + 1$
- Nodes on level i:
  at most 2^{i-1}
- Array length in level i:
  at most $\lceil \frac{n}{2^{i-1}} \rceil$

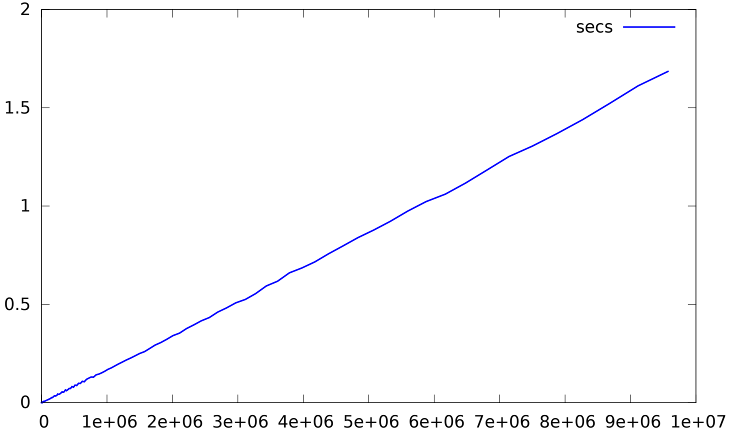


Worst-case Runtime:

$$\sum_{i=1}^{\lceil \log n \rceil + 1} 2^{i-1} O\left(\lceil \frac{n}{2^{i-1}} \rceil\right) = \sum_{i=1}^{\lceil \log n \rceil + 1} 2^{i-1} O\left(\frac{n}{2^{i-1}}\right)$$

$$= \sum_{i=1}^{\lceil \log n \rceil + 1} O(n) = (\lceil \log n \rceil + 1)\, O(n) = O(n \log n) \; .$$

Merge sort in Practice on Worst-case Instances



n	46929	102428	364178	1014570
secs	1.03084	4.81622	61.2737	497.879 (Insertion-sort)
secs	0.007157	0.015802	0.0645791	0.169165 (Merge-sort)

Generalizing the Analysis

Divide and Conquer Algorithm:

Let **A** be a divide and conquer algorithm with the following properties:

1. **A** performs two recursive calls on input sizes at most $n/2$
2. The conquer operation in **A** takes $O(n)$ time

Then:

$$\boxed{\textbf{A} \text{ has a runtime of } O(n \log n)\ .}$$

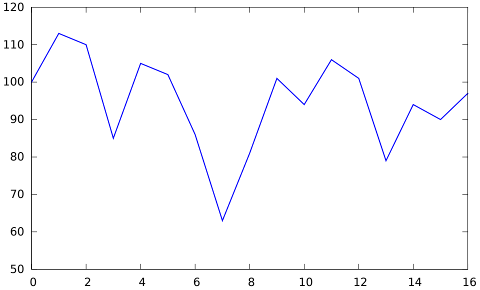
Stability and In Place Property?

- Merge sort is stable
- Merge sort does not sort in place

Maximum Subarray Problem

Buy Low, Sell High Problem

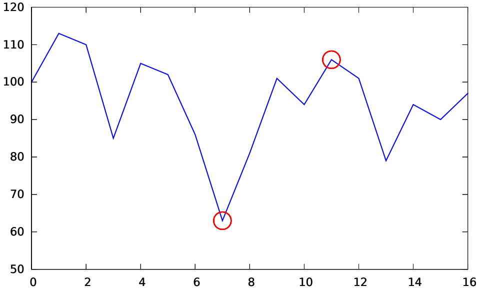
- **Input:** An array of n integers
- **Output:** Indices $0 \leq i < j \leq n-1$ such that $A[j] - A[i]$ is maximized



Maximum Subarray Problem

Buy Low, Sell High Problem

- **Input:** An array of n integers
- **Output:** Indices $0 \leq i < j \leq n-1$ such that $A[j] - A[i]$ is maximized



Maximum Subarray Problem

Focus on Array of Changes:

Day	0	1	2	3	4	5	6	7	8	9	10	11
$	100	113	110	85	105	102	86	63	81	101	94	106
Δ		13	-3	-25	20	-3	-16	-23	18	20	-7	12

Maximum Subarray Problem

- **Input:** Array A of n numbers
- **Output:** Indices $0 \le i \le j \le n-1$ such that $\sum_{l=i}^{j} A[l]$ is maximum.

Trivial Solution: $O(n^3)$ runtime

- Compute subarrays for every pair i, j
- There are $O(n^2)$ pairs, computing the sum takes time $O(n)$.

Maximum Subarray Problem

Focus on Array of Changes:

Day	0	1	2	3	4	5	6	7	8	9	10	11
\$	100	113	110	85	105	102	86	63	81	101	94	106
Δ		13	-3	-25	20	-3	-16	-23	18	20	-7	12

Maximum Subarray Problem

- **Input:** Array A of n numbers
- **Output:** Indices $0 \leq i \leq j \leq n-1$ such that $\sum_{l=i}^{j} A[l]$ is maximum.

Trivial Solution: $O(n^3)$ runtime

- Compute subarrays for every pair i, j
- There are $O(n^2)$ pairs, computing the sum takes time $O(n)$.

Divide and Conquer Algorithm for Maximum Subarray

Divide and Conquer:
Compute maximum subarrays in left and right halves of initial array

$$A = L \circ R$$

Combine:
Given maximum subarrays in L and R, we need to compute maximum subarray in A

Three cases:

1. Maximum subarray is entirely included in L ✓
2. Maximum subarray is entirely included in R ✓
3. Maximum subarray crosses midpoint, i.e., i is included in L and j is included in R

Divide and Conquer Algorithm for Maximum Subarray

Maximum Subarray Crosses Midpoint:

- Find maximum subarray $A[i,j]$ such that $i \leq \frac{n}{2}$ and $j > \frac{n}{2}$ (assume that n is even)
- Observe that: $\sum_{l=i}^{j} A[l] = \sum_{l=i}^{\frac{n}{2}} A[i] + \sum_{l=\frac{n}{2}+1}^{j} A[l]$.

Two Independent Subproblems:

- Find index i such that $\sum_{l=i}^{\frac{n}{2}} A[i]$ is maximized
- Find index j such that $\sum_{l=\frac{n}{2}+1}^{j} A[l]$ is maximized

We can solve these subproblems in time $O(n)$. (how?)

Maximum Subarray Problem - Summary

Require: Array A of n numbers
  if $n = 1$ **then**
    return A
  Recursively compute max. subarray S_1 in $A[0, \lfloor \frac{n}{2} \rfloor]$
  Recursively compute max. subarray S_2 in $A[\lfloor \frac{n}{2} \rfloor + 1, n - 1]$
  Compute maximum subarray S_3 that crosses midpoint
  return Heaviest of the three subarrays S_1, S_2, S_3

Recursive Algorithm for the Maximum Subarray Problem

Analysis:

- Two recursive calls with inputs that are only half the size
- Conquer step requires $O(n)$ time
- Identical to Merge Sort, runtime $O(n \log n)$!