Advanced Algorithms and Data Structures

Prof. Tapio Elomaa
tapio.elomaa@tut.fi

Course Prerequisites

- A seven credit unit course
- Replaced OHJ-2156 *Analysis of Algorithms*
- We take things a bit further than OHJ-2156

- We will assume familiarity with
  - Necessary mathematics
  - Elementary data structures
  - Programming
Course Basics

- There will be 4 hours of lectures per week
- Weekly exercises start in two weeks time
- We will not have a programming exercise this year (unless you demand to have one)
- We might consider organizing a seminar with voluntary presentations (yielding extra points) at the end of the course

Organization & Timetable

- **Lectures**: Prof. Tapio Elomaa
  - Tue & Thu 2–4 PM in TB223 (& TB219 in PII)
    - Period break Oct. 12–18, 2015

- **Exercises**: M.Sc. Juho Lauri & M.Sc. Teemu Heinimäki,
  - Mon 10–12 TC315, Start: Sept. 9

- **Exam**: Fri Dec. 18, 2015
Guest Lecture

- Dr. Timo Aho from Solita
- In October
  - Big data
  - Definitions of data analytics
  - Methods
  - Case studies

Course Grading

- **Exam**: Maximum of 30 points
- **Weekly exercises** yield extra points
  - 40% of questions answered: 1 point
  - 80% answered: 6 points
  - In between: linear scale (so that decimals are possible)
- Final grading depends on what we agree as course components
Material

- The textbook of the course is
- There is no prepared material, the slides appear in the web as the lectures proceed
- The exam is based on the lectures (i.e., not on the slides only)

Content (Plan)

I Foundations
II (Sorting and) Order Statistics
III Data Structures
IV Advanced Design and Analysis Techniques
V Advanced Data Structures
VI Graph Algorithms
VII Selected Topics
I Foundations

The Role of Algorithms in Computing
Getting Started
Growth of Functions
Recurrences
Probabilistic Analysis and Randomized Algorithms

II (Sorting and) Order Statistics

Heapsort
Quicksort
Sorting in Linear Time
Medians and Order Statistics
III Data Structures

Elementary Data Structures
   Hash Tables
   Binary Search Trees
   Red-Black Trees

IV Advanced Design and Analysis Techniques

Dynamic Programming
   Greedy Algorithms
   Amortized Analysis
V Advanced Data Structures

- B-Trees
- Binomial Heaps
- Fibonacci Heaps

VI Graph Algorithms

- MAT-62756 GRAPH THEORY
- Elementary Graph Algorithms
- Minimum Spanning Trees
- Single-Source Shortest Paths
- All-Pairs Shortest Paths
- Maximum Flow
Online ChiMerge

- In machine learning and data mining algorithms one often needs to discretize numerical attributes
- There are initial intervals that cannot be divided (examples that have the same value for the attribute in question)
- ChiMERGE is intended as a preprocessing technique for learning algorithms
- It combines adjacent initial intervals bottom-up to come up with the final discretization of the value range of the numerical attribute in question
The worst-case time complexity of the batch algorithm is $O(n \lg n)$.

The main requirement for the online version of CHIMERGE (OCM) is a processing time that is, per each received training example, logarithmic in the number of intervals.

Thus, we altogether match the $O(n \lg n)$ time requirement of the batch version of CHIMERGE.
• If each new example drawn from DATA SOURCE would update the current discretization immediately, this would yield a slowing down of OCM by a factor $n$ as compared to the batch version.
• Therefore, we update the active discretization only periodically, freezing it for the intermediate period.
• Examples received during this time can be handled at a logarithmic time.
• The price to be paid is maintaining some other data structures in addition to the BST.

• OCM uses a balanced binary search tree $T$ into which each example drawn from DATA SOURCE is (eventually) submitted.
• The algorithm continuously takes in new examples from DATA SOURCE, but instead of just storing them to $T$, it updates the required data structures simultaneously.
• We do not halt the flow of the data stream because of extra processing of the data structures, but amortize the required work to normal example handling.
• BST $T$ plus a queue, two doubly linked lists, and a priority queue (heap) are maintained.
The discretization process works in three phases:

1. In Phase 1 (of length \( V \) time steps), the examples from \( \text{DATA} \text{SOURCE} \) are cached for temporary storage to the queue \( Q \).
2. Meanwhile, the initial intervals are collected one at a time from \( T \) to a list \( L_I \) that holds the intervals in their numerical order.
3. For each pair of two adjacent intervals, the \( D \)-value is computed and inserted into the priority queue \( P \).
4. The queue \( Q \) is needed to freeze \( T \) for the duration of the Phase 1.
Procedure $\text{OCM}(M, D_{th})$

**Input:** Integer $M$ giving the number of examples on which initial discretization is based on and real $D_{th}$ which is the $\chi^2$ threshold for interval merging

**Data structures:** A BST $T$, a queue $Q$, two doubly linked lists $L_I$ and $L_D$, and a priority queue $P$

1. Initialize $T$, $Q$, $L_I$, and $L_D$ as empty;
2. Read $M$ training examples into tree $T$;
3. phase $\leftarrow 1$; $b \leftarrow \text{Tree-Minimum}(T)$;
4. Make $P$ an empty priority queue of size $|T| - 1$;
5. while $E \leftarrow \text{DataSource}() \neq \text{nil}$ do

   6. if phase = 1 then $\text{ENQUEUE}(Q, E)$
   7. else $\text{Tree-Insert}(T, E)$ fi;
   8. if phase = 1 then

      9. Insert a copy of $b$ as the last item in $L_I$;
     10. if $|L_I| > 1$ then
     11. Compute the $D$-value of the two last intervals in $L_I$;
     12. Using the obtained value as a key, add a pointer to the two intervals in $L_I$ into $P$;
     13. fi
     14. $b \leftarrow \text{Tree-Successor}(T, b)$;
     15. if $b = \text{nil}$ then phase $\leftarrow 2$ fi
• After seeing $V$ examples, all the initial intervals in $T$ have been inserted to $L_I$ and all the $D$-values have been computed and added to $P$
• At this point $Q$ holds $V$ (unprocessed) examples
• At the end of Phase 1 all required preparations for interval merging have been done
• Phase 2 implements the merging as in batch CHIMERGE, but without stopping the processing of DATA SOURCE
• As a result of each merging, the intervals in $L_I$ as well as the related $D$-values in $P$ are updated

• In Phase 2 the example received from DATA SOURCE is submitted directly to $T$ along with another, previously received example that is cached in $Q$
• The lowest $D$-value, $D_{i,j}$, corresponding to some adjacent pair of intervals $I$ and $J$, is drawn from $P$
• If it exceeds the $\chi^2$ merging threshold, Phase 2 is complete
• Otherwise, the intervals $I$ and $J$ are merged to obtain a new interval $K$, and the $D$-values on both sides of $K$ are updated into $P$
16. else if phase = 2 then

\[ \text{17. } d \leftarrow \text{DEQUEUE}(Q); \text{TREE-INSERT}(T, d); \]

\[ \text{18. } (D_{i,j}, \langle I, J \rangle) \leftarrow \text{EXTRACT-MIN}(P); \]

19. if \( \langle I, J \rangle \neq \text{nil} \text{ and } D_{i,j} < D_{th} \) then

20. Remove from P the D-values of I and J;

21. \text{LIST-DELETE}(L_I, I); \text{LIST-DELETE}(L_I, J);

22. \( K \leftarrow \text{MERGE}(I, J); \)

23. Compute the \( D \)-values of \( K \) with its neighbors in \( L_I; \)

24. Update the \( D \)-values into priority queue \( P; \)

25. else

26. phase \( \leftarrow 3; \) fi

27. else

\[ \text{28. } d \leftarrow \text{DEQUEUE}(Q); \text{TREE-INSERT}(T, d); \]

\[ \text{29. } e \leftarrow \text{LIST-DELETE}(L_I, \text{HEAD}(L_I)); \]

30. \text{LIST-INSERT}(L_D, e);\]

31. if \text{EMPTY}(Q) then

32. phase \( \leftarrow 1; \) \( b \leftarrow \text{TREE-MINIMUM}(T); \)

33. Make \( P \) an empty priority queue, size \( |T| - 1; \)

34. fi

35. fi

36. od
**Time Complexity of OCM**

- The insertion of the received example $E$ to $Q$ clearly takes constant time, as well as the insertion of an interval from $T$ to $L_I$.
- Also computing the $D$-value can be considered a constant-time operation.
- Inserting the obtained value to $P$ requires time $O(\lg V)$.
- Finding a successor in $T$ also takes time $O(\lg V)$.
- Thus, the total time consumption of one iteration of Phase 1 is $O(\lg V)$.

- Insertion of the new example $E$ and one from the queue $Q$ to the BST $T$ both take time $O(\lg V)$.
- Extracting the lowest $D$-value from the priority queue $P$ is also an $O(\lg V)$ time operation.
- As $P$ contains pointers to the items holding intervals $I$ and $J$ in $L_I$, removing them can be implemented in constant time.
- Merging of $I$ and $J$ to obtain $K$ can be considered a constant-time operation as well as computing the new $D$-values for $K$ and its neighbors in $L_I$.
- Updating each of the new $D$-values into $P$ takes $O(\lg V)$ time.
- Thus, the total time consumption of one iteration of Phase 2 is also $O(\lg V)$.
The sorting problem

- **Input:** A sequence of \( n \) numbers \( \langle a_1, a_2, \ldots, a_n \rangle \)

- **Output:** A *permutation* (reordering) \( \langle a'_1, a'_2, \ldots, a'_n \rangle \) of the input sequence such that \( a'_1 \leq a'_2 \leq \cdots \leq a'_n \)

- The numbers that we wish to sort are also known as *keys*

---

**INSERTION-SORT** \( (A) \)

1. for \( j \leftarrow 2 \) to \( A\.length \)
2. \( key \leftarrow A[j] \)
3. // Insert \( A[j] \) into the sorted sequence \( A[1..j-1] \)
4. \( i \leftarrow j - 1 \)
5. while \( i > 0 \) and \( A[i] > key \)
6. \( A[i+1] \leftarrow A[i] \)
7. \( i \leftarrow i - 1 \)
8. \( A[i+1] \leftarrow key \)
Correctness of the Algorithm

- The following loop invariant helps us understand why the algorithm is correct:

At the start of each iteration of the for loop of lines 1–8, the subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$ but in sorted order.
Initialization

- The loop invariant holds before the first loop iteration, when $j = 2$:
  - The subarray, therefore, consists of just the single element $A[1]$
  - It is in fact the original element in $A[1]$
  - This subarray is trivially sorted
  - Therefore, the loop invariant holds prior to the first iteration of the loop

Maintenance

- Each iteration maintains the loop invariant:
  - The body of the for loop works by moving $A[j-1], A[j-2], A[j-3], \ldots$ by one position to the right until the proper position for $A[j]$ is found (lines 4–7)
  - At this point the value of $A[j]$ is inserted (line 8)
  - The subarray $A[1..j]$ then consists of the elements originally in $A[1..j]$, but in sorted order
Termination

- The condition causing the for loop to terminate is that \( j > A.\text{length} = n \)
- Because each loop iteration increases \( j \) by 1, we must have \( j = n + 1 \) at that time
- Substituting \( n + 1 \) for \( j \) in the wording of loop invariant, we have that the subarray \( A[1..n] \) consists of the elements originally in \( A[1..n] \), but in sorted order
- \( A[1..n] \) is the entire array

Analysis of insertion sort

- The time taken by the \textsc{insertion-sort} depends on the input:
  - sorting a thousand numbers takes longer than sorting three numbers
- Moreover, the procedure can take different amounts of time to sort two input sequences of the same size
  - depending on how nearly sorted they already are
Input size

- The time taken by an algorithm grows with the size of the input
- Traditional to describe the running time of a program as a function of the size of its input
- For many problems, such as sorting, the most natural measure for input size is the number of items in the input—i.e., the array size $n$

- For, e.g., multiplying two integers, the best measure is the total number of bits needed to represent the input in binary notation
- Sometimes, more appropriate to describe the size with two numbers rather than one
- E.g., if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in it
Running time

- Running time of an algorithm on an input:
  - The number of primitive operations ("steps") executed
- Step as machine-independent as possible
- For the moment:
  - Constant amount of time to execute each line of pseudocode
  - We assume that each execution of the $i$th line takes time $c_i$, where $c_i$ is a constant

<table>
<thead>
<tr>
<th>Insertion-Sort($A$)</th>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 for $j \leftarrow 2$ to $A$.length</td>
<td>$c_1$</td>
<td>$n$</td>
</tr>
<tr>
<td>2 key $\leftarrow A[j]$</td>
<td>$c_2$</td>
<td>$n-1$</td>
</tr>
<tr>
<td>3 // Insert $A[j]$ into the sorted sequence $A[1..j]$</td>
<td>0</td>
<td>$n-1$</td>
</tr>
<tr>
<td>4 $i \leftarrow j - 1$</td>
<td>$c_4$</td>
<td>$n-1$</td>
</tr>
<tr>
<td>5 while $i &gt; 0$ and $A[i] &gt; key$</td>
<td>$c_5$</td>
<td>$\sum_{j=2}^{n} t_j$</td>
</tr>
<tr>
<td>6 $A[i+1] \leftarrow A[i]$</td>
<td>$c_6$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
</tr>
<tr>
<td>7 $i \leftarrow i - 1$</td>
<td>$c_7$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
</tr>
<tr>
<td>8 $A[i+1] \leftarrow key$</td>
<td>$c_8$</td>
<td>$n-1$</td>
</tr>
</tbody>
</table>
• $t_j$ denotes the number of times the `while` loop test in line 5 is executed for that value of $j$

• When a `for` or `while` loop exits in the usual way, the test is executed one time more than the loop body

• Comments are not executable statements, so they take no time

• Running time of the algorithm is the sum of those for each statement executed

To compute $T(n)$, the running time of `INSERTION-SORT` on an input of $n$ values,

– we sum the products of the cost and times columns, obtaining

$$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j$$

$$+c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1)$$
**Best case**

- The best case occurs if the array is already sorted.
- For each $j = 2, 3, \ldots, n$, we then find that $A[i] < \text{key}$ in line 5 when $i$ has its initial value of $j - 1$.
- Thus $t_j = 1$ for $j = 2, 3, \ldots, n$, and the best-case running time is

$$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 (n - 1) + c_8 (n - 1) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

**Worst case**

- We can express this as $an + b$ for constants $a$ and $b$ that depend on the statement costs $c_i$.
- It is a \textit{linear function} of $n$.
- The worst case results when the array is in reverse sorted order — in decreasing order.
- We must compare each element $A[j]$ with each element in the entire sorted subarray $A[1..j-1]$, and so $t_j = j$ for $j = 2, 3, \ldots, n$. 

• Note that
\[
\sum_{j=2}^{n} j = \frac{n(n + 1)}{2} - 1
\]
and
\[
\sum_{j=2}^{n} (j - 1) = \frac{n(n - 1)}{2}
\]
by the summation of an arithmetic series
\[
\sum_{j=1}^{n} j = \frac{n(n + 1)}{2}
\]

• The worst-case running time of INSERTION-SORT is
\[
T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) \\
+ c_5 \left( \frac{n(n + 1)}{2} - 1 \right) + c_6 \left( \frac{n(n - 1)}{2} \right) \\
+ c_7 \left( \frac{n(n - 1)}{2} \right) + c_8 (n - 1) \\
= \left( \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + (c_1 + \cdots + c_8) n \\
- (c_2 + \cdots + c_8)
\]
• We can express this worst-case running time as $an^2 + bn + c$ for constants $a$, $b$, and $c$ that depend on the statement costs $c_i$
• It is a **quadratic function** of $n$
• The **rate of growth**, or **order of growth**, of the running time really interests us
• We consider only the leading term of a formula ($an^2$); the lower-order terms are relatively insignificant for large values of $n$

• We also ignore the leading term's coefficient, constant factors are less significant than the rate of growth in determining computational efficiency for large inputs
• For insertion sort, we are left with the factor of $n^2$ from the leading term
• We write that insertion sort has a worst-case running time of $\Theta(n^2)$
  ("theta of $n$-squared")
2.3 Designing algorithms

- Insertion sort is an incremental approach: having sorted $A[1..j-1]$, we insert $A[j]$ into its proper place, yielding sorted subarray $A[1..j]$
- Let us examine an alternative design approach, known as “divide-and-conquer”
- We design a sorting algorithm whose worst-case running time is much lower
- The running times of divide-and-conquer algorithms are often easily determined

The divide-and-conquer approach

- Many useful algorithms are **recursive**:
  - to solve a problem, they call themselves to deal with closely related subproblems
- These algorithms typically follow a divide-and-conquer approach:
  - Break the problem into subproblems that resemble the original problem but are smaller,
  - Solve the subproblems recursively,
  - Combine these solutions to create a solution to the original problem
The paradigm involves three steps at each level of the recursion:

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

---

**The merge sort algorithm**

- **Divide:** Divide the $n$-element sequence into two subsequences of $n/2$ elements each
- **Conquer:** Sort the two subsequences recursively using merge sort
- **Combine:** Merge the two sorted subsequences to produce the sorted answer
  - Recursion “bottoms out” when the sequence to be sorted has length 1: a sequence of length 1 is already in sorted order
- The key operation is the merging of two sorted sequences in the “combine” step
- We call auxiliary procedure \( \text{MERGE}(A, p, q, r) \), where \( A \) is an array and \( p, q, \) and \( r \) are indices such that \( p \leq q < r \)
- The procedure assumes that the subarrays \( A[p..q] \) and \( A[q+1..r] \) are in sorted order and
- merges them to form a single sorted subarray that replaces the current subarray \( A[p..r] \)

\[
\begin{align*}
\text{MERGE}(A, p, q, r) & \\
1. & n_1 \leftarrow q - p + 1 \\
2. & n_2 \leftarrow r - q \\
3. & \text{Let } L[1..n_1 + 1] \text{ and } R[1..n_2 + 1] \text{ be new arrays} \\
4. & \text{for } i \leftarrow 1 \text{ to } n_1 \\
5. & L[i] \leftarrow A[p + i - 1] \\
6. & \text{for } j \leftarrow 1 \text{ to } n_2 \\
7. & R[j] \leftarrow A[q + j] \\
8. & L[n_1 + 1] \leftarrow \infty \\
9. & R[n_2 + 1] \leftarrow \infty \\
10. & i \leftarrow 1 \\
11. & j \leftarrow 1 \\
12. & \text{for } k \leftarrow p \text{ to } r \\
13. & \text{if } L[i] \leq R[j] \\
14. & A[k] \leftarrow L[i] \\
15. & i \leftarrow i + 1 \\
16. & \text{else } A[k] \leftarrow R[j] \\
17. & j \leftarrow j + 1
\end{align*}
\]
Line 1 computes the length \( n_1 \) of the subarray \( A[p..q] \); similarly for \( n_2 \) and \( A[q+1..r] \) on line 2.

Line 3 creates arrays \( L \) (left) and \( R \) (right), of lengths \( n_1 + 1 \) and \( n_2 + 1 \), respectively. The extra position will hold the sentinel \( \infty \).

The \textbf{for} loop of lines 4–5 copies \( A[p..q] \) into \( L[1..n_1] \);

Lines 6–7 copy \( A[q+1..r] \) into \( R[1..n_2] \).

Lines 8–9 put the sentinels at the ends of \( L \) and \( R \).

Lines 10–17 perform the \( r - p + 1 \) basic steps by maintaining the following loop invariant:

- At the start of each iteration of the \textbf{for} loop of lines 12–17, \( A[p..k-1] \) contains the \( k - p \) smallest elements of \( L[1..n_1 + 1] \) and \( R[1..n_2 + 1] \), in sorted order.

- Moreover, \( L[i] \) and \( R[j] \) are the smallest elements of their arrays that have not been copied back into \( A \).
\textbf{MERGE}(A, 9, 12, 16)

\begin{align*}
A & \quad 2 \quad 4 \quad 5 \quad 7 \quad 1 \quad 2 \quad 3 \quad 6 \quad \ldots \\
L & \quad 2 \quad 4 \quad 5 \quad 7 \quad \infty \\
R & \quad 1 \quad 2 \quad 3 \quad 6 \quad \infty \\
\end{align*}

\begin{align*}
A & \quad 1 \quad 4 \quad 5 \quad 7 \quad 1 \quad 2 \quad 3 \quad 6 \quad \ldots \\
L & \quad 2 \quad 4 \quad 5 \quad 7 \quad \infty \\
R & \quad 1 \quad 2 \quad 3 \quad 6 \quad \infty \\
\end{align*}

\begin{align*}
A & \quad 1 \quad 2 \quad 5 \quad 7 \quad 1 \quad 2 \quad 3 \quad 6 \quad \ldots \\
L & \quad 2 \quad 4 \quad 5 \quad 7 \quad \infty \\
R & \quad 1 \quad 2 \quad 3 \quad 6 \quad \infty \\
\end{align*}
• The needed \( r - p + 1 \) iterations of the last for loop have been executed:
  - \( A[9..16] \) is sorted, and
  - the two sentinels in \( L \) and \( R \) are the only two elements in these arrays that have not been copied into \( A \)

• \textsc{merge} procedure runs in \( \Theta(n) \) time, where
  \( n = r - p + 1 \)
  – each of lines 1–3 and 8–11 takes constant time
  – the for loops of lines 4–7 take \( \Theta(n_1 + n_2) = \Theta(n) \) time
  – there are \( n \) iterations of the for loop of lines 12–17, each of which takes constant time
**Merge sort**

- The procedure \( \text{MERGE-SORT}(A, p, r) \) sorts the elements in \( A[p..r] \)
- If \( p \geq r \), the subarray has at most one element and is therefore already sorted
- Otherwise, the divide step computes an index \( q \) that partitions \( A[p..r] \) into two subarrays:
  - \( A[p..q] \), containing \( \lfloor n/2 \rfloor \) elements
  - \( A[q + 1..r] \), containing \( \lceil n/2 \rceil \) elements

\[
\text{MERGE-SORT}(A, p, r) \\
1. \text{if } p < r \\
2. \quad q \leftarrow \lfloor (p + r)/2 \rfloor \\
3. \quad \text{MERGE-SORT}(A, p, q) \\
4. \quad \text{MERGE-SORT}(A, q + 1, r) \\
5. \quad \text{MERGE}(A, p, q, r)
\]
Analysis of merge sort

- Our analysis assumes that the original problem size is a power of 2
- Each divide step then yields two subsequences of size exactly $n/2$
- We set up the recurrence for $T(n)$, the worst-case running time of merge sort on $n$ numbers
- Merge sort on just one element takes constant time

When we have $n > 1$ elements, we break down the running time as follows:
- **Divide**: The step just computes the middle of the subarray, which takes constant time:
  \[ D(n) = \Theta(1) \]
- **Conquer**: We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time
- **Combine**: the MERGE procedure on an $n$-element array takes time $\Theta(n)$, and so
  \[ C(n) = \Theta(n) \]
When we add the $D(n)$ and $C(n)$, we are adding functions that are $\Theta(n)$ and $\Theta(1)$.

This sum is a linear function of $n$.

Adding it to the $2T(n/2)$ term from the “conquer” step gives the recurrence for $T(n)$:

$$T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
2T(n/2) + \Theta(n) & \text{if } n > 1
\end{cases}$$

To intuitively see that the solution to the recurrence is $T(n) = \Theta(n \lg n)$, where $\lg n$ stands for $\log_2 n$, let us rewrite it as

$$T(n) = \begin{cases} 
c & \text{if } n = 1 \\
2T(n/2) + cn & \text{if } n > 1
\end{cases}$$

where constant $c$ represents time required to solve problems of size 1 and that per array element of the divide and combine steps.