**General remarks**

- First we continue with an important example for Divide-and-Conquer, namely **Merge Sort**.
- Then we present a basic tool for analysing algorithms by **Solving Recurrences**.
- We conclude by considering another example, namely **Matrix Multiplication**.

**Reading from CLRS for week 3**

- Chapter 4

---

**Another example: Merge-Sort**

A sorting algorithm based on divide and conquer. The worst-case running time has a lower order of growth than insertion sort.

Again we are dealing with subproblems of sorting subarrays $A[p\ldots q]$ Initially, $p = 1$ and $q = A.length$, but these values change again as we recurse through subproblems.

To sort $A[p\ldots q]$:

**Divide** by splitting into two subarrays $A[p\ldots r]$ and $A[r+1\ldots q]$, where $r$ is the halfway point of $A[p\ldots q]$.

**Conquer** by recursively sorting the two subarrays $A[p\ldots r]$ and $A[r+1\ldots q]$.

**Combine** by merging the two sorted subarrays $A[p\ldots r]$ and $A[r+1\ldots q]$ to produce a single sorted subarray $A[p\ldots q]$.

The recursion bottoms out when the subarray has just 1 element, so that it is trivially sorted.

**Merge-Sort**($A, p, q$)

1. **if** $p < q$ // check for base case
2. $r = 
[(p+q)/2]$ // divide
3. **Merge-Sort**($A, p, r$) // conquer
4. **Merge-Sort**($A, r+1, q$) // conquer
5. **Merge**($A, p, r, q$) // combine

**Initial call**: **Merge-Sort**($A, 1, A.length$)
Merge

**Input:** Array $A$ and indices $p, r, q$ such that
- $p \leq r < q$
- Subarrays $A[p . . r]$ and subarray $A[r+1 . . q]$ are sorted. By the restriction on $p, r, q$ neither subarray is empty.

**Output:** The two subarrays are merged into a single sorted subarray in $A[p . . q]$.

We implement is so that it takes $\Theta(n)$ time, with $n = q - p + 1 = \text{the number of elements being merged}$. 

```
MERGE(A, p, r, q)
1  n1 = r - p + 1
2  n2 = q - r
3  let L[1 . . n1+1] and R[1 . . n2+1] be new arrays
4  for i = 1 to n1
5      L[i] = A[p+i-1]
6  for j = 1 to n2
7      R[j] = A[r+j]
8  L[n1+1] = R[n2+1] = \infty
9  i = j = 1
10  for k = p to q
11     if L[i] \leq R[j]
12        A[k] = L[i]
13        i = i+1
14     else A[k] = R[j]
15        j = j+1
```

### Example

- $A = [0, 2, 5, 6, 8]$ (the two subarrays to be merged)
- $A[p . . q] = [0, 1, 2, 3, 4, 5, 6, 8, 8, 9]$

#### Analysis of Merge-Sort

The runtime $T(n)$, where $n = q-p+1 > 1$, satisfies:

$$T(n) = 2T(n/2) + \Theta(n).$$

We will show that $T(n) = \Theta(n \lg n)$.

- It can be shown (see tutorial-section) that $\Omega(n \lg n)$ comparisons are necessary in the worst case to sort $n$ numbers for any comparison-based algorithm: this is thus an (asymptotic) lower bound on the problem.
- Hence **Merge-Sort** is provably (asymptotically) optimal.
Analysing divide-and-conquer algorithms

Recall the divide-and-conquer paradigm:

Divide the problem into a number of subproblems that are smaller instances of the same problem.
Conquer the subproblems by solving them recursively.

Base case: If the subproblem are small enough, just solve them by brute force.
Combine the subproblem solutions to give a solution to the original problem.

We use recurrences to characterise the running time of a divide-and-conquer algorithm. Solving the recurrence gives us the asymptotic running time.

A recurrence is a function defined in terms of

- one or more base cases, and
- itself, with smaller arguments

Main technical issues with recurrences

Floors and ceilings: The recurrence describing worst-case running time of Merge-Sort is really

\[ T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1 \end{cases} \]

Exact vs. asymptotic functions Sometimes we are interested in the exact analysis of an algorithm (as for the Min-Max-Problem), at other times we are concerned with the asymptotic analysis (as for the Sorting Problem).

Boundary conditions Running time on small inputs is bounded by a constant: \( T(n) = \Theta(1) \) for small \( n \). We usually do not mention this constant, as it typically doesn’t change the order of growth of \( T(n) \). Such constants only play a role if we are interested in exact solutions.

When we state and solve recurrences, we often omit floors, ceilings, and boundary conditions, as they usually do not matter.

Examples for recurrences

- \( T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + 1 & \text{if } n > 1 \end{cases} \)
  Solution: \( T(n) = n \).

- \( T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases} \)
  Solution: \( T(n) \approx n \log n + n \).

- \( T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/3) + T(2n/3) + n & \text{if } n > 1 \end{cases} \)
  Solution: \( T(n) = \Theta(n \log n) \).

Recursion trees (quadratic growth)

Draw the unfolding of the recurrence

\( T(n) = n + 4T(n/2) \).

We exploited that \( 1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 = \Theta(2^k) \).
Recursion trees (quasi-linear and linear growth)

What about the “merge-sort” recurrence

\[ T(n) = n + 2T(n/2) \]

- Again the height of the tree is \( \log n \).
- However now the “workload” of each level is equal to \( n \). So all workloads are the same.
So here we get
\[ T(n) = \Theta(n \cdot \log n). \]

And what about the recurrence

\[ T(n) = 1 + 2T(n/2) \]

- Again the height of the tree is \( \log n \).
- The “workload” of the level is 1, 2, 4, 8, \ldots, \( 2^{\log n} \). Back to the original method, we can exploit the exponential sum.
So here we get
\[ T(n) = \Theta(n). \]

In other words

We need to give an equation for \( T(n) \) of the form

\[ T(n) = b^x \cdot T(n/b) + \Theta(n^c), \]

where the \( x \) you have to find: \( x = \log_b a \).

Then \( T(n) \) is bounded asymptotically as follows:
- If \( x < c \) then \( T(n) = \Theta(n^c) \).
- If \( x = c \) then \( T(n) = \Theta(n^c \log n) \).
- If \( x > c \) then \( T(n) = \Theta(n^c) \).

The meaning of the three parameters in a divide-and-conquer scheme:
- \( a = b^x \): the number of subproblems to be solved
- \( b \): how often the subproblems (all of the same size) fit into the full problem
- \( c \): power in the runtime of combination-computation.

Master Theorem (simplified version)

Let \( a \geq 1 \) and \( b > 1 \) and \( c \geq 0 \) be constants.

Let \( T(n) \) be defined by the recurrence

\[ T(n) = aT(n/b) + \Theta(n^c), \]

where \( n/b \) represents either \( \lfloor n/b \rfloor \) or \( \lceil n/b \rceil \).

Then \( T(n) \) is bounded asymptotically as follows:
- If \( c < \log_b a \) then \( T(n) = \Theta(n^{\log_b a}) \).
- If \( c = \log_b a \) then \( T(n) = \Theta(n^c \log n) \).
- If \( c > \log_b a \) then \( T(n) = \Theta(n^c) \).

(General version: CLRS, Thm 4.1, p94.)

Using the Master Theorem

- The runtime for MIN-MAX satisfies the recurrence:
  \[ T(n) = 2T(n/2) + \Theta(1). \]
  The Master Theorem (case 1) applies:
  \( a = b = 2 \) and \( c = 0 < 1 = \log_b a, \)
  giving \( T(n) = \Theta(n^{\log_b a}) = \Theta(n). \)

- The runtime for MERGE-SORT satisfies the recurrence:
  \[ T(n) = 2T(n/2) + \Theta(n). \]
  The Master Theorem (case 2) applies:
  \( a = b = 2 \) and \( c = 1 = \log_b a, \)
  giving \( T(n) = \Theta(n^{\log_b n}) = \Theta(n \log n). \)
Easy decision between the three cases

Consider (again)  

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^c).$$

The main question to start with is always:

Which of the three cases applies?

Apparently you needed to compute $x = \log_b a$ for that. But it is actually easier:

- If $b^c < a$ then Case 1 applies.
- If $b^c = a$ then Case 2 applies.
- If $b^c > a$ then Case 3 applies.

(Try to understand why this holds — it’s easy.)
Computing the two largest entries of an array

Develop a Divide-and-Conquer algorithm, which for

input array A
computes the largest and second-largest entry.

That’s easy (similar to Min-Max-computation):

1. Divide the array A into two equal parts.
2. Compute the largest and second-largest entries recursively for both sub-arrays (this yields altogether four numbers).
3. Merge the two sorted lists of length two into one sorted list, and return the first two numbers of that list.
4. Recursion basis: If A has length 1, return that number twice, if it has length 2, sort it and return the result.

Finding all different entries

Find a D&D-algorithm, which for

input array A
computes an array with all the different entries.

Again, that’s easy:

1. Divide the array A into two equal parts.
2. Compute the different entries for each part — sorted!
3. Use the Merge algorithm to merge the two parts, removing doubled elements.
4. Recursion basis: IF A has length 1, return that number.

Analysis

What’s the recurrence?

\[ T(n) = 2T(n/2) + 1 \]

What’s the solution?

\[ T(n) = \Theta(n). \]

First case.

What’s the recurrence?

\[ T(n) = 2T(n/2) + n \]

What’s the solution?

\[ T(n) = \Theta(n \log n). \]

Second case.

Now we could have achieved that by first using MergeSort, and then removing duplicates.

Is our algorithm nevertheless better?
Having only few different elements

Assume we have at most \( k \) different elements in \( A \).

For example, \( k = 1 \) means all entries are equal.

We consider \( k \) as constant.

What is now the recurrence?

\[
T(n) = 2T(n/2) + 1
\]

What’s the solution?

\[
T(n) = \Theta(n).
\]

First case.

Remarks on Merge-Sort

- **Stability**
  - For many sorting-applications, the objects to be sorted consist of a key which provides the sorting criterion, and a lot of other data; for example the last name as part of an employee-record.
  - Then it is quite natural that different objects have the same key. Often, such arrays are then pre-sorted according to other criterions.
  - “Stability” of a sorting algorithm is now the property that the order of equal elements (according to their keys) is not changed.
  - Merge-Sort is stable (at least in our implementation — provided we take the left element in case of equality!).
  - Also Insertion-Sort is stable.

**Remarks on Merge-Sort** (cont.)

- **In-place**
  - A sorting algorithms sorts “in-place”, if besides the given array and some auxiliary data it doesn’t need more memory. This is important if the array is very large (say, \( n \approx 10^9 \)).
  - Insertion-Sort is in-place, while our algorithm for Merge-Sort is not (needing \( \approx 2n \) memory cells). One can make Merge-Sort in-place, but this (apparently) only with a complicated algorithm, which in practice seems not to be applied. If in-place sorting is required, then often one uses “Heap-Sort”.

- **Already sorted**
  - If the array is already sorted, then only \( n - 1 \) comparisons are needed (however overall it still needs time \( \Theta(n \log n) \) because of the swapping, and it stills needs space \( 2n \)).

- **Combination-cost**
  - The general combination-cost of Merge-Sort (due to the swapping) is somewhat higher than what can be achieved with “Quick-Sort”, which typically is the default sorting-algorithm in libraries.
Addendum: The minimal numbers of comparisons

Let $S(n)$ be the minimum number of comparisons that will (always!) suffice to sort $n$ elements (using only comparisons between the elements, and no other properties of them). It holds

$$S(N) \geq \lceil \log(n!) \rceil = \Theta(n \log n).$$

This is the so-called information-theoretic lower bound: It follows by observing that the $n!$ many ordering of $1, \ldots, n$ need to be handled, where every comparison establishes 1 bit of information.

The initial known values for $S(n)$ and the lower bounds are:

<table>
<thead>
<tr>
<th>$n$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S(n)$</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>16</td>
<td>19</td>
<td>22</td>
<td>26</td>
<td>30</td>
<td>34</td>
<td>38</td>
<td>42</td>
</tr>
<tr>
<td>$\lceil \log(n!) \rceil$</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>16</td>
<td>19</td>
<td>22</td>
<td>26</td>
<td>29</td>
<td>33</td>
<td>37</td>
<td>41</td>
</tr>
</tbody>
</table>

The first open value is $S(16)$ (see http://oeis.org/A036604).