Week 3

Divide and Conquer

General remarks

- We start with a general approach to bring down time-complexity dramatically by an important algorithmic paradigm: **Divide-and-Conquer**.
- First we consider a simple example, the Min-Max computation.
- An important example for Divide-and-Conquer is **Merge Sort**, which we discuss next.

Reading from CLRS for week 3

- Chapter 2, Section 3

Divide-and-Conquer Approach

There are many ways to design algorithms.

For example, Insertion-Sort is incremental: having sorted $A[1 \mathinner{..} j-1]$, place $A[j]$ correctly, so that $A[1 \mathinner{..} j]$ is sorted.

Divide-and-Conquer Approach

There are many ways to design algorithms.

For example, Insertion-Sort is incremental: having sorted $A[1\,..\,j-1]$, place $A[j]$ correctly, so that $A[1\,..\,j]$ is sorted.

Divide-and-Conquer is another common approach:

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.

Naive Min-Max

Find minimum and maximum of a list A of $n \geq 1$ numbers.

```
NAIVE-MIN-MAX(A)
least = A[1]
for i = 2 to A.length
    if A[i] < least
        least = A[i]
greatest = A[1]
for i = 2 to A.length
    if A[i] > greatest
        greatest = A[i]
return (least, greatest)
```

Naive Min-Max

Find minimum and maximum of a list A of $n \geq 1$ numbers.

```
NAIVE-MIN-MAX(A)
1  least = A[1]
2  for i = 2 to A.length
3      if A[i] < least
4          least = A[i]
5  greatest = A[1]
6  for i = 2 to A.length
7      if A[i] > greatest
8          greatest = A[i]
9  return (least, greatest)
```

The **for**-loop on line 2 makes $n-1$ comparisons in line 3, as does the **for**-loop on line 6 (in line 7), making a total of $\underline{\underline{2n-2}}$ comparisons.

Naive Min-Max

Find minimum and maximum of a list A of $n \geq 1$ numbers.

```
NAIVE-MIN-MAX($A$)
1   $least = A[1]$
2   for $i = 2$ to $A.length$
3       if $A[i] < least$
4           $least = A[i]$
5   $greatest = A[1]$
6   for $i = 2$ to $A.length$
7       if $A[i] > greatest$
8           $greatest = A[i]$
9   return $(least, greatest)$
```

The **for**-loop on line 2 makes $n-1$ comparisons in line 3, as does the **for**-loop on line 6 (in line 7), making a total of $\underline{\underline{2n-2}}$ comparisons.

Can we do better? Yes!

Divide-and-Conquer Min-Max

As we are dealing with subproblems, we state each subproblem as computing minimum and maximum of a subarray $A[p\,..\,q]$. Initially, $p = 1$ and $q = A.length$, but these values change as we recurse through subproblems.

Divide-and-Conquer Min-Max

As we are dealing with subproblems, we state each subproblem as computing minimum and maximum of a subarray $A[p\,..\,q]$. Initially, $p = 1$ and $q = A.\mathit{length}$, but these values change as we recurse through subproblems.

To compute minimum and maximum of $A[p\,..\,q]$:

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

Divide-and-Conquer Min-Max

As we are dealing with subproblems, we state each subproblem as computing minimum and maximum of a subarray $A[p\,..\,q]$. Initially, $p = 1$ and $q = A.\mathit{length}$, but these values change as we recurse through subproblems.

To compute minimum and maximum of $A[p\,..\,q]$:

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

Conquer by recursively computing minimum and maximum of the two subarrays $A[p\,..\,r]$ and $A[r{+}1\,..\,q]$.

Divide-and-Conquer Min-Max

As we are dealing with subproblems, we state each subproblem as computing minimum and maximum of a subarray $A[p\,..\,q]$. Initially, $p = 1$ and $q = A.\mathit{length}$, but these values change as we recurse through subproblems.

To compute minimum and maximum of $A[p\,..\,q]$:

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

Conquer by recursively computing minimum and maximum of the two subarrays $A[p\,..\,r]$ and $A[r{+}1\,..\,q]$.

Combine by computing the overall minimum as the min of the two recursively computed minima, similar for the overall maximum.

Divide-and-Conquer Min-Max Algorithm

Initially called with $\textsc{Min-Max}(A, 1, A.\mathit{length})$.

```
$\textsc{Min-Max}(A, p, q)$
 1  if $p = q$
 2      return $(A[p], A[p])$
 3  if $p = q-1$
 4      if $A[p] < A[q]$
 5          return $(A[p], A[q])$
 6      else return $(A[q], A[p])$
 7  $r = \lfloor (p+q)/2 \rfloor$
 8  $(min1, max1) = \textsc{Min-Max}(A, p, r)$
 9  $(min2, max2) = \textsc{Min-Max}(A, r+1, q)$
10  return $(\min(min1, min2), \max(max1, max2))$
```

- In line 7, r computes the halfway point of $A[p\,.\,.\,q]$. For example, for $p = 1$, $q = n$, and $n = 1, 2, 3, 4, 5, 6$ we get $r = 1, 1, 2, 2, 3, 3$.
- $n = q - p + 1$ is the number of elements from which we compute the min and max.
- $r - p + 1 = \lceil n/2 \rceil$ is the size of the first (left) subarray, $q - r = \lfloor n/2 \rfloor$ the size of the second (right) subarray.

Solving the Min-Max Recurrence

Let $T(n)$ be the number of comparisons made by $\textsc{Min-Max}(A, p, q)$, where $n = q-p+1$ is the number of elements from which we compute the min and max.

Then $T(1) = 0$, $T(2) = 1$, and for $n > 2$:

$$T(n) \ = \ T(\lceil n/2 \rceil) \ + \ T(\lfloor n/2 \rfloor) \ + \ 2.$$

This is a Recurrence:

- Some initial values for $T(n)$ are given (here for $n \in \{0, 1\}$).
- All further values of $T(n)$ are computed **recursively**.

Unfolding the recursion for Min-Max

We have

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + 2 & \text{else} \end{cases}.$$

1. $T(1) = 0$
2. $T(2) = 1$
3. $T(3) = T(2) + T(1) + 2 = 1 + 0 + 2 = 3$
4. $T(4) = T(2) + T(2) + 2 = 1 + 1 + 2 = 4$
5. $T(5) = T(3) + T(2) + 2 = 3 + 1 + 2 = 6$
6. $T(6) = T(3) + T(3) + 2 = 3 + 3 + 2 = 8$
7. $T(7) = T(4) + T(3) + 2 = 4 + 3 + 2 = 9$
8. $T(8) = T(4) + T(4) + 2 = 4 + 4 + 2 = 10$
9. $T(9) = T(5) + T(4) + 2 = 6 + 4 + 2 = 12$
10. $T(10) = T(5) + T(5) + 2 = 6 + 6 + 2 = 14.$
11. $T(11) = T(6) + T(5) + 2 = 8 + 6 + 2 = 16.$
12. $T(12) = T(6) + T(6) + 2 = 8 + 8 + 2 = 18.$

We guess $T(n) \approx \frac{3}{2}n$.

Solving the Min-Max Recurrence

Claim

$$T(n) = \tfrac{3}{2}n - 2 \quad \textit{for} \quad n = 2^k \geq 2, \textit{ i.e., powers of } 2.$$

Solving the Min-Max Recurrence

Claim

$$T(n) = \tfrac{3}{2}n - 2 \quad \textit{for} \quad n = 2^k \geq 2, \textit{ i.e., powers of } 2.$$

Proof.

The proof is by induction on k (using $n = 2^k$).
Base case: true for $k = 1$, as $T(2^1) = 1 = \frac{3}{2} \cdot 2^1 - 2$.
Induction step: Assume $T(2^k) = \frac{3}{2}2^k - 2$. We want to derive $T(2^{k+1}) = \frac{3}{2}2^{k+1} - 2$. And indeed:

$$T(2^{k+1}) \;=\; T(\left\lceil \frac{2^{k+1}}{2} \right\rceil) + T(\left\lfloor \frac{2^{k+1}}{2} \right\rfloor) + 2 \;=\; 2T(2^k) + 2$$
$$=\; 2\left(\frac{3}{2}2^k - 2\right) + 2 \;=\; 3 \cdot 2^k - 4 + 2 \;=\; \frac{3}{2}2^{k+1} - 2 \qquad \square$$

Solving the Min-Max Recurrence (cont'd)

Some remarks:

1. If we replace line 7 of the algorithm by $r = p+1$ (thus the first subarray has size two), then the resulting runtime $T'(n)$ satisfies $T'(n) = \lceil \frac{3n}{2} \rceil - 2$ for all $n > 0$.
2. For example, $T'(6) = 7$ whereas $T(6) = 8$.
3. It can be shown that at least $\lceil \frac{3n}{2} \rceil - 2$ comparisons are necessary in the worst case to find the maximum and minimum of n numbers for any comparison-based algorithm: this is thus a lower bound on the problem.
4. Hence this (last) algorithm is provably optimal.

HOWEVER: What is the source of the improvement from $2n$ to $\frac{3}{2}n$?!
We should understand this better.

Main 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.

Main 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 $A[p\,..\,q]$. Initially, $p = 1$ and $q = A.\mathit{length}$, but these values change again as we recurse through subproblems.

Main 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 $A[p \,..\, q]$. Initially, $p = 1$ and $q = A.\mathit{length}$, but these values change again as we recurse through subproblems.

To sort $A[p \,..\, q]$:

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.

Main example: Merge-Sort

```
MERGE-SORT($A, p, q$)
1  if $p < q$                              // check for base case
2        $r = \lfloor (p+q)/2 \rfloor$              // 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 $A[p \mathinner{.\,.} r]$ and subarray $A[r+1 \mathinner{.\,.} q]$ 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 $A[p \mathinner{.\,.} q]$.

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

Example

1. 1 3 4 8 9 — 0 2 5 6 8 $\rightsquigarrow$ []

Example

1. 1 3 4 8 9 — 0 2 5 6 8 $\rightsquigarrow$ []
2. 1 3 4 8 9 — 2 5 6 8 $\rightsquigarrow$ [0]

Example

1. 1 3 4 8 9 — 0 2 5 6 8 $\rightsquigarrow$ $[]$
2. 1 3 4 8 9 — 2 5 6 8 $\rightsquigarrow$ $[0]$
3. 3 4 8 9 — 2 5 6 8 $\rightsquigarrow$ $[0, 1]$

Example

1. 1 3 4 8 9 — 0 2 5 6 8 $\rightsquigarrow$ $[]$
2. 1 3 4 8 9 — 2 5 6 8 $\rightsquigarrow$ $[0]$
3. 3 4 8 9 — 2 5 6 8 $\rightsquigarrow$ $[0, 1]$
4. 3 4 8 9 — 5 6 8 $\rightsquigarrow$ $[0, 1, 2]$

Example

1. $1\ 3\ 4\ 8\ 9 — 0\ 2\ 5\ 6\ 8 \rightsquigarrow []$
2. $1\ 3\ 4\ 8\ 9 — 2\ 5\ 6\ 8 \rightsquigarrow [0]$
3. $3\ 4\ 8\ 9 — 2\ 5\ 6\ 8 \rightsquigarrow [0, 1]$
4. $3\ 4\ 8\ 9 — 5\ 6\ 8 \rightsquigarrow [0, 1, 2]$
5. $4\ 8\ 9 — 5\ 6\ 8 \rightsquigarrow [0, 1, 2, 3]$

Example

1. 1 3 4 8 9 — 0 2 5 6 8 $\rightsquigarrow$ []
2. 1 3 4 8 9 — 2 5 6 8 $\rightsquigarrow$ [0]
3. 3 4 8 9 — 2 5 6 8 $\rightsquigarrow$ [0, 1]
4. 3 4 8 9 — 5 6 8 $\rightsquigarrow$ [0, 1, 2]
5. 4 8 9 — 5 6 8 $\rightsquigarrow$ [0, 1, 2, 3]
6. 8 9 — 5 6 8 $\rightsquigarrow$ [0, 1, 2, 3, 4]

Example

1. 1 3 4 8 9 — 0 2 5 6 8 $\rightsquigarrow$ $[]$
2. 1 3 4 8 9 — 2 5 6 8 $\rightsquigarrow$ $[0]$
3. 3 4 8 9 — 2 5 6 8 $\rightsquigarrow$ $[0, 1]$
4. 3 4 8 9 — 5 6 8 $\rightsquigarrow$ $[0, 1, 2]$
5. 4 8 9 — 5 6 8 $\rightsquigarrow$ $[0, 1, 2, 3]$
6. 8 9 — 5 6 8 $\rightsquigarrow$ $[0, 1, 2, 3, 4]$
7. 8 9 — 6 8 $\rightsquigarrow$ $[0, 1, 2, 3, 4, 5]$

Example

1. $1\ 3\ 4\ 8\ 9 — 0\ 2\ 5\ 6\ 8 \rightsquigarrow []$
2. $1\ 3\ 4\ 8\ 9 — 2\ 5\ 6\ 8 \rightsquigarrow [0]$
3. $3\ 4\ 8\ 9 — 2\ 5\ 6\ 8 \rightsquigarrow [0, 1]$
4. $3\ 4\ 8\ 9 — 5\ 6\ 8 \rightsquigarrow [0, 1, 2]$
5. $4\ 8\ 9 — 5\ 6\ 8 \rightsquigarrow [0, 1, 2, 3]$
6. $8\ 9 — 5\ 6\ 8 \rightsquigarrow [0, 1, 2, 3, 4]$
7. $8\ 9 — 6\ 8 \rightsquigarrow [0, 1, 2, 3, 4, 5]$
8. $8\ 9 — 8 \rightsquigarrow [0, 1, 2, 3, 4, 5, 6]$

Example

1. $1\ 3\ 4\ 8\ 9 — 0\ 2\ 5\ 6\ 8 \leadsto []$
2. $1\ 3\ 4\ 8\ 9 — 2\ 5\ 6\ 8 \leadsto [0]$
3. $3\ 4\ 8\ 9 — 2\ 5\ 6\ 8 \leadsto [0, 1]$
4. $3\ 4\ 8\ 9 — 5\ 6\ 8 \leadsto [0, 1, 2]$
5. $4\ 8\ 9 — 5\ 6\ 8 \leadsto [0, 1, 2, 3]$
6. $8\ 9 — 5\ 6\ 8 \leadsto [0, 1, 2, 3, 4]$
7. $8\ 9 — 6\ 8 \leadsto [0, 1, 2, 3, 4, 5]$
8. $8\ 9 — 8 \leadsto [0, 1, 2, 3, 4, 5, 6]$
9. $9 — 8 \leadsto [0, 1, 2, 3, 4, 5, 6, 8]$

Example

1. 1 3 4 8 9 — 0 2 5 6 8 $\leadsto$ []
2. 1 3 4 8 9 — 2 5 6 8 $\leadsto$ [0]
3. 3 4 8 9 — 2 5 6 8 $\leadsto$ [0, 1]
4. 3 4 8 9 — 5 6 8 $\leadsto$ [0, 1, 2]
5. 4 8 9 — 5 6 8 $\leadsto$ [0, 1, 2, 3]
6. 8 9 — 5 6 8 $\leadsto$ [0, 1, 2, 3, 4]
7. 8 9 — 6 8 $\leadsto$ [0, 1, 2, 3, 4, 5]
8. 8 9 — 8 $\leadsto$ [0, 1, 2, 3, 4, 5, 6]
9. 9 — 8 $\leadsto$ [0, 1, 2, 3, 4, 5, 6, 8]
10. 9 — $\leadsto$ [0, 1, 2, 3, 4, 5, 6, 8, 8]

Example

1. 1 3 4 8 9 — 0 2 5 6 8 $\rightsquigarrow$ []
2. 1 3 4 8 9 — 2 5 6 8 $\rightsquigarrow$ [0]
3. 3 4 8 9 — 2 5 6 8 $\rightsquigarrow$ [0, 1]
4. 3 4 8 9 — 5 6 8 $\rightsquigarrow$ [0, 1, 2]
5. 4 8 9 — 5 6 8 $\rightsquigarrow$ [0, 1, 2, 3]
6. 8 9 — 5 6 8 $\rightsquigarrow$ [0, 1, 2, 3, 4]
7. 8 9 — 6 8 $\rightsquigarrow$ [0, 1, 2, 3, 4, 5]
8. 8 9 — 8 $\rightsquigarrow$ [0, 1, 2, 3, 4, 5, 6]
9. 9 — 8 $\rightsquigarrow$ [0, 1, 2, 3, 4, 5, 6, 8]
10. 9 — $\rightsquigarrow$ [0, 1, 2, 3, 4, 5, 6, 8, 8]
11. [0, 1, 2, 3, 4, 5, 6, 8, 8, 9]

```
MERGE($A, p, r, q$)
$n_1 = r - p + 1$
$n_2 = q - r$
let $L[1\,..\,n_1{+}1]$ and $R[1\,..\,n_2{+}1]$ be new arrays
for $i = 1$ to $n_1$
    $L[i] = A[p{+}i{-}1]$
for $j = 1$ to $n_2$
    $R[j] = A[r{+}j]$
$L[n_1{+}1] = R[n_2{+}1] = \infty$
$i = j = 1$
for $k = p$ to $q$
    if $L[i] \leq R[j]$
        $A[k] = L[i]$
        $i = i{+}1$
    else $A[k] = R[j]$
        $j = j{+}1$
```

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 Addendum) 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.

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.

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 (only those!) 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.

Analysis

What's the recurrence?

Analysis

What's the recurrence?

$$T(n) = 2T(n/2) + 1$$

What's the solution? Guess it!

Analysis

What's the recurrence?

$$T(n) = 2T(n/2) + 1$$

What's the solution? Guess it!

$$T(n) = \Theta(n).$$

That'll be the "first case" of the Master Theorem.

Finding all different entries

Find a D&C-algorithm, which for

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

Finding all different entries

Find a D&C-algorithm, which for

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

Again, that's easy, when we strengthen the task to compute all different entries in *sorted order*:

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

Analysis

What's the recurrence?

Analysis

What's the recurrence?

$$T(n) = 2T(n/2) + n$$

What's the solution? Recall it!

Analysis

What's the recurrence?

$$T(n) = 2T(n/2) + n$$

What's the solution? Recall it!

$$T(n) = \Theta(n \lg n).$$

Same as MergeSort.

That'll be the "second case" of the Master Theorem next week.

Analysis

What's the recurrence?

$$T(n) = 2T(n/2) + n$$

What's the solution? Recall it!

$$T(n) = \Theta(n \lg n).$$

Same as MergeSort.

That'll be the "second case" of the Master Theorem next week.

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 the algorithm computing the different entries (the previous algorithm).

For example, $k = 1$ means all entries are equal.

We consider k as **constant**.

What is now the recurrence?

Having only few different elements

Assume we have at most k different elements in A, for the algorithm computing the different entries (the previous algorithm).

For example, $k = 1$ means all entries are equal.

We consider k as **constant**.

What is now the recurrence?

Note that now we only have to merge at most $2k$ elements.

Having only few different elements

Assume we have at most k different elements in A, for the algorithm computing the different entries (the previous algorithm).

For example, $k = 1$ means all entries are equal.

We consider k as **constant**.

What is now the recurrence?

Note that now we only have to merge at most $2k$ elements.

$$T(n) = 2T(n/2) + k \approx 2T(n/2) + 1$$

What's the solution?

Having only few different elements

Assume we have at most k different elements in A, for the algorithm computing the different entries (the previous algorithm).

For example, $k = 1$ means all entries are equal.

We consider k as **constant**.

What is now the recurrence?

Note that now we only have to merge at most $2k$ elements.

$$T(n) = 2T(n/2) + k \approx 2T(n/2) + 1$$

What's the solution?

$$T(n) = \Theta(n).$$

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.

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".

Further properties

Already sorted: If the array is already sorted, then only $n-1$ comparisons are needed with Merge-Sort. 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 shifts the burden from the combination- to the division-step, and that in-place). Thus Quick-Sort is typically the basis of the default sorting-algorithm in libraries (though Quick-Sort is not stable).

Finding the best min-max algorithm

For such a relatively simple problem like Min-Max we should be able to understand better where the "$\frac{3}{2}$" comes from. And perhaps for this simple case, a simpler (non-recursive, and thus faster) algorithms can then be developed. And indeed:

The key of the improvement is the special handling of the (additional) base case $n = 2$, with just ONE comparison. Integration of new min-max-results needs TWO comparisons.

Thus the real idea is: Grab always *two more elements*, and integrate their min-max into the old min-max.

The point is:
For two elements one gets
min AND max
with one comparison at the same time.

Finding the best min-max algorithm (cont.)

1. For even n, find the min-max of the first two elements, otherwise take the first element as the min-max.
2. It remains an even number of elements.
3. Iteratively find the min-max of the **next two elements** using 1 comparison, and compute the new overall min-max using 2 further comparisons.

So for every 2 additional elements, 3 comparisons are needed.

- This algorithm uses precisely $\lceil \frac{3}{2} n \rceil - 2$ comparisons.
- And this is **precisely optimal for all** n.

We learn: Here divide-and-conquer provided a good stepping stone for finding a really good algorithm.

Java code for improved min-max algorithm

```
// Compute the array {min,max} for input a:
static int[] minmax(final int[] a) {
  if (a == null) return null;
  final int N = a.length;
  if (N == 0) return new int[]
    {Integer.MAX_VALUE, Integer.MIN_VALUE};
  final int begin;
  int min, max;
  if (N % 2 == 0) {
    begin = 2; final int x=a[0], y=a[1];
    if (x < y) {min=x; max=y;}
    else {min=y; max=x;}
  }
  else { begin = 1; min = a[0]; max = min; }

  assert(0 <= begin && begin <= N);
  assert((N - begin) % 2 == 0);
```

Java code for improved min-max algorithm (cont.)

```
  for (int i = begin; i < N;) {
    final int x = a[i++], y = a[i++];
    if (x < y) {
      if (x < min) min = x;
      if (y > max) max = y;
    }
    else {
      if (y < min) min = y;
      if (x > max) max = x;
    }
  }
  return new int[] {min, max};
}
```

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

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

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

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

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$S(n)$	0	1	3	5	7	10	13	16	19	22	26	30	34	38	42
$\lceil \lg(n!) \rceil$	0	1	3	5	7	10	13	16	19	22	26	29	33	37	41

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