The divide and conquer algorithm design technique
Analysing a divide and conquer algorithm: Mergesort
Counting inversions
Closest pair of points
Refresher on proof by induction

**Lemma**

\[ S(n) = 1 + 2 + \ldots + n = \frac{n(n + 1)}{2} \]

**Proof.**

- Induction on \( n \)
- Base case \( n = 1 \): \( S(1) = 1(1 + 1)/2 = 1 \)
- Induction hypothesis: \( S(n) \) is true for \( i = 1, 2, \ldots, n \)
- Show that \( S(n + 1) \) is true using the induction hypothesis

\[
S(n + 1) = S(n) + n + 1 \\
= \frac{n(n + 1)}{2} + n + 1 \\
= \frac{n(n + 1)}{2} + 2(n + 1) \\
= \frac{n(n + 1) + 2(n + 1)}{2} \\
= \frac{n^2 + n + 2n + 2}{2} \\
= \frac{(n+1)(n+2)}{2} = S(n + 1)
\]
Suppose you have $n$ items of input and the simplest technique to process it would be two nested for loops with a $\Theta(n^2)$ running time.

If $n$ is small then this is fine.

With divide and conquer we instead aim at:

- Divide in linear time the problem into two subproblems with $n/2$ items.
- Solve each subproblem.
- Combine the solutions to the subproblems in linear time into a solution for the $n$ item problem.

The resulting running time becomes $\Theta(n \log n)$.

We will next study Mergesort.
### 4 GHz modern CPU

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>$n \log n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2.5 ns</td>
<td>8.3 ns</td>
<td>25.0 ns</td>
<td>250.0 ns</td>
<td>14.4 ns</td>
<td>256.0 ns</td>
<td>907.2 μs</td>
</tr>
<tr>
<td>11</td>
<td>2.8 ns</td>
<td>9.5 ns</td>
<td>30.2 ns</td>
<td>332.8 ns</td>
<td>21.6 ns</td>
<td>512.0 ns</td>
<td>10.0 ns</td>
</tr>
<tr>
<td>12</td>
<td>3.0 ns</td>
<td>10.8 ns</td>
<td>36.0 ns</td>
<td>432.0 ns</td>
<td>32.4 ns</td>
<td>1.0 μs</td>
<td>119.8 ms</td>
</tr>
<tr>
<td>13</td>
<td>3.2 ns</td>
<td>12.0 ns</td>
<td>42.2 ns</td>
<td>549.2 ns</td>
<td>48.7 ns</td>
<td>2.0 μs</td>
<td>1.6 s</td>
</tr>
<tr>
<td>14</td>
<td>3.5 ns</td>
<td>13.3 ns</td>
<td>49.0 ns</td>
<td>686.0 ns</td>
<td>73.0 ns</td>
<td>4.1 μs</td>
<td>21.8 s</td>
</tr>
<tr>
<td>15</td>
<td>3.8 ns</td>
<td>14.7 ns</td>
<td>56.2 ns</td>
<td>843.8 ns</td>
<td>109.5 ns</td>
<td>8.2 μs</td>
<td>5 min</td>
</tr>
<tr>
<td>16</td>
<td>4.0 ns</td>
<td>16.0 ns</td>
<td>64.0 ns</td>
<td>1.0 μs</td>
<td>164.2 ns</td>
<td>16.4 μs</td>
<td>1 hour</td>
</tr>
<tr>
<td>17</td>
<td>4.2 ns</td>
<td>17.4 ns</td>
<td>72.2 ns</td>
<td>1.2 μs</td>
<td>246.3 ns</td>
<td>32.8 μs</td>
<td>1.0 days</td>
</tr>
<tr>
<td>18</td>
<td>4.5 ns</td>
<td>18.8 ns</td>
<td>81.0 ns</td>
<td>1.5 μs</td>
<td>369.5 ns</td>
<td>65.5 μs</td>
<td>18.5 days</td>
</tr>
<tr>
<td>19</td>
<td>4.8 ns</td>
<td>20.2 ns</td>
<td>90.2 ns</td>
<td>1.7 μs</td>
<td>554.2 ns</td>
<td>131.1 μs</td>
<td>352.0 days</td>
</tr>
<tr>
<td>20</td>
<td>5.0 ns</td>
<td>21.6 ns</td>
<td>100.0 ns</td>
<td>2.0 μs</td>
<td>831.3 ns</td>
<td>262.1 μs</td>
<td>19 years</td>
</tr>
<tr>
<td>30</td>
<td>7.5 ns</td>
<td>36.8 ns</td>
<td>225.0 ns</td>
<td>6.8 μs</td>
<td>47.9 μs</td>
<td>268.4 ms</td>
<td>10^{15} years</td>
</tr>
<tr>
<td>40</td>
<td>10.0 ns</td>
<td>53.2 ns</td>
<td>400.0 ns</td>
<td>16.0 μs</td>
<td>2.8 ms</td>
<td>5 min</td>
<td>10^{31} years</td>
</tr>
<tr>
<td>50</td>
<td>12.5 ns</td>
<td>70.5 ns</td>
<td>625.0 ns</td>
<td>31.2 μs</td>
<td>159.4 ms</td>
<td>3.3 days</td>
<td>10^{47} years</td>
</tr>
<tr>
<td>100</td>
<td>25.0 ns</td>
<td>166.1 ns</td>
<td>2.5 μs</td>
<td>250.0 μs</td>
<td>3 years</td>
<td>10^{13} years</td>
<td>10^{141} years</td>
</tr>
<tr>
<td>1000</td>
<td>250.0 ns</td>
<td>2.5 μs</td>
<td>250.0 μs</td>
<td>250.0 ms</td>
<td>10^{159} years</td>
<td>10^{284} years</td>
<td>huge</td>
</tr>
<tr>
<td>10^4</td>
<td>2.5 μs</td>
<td>33.2 μs</td>
<td>25.0 ms</td>
<td>4 min</td>
<td>huge</td>
<td>huge</td>
<td>huge</td>
</tr>
<tr>
<td>10^5</td>
<td>25.0 μs</td>
<td>415.2 μs</td>
<td>2.5 s</td>
<td>2.9 days</td>
<td>huge</td>
<td>huge</td>
<td>huge</td>
</tr>
<tr>
<td>10^6</td>
<td>250.0 μs</td>
<td>5.0 ms</td>
<td>4 min</td>
<td>8 years</td>
<td>huge</td>
<td>huge</td>
<td>huge</td>
</tr>
<tr>
<td>10^7</td>
<td>2.5 ms</td>
<td>58.1 ms</td>
<td>7 hour</td>
<td>10^4 years</td>
<td>huge</td>
<td>huge</td>
<td>huge</td>
</tr>
<tr>
<td>10^8</td>
<td>25.0 ms</td>
<td>664.4 ms</td>
<td>28.9 days</td>
<td>10^7 years</td>
<td>huge</td>
<td>huge</td>
<td>huge</td>
</tr>
<tr>
<td>10^9</td>
<td>250.0 ms</td>
<td>7.5 s</td>
<td>8 years</td>
<td>10^{10} years</td>
<td>huge</td>
<td>huge</td>
<td>huge</td>
</tr>
</tbody>
</table>
Mergesort is a stable sort algorithm
Running time \( \Theta(n \log n) \)
See `mergesort.c` e.g. in the book
A recurrence relation or just recurrence is a set of equalities or inequalities such as

\[ T(n) = \begin{cases} 
0, & n = 1 \\
2T(n/2) + n, & n > 1 
\end{cases} \]

The value of \( T(n) \) is expressed using smaller instances of itself and a boundary value.

To analyze the running time of a divide and conquer algorithm, recurrences are very natural.

But we want to have an expression for \( T(n) \) in closed form.

Closed form means an expression only involving functions and operations from a generally accepted set — i.e. "common knowledge".

Closed form can also be called explicit form.

So our next goal is to rewrite \( T(n) \) into closed form.
Mergesort recurrence

- \( T(n) = \text{max comparisons to mergesort } n \text{ items} \)
- Mergesort recurrence:

\[
T(n) \leq \begin{cases} 
0, & n = 1 \\
T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n, & n > 1 
\end{cases}
\]

- We initially ignore ceil and floor:

\[
T(n) \leq \begin{cases} 
0, & n = 1 \\
2T(n/2) + n, & n > 1 
\end{cases}
\]

- We also assume \( n \) is a power of 2
- We will show that these simplifications do not affect our running time analysis, i.e. they are valid
Rewriting a recurrence to closed form

- The easiest way to understand what the closed form is, may be to "expand" or "unroll" the recurrence and simply "see" what is happening
- For Mergesort the closed form will be easy to find this way
- Another way is to look at small inputs and try to guess the closed form
- When we have a guess which works for the small inputs, we then prove by induction that our guess is correct
- In both cases we prove our closed form by induction
- We will start with expanding $T(n)$
Expanding the recurrence and count

\[ T(n) \leq \begin{cases} 
0, & n = 1 \\
2T(n/2) + n, & n > 1 
\end{cases} \]

- Assume \( n \) is power of 2
- \( \log_2 n \) levels
- \( n \) comparisons per level
- In total \( n \log n \) comparisons
- \( T(n) = n \log n \)
Proof by induction

Lemma

The recurrence

\[ T(n) = \begin{cases} 
0, & n = 1 \\
2T(n/2) + n, & n > 1 
\end{cases} \]

has the closed form \( T(n) = n \log_2 n \).

Proof.

- Induction on \( n \). Below we use: \( 2^3 = 8 \) and \( \log_2 8 = 3 = (\log_2 16) - 1 \)
- Base case: \( n = 1 \): \( T(1) = 1 \log_2 1 = 0 \)
- Induction hypothesis: assume \( T(n) = n \log_2 n \)
- Show \( T(2n) = (2n) \log_2 (2n) \).
- \( T(2n) = 2T(n) + 2n = 2n \log_2 n + 2n = 2n((\log_2 2n) - 1) + 2n \)
- \( 2n((\log_2 2n) - 1) + 2n = 2n \log_2 2n - 2n + 2n = 2n \log_2 2n \)
Remark about previous proof

- Normally we assume $S(i)$ is true and prove $S(i + 1)$
- On previous slide we did not increment by one but rather doubled our variable
- We could have stated the lemma in terms of $S(i)$ and let $n = 2^i$
- Then we use induction on $i$ and assume $S(i)$ and prove $S(i + 1)$
Proof by induction, removing assumption $n = 2^k$

Lemma

The recurrence

$$T(n) \leq \begin{cases} 
0, & n = 1 \\
T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n, & n > 1 
\end{cases}$$

has the closed form $T(n) \leq n\lceil \log_2 n \rceil$.

Proof.

- Induction on $n$
- Base case: $n = 1$: $T(1) = 1 \log_2 1 = 0$
- Let $n_1 = \lfloor n/2 \rfloor$ and $n_2 = \lceil n/2 \rceil$
- Induction hypothesis: assume true for $n = 1, 2, \ldots, n - 1$
- $T(n) \leq T(n_1) + T(n_2) + n \leq n_1 \log_2 n_1 + n_2 \log_2 n_2 + n$
- Previous line follows from induction hypothesis
Proof by induction, removing assumption $n = 2^k$

**Lemma**

The recurrence

$$T(n) \leq \begin{cases} 
0, & n = 1 \\
T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n, & n > 1 
\end{cases}$$

has the closed form $T(n) \leq n \lceil \log_2 n \rceil$.

**Proof.**

\[
T(n) \leq T(n_1) + T(n_2) + n \\
\leq n_1 \log_2 n_1 + n_2 \log_2 n_2 + n \\
\leq (n_1 + n_2) \log_2 n_2 + n \\
= n \log_2 n_2 + n \\
\leq n(\lceil \log_2 n \rceil - 1) + n \\
= n \lceil \log_2 n \rceil
\]
Looking at small inputs

\[ T(n) \leq \begin{cases} 
0, & n = 1 \\
2T(n/2) + n, & n > 1 
\end{cases} \]

- Let us try out some small values:

\[
\begin{array}{cccccccc}
 n & 1 & 2 & 4 & 8 & 16 & 32 & 64 \\
T(n) & 0 & 2 & 8 & 24 & 64 & 160 & 384 \\
\end{array}
\]

- Can we identify a pattern?

\[
\begin{array}{cccccccc}
 n & 1 & 2 & 4 & 8 & 16 & 32 & 64 \\
T(n) & 0 & 2 & 8 & 24 & 64 & 160 & 384 \\
T(n)/n & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\]

- \( \log_2 n \) is incremented by one when \( n \) is doubled: \( \log_2 2n = 1 + \log_2 n \)
- So \( T(n) = n \log_2 n \) is tempting to try to prove by induction, which we already know is true
Master Theorem

- This is new 2020 and I have not prepared slides for it and I doubt I will have time for it in LP4 2020.
- My focus is to finish all lectures first, and then there are tons of other things which must be done.
Finding people with similar tastes

- Consider a category such as text editor, programming language, preferred tab width, or the 22 Mozart operas.
- To compare how similar tastes within a category three people have, they can rank a list of say 5 operas A-E.
  - Vladimir: A D C E B
  - Alexander: A C B D E
  - Catherine: A B D C E
- All agree opera A is best.
- Who have most similar tastes?
Inversions

Vladimir: A D C E B
Alexander: A C B D E
Catherine: A B D C E

We have 5 positions in each list

Start with Vladimir’s list and label each item 1, 2, ..., 5:
Vladimir: A D C E B
Vladimir: 1 2 3 4 5

Then we put these labels according to Alexander’s ranking:
Alexander: 1 3 5 2 4

\[ a_1 \ a_2 \ a_3 \ a_4 \ a_5 \]

\( i \) and \( j \) are \textbf{inverted} if \( i < j \) and \( a_i > a_j \)

Inversions: (3,2), (5,2), and (5,4)

The fewer inversions, the more similar tastes (obviously)
for (c = i = 0; i < n; i += 1)
    for (j = i+1; j < n; j += 1)
        if (a[i] > a[j])
            c += 1;

printf("%d inversions\n", c);

- Running time is $O(n^2)$
- How can we use divide and conquer to achieve $O(n \log n)$?
  - 1  3  |  5  2  4
- Count inversions in left part
- Count inversions in right part
- Somehow combine these parts and add number of inversions...???
What can we do to simplify the problem?

- 1 3 | 5 2 4
- Assume you know there are no inversions in the left part and two in the right part
- It is OK to “destroy” the array, such as sorting it, if that helps...
- If modifying the array is forbidden, we can always make a copy and work with the copy instead
- Copying the array is fine since that is faster than $O(n \log n)$
- Copying the array is $O(n)$ but memory allocation can be costly so don’t do it too much
- For Mergesort, it is non-trivial to not use a second array
By subarray is meant the part our recursive subproblem is going to work with.

Sorting the subarray **after** counting the inversions may help.

1 3 | 5 2 4

After having counted in the subarrays we have: 1 3 | 2 4 5

Combining two sorted parts can be done in linear time as in Mergesort.

```
  3  2  4  5  |  1
  3  4  5  |  1  2
```

The 2 was inverted with each remaining in left part — only the 3 in this example so one inversion is counted when the parts are combined.

```
  4  5  |  1  2  3
  5  |  1  2  3  4
  |  1  2  3  4  5
```

In total 3 inversions.

Jonas Skeppstedt (jonasskeppstedt.net)
Implementing the $n \log n$ algorithm

- As always: first make a simple reference implementation for verifying correctness
- In this case the $n^2$ algorithm is ideal if used with small inputs
Lab 4 is about the field of **computational geometry**

Consider $n$ points $(x_i, y_i)$ in a plane

We want to find which points are closest

Comparing all points with each other in an $n^2$ algorithm is simple

But comparing points "obviously" far from each other is a waste

How can divide and conquer be used to find an $n \log n$ algorithm?

We cut the plane in two halves and find closest points in each half

We have then three categories of point pairs which can be closest:

1. Point pairs in the left half
2. Point pairs in the right half
3. Point pairs with one point in the left and the other in the right half

Can we find close points from the last category in linear time???
We cut the plane in two halves with 10 points in each half.

We compute the nearest points in each half.

\[ \delta = \min(146, 113) \]

We only have to consider points within \( \delta \) from the vertical line.

If there are none, then \( \delta \) is the answer.

If there are, then they must be checked with points from the other side which also must be within \( \delta \) from the vertical line, of course.
The point $p$ on the vertical line $x_p$ belongs to the left half but there could also be points in the right half with the same $x$-coordinate.

Let the set $S$ consist of all points with a distance within $\delta$ from the line $x_p$, (5 points here).

Clearly it is sufficient to compare only points $q$ and $r$ from $S$ such that $p$ comes from the left half and $q$ from the right part.
Each dashed box has a side of \( \delta/2 \)

How many points can each such box contain at most?

The diagonal of a dashed box is \( \sqrt{2} \times \delta/2 < \delta \)

With two points in a dashed box, their distance would be less than \( \delta \) so at most one point
With at most one point per dashed box, we can do as follows.

Let $S$ be sorted on $y$-coordinates

Each point $p \in S$ is inspected at a time.

The distances from $p$ to each of the next 15 points in $S$ (according to $y$-coordinates) are checked to see if it less than the shortest distance found so far.
What do we need for this?
- Input is a set of $n$ points $P$
- We produce two sorted arrays $P_x$ and $P_y$ before starting our recursion
- We divide $P_x$ into two arrays $L_x$ and $R_x$ (left and right)
- We divide $P_y$ into two arrays $L_y$ and $R_y$
- We solve the two subproblems $(L_x, L_y, n/2)$ and $(R_x, R_y, n/2)$
- Then we compute $\delta$ as the minimum from these subproblems
- Then we create the set $S_y$ from $P_y$
- All dividing and combining can be done in linear time, so we solve this in $\Theta(n \log n)$ time