1 Simplifications

1.1 Logarithms

\[ \log_a (a^x \cdot a^2) = x + 2 \]  \hspace{1cm} (1)

since

\[ a^x \cdot a^y = a^{x+y} \]

and thus

\[ \log_a (a^x) = z, \]

\[ \log_a (a^{x+2}) = x + 2. \]

One can also use

\[ \log(x \cdot y) = \log(x) + \log(y), \]

and thus

\[ \log_a (a^x \cdot a^2) = \log_a (a^x) + \log_a (a^2) = x + 2. \]

Remark: We assume \( a \in \mathbb{R}_{>0} \) and \( x \in \mathbb{R} \).
1.2 Floors and ceilings

For $n \in \mathbb{N}_0$ we have
\[
\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil = n. \tag{2}
\]
Small examples:
\[
\begin{align*}
\left\lfloor \frac{0}{2} \right\rfloor + \left\lceil \frac{0}{2} \right\rceil &= 0 + 0 = 0 \\
\left\lfloor \frac{1}{2} \right\rfloor + \left\lceil \frac{1}{2} \right\rceil &= 1 + 0 = 1 \\
\left\lfloor \frac{2}{2} \right\rfloor + \left\lceil \frac{2}{2} \right\rceil &= 1 + 1 = 2 \\
\left\lfloor \frac{3}{2} \right\rfloor + \left\lceil \frac{3}{2} \right\rceil &= 2 + 1 = 3 \\
\left\lfloor \frac{4}{2} \right\rfloor + \left\lceil \frac{4}{2} \right\rceil &= 2 + 2 = 4 \\
\left\lfloor \frac{5}{2} \right\rfloor + \left\lceil \frac{5}{2} \right\rceil &= 3 + 2 = 5.
\end{align*}
\]
We see:

- For even $n$ we have $\left\lfloor \frac{n}{2} \right\rfloor = \frac{n}{2} = \left\lceil \frac{n}{2} \right\rceil$, since 2 divides $n$, and thus $\frac{n}{2}$ is an integer (recall that for integers $z$ we have $\lfloor z \rfloor = \lceil z \rceil = z$).

  It follows that for even $n$ we have $\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil = \frac{n}{2} + \frac{n}{2} = n$.

- For odd $n$ we have $\left\lfloor \frac{n}{2} \right\rfloor = \frac{n+1}{2}$ and $\left\lceil \frac{n}{2} \right\rceil = \frac{n-1}{2}$, since $\frac{n-1}{2}, \frac{n+1}{2}$ are integers (due to $n$ odd) with

\[
\frac{n-1}{2} = \frac{n}{2} - \frac{1}{2} < \frac{n}{2} < \frac{n}{2} + \frac{1}{2} = \frac{n+1}{2}.
\]

  It follows that for odd $n$ we have $\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil = \frac{n+1}{2} + \frac{n-1}{2} = n$.

1.3 Sums

For $n \in \mathbb{N}_0$ we have
\[
\sum_{i=1}^{n} (i^2 - (i-1)^2) = n^2. \tag{3}
\]
Small examples:

\[
\sum_{i=1}^{0} (i^2 - (i - 1)^2) = 0
\]

\[
\sum_{i=1}^{1} (i^2 - (i - 1)^2) = (1^2 - 0^2) = 1
\]

\[
\sum_{i=1}^{2} (i^2 - (i - 1)^2) = (2^2 - 1^2) + 1 = 4
\]

\[
\sum_{i=1}^{3} (i^2 - (i - 1)^2) = (3^2 - 2^2) + 4 = 9
\]

\[
\sum_{i=1}^{4} (i^2 - (i - 1)^2) = (4^2 - 3^2) + 9 = 16.
\]

Thus we can guess that the sum evaluates to \( n^2 \). Prove by induction is easy, or one uses directly a **telescope sum argument** (exploiting cancellation of neighbouring terms):

\[
\sum_{i=1}^{n} (i^2 - (i - 1)^2) = \left( (n^2 - (n-1)^2) + (n-1)^2 - (n-2)^2 \right) + \cdots + (1^2 - 0^2) = n^2 - 0^2 = n^2.
\]

### 2 Theta-Expressions

\[
5n^3 - 6n^2 + \sqrt{|\sin(n)|} = \Theta(n^3)
\]

since terms of lower order can be removed, and we have

\[
5n^3 = \Theta(n^3) \\
6n^2 = \Theta(n^2) \\
\sqrt{|\sin(n)|} = \Theta(1).
\]

\[
2^n + n^{1000} = \Theta(2^n)
\]

since for every \( a \) and every \( b > 1 \) we have \( n^a = O(b^n) \) (exponential growth is asymptotically stronger than polynomial growth), and thus one can argue

\[
2^n + n^{1000} = 2^n + O(2^n) = O(2^n) \\
2^n + n^{1000} = \Omega(2^n).
\]
\[ 2^n + 3^n = \Theta(3^n) \] (6)

since for \( n \geq 0 \) we have \( 2^n \leq 3^n \), and thus

\[ 2^n + 3^n \leq 2 \cdot 3^n = O(3^n) \]
\[ 2^n + 3^n = \Omega(3^n). \]

### 3 Growth order

We have

- \( 2^{\log n} = n \)
- \( \sqrt{n} = n^{\frac{1}{2}} \)
- \( \log_{10} n = \frac{\log_2 n}{\log_2 10} \), and thus \( 2^{\log_{10} n} = 2^{\frac{\log_2 n}{\log_2 10}} = \left(2^{\log_2 n}\right)^{\frac{1}{\log_2 10}} = n^{\frac{1}{\log_2 10}} \)

where \( \log_2 10 > 3 \) since \( 2^3 = 8 < 10 \).

- \( 2^n > n \) for \( n \geq 0 \), and thus \( 2^n = (2^n)^n > n^n \).

So the sorting of the given functions (of \( n \)) by ascending order of growth is

\[ 2^{\log_{10} n}, \sqrt{n}, 2^{\log n}, n^2, n^3, 2^n, e^n, n!, n^n, 2^{(n^2)} \].

Remarks:

- We use \( \log(x) = \log_2(x) \) (common is also \( \text{ld}(x) = \log_2(x) \) for “logarithm dualis”).
- Common is \( \ln(x) = \log_e(x) \) with \( e = 2.71828182845904 \ldots \) (for “logarithm naturalis”).
- When we use \( \log(x) \), then typically the basis of the logarithm is left “open”. For example in

\[ \log_a(x) = \frac{\log(x)}{\log(a)}, \]

where on the right-hand side actually any basis can be used (but consistently!), that is, more precisely we have for every \( a, b > 1 \):

\[ \log_a(x) = \frac{\log_b(x)}{\log_b(a)}. \]
4 Master theorem

\[ T(n) = 3T\left(\frac{n}{2}\right) + 5 \implies T(n) = \Theta(n^{\log_2 3}). \] (7)

This is case 1 of the Master Theorem, with \( a = 3, b = 2 \) and \( c = 0 \). We can also write it as \( T(n) = 2^{\log_2 3}T\left(\frac{n}{2}\right) + \Theta(1) \).

\[ T(n) = 16T\left(\frac{n}{4}\right) + n^2 \implies T(n) = \Theta(n^2 \cdot \log n). \] (8)

This is case 2, with \( a = 16, b = 4 \) and \( c = 2 \). We can also write it as \( T(n) = 4^2T\left(\frac{n}{4}\right) + n^2 \).

\[ T(n) = T\left(\frac{n}{2}\right) + \sqrt{n} \implies T(n) = \Theta(\sqrt{n}). \] (9)

This is case 3, with \( a = 1, b = 2 \) and \( c = \frac{1}{2} \). We can also write it as \( T(n) = 2^0T\left(\frac{n}{2}\right) + n^{\frac{1}{2}} \).

5 Sorting 4 numbers

When using insertion-sort for sorting 3 numbers, we need \( 2+1 = 3 \) comparisons. In general, for \( n \) items, we need (precisely)

\[ \sum_{i=1}^{n-1} i = \frac{1}{2}n(n - 1) \]

comparisons. This can be achieved also by the following simple function (using C with reference-parameters):

```c
void sort3 (int & a, int & b, int & c) {
    if (b < a) swap(a, b);
    if (c < b) swap(b, c);
    if (b < a) swap(a, b);
}
```

The key ideas for sorting 4 numbers \( a, b, c, d \) with (only) 5 comparisons are:

1. Sort the first three numbers \( a, b, c \) with 3 comparisons.

2. Compare \( d \) with the middle number \( b \): then only further comparison with \( a \) resp. \( c \) is needed.

The C++-code is as follows (this time not modifying the arguments, but returning a vector with 4 elements):
```cpp
vector<int> sort4(int a, int b, int c, const int d) {
    sort3(a, b, c); // now a <= b <= c
    if (d < b) {
        if (d < a) return {d, a, b, c};
        return {a, d, b, c};
    }
    if (d < c) return {a, b, d, c};
    return {a, b, c, d};
}

6 Analysing Insertion-Sort

The number of inversions of an array $A$ of length $n \geq 1$, denoted here by $\text{inv}(A)$, is the number of pairs $(i,j)$ with $1 \leq i < j \leq n$ and $A[i] > A[j]$. (In the solution we drop the simplifying assumptions that $A$ is repetition-free.)

The inversions of $(2, 3, 8, 6, 1)$ are

$$(1, 5), (2, 5), (3, 4), (3, 5), (4, 5).$$

When there are no duplicate elements in $A$, then for readability we can write the elements instead of the indices:

$$(2, 1), (3, 1), (8, 6), (8, 1), (6, 1).$$

Amongst the arrays with $n$ elements, the array $(n, n-1, \ldots, 1)$ has the maximal number of inversions, namely

$$(n - 1) + (n - 2) + \cdots + 1 + 0 = \sum_{i=1}^{n-1} i = \frac{1}{2} n(n - 1).$$

This is maximal over all arrays (which might contain repetitions), which can be proven by induction:

- The statement is true for $n = 1$ (zero inversions).
- Assume $n > 1$, and that it is true for length $n - 1$. Consider any array $A = (a_1, \ldots, a_n)$. Without loss of generality we can assume \(a_1, \ldots, a_n\) = \(1, \ldots, n\).

  (Essential here the claim (in the “without loss of generality”) that repetitions of values do not increase the number of inversions. That actually needs a proof. Harmless is the renaming, bringing arbitrary values into the range from 1 to $n$.)

- Now the number of inversions of $A$ is not decreased if we bring $n$ to the front. So without loss of generality we can assume $a_1 = n$.}
```

6 Analysing Insertion-Sort

The number of inversions of an array $A$ of length $n \geq 1$, denoted here by $\text{inv}(A)$, is the number of pairs $(i,j)$ with $1 \leq i < j \leq n$ and $A[i] > A[j]$. (In the solution we drop the simplifying assumptions that $A$ is repetition-free.)

The inversions of $(2, 3, 8, 6, 1)$ are

$$(1, 5), (2, 5), (3, 4), (3, 5), (4, 5).$$

When there are no duplicate elements in $A$, then for readability we can write the elements instead of the indices:

$$(2, 1), (3, 1), (8, 6), (8, 1), (6, 1).$$

Amongst the arrays with $n$ elements, the array $(n, n-1, \ldots, 1)$ has the maximal number of inversions, namely

$$(n - 1) + (n - 2) + \cdots + 1 + 0 = \sum_{i=1}^{n-1} i = \frac{1}{2} n(n - 1).$$

This is maximal over all arrays (which might contain repetitions), which can be proven by induction:

- The statement is true for $n = 1$ (zero inversions).
- Assume $n > 1$, and that it is true for length $n - 1$. Consider any array $A = (a_1, \ldots, a_n)$. Without loss of generality we can assume $a_1, \ldots, a_n$ = $1, \ldots, n$.

  (Essential here the claim (in the “without loss of generality”) that repetitions of values do not increase the number of inversions. That actually needs a proof. Harmless is the renaming, bringing arbitrary values into the range from 1 to $n$.)

- Now the number of inversions of $A$ is not decreased if we bring $n$ to the front. So without loss of generality we can assume $a_1 = n$.}
• Now we can apply the induction hypothesis to \((a_2, \ldots, a_n)\) and deduce that there can be at most \(\frac{1}{2}(n-1)(n-2)\) inversions in it.

• \(a_1\) contributes further \(n-1\) inversions, and so altogether we have at most 
\[
\frac{1}{2}(n-1)(n-2) + (n-1) = (n-1)(\frac{1}{2}(n-2) + 1) = (n-1)\frac{1}{2}n
\]

inversions. QED

Recall the analysis of insertion-sort in week 1:

1. Eight different operations were identified, with associated costs \(c_1, \ldots, c_8\), corresponding to the eight lines of code. (These \(c_i\) are parameters, depending on the execution environment; only \(c_3\) is known in advance, namely \(c_3 = 0\).)

2. The total run-time on input \(A\) is (precisely)
\[
T(A) = c_1 \cdot n + (c_2 + c_4 + c_8) \cdot (n-1) + c_5 \cdot \sum_{j=2}^{n} t_j + (c_6 + c_7) \cdot \sum_{j=2}^{n} (t_j - 1) = \\
c_1 \cdot n + (c_2 + c_4 + c_8 - c_6 - c_7) \cdot (n-1) + (c_5 + c_6 + c_7) \cdot \sum_{j=2}^{n} t_j.
\]

3. The number \(t_j\) for \(j \in \{2, \ldots, n\}\) here is just the number of executions of line 5 (the \texttt{while}-check) for the execution of the outer loop with loop-variable value \(j\).

The crucial observation now is that \(t_j\) is

1 plus the number of inversions \(A[j]\) is involved

which are to the left of position \(j\).

We get
\[
\sum_{j=2}^{n} t_j = \text{inv}(A) + n - 1.
\]

So
\[
T(A) = c_1 \cdot n + (c_2 + c_4 + c_8 - c_6 - c_7) \cdot (n-1) + (c_5 + c_6 + c_7) \cdot \text{inv}(A) + n - 1 = \\
c_1 \cdot n + (c_2 + c_4 + c_5 + c_8) \cdot (n-1) + (c_5 + c_6 + c_7) \cdot \text{inv}(A) = \\
(c_5 + c_6 + c_7) \cdot \text{inv}(A) + (c_1 + c_2 + c_4 + c_5 + c_8) \cdot n - (c_2 + c_4 + c_5 + c_8) \cdot n.
\]

Thus the run-time of insertion sort on input \(A\) is \(\Theta(n + \text{inv}(A))\). (Note that this means here that there are constants \(\alpha, \beta\), depending on the execution environment, such that \(\alpha \cdot (n + \text{inv}(A)) \leq T(A) \leq \beta \cdot (n + \text{inv}(A))\). Also note that here we have the precise dependency on the input, not just on the input-length \(n\), and so it’s not a worst-case statement, but an exact-case statement.)

Finally let’s consider how to computer \(\text{inv}(A)\). We could compute it via insertion-sort in time \(O(n^2)\). Can we do better?
• Insertion-sort encounters ("repairs") the inversions one by one.

• We need to process them in large chunks.

• Merge-sort also handles inversions in larger chunks — perhaps we can adapt it?

Consider an array \( A \) (again, length \( n \), indices from 1 to \( n \)) and some \( 1 \leq r \leq n \). Let \( A[p, q] \) be the subarray for indices \( p \leq i \leq q \). Now the fundamental observation, enabling us to use divide and conquer, is

\[
\text{inv}(A) = \text{inv}(A[1, r]) + \text{inv}(A[r + 1, n]) +
|\{(i, j) : i \in \{1, \ldots, r\} \land j \in \{r + 1, \ldots, n\} \land i < j \land A[j] < A[i]\}|.
\]

In words: the inversions in the whole array \( A \) are the inversions in the sub-arrays \( A[1, r] \) and \( A[r + 1, n] \) plus the inversions which "cross" these two sub-sections (and these three cases do not overlap).

The problem is now how to count the cross-inversions:

• We need to employ the merge-step of merge-sort (recall the script from week 3).

• Every time we merge-in an element from the right sub-array \( R \) (above: \( A[r + 1, n] \)), we get as many inversions as there are remaining elements in the left sub-array \( L \).

• This also works with duplicated elements: The merge-procedure delays insertion of elements from the right sub-array \( R \) as long as possible — this is needed for the property of being a stable sorting algorithm, and it also makes sure, that an element from \( R \) duplicated in \( L \) does not yield an inversion.

Instead of \( \text{MergeSort}(A, 1, n) \) we will now call \( \text{inversions}(A, 0, n) \), which, as before, will sort the array \( A \), and will return the number of inversions of \( A \). C++ code is as follows (the differences to the pseudo-code and to Java-code should be small enough, so that the code can be easily understood; as in C and Java, we use 0-based arrays here, and the right bound is "one-past-the-end"):

```c++
int inversions(int [] A, const int p, const int q) {
    if (p >= q-1) return 0;
    const int r = (p+q)/2;
    const int il = inversions(A, p, r);
    const int ir = inversions(A, r, q);
    const int im = merge(A, p, r, q);
    return il + ir + im;
}
```
int merge(int[] A, const int p, const int r, const int q) {
    const int n1 = r - p, n2 = q - r;
    vector<int> L(A + p, A + r), R(A + r, A + q);
    int i = 0, j = 0, k = p, inv = 0;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) A[k++] = L[i++];
        else {
            A[k++] = R[j++];
            inv += n1 - i;
        }
    }
    if (i == n1) while (k < q) A[k++] = R[j++];
    else while (k < q) A[k++] = L[i++];
    return inv;
}