Maths introduction: Exponentials

Basic exponentials (for a positive integer n):

$$a^n = a \times a \times \ \dots \ \times a \qquad \text{(repeated } n \text{ times)}$$

Example: $2^3 = 2 \times 2 \times 2 = 8, \quad 10^1 = 10, \quad 10^2 = 100.$

Exponentiation is also defined for negative numbers or fractions:

$$a^{-n} = \frac{1}{a^n} \qquad \text{and} \qquad a^{1/n} = \sqrt[n]{a}$$

Basic rules:

$$a^m \times a^n = a^{m+n} \qquad (a^m)^n = a^{m \times n}$$

$$a^m / a^n = a^{m-n} \qquad a^0 = 1$$

Example: $9^{1/2} = 3 \quad 2^{-3} = 1/8$

$$\sqrt{5} \times \sqrt{5} = 5^{\frac{1}{2}} \times 5^{\frac{1}{2}} = 5^{\frac{1}{2}+\frac{1}{2}} = 5^1 = 5$$

$$16^{3/2} = (16^{1/2})^3 = 4^3 = 64$$

Maths introduction: Logarithms

Basic idea (for numbers a and b):

$$\log_a b = c \quad \text{such that} \quad a^c = b$$

Example:
$$\begin{aligned}
&\log_{10} 1000000 = 6\\
&\log_{10} 0.0001 = -4\\
&\log_2 32 = 5\\
&\log_8 32 = \log_8(2^5) = \log_8\left((\sqrt[3]{8})^5\right) = 5/3
\end{aligned}$$

From the rules for exponentiation we derive the following rules:

$$\begin{aligned}
\log_a(bc) &= \log_a b + \log_a c\\
\log_a(b/c) &= \log_a b - \log_a c\\
\log_a(b^n) &= n \times \log_a b
\end{aligned}
\qquad\qquad
\log_a b = \frac{\log_c b}{\log_c a}$$

$\log_{10} 1000000 = 6$ because $10^6 = 1000000$ and

$\log_{10} 0.0001 = -4$ because $10^{-4} = \frac{1}{10^4} = \frac{1}{10000} = 0.0001$.

Proof of the last rule (for curious students):

Observe that $b = a^{\log_a b}$ (this is exactly how logarithms are defined).

Therefore

$$\log_c b = \log_c(a^{\log_a b}) = \log_a b \times \log_c a \qquad (\star)$$

where the last equality follows from the rule: $\log_c(a^n) = n \times \log_c a$.

Finally, we obtain that $\log_a b = \frac{\log_c b}{\log_c a}$ from $(\star)$ by dividing both sides by $\log_c a$.

Example

Starting amount on your bank account £1000, interest is 1%, in five years you have 1000×1.01^5 pounds.

To compute when you reach £1100, you ask for which n is

$$1000 \times 1.01^n \geq 1100$$

or, equivalently, $1.01^n \geq 1.1$ (i.e. after dividing by 1000).

The actual number of years is $n \geq \log_{1.01} 1.1$.

(In fact, it is the ceiling $\lceil \log_{1.01} 1.1 \rceil$. You get the money at the end of the year.)

Time Complexity

Linear search (worst case complexity)

```
1  int search (int[] array, int x) {
2    int n = array.length;
3    int i = 0;
4
5    while (i < n) {            // iterate over the elements
6      if (array[i] == x) {
7        return i;              // found it!
8      } else {
9        i = i + 1;             // try the next one
10     }
11   }
12
13   return -1;                 // the value not found
14 }
```

Worst case: the value `x` is not in the array.

Number of steps: $2 + n \times (1 + 1 + 1) + 2 = 3n + 4$

(2nd and 3rd lines, then `n` -times 5th, 6th and 9th lines, and finally the 5th and 13th line)

Linear search (worst case complexity), recursively

```
 1 int search (int[] array, int x) {
 2   search_rec(array, 0, x);  // iterate over the elements
 3 }
 4
 5 int search_rec(int[] array, int i, int x) {
 6   if (i == array.length)
 7     return -1;              // the value not found
 8
 9   if (array[i] == x)
10     return i;               // found it!
11
12   int i_next = i + 1;       // try the next one
13   return search_rec(array, i_next, x);
14 }
```

Worst case: the value `x` is not in the array.

Number of steps: $1 + n \times (1 + 1 + 1 + 1) + 1 = 4n + 2$

(2nd line, then `n` -times 6th, 9th, 12th and 13th lines, and finally 6th line)

What is the difference between the two?

From the theoretical perspective we are more interested in *proportional growth* of the number of steps rather than the actual number of steps. This is because the actual speed depends on

- the hardware on which it runs,
- programming language used (or its compiler),
- how well is the implementation optimised, ...

Therefore, we prefer to compare *algorithms* in terms their *complexity classes*, that is, we do not care about constant factors.

$\Longrightarrow$ Both the non-recursive and recursive versions of the search algorithm belong to $\mathcal{O}(n)$, that is, *linear complexity class*.

In practise, it does not make sense to count the exact number of steps because

- it gets trickier and trickier as your program grows, and also
- there are many hidden constants that we can't compute, for example it takes longer to execute `if (i < n)` than `i+1`.

Instead of the number of steps, we care about the proportional growth of the number of steps. For example, if the size of the input doubles, how much longer will the algorithm take to finish? Will it be approximately twice as long? Four times as long? (The first case corresponds to $\mathcal{O}(n)$, the second to $\mathcal{O}(n^2)$).

This is important for the theoretical analysis of your programs. Before actually implementing an algorithm, it is better to estimate its complexity class and based on that you can see if such algorithm is worth implementing or if it is just too slow.

“Big Oh” denotes proportional or less

For an algorithm operating on data of size n, how do we determine its time complexity?

The algorithm is in $\mathcal{O}(n)$ whenever the number of steps is

1. $\leq 4n$
2. $\leq 23492n$
   (Big-Oh doesn’t care about the constants.)
3. $\leq 4n + 3123$
4. $\leq 23492n$, provided that $n \geq 312$
   (Big-Oh doesn’t care about the first few n.)

Examples:

- $\leq 33n^3$ steps is in $\mathcal{O}(n^3)$
- ≤ 15 steps is in $\mathcal{O}(1)$ (constant time)
- $\leq 33n^3 + 2.5n^2 + 444n + 30$ steps is in $\mathcal{O}(n^3)$

1. This generalises to other complexity classes as well. For example, if an algorithm takes $\leq 131 \times n \times \log_{10} n$ steps, then it is in $\mathcal{O}(n \log_{10} n)$. For a proper mathematical definition of $\mathcal{O}(\cdot)$ I recommend you to read Martín Escardó's lecture notes which you can find on Canvas.

2. The case (3) can happen if, for example, the initialization of a data structure takes many steps but then all operations on the data structure are fast.

3. Why is $\leq 33n^3 + 2.5n^2 + 444n + 30$ steps in $\mathcal{O}(n^3)$? Observe that, for $n \geq 1$,

- $2.5n^2 \leq 2.5n^3$ and
- $444n \leq 444n^3$.

Therefore, the number of steps is

$\leq 33n^3 + 2.5n^3 + 444n^3 + 30 = 479.5n^3 + 30 \implies$ the alg. is in $\mathcal{O}(n^3)$.

Remarks

1. Because $\log_2 n = \frac{1}{\log_{10} 2} \cdot \log_{10} n$ and $\frac{1}{\log_{10} 2}$ is a constant,

$$\mathcal{O}(\log_2 n) = \mathcal{O}(\log_{10} n) = \mathcal{O}(\log_{35} n) = \ldots = \mathcal{O}(\log n)$$

2. Algorithms with the time complexity in $\mathcal{O}(n)$ are also in $\mathcal{O}(n^2)$. We denote this by

$$\mathcal{O}(n) \subseteq \mathcal{O}(n^2).$$

Similarly, we have

- $\mathcal{O}(\log n) \subseteq \mathcal{O}(n)$ and
- $\mathcal{O}(1) \subseteq \mathcal{O}(\log n)$.

Example:

- an algorithm which finishes in $\leq 2n + 4 \log n$ steps is in $\mathcal{O}(n)$

1. Logarithm with its base omitted, i.e. $\log a$, is usually just a shorthand for $\log_{10} a$.

2. If an algorithm takes $\leq n$ steps, then it also takes $\leq n^2$ steps. This is because n is always smaller or equal to n^2. A similar reasoning applies to the complexity classes.

For example, if an algorithm takes $\leq 32n + 11$ steps (which means that it is in $\mathcal{O}(n)$, it also takes $\leq 32n^2 + 11$ steps (which means that it is in $\mathcal{O}(n^2)$).

In other words, any algorithm which is in $\mathcal{O}(n)$ is also in $\mathcal{O}(n^2)$. We express this by writing

$$\mathcal{O}(n) \subseteq \mathcal{O}(n^2).$$

Linear search (average case complexity)

```
int search (int[] array, int x) {
  int n = array.length;
  int i = 0;

  while (i < n) {
    if (array[i] == x) {
      return i;
    } else {
      i++;
    }
  }

  return -1; // the value not found
}
```

Average case: the value `x` is on the position $\frac{n}{2}$

(We assume that `x` appears once in the array)

Number of steps: $2 + \frac{n}{2} \times 3 = \frac{3}{2}n + 2 \quad \Longrightarrow \quad$ it is in $\mathcal{O}(n)$

(one iteration of the while loop is 3 steps, no matter if we found `x` or not)

Previously we computed that the **worst case** complexity of linear search is $\mathcal{O}(n)$. This happens if the value is not in the array and we need to search through the whole array.

Next, we consider a situation when the value `x` is in the array. (And for simplicity we assume that it is there only once). How many steps does it take **on average** to find `x`?

Because `x` can be on any position, it is on average in the middle of the array. This means that we find it on position $\frac{n}{2}$ and so the while-loop evaluates $\frac{n}{2}$-many times.

Binary Search (worst case and average case)

Searching `x` in a **sorted** array `arr`:

1. Compare `x` and `arr[arr.length div 2]`.
2. If `x` is bigger, recursively search `arr` on positions `(arr.length div 2) + 1`, ..., `arr.length - 1`.
3. Otherwise, recursively search `arr` on positions `0`, ..., `arr.length div 2`
4. We continue like this until we are left with only one element in the array. Then, return whether this element equals `x`.

The length of the array we search through reduces by one half in every step and we continue until the length is 1.

For simplicity assume that the length of `arr` is 2^k

$\implies$ the number of steps is $\mathcal{O}(k) = \mathcal{O}(\log_2 n)$.

see Paul's notes,

includes exact computation even when the array is not 2^k

If you want to compute this for general n, you have to do a bit more involved calculation (involving floor and ceiling)