ALGO

Alexandre Duret-Lutz

October 28, 2019

These are incomplete[1] lecture notes for the "ALGO" course taught to ING1 students at EPITA. This course studies the *complexity of algorithms*, and introduces students to several mathematical tools useful to analyze algorithms and derive complexity bounds.

[1] The latest version can be retrieved from `https://www.lrde.epita.fr/~adl/ens/algo/algo.pdf`. Please email me any correction to `adl@lrde.epita.fr`.

Contents

✲ Sections or paragraphs introduced with this mark contain more advanced material that is not strictly necessary to understand the rest of the text. You may want to skip them on first read, and decide later if you want to read more.

NOTATION

Although this document is written in English, it targets French students. As such, it mixes conventions from different origins. For instance, I prefer to write $\subseteq$ and $\subset$ (for the analogy with $\leq$ and $<$) rather than the $\subset$ and $\subsetneq$ convention commonly used in France.

$\mathbb{N}$	the set of natural numbers, including 0. $\mathbb{N} = \{0, 1, 2, \ldots\}$	
$\mathbb{N}^+$	the set of natural numbers, excluding 0. $\mathbb{N}^+ = \{1, 2, 3, \ldots\}$	
$\mathbb{Z}$	the set of integers	
$\mathbb{R}$	the set of real numbers	
$A \subseteq B$	A is a subset of (and possibly equal to) B	
$A \subset B$	A is a proper subset of B	
$\log_a x$	the logarithm of x in base a	page 5
$\ln x$	$\log_e x$, the natural logarithm	page 5
$\lfloor x \rfloor$	the floor function, largest integer less than or equal to x	page 6
$\lceil x \rceil$	the ceiling function, smallest integer greater than or equal to x	page 6
$x^{\underline{n}}$	the nth falling power of x: $x^{\underline{n}} = x(x-1)(x-2)\ldots(x-n+1)$	page 7
$\binom{n}{k}$	a binomial coefficient: there are $\binom{n}{k} = \frac{n^{\underline{k}}}{k!}$ ways to choose k items out of n	page 7
$\mathrm{O}(f(n))$	the set of functions that, up to some multiplicative factor, are dominated by $f(n)$ asymptotically	pages 23–24
$\Omega(f(n))$	the set of functions that, up to some multiplicative factor, dominate $f(n)$ asymptotically	pages 23–24
$\Theta(f(n))$	the set of functions that, up to some multiplicative factors, dominate and are dominated by $f(n)$ asymptotically	pages 23–24
$f(n) \sim g(n)$	$f(n)$ is asymptotically equivalent to $g(n)$, i.e., $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$	page 24
iff	if and only if	
positive	(strictly) greater than zero ($x > 0$)	
negative	(strictly) less than zero ($x < 0$)	
non-positive	less than or equal to zero ($x \leq 0$)	
non-negative	greater than or equal to zero ($x \geq 0$)	

Mathematical Background

There are a couple of mathematical notions we need to review before we turn our attention to algorithms. I expect you to be already familiar with many of those, but you might learn of few tricks along the way.

Two $\sum$ Notations for Sums

A sum such as $a_0 + a_1 + \cdots + a_{n-1}$ is more compactly written

$$\sum_{k=0}^{n-1} a_k \qquad \text{or, using a more general form,} \qquad \sum_{0 \le k < n} a_k.$$

The latter form has a couple of advantages over the former one.

- As this example demonstrates, we can use the semi-open interval[2] $0 \le k < n$ instead of the more tedious $0 \le k \le n-1$.
- The sum $\sum_{a \le k \le b} k$ clearly evaluates to 0 when $b < a$ since there is nothing to sum. This is not so obvious[3] with $\sum_{k=a}^{b} k$.
- The general form supports the addition of more constraints. The sum of all odd numbers below 1000 can be expressed as

$$\sum_{\substack{1 \le k < 1000 \\ k \text{ odd}}} k \qquad \text{or less intuitively} \qquad \sum_{k=0}^{499} 2k+1. \tag{1}$$

- The general form makes variable substitutions much less error-prone. Let us look at the sum of all odd numbers from (1) and see how we can derive the right-hand expression starting from the left-hand one.

  Since k should be odd, let us replace all occurrences of k by $2k+1$:

$$\sum_{\substack{1 \le k < 1000 \\ k \text{ odd}}} k = \sum_{\substack{1 \le 2k+1 < 1000 \\ 2k+1 \text{ odd}}} 2k+1$$

  As $2k+1$ is always odd, the constraint is now superfluous:

$$\sum_{\substack{1 \le 2k+1 < 1000 \\ 2k+1 \text{ odd}}} 2k+1 = \sum_{1 \le 2k+1 < 1000} 2k+1$$

  We can simplify $1 \le 2k+1 < 1000$ by subtracting 1 from all sides, and then halving them:

$$\sum_{1 \le 2k+1 < 1000} 2k+1 = \sum_{0 \le k < 499.5} 2k+1$$

  Now since k is an integer changing $0 \le k < 499.5$ into the equivalent $0 \le k \le 499$ gives us the right-hand expression of (1).

[2] As programmers you should learn to love semi-open intervals. Think for instance about how `begin()` and `end()` are used in the C++ standard library. If the interval was closed (i.e., if it included the value pointed to by `end()`) you would not be able to specify an empty range.

[3] But true nonetheless. I.e., it is wrong to write $\sum_{k=a}^{b} k = \sum_{k=b}^{a} k$ because the $\sum_{k=a}^{b}$ notation is about going from a to b using an *increment* of 1.

Logarithms

The logarithm in base a, i.e., the function $x \mapsto \log_a x$, is the reciprocal function of $x \mapsto a^x$. Figure 1 shows a few examples.

It is common to write $\ln x = \log_e x$ for the natural logarithm[4], i.e., the logarithm in base e. But this natural logarithm will have almost no use to us. When analyzing algorithms, we will usually encounter $\log_a x$ for various integer values of a, and most often $a = 2$.

There is a simple algorithm for computing a logarithm in any base, using only elementary operations. This algorithm is also a perfect exercise to practice logarithms.

Let us compute $\log_{10}(1385)$ up to two decimal places. Because $x \mapsto \log_{10} x$ is the reciprocal function of $x \mapsto 10^x$ we know that $\log_{10}(1000) = \log_{10}(10^3) = 3$ and $\log_{10}(10000) = \log_{10}(10^4) = 4$. Furthermore, since $1000 < 1385 < 10000$ and $\log_{10}$ is an increasing function, it follows that $3 < \log_{10}(1385) < 4$. We are therefore looking for two digits a and b such that

$$\log_{10}(1385) = 3.ab\ldots \tag{2}$$

To find a, we should subtract 3 from both sides, multiply everything by 10 and rework the left-hand side as a $\log_{10}$:

$$\begin{aligned}
\log_{10}(1385) - 3 &= 0.ab\ldots \\
\log_{10}(1385) - \log_{10}(10^3) &= 0.ab\ldots \\
\log_{10}\left(\frac{1385}{1000}\right) &= 0.ab\ldots \\
\log_{10}(1.385) &= 0.ab\ldots && (3)\\
10\log_{10}(1.385) &= a.b\ldots \\
\log_{10}(1.385^{10}) &= a.b\ldots && (4)\\
\log_{10}(25.9715419\ldots) &= a.b\ldots
\end{aligned}$$

Since $10^1 < 25.9715419\ldots < 10^2$ we conclude that $a = 1$. Did you notice what happened between (2) and (3)? When we have $\log_{10} x = y$, removing k from y is equivalent to shifting the decimal point by k places in x.[5] Also, looking at (3) and (4), multiplying y by 10 is equivalent to raising x to its 10th power.[6] We can now use a similar procedure to find b:

$$\begin{aligned}
\log_{10}(25.9715419\ldots) &= 1.b\ldots \\
\log_{10}(2.59715419\ldots) &= 0.b\ldots \\
\log_{10}(2.59715419\ldots^{10}) &= b.\ldots \\
\log_{10}(13962.955\ldots) &= b.\ldots
\end{aligned}$$

Since $10^4 < 13962.955\ldots < 10^5$ we conclude that $b = 4$ and we have just computed that $\log_{10}(1385) \approx 3.14$.

You can adjust this algorithm to compute a logarithm in any base. Using paper and pen, the only difficult step is to compute x^{10}. However, unless you plan to compute a lot of decimal places, you do not necessary need a very precise result.[7]

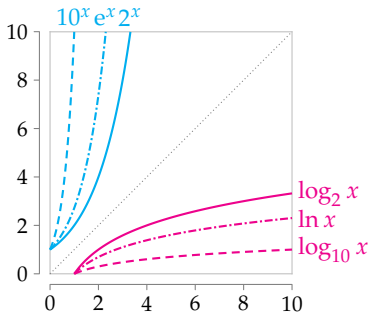


Figure 1: Various logarithms and their reciprocal functions. This figure is restricted to positive values because negative values will never occur in the analysis of algorithms.

[4] Trivia: $x \mapsto \ln x$ is sometimes called *Napierian logarithm* after John Napier (known as Neper in France), despite the fact that the function he defined in 1614 was different. The natural logarithm was introduced by Nicolaus Mercator in 1668 using the series expansion of $\ln(1+x)$. Finally around 1730 Leonhard Euler defined the functions $e^x = \lim_{n\to\infty}(1+x/n)^n$ and $\ln x = \lim_{n\to\infty} n(x^{1/n} - 1)$ and proved that they are the reciprocal of each other.

[5] This is because we are working with a base-10 logarithm.

[6] This is independent on the base of the logarithm: this 10 is the base in which we represent numbers on the right-hand side.

[7] You can compute x^{10} using only 4 multiplications. Can you see how? Hint: x^4 requires 2 multiplications.

Floor $\lfloor x \rfloor$ and Ceiling $\lceil x \rceil$ of x

Given a real number x, the notation $\lfloor x \rfloor$ denotes the largest integer smaller than x. Conversely, $\lceil x \rceil$ denotes the smallest integer larger than x. Figure 2 illustrate both functions, called respectively *floor* and *ceiling*.[8] For instance $\lfloor \pi \rfloor = 3$ and $\lceil \pi \rceil = 4$. These two functions have no effect on integers: $\lfloor 12 \rfloor = \lceil 12 \rceil = 12$. In fact for any real number x we have:

$$\lfloor x \rfloor \leq \lceil x \rceil$$
$$\lfloor x \rfloor = \lceil x \rceil \text{ iff } x \in \mathbb{Z}$$
$$1 + \lfloor x \rfloor = \lceil x \rceil \text{ iff } x \notin \mathbb{Z}$$

For any $n \in \mathbb{Z}$ and any $x \in \mathbb{R}$ the following properties hold[9]:

$$\lfloor x \rfloor < n \iff x < n$$
$$\lceil x \rceil \leq n \iff x \leq n$$
$$n < \lceil x \rceil \iff n < x$$
$$n \leq \lfloor x \rfloor \iff n \leq x$$
$$\lfloor x \rfloor = n \iff x - 1 < n \leq x \iff n \leq x < n + 1$$
$$\lceil x \rceil = n \iff x \leq n < x + 1 \iff n - 1 < x \leq n$$

For any $n \in \mathbb{N}$ we have $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$. We can prove this equation by considering the parity of n. If n is even, $\lfloor n/2 \rfloor = \lceil n/2 \rceil = n/2$ and the equation holds trivially. If n is odd, then $\lfloor n/2 \rfloor = n/2 - 1/2$ and $\lceil n/2 \rceil = n/2 + 1/2$ so the sum is indeed n.

Rounding to the *nearest* integer can be done with $\lfloor x + 0.5 \rfloor$ or $\lceil x - 0.5 \rceil$ depending on how you want to round half-integers[10].

Now let us nest these rounding notations. It should be easy to see that $\lceil \lceil x \rceil \rceil = \lfloor \lceil x \rceil \rfloor = \lceil x \rceil$ and $\lceil \lfloor x \rfloor \rceil = \lfloor \lfloor x \rfloor \rfloor = \lfloor x \rfloor$, i.e., only the innermost rounding function matters.

Furthermore, for any $n \in \mathbb{N}^+$, $m \in \mathbb{N}^+$ and $x \in \mathbb{R}$ we have[11]:

$$\lfloor \lfloor x/n \rfloor / m \rfloor = \lfloor x/nm \rfloor$$
$$\lceil \lceil x/n \rceil / m \rceil = \lceil x/nm \rceil$$

The floor notation should be used any time we want to represent an *integer division*, for instance as in Figure 3.

When rounding logarithms you should know the following identity:

$$\lceil \log_2(n+1) \rceil = \lfloor \log_2(n) \rfloor + 1$$

To prove that, rewrite n as $2^m + p$ where $m \in \mathbb{N}$ and $0 \leq p < 2^m$. Then:

$$\lfloor \log_2(n) \rfloor + 1 = \left\lfloor \log_2 \left(2^m \left(1 + \frac{p}{2^m} \right) \right) \right\rfloor + 1 = m + \Big\lfloor \underbrace{\log_2 \left(1 + \frac{p}{2^m} \right)}_{1 \leq \cdots < 2} \Big\rfloor + 1$$
$$= m + 0 + 1 = m + 1, \text{ and}$$
$$\lceil \log_2(n+1) \rceil = \left\lceil \log_2 \left(2^m \left(1 + \frac{p+1}{2^m} \right) \right) \right\rceil = m + \Big\lceil \underbrace{\log_2 \left(1 + \frac{p+1}{2^m} \right)}_{1 < \cdots \leq 2} \Big\rceil$$
$$= m + 1.$$

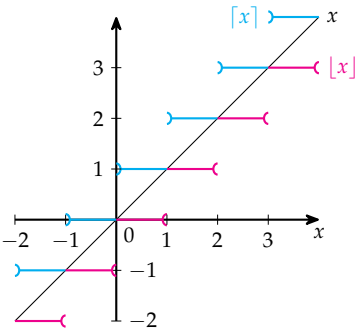


Figure 2: The functions $\lfloor x \rfloor$ and $\lceil x \rceil$.

[8] The C standard library has two functions `floor()` and `ceil()` that round a `double` accordingly.

[9] **Exercise**: demonstrate $\lfloor -x \rfloor = -\lceil x \rceil$ using these properties.

[10] i.e., values of the form $n + 0.5$ with $n \in \mathbb{N}$. Should they be rounded down to n, or up to $n + 1$?

[11] Note that these two equations are only good when n and m are integers. For instance $\lfloor \lfloor 10/.3 \rfloor / .3 \rfloor = 111$ but $\lfloor 10/.3/.3 \rfloor = 110$.

```
int avg(int a, int b)
{
  return (a + b) / 2;
}
```

Figure 3: If we ignore overflows, this function computes $\lfloor \frac{a+b}{2} \rfloor$ because dividing an `int` by another `int` will always round the result towards zero.

Simple Combinatorics

- Assume you have a set S of n different letters. How many different words[12] of length k can we build using only letters from S, assuming we can use the same letter multiple times? There are n possible letters for each of the k positions in the word, so the number of choices is $\underbrace{n \times n \times n \times \cdots \times n}_{k \text{ terms}} = n^k$. See Figure 4.

- What if we are only allowed to use each letter of S at most once? Then after we have selected the first letter among the n available, we are left with only $n-1$ choices for the second letter, $n-2$ for the third letter, etc. The number of words of length $k \leq n$ we can build without repeated letter is therefore

$$\underbrace{n \times (n-1) \times (n-2) \times \cdots \times (n-k+1)}_{k \text{ terms}} = n^{\underline{k}} = \frac{n!}{(n-k)!}$$

  See Figure 5 for an example. The notation $n^{\underline{k}}$, with an underlined exponent, is the k^{th} *falling power* of n: it works like a power except that its argument is decremented by one after each product.[13] We can define the falling power recursively as $n^{\underline{0}} = 1$ and for $k > 0$, $n^{\underline{k}} = n \times (n-1)^{\underline{k-1}}$. In particular we have $n^{\underline{n}} = n!$.

- Let us now build *subsets* of S that contain k letters. We could proceed as we did for building words of length k with unique letters: choosing the first letter among n, then the second among $n-1$, etc. We can actually associate each word to a set. For instance, the word ab would correspond to the set $\{a, b\}$, the word bc to $\{b, c\}$. The problem is that this correspondence is not a one-to-one mapping: the word ba would also be mapped to the set $\{a, b\}$ since sets are not ordered. For a given set with k letters, there are $k^{\underline{k}} = k!$ different words. So the number of subsets of size k built from a set of size n, is equal to the number of k-letter words we can build without repeating letters from n letters, divided by the $k!$ numbers of ways to order these k letters.

$$\frac{n^{\underline{k}}}{k!} = \frac{n!}{(n-k)!k!} = \binom{n}{k}$$

  The number $\binom{n}{k}$, pronounced 'n choose k', is called *binomial coefficient* because it is the coefficient of $x^k y^{n-k}$ in the polynomial expansion of the n^{th} power of the binomial $x + y$:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$$

- What is the total number of subsets of S (of any size)? To build one subset, we iterate over each letter of S and decide whether we take it or not. We have 2 possibilities for each of the n letters, that makes 2^n different subsets. On the other hand, this number of subsets is also the sum of all subsets of different sizes, as computed in the previous paragraph. So we have $\sum_{k=0}^{n} \binom{n}{k} = 2^n$ as illustrated by Figure 7.

[12] By word, we mean just any sequence of letters, not necessarily a meaningful word in some dictionary.

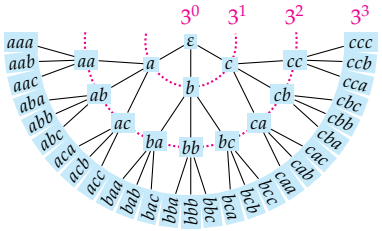


Figure 4: Over the alphabet $\{a, b, c\}$ there are 3^1 ways to build a 1-letter word, 3^2 ways to build a 2-letter word, and 3^3 ways to build a 3-letter word. There is only $3^0 = 1$ way to build the empty word (denoted ε).

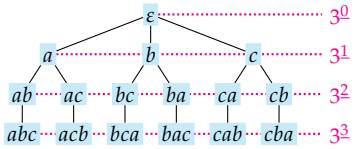


Figure 5: Without repeating letter there are only $3^{\underline{1}} = 3$ ways to build a 1-letter word, $3^{\underline{2}} = 3 \times 2$ ways to build a 2-letter word, and $3^{\underline{3}} = 3 \times 2 \times 1$ ways to build a 3-letter word.

[13] When both n and k are natural numbers such that $k \leq n$, we have $n^{\underline{k}} = n!/(n-k)!$. However, the falling power can be used even when n is a complex number, or when k is larger than n, two cases that are not supported by the expression using factorials.

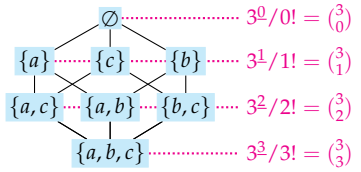


Figure 6: When the words of Figure 5 are converted to sets, the tree collapses into this lattice.

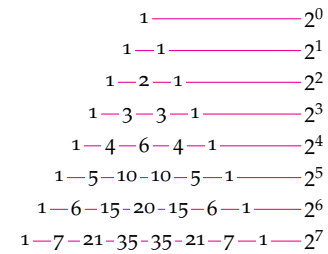


Figure 7: The sum of each line of Pascal's triangle is a power of 2.

Triangular Numbers

The numbers $A_n = 0+1+2+\cdots+n = \sum_{k=0}^{n} k$ are called *triangular numbers* because they can be represented as in Figure 8.

The equality $A_n = n(n+1)/2$ can be demonstrated in several ways.

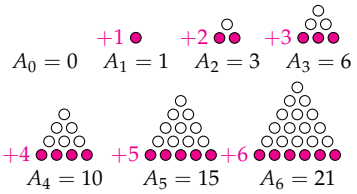


Figure 8: The first triangular numbers.

- By induction[14]. You probably already did it when you learned induction. The proof is based on the fact that $A_n = A_{n-1} + n$.

- By summing twice: once forward, and once backward.[15]

$$\begin{aligned} A_n &= 0 + & 1 + & & 2 + \cdots + (n-1) + n \\ A_n &= n + (n-1) + (n-2) + \cdots + & & & 1 + 0 \\ 2A_n &= n + & n + & & n + \cdots + & & n + n \end{aligned}$$

  Since there are $n+1$ terms on the right-hand side of the last line, we find that $2A_n = n(n+1)$.

  Figure 9 shows a graphical version of this demonstration.

- The previous demonstration is easily performed using the $\sum$ notation as well:

$$2A_n = \left(\sum_{0\le k\le n} k\right) + \left(\sum_{0\le k\le n} k\right)$$

  Replace k by $n-k$ in the second sum:

$$2A_n = \left(\sum_{0\le k\le n} k\right) + \left(\sum_{0\le n-k\le n} n-k\right)$$

  Simplify the constraint of the second sum:

$$2A_n = \left(\sum_{0\le k\le n} k\right) + \left(\sum_{-n\le -k\le 0} n-k\right)$$

$$2A_n = \left(\sum_{0\le k\le n} k\right) + \left(\sum_{0\le k\le n} n-k\right)$$

  Finally merge the two sums:

$$2A_n = \sum_{0\le k\le n} (k+n-k) = \sum_{0\le k\le n} n = n(n+1)$$

- As seen on page 7, there are $\binom{n+1}{2}$ subsets of length 2 in $\{0,1,\ldots,n\}$. Let $\{x,y\}$ be such a subset, and assume $x < y$. Let us count all subsets existing for the different values of x. If $x = 0$, there are n possible values for y; if $x = 1$ we have $n-1$ possible values for y; etc. If $x = n$ there is no value available for y. The sum of all these $n + (n-1) + \cdots + 0$ just happens to be A_n. So we have

$$A_n = \binom{n+1}{2} = \frac{(n+1)^{\underline{2}}}{2!} = \frac{(n+1)n}{2}.$$

  Figure 10 should therefore not be a surprise.[16]

[14] Induction is of no use to you if you do not already know the solution. If this was a new problem for which you suspected (maybe after looking at a few values) that $A_n = n(n+1)/2$, then induction would be a way to prove that your intuition is correct.

[15] Gauss reportedly found this trick while he was a child.

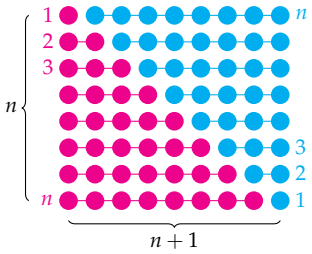


Figure 9: Another way to see that $2A_n = n(n+1)$: there are A_n dots of each color arranged in a n by $n+1$ rectangle.

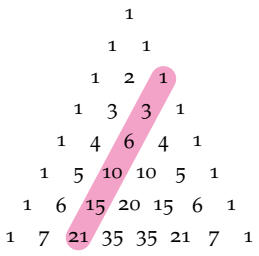


Figure 10: Triangular numbers form a diagonal of Pascal's triangle.

[16] By convention $\binom{n}{k} = 0$ when $k > n$ or $k < 0$ (i.e., outside of Pascal's triangle) so our $\binom{n+1}{2}$ is also valid for A_0.

Tetrahedral Numbers

What happens when we sum all consecutive triangle numbers?

$$B_n = A_0 + A_1 + \cdots + A_n = \sum_{j=0}^{n} A_j = \sum_{j=0}^{n} \sum_{k=0}^{j} k$$

We get *tetrahedral numbers*, so called because stacking the triangles of Figure 8 gives you a triangular pyramid as shown in Figure 11.

The closed formula is $B_n = n(n+1)(n+2)/6$ and there are again a couple of ways to prove it.[17]

- Induction is still a possible option. The key step is that $B_n = B_{n-1} + A_n = \frac{(n-1)n(n+1)}{6} + \frac{n(n+1)}{2} = \frac{n(n+1)(n+2)}{6}$.
- Note that the above formula $\frac{(n-1)n(n+1)}{6} + \frac{n(n+1)}{2} = \frac{n(n+1)(n+2)}{6}$ is simply a long way to write $\binom{n+1}{3} + \binom{n+1}{2} = \binom{n+2}{3}$. You may find it easier to remember that $B_n = \binom{n+2}{3}$, forming another diagonal of Pascal's triangle (Figure 12).

  Since each diagonal of Pascal's triangle is made of the partial sum of the previous diagonal, you should find very easy to guess a formula for the sum of consecutive tetrahedral numbers:

$$\sum_{k=0}^{n} k = \binom{n+1}{2}, \quad \sum_{j=0}^{n} \sum_{k=0}^{j} k = \binom{n+2}{3}, \quad \sum_{i=0}^{n} \sum_{j=0}^{i} \sum_{k=0}^{j} k = \binom{n+3}{4}.$$

- The above two points require you to know (or suspect) that $B_n = \frac{n(n+1)(n+2)}{6}$ or $B_n = \binom{n+2}{3}$ in order to prove it by induction.

  How can we find a closed formula for B_n if we do not know that? Looking at how balls are stacked in 3D in Figure 11, we can assume that B_n should represent some volume, i.e., a cubic polynomial. Or if you prefer a more mathematical view: A_j is a quadratic polynomial, B_n, as the sum of n of these terms, should be expressible as a cubic polynomial. So we guess $B_n = an^3 + bn^2 + cn + d$ and we just need to evaluate this for a couple of values of n to find a, b, c, and d. Evaluating $B_0 = 0$ tells us that $d = 0$. From $B_1 = 1$, $B_2 = 4$, and $B_3 = 10$ we get:

$$\begin{cases} a+b+c=1 \\ 8a+4b+2c=4 \\ 27a+9b+3c=10 \end{cases} \text{ hence } \begin{cases} c=1-a-b \\ 6a+2b=2 \\ 24a+6b=7 \end{cases}$$

$$\text{hence } \begin{cases} c=1-a-b \\ b=1-3a \\ 6a+6=7 \end{cases} \text{ hence } \begin{cases} c=2/6 \\ b=3/6 \\ a=1/6 \end{cases}$$

  Thus we have found that $\frac{n^3+3n^2+2n}{6}$, which happens to be equal to $\frac{n(n+1)(n+2)}{6}$, is a polynomial that will work for $n = 0, 1, 2, 3$, and we can prove by induction that it is correct for any $n \in \mathbb{N}$.

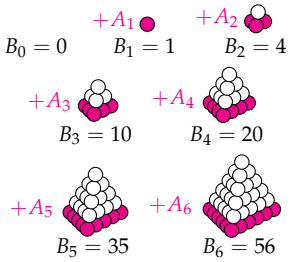


Figure 11: The first tetrahedral numbers.

[17] Do not confuse this formula with the $C_n = n(n+1)(2n+1)/6$ from page 10.

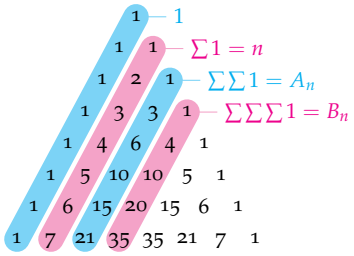


Figure 12: Tetrahedral numbers form another diagonal of Pascal's triangle. (Note that these sums implicitly start at 1, not 0 like in the rest of the page; do you see why it matters in this picture?)

Pyramidal Numbers

The numbers $C_n = 0^2 + 1^2 + 2^2 + \cdots + n^2 = \sum_{k=0}^{n} k^2$ are called *pyramidal numbers* because they represent the number of spheres stacked in a pyramid with a square base, as shown in Figure 13.

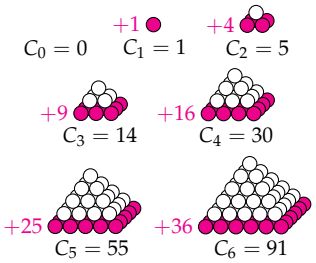


Figure 13: The first pyramidal numbers.

Unlike previous numbers, we will not give the closed formula directly. It seems remembering the formula is hard for many students, so maybe it is best to learn three ways to rediscover it.

- Since this is a sum of n squares, and Figure 13 gives a 3D interpretation, we can, as we did on page 9 for tetrahedral numbers, assume that C_n is cubic polynomial $an^3 + bn^2 + cn + d$ and use the first values of C_n to find its coefficients. From $C_0 = 0$, we learn that $d = 0$. Using $C_1 = 1$, $C_2 = 5$, and $C_3 = 14$, we get:

$$\begin{cases} a + b + c = 1 \\ 8a + 4b + 2c = 5 \\ 27a + 9b + 3c = 14 \end{cases} \quad \text{whose solution is} \quad \begin{cases} c = 1/6 \\ b = 3/6 \\ a = 2/6 \end{cases}$$

  Hence our polynomial is $\frac{2n^3+3n^2+n}{6}$ and without too much effort[18] we can factorize it as $\frac{n(n+1)(2n+1)}{6}$.

  By construction this formula is correct from C_0 to C_3. If we assume that $C_{n-1} = \frac{(n-1)n(2n-1)}{6}$, then $C_n = C_{n-1} + n^2 = \frac{n(2n^2-3n+1)+6n^2}{6} = \frac{n(2n^2+3n+1)}{6} = \frac{n(n+1)(2n+1)}{6}$. Hence by induction our formula is correct for all $n \in \mathbb{N}$.

[18] Because two of the three roots are easy to find: 0 and -1.

- Let us compute $S = \sum_{i=0}^{n} \left((i+1)^3 - i^3 \right)$ in two ways. First, we separate it in two sums which almost cancel out each other[19]:

$$S = \sum_{i=0}^{n} (i+1)^3 - \sum_{i=0}^{n} i^3 = \sum_{i=1}^{n+1} i^3 - \sum_{i=1}^{n} i^3 = (n+1)^3 \qquad (5)$$

  In a second approach, we develop the summand and express the result as a sum of triangular (page 8) and pyramidal numbers:

$$S = \sum_{i=0}^{n} (3i^2 + 3i + 1) = 3C_n + 3A_n + n + 1 \qquad (6)$$

  Since (5) and (6) are two expressions for S, we get that $3C_n + 3A_n + n + 1 = (n+1)^3$. Knowing a formula for A_n, we get $3C_n = (n+1)((n+1)^2 - \frac{3}{2}n - 1)$ hence $C_n = \frac{n(n+1)(2n+1)}{6}$.

[19] Watch out for the indices in these two sums! The first sum is changed by replacing i by $i-1$ and rewriting the range $0 \le i-1 \le n$ into $1 \le i \le n+1$. In the second sum we just omit the first term, because it is equal to 0.

- Consider each square used in the layers of a pyramid in Figure 13, and split them into two triangles by the diagonal. One triangle (the larger one, drawn using ● in Figure 14) includes the diagonal, and the other does not. The sum of the larger triangles of all layers of C_n is the tetrahedral number B_n (page 9) while the sum of all smaller triangles is B_{n-1}. Hence

$$C_n = B_n + B_{n-1} = \binom{n+2}{3} + \binom{n+1}{3}$$
$$= \frac{n(n+1)(n+2)}{6} + \frac{(n-1)n(n+1)}{6} = \frac{n(n+1)(2n+1)}{6}$$

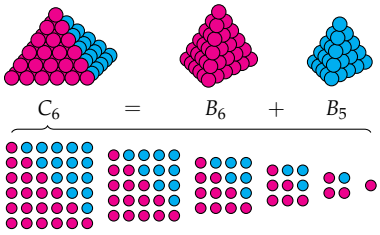


Figure 14: A pyramidal number is the sum of two consecutive tetrahedral numbers.

Sum of an Arithmetic Progression

When analyzing algorithms, it often happens that the number of operations performed in a loop is a linear function of the loop counter. Then, the sum of all performed operations has the following form, for some value of a and b:

$$D_n = a + (a+b) + (a+2b) + \cdots + (a+nb) = \sum_{k=0}^{n} a + kb$$

Triangular numbers (page 8) are a special case of this sum with $a = 0$ and $b = 1$. In the general case we can rewrite D_n using A_n:

$$\begin{aligned} D_n &= \left(\sum_{k=0}^{n} a\right) + \left(\sum_{k=0}^{n} kb\right) = a(n+1) + b\sum_{k=0}^{n} k = a(n+1) + bA_n \\ &= a(n+1) + \frac{bn(n+1)}{2} = \frac{(2a+nb)(n+1)}{2} \end{aligned}$$

But the same result is in fact even easier to obtain using Gauss' trick of summing forward and backward:

$$\begin{aligned} D_n &= a + (a+b) + (a+2b) + \cdots + (a+nb) \\ D_n &= (a+nb) + (a+(n-1)b) + (a+(n-2)b) + \cdots + a \\ 2D_n &= (2a+nb) + (2a+nb) + (2a+nb) + \cdots + (2a+nb) \end{aligned}$$

Hence $2D_n = (2a+nb)(n+1)$. Figure 15 gives an example with $a = 1$ and $b = 2$.

The above trick has a huge advantage over expressing D_n using A_n: it can be generalized very easily to any *partial* sum of an arithmetic progression. For instance, let us assume you want to sum all the terms $3 + 5i$ for $100 \leq i \leq 1000$. Calling S the result, you would write

$$\begin{aligned} S &= 503 + 508 + 513 + \cdots + 5003 \\ S &= 5003 + 4998 + 4993 + \cdots + 503 \\ 2S &= 5506 + 5506 + 5506 + \cdots + 5506 \end{aligned}$$

The number of terms[20] in these sums is 901 since we go from $i = 100$ to $i = 1000$. Therefore $2S = 5506 \times 901$ and $S = 2480453$.

For any a, b, $v \leq w$, we have

$$\sum_{k=v}^{w} a + kb = \frac{(2a + (v+w)b)(w-v+1)}{2}.$$

You might find the above formula easier to remember as

$$((a+vb) + (a+wb))\frac{w-v+1}{2},$$

that is: the sum of the first and last terms, multiplied by half the number of terms.[21]

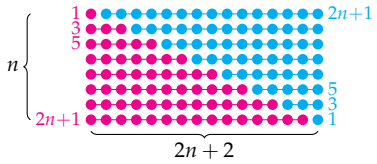


Figure 15: The sum $O_n = \sum_{k=0}^{n} 2k+1$ of the first n odd numbers is such that $2O_n = n(2n+2)$ hence $O_n = n(n+1)$.

[20] Always be cautious when calculating the length of an interval: it is a frequent source of off-by-one errors.

[21] But do also remember that this is only valid for arithmetic progressions.

Sum of a Geometric Progression

Consider the sum of the terms of a geometric progression of ratio r:

$$E_n = 1 + r + r^2 + \cdots + r^n = \sum_{k=0}^{n} r^k$$

An easy way to find a closed formula for this sum is to notice that E_n and rE_n have many terms in common:

$$\begin{aligned} E_n &= 1 + r + r^2 + \cdots + r^n \\ rE_n &= \quad r + r^2 + \cdots + r^n + r^{n+1} \end{aligned}$$

hence $$E_n - rE_n = 1 - r^{n+1}$$

and assuming $r \neq 1$, $$E_n = \frac{1 - r^{n+1}}{1 - r}$$

The formula to remember is therefore:

$$\text{For any } r \neq 1, \qquad \sum_{k=0}^{n} r^k = \frac{1 - r^{n+1}}{1 - r} \tag{7}$$

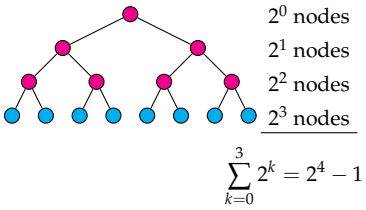


Figure 16: A complete binary tree of height 3 has $2^4 - 1 = 15$ nodes.

- When $r = 2$, we have $\sum_{k=0}^{n} 2^k = \frac{1-2^{n+1}}{1-2} = 2^{n+1} - 1$, a formula that should be known by any programmer. For instance the number of nodes in a complete binary tree of height n (see Figure 16). A binary number $(111\ldots1)_2$ that has all its n bits set to 1 represents the value $\sum_{k=0}^{n-1} 2^k = 2^n - 1$. In particular $2^8 - 1$ is the maximum value you can represent with a `unsigned char` variable, since this type uses 8 bits.

- We had to assume $r \neq 1$ because of the division by $1 - r$, but the limit[22] of $\frac{1 - r^{n+1}}{1 - r}$ when r tends to 1 is actually what we expect:

$$\sum_{k=0}^{n} 1^k = \sum_{k=0}^{n} 1 = n + 1$$

$$\lim_{r \to 1} \frac{1 - r^{n+1}}{1 - r} = \lim_{r \to 1} \frac{-(n+1)r^n}{-1} = n + 1$$

- Equation (7) can be used to rediscover the formula for Triangular Numbers (page 8). To transform $\sum r^k$ into $\sum k$, we differentiate $\sum r^k$ with respect to r, giving $\sum k r^{k-1}$, and then we set $r = 1$. Of course we must do these operations on both sides of (7), and we have to take a limit for $r \to 1$ on the right:

$$\frac{d}{dr} \sum_{k=0}^{n} r^k = \frac{d}{dr} \frac{1 - r^{n+1}}{1 - r}$$

$$\sum_{k=1}^{n} k r^{k-1} = \frac{-(n+1)r^n(1-r) + (1 - r^{n+1})}{(1-r)^2}$$

$$\sum_{k=1}^{n} k = \lim_{r \to 1} \frac{nr^{n+1} - (n+1)r^n + 1}{(1-r)^2} = \lim_{r \to 1} \frac{(n+1)nr^n - (n+1)nr^{n-1}}{2(r-1)}$$

$$= \lim_{r \to 1} \frac{(n+1)nr^{n-1}(r-1)}{2(r-1)} = \frac{(n+1)n}{2}$$

Similarly $\frac{d}{dr}\left(r \frac{d}{dr} \sum r^k\right) = \sum k^2 r^{k-1}$ so by setting $r = 1$ we get the formula for the Pyramidal Numbers (page 8).[23]

[22] Limits on this page are computed using L'Hôpital's rule: if $\lim_{x \to c} f(x) = \lim_{x \to c} g(x) = 0$ and $\lim_{x \to c} \frac{f'(x)}{g'(x)}$ exists, then $\lim_{x \to c} \frac{f(x)}{g(x)} = \lim_{x \to c} \frac{f'(x)}{g'(x)}$.

[23] Doing so is left as an exercise to the reader. If you survive the double differentiation and the computation of the limit, and obtain the expected $\frac{n(n+1)(2n+1)}{6}$, treat yourself with a well-earned lollipop.

⚙ *Catalan Numbers*

A *Dyck word* of length $2n$ is a string built using n opening parentheses and n closing parentheses, in such a way that a closing parenthesis always matches an opening one. For instance $w_1 =$ '`( ( ( ) ( ) ) ) ( ( ) )`' is a Dyck word, but $w_2 =$ '`( ( ) ) ) ( ( ) ) ( ) (`' is not.

Let P_n be the number of Dyck words of length $2n$. This integer sequence (Table 1) is known as the *Catalan numbers*[24].

A string with n opening parentheses and n closing parentheses can be interpreted as a path on a square grid (Figure 17). Starting from the lower left corner and interpreting the letter '`(`' and '`)`' respectively as *up* and *right*, we necessarily reach the above right corner. The number of paths that join the two corners using only n *up* and n *right* movements is $\binom{2n}{n}$: from the total of $2n$ movements we simply have to choose n which will be the *up*s. (Or if you prefer working with words: in a string of $2n$ characters we have to choose n positions among the $2n$ positions available to put the '`(`' letters.)

Not all these $\binom{2n}{n}$ paths correspond to Dyck words, only those that stay above the diagonal. To count the number of paths that do *not* correspond to Dyck words, let us consider the first segment of the path that goes below the diagonal, and flip all *up* and *right* movements afterwards (Figure 18). This is a reversible operation that can only be done on paths that do not represent a Dyck word. Since the resulting path has only $n-1$ *up* movements, there are $\binom{2n}{n-1}$ words of length $2n$ that are not Dyck words. We have established that

$$P_n = \binom{2n}{n} - \binom{2n}{n-1} \tag{8}$$

which we can simplify:

$$\begin{aligned} &= \frac{(2n)(2n-1)\cdots(n+2)(n+1)}{n!} - \frac{(2n)(2n-1)\cdots(n+2)}{(n-1)!} \\ &= \frac{(2n)(2n-1)\cdots(n+2)(n+1)}{n!}\left(1 - \frac{n}{n+1}\right) \\ P_n &= \frac{1}{n+1}\binom{2n}{n} \end{aligned} \tag{9}$$

Note that (8) tells us that P_n is an integer even if it is not that obvious from (9).

Catalan numbers have a vast number of applications.[25] For instance the number of full[26] binary trees with n internal nodes is P_n. To see that, make a depth-first traversal of some full binary tree and write '`(`' each time you get down a left edge, and '`)`' each time you get down a right edge (Figure 19 below).

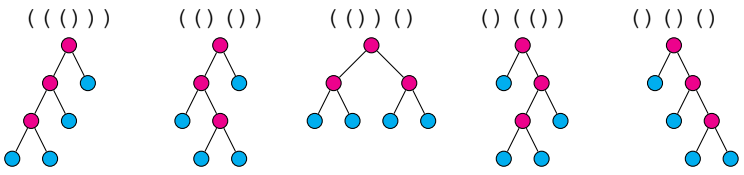


Figure 19: The $P_3 = 5$ full binary trees with 3 internal nodes and their relation to Dyck words of length 6.

n	P_n	Dyck words
0	1	ε (empty word)
1	1	`( )`
2	2	`( ) ( )`, `( ( ) )`
3	5	`( ) ( ) ( )`, `( ) ( ( ) )`, `( ( ) ) ( )`, `( ( ) ( ) )`, `( ( ( ) ) )`
4	14	...
6	42	...
7	132	...

Table 1: Number of Dyck words for various n, a.k.a. Catalan numbers.

[24] Named after Eugène Charles Catalan (1814–1894).

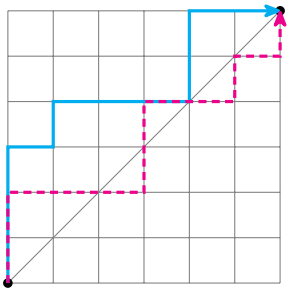


Figure 17: The words $w_1 =$ '`( ( ( ) ( ) ) ) ( ( ) )`' and $w_2 =$ '`( ( ) ) ) ( ( ) ) ( ) (`' interpreted as paths on a grid. The letter '`(`' is *up*, while '`)`' is *right*. Dyck words corresponds to paths that stay above the diagonal.

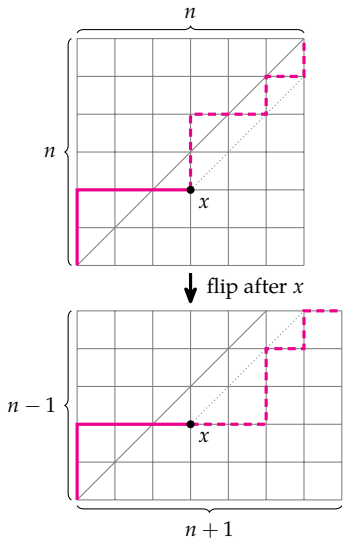


Figure 18: Flipping all *up*s and *right*s that occur after the first segment below the diagonal transform a path with n *up*s and n *right*s into a path with $n-1$ *up*s and $n+1$ *right*s.

[25] And also many different proofs.

[26] A binary tree is *full* if all its internal nodes have degree 2.

✲ *Bounding Sums with Integrals*

The technique presented on this page justifies (and generalizes) the intuition we used on pages 9 and 10 that the sum of n quadratic terms should be a cubic polynomial.

For more generality, let us consider the sum $f(0) + f(1) + \cdots + f(n)$ where f is some monotonically increasing function. Showing these terms under the graph of f as in Figures 20 and 21 we have

$$\int_{-1}^{n} f(k)\mathrm{d}k \leq \sum_{k=0}^{n} f(k) \leq \int_{0}^{n+1} f(k)\mathrm{d}k$$

Note that the length of the two integration intervals is equal to the number of terms in the sum.[27]

These inequalities come in handy to bound a sum that we do not know how to simplify. For instance, let us pretend that we do not know how to compute triangular numbers (page 8). We simply rewrite the above inequalities with $f(k) = k$:

$$\int_{-1}^{n} k\mathrm{d}k \leq \sum_{k=0}^{n} k \leq \int_{0}^{n+1} k\mathrm{d}k$$

Since the antiderivative[28] of k is $k^2/2$ we get:

$$\left[\frac{k^2}{2}\right]_{-1}^{n} \leq \sum_{k=0}^{n} k \leq \left[\frac{k^2}{2}\right]_{0}^{n+1}$$

$$\frac{n^2-1}{2} \leq \sum_{k=0}^{n} k \leq \frac{(n+1)^2}{2}$$

We do not have an exact value for this sum, but from these bounds we can at least derive some asymptotic equivalence[29]:

$$\sum_{k=0}^{n} k \sim \frac{n^2}{2}$$

A complexity we will encounter later is $\log_2(n!)$. Do you think that using $\log_2(n!)$ operations to sort n value is efficient? It is hard to tell if you have no idea how fast $\log_2(n!)$ grows. Luckily, we can rewrite $\log_2(n!)$ as a sum:

$$\log_2(n!) = \log_2\left(\prod_{k=1}^{n} k\right) = \sum_{k=1}^{n} \log_2(k) = \sum_{k=2}^{n} \log_2(k)$$

and then we simply apply the bound-by-integral technique[30]:

$$\int_{1}^{n} \log_2 k\mathrm{d}k \leq \sum_{k=2}^{n} \log_2(k) \leq \int_{2}^{n+1} \log_2 k\mathrm{d}k$$

$$\left[k\log_2\left(\frac{k}{e}\right)\right]_{1}^{n} \leq \log_2(n!) \leq \left[k\log_2\left(\frac{k}{e}\right)\right]_{2}^{n+1}$$

$$n\log_2 n - n\log_2(e) + \log_2(e) \leq \log_2(n!) \leq (n+1)\log_2(n+1) - (n+1)\log_2(e) - 2\log_2\left(\frac{2}{e}\right)$$

From that we easily conclude $\log_2(n!) \sim n\log_2 n$. A sorting algorithm that performs in the order of $n\log_2 n$ operations is actually pretty good.[31]

For a more precise tie between sums and integrals, look up the Euler-Maclaurin formula in your preferred encyclopedia.

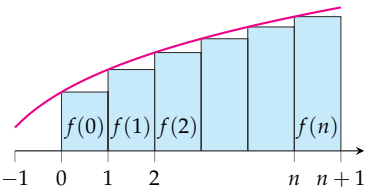


Figure 20: When $f(i)$ is interpreted as an area between i and $i+1$, we have $f(0) + \cdots + f(n) \leq \int_0^{n+1} f(k)\mathrm{d}k$.

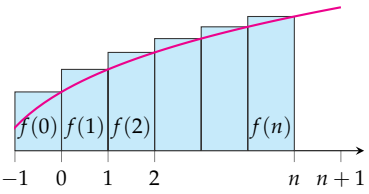


Figure 21: If $f(i)$ is interpreted as an area between $i-1$ and i, we have $f(0) + \cdots + f(n) \geq \int_{-1}^{n} f(k)\mathrm{d}k$.

[27] Using a semi-open interval for the sum, we can rewrite these inequalities using the same bounds for the sum and integrals:

$$\int_{0}^{n+1} f(k-1)\mathrm{d}k \leq \sum_{0 \leq k < n+1} f(k) \leq \int_{0}^{n+1} f(k)\mathrm{d}k$$

[28] a.k.a. primitive

[29] $f \sim g$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$

[30] If you learned that the antiderivative of $\ln(x)$ is $x\ln(x) - x$, just erase it from your memory, and use the freed space to store a formula that will work for all bases instead: the antiderivative of $\log_a(x)$ is $x\log_a(x/e)$.

[31] Later we will demonstrate that any sorting algorithm that uses comparisons to order values requires at least $n\log_2 n$ comparisons in the worst case.

❁ *Summing Using the Reciprocal*

Here is a nifty trick to deal with sums such as $\sum_i \lfloor \log_2 i \rfloor$. Let us consider the following sum, which we will encounter later.

$$F_n = \sum_{i=1}^{n} (\lfloor \log_2 i \rfloor + 1) = n + \sum_{i=1}^{n} \lfloor \log_2 i \rfloor$$

The trick, pictured on Figure 22, is to express the sum[32] of a strictly increasing function f using the sum of its reciprocal f^{-1}.

[32] This trick can be applied to integrals as well.

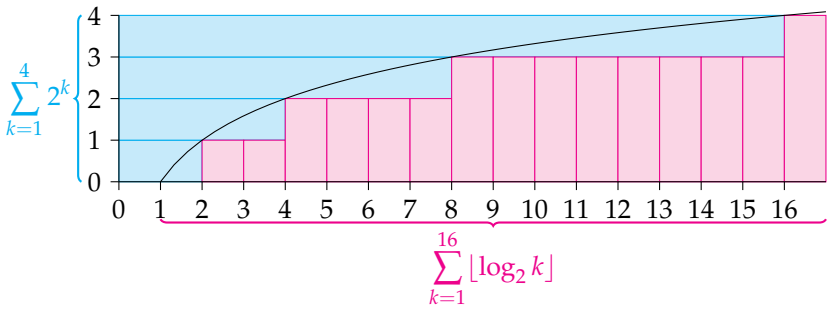


Figure 22: The total area covered by those two sums is a rectangle, and we have $\sum_{k=1}^{16} \lfloor \log_2 k \rfloor = 17 \times 4 - \sum_{k=1}^{4} 2^k$.

Generalizing this figure for any n, we have

$$\sum_{k=1}^{n} \lfloor \log_2 k \rfloor = (n+1)\lfloor \log_2 n \rfloor - \sum_{k=1}^{\lfloor \log_2 n \rfloor} 2^k = (n+1)\lfloor \log_2 n \rfloor + 1 - \sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^k$$
$$= (n+1)\lfloor \log_2 n \rfloor + 2 - 2^{\lfloor \log_2 n \rfloor + 1}$$

Finally, $F_n = n + \sum_{i=1}^{n} \lfloor \log_2 i \rfloor = n + (n+1)\lfloor \log_2 n \rfloor + 2 - 2^{\lfloor \log_2 n \rfloor + 1}$.

Why do we care about such a function? Because $\lfloor \log_2(n) \rfloor + 1$ is the number of bits required to represent the number n in binary, and many algorithms have a run time proportional to that. F_n is the sum of the bits needed to represent each number between 1 and n (see Figure 23).

For instance, running a recursive implementation of BinarySearch (page 22) on an array of length n involves at most $2 + \lfloor \log_2(n) \rfloor$ calls (the first one plus the recursive ones) to BinarySearch.

Now let us assume that you are doing a binary search to insert a new element into a sorted array, and that you do this in a loop, so that each binary search is applied to an array that has one entry more than the previous one. The total number of calls the BinarySearch (including recursive calls) will therefore have the form

$$\sum_{k=i}^{j} (2 + \lfloor \log_2 k \rfloor)$$

where i and j depends on the initial size of the array and the number of iterations (i.e., binary search + insertion) performed.

1	0	0	0	0	1
2	0	0	0	1	0
3	0	0	0	1	1
4	0	0	1	0	0
5	0	0	1	0	1
6	0	0	1	1	0
7	0	0	1	1	1
8	0	1	0	0	0
9	0	1	0	0	1
10	0	1	0	1	0
11	0	1	0	1	1
12	0	1	1	0	0
13	0	1	1	0	1
14	0	1	1	1	0
15	0	1	1	1	1
16	1	0	0	0	0

Figure 23: The first 16 positive integers with their binary representation. The $\log_2(i) + 1$ bits needed to represent the number i are highlighted. If you omit the last column (a total of n bits), the two colored areas are the same as in Figure 22.

✲ *Finite Calculus*

For those of you curious to learn new tricks, here is something called *Finite Calculus*. This page should by no means be understood as a reference on this subject, instead, consider it as a *teaser*[33].

The idea is that a sum of $k^{\underline{\alpha}}$ over a half-open interval behaves like the integral of k^α over the same interval. So summing these falling powers[34] should be very natural (at least if you remember how to integrate). Compare the following equations:

$$\sum_{0\le k<n} 1 = n \qquad \int_0^n 1\mathrm{d}k = n$$

$$\sum_{0\le k<n} k = \frac{n^{\underline{2}}}{2} \qquad \int_0^n k\mathrm{d}k = \frac{n^2}{2}$$

$$\sum_{0\le k<n} k^{\underline{2}} = \frac{n^{\underline{3}}}{3} \qquad \int_0^n k^2\mathrm{d}k = \frac{n^3}{3}$$

$$\sum_{0\le k<n} k^{\underline{\alpha}} = \frac{n^{\underline{\alpha+1}}}{\alpha+1} \qquad \int_0^n k^\alpha\mathrm{d}k = \frac{n^{\alpha+1}}{\alpha+1}$$

These work on non-zero based intervals as you would expect from an integral. For instance

$$\sum_{i\le k<j} k^{\underline{\alpha}} = \left[\frac{k^{\underline{\alpha+1}}}{\alpha+1}\right]_i^j = \frac{j^{\underline{\alpha+1}} - i^{\underline{\alpha+1}}}{\alpha+1}$$

Following these rules, we can for instance compute the tetrahedral numbers of page 9 very easily (just remember to use semi-open intervals[35]):

$$B_n = \sum_{0\le j<n+1}\ \sum_{0\le k<j+1} k \;= \sum_{0\le j<n+1} \frac{(j+1)^{\underline{2}}}{2} = \frac{(n+2)^{\underline{3}}}{6} \underbrace{- \frac{1^{\underline{3}}}{6}}_{0} = \frac{(n+2)^{\underline{3}}}{6}$$

If we look at functions other than falling powers, the analogy between sum and integral does not always exist. For instance, it would be tempting to see the sum of 2^x as the analogous of the integral of e^x:

$$\sum_{i\le k<j} 2^k = 2^j - 2^i \qquad \int_i^j \mathrm{e}^k\mathrm{d}k = e^j - e^i$$

but the sum and integral of x^k do not actually exhibit that much similarities:

$$\sum_{i\le k<j} x^k = \frac{x^j - x^i}{x-1} \qquad \int_i^j \mathrm{x}^k\mathrm{d}k = \frac{x^j - x^i}{\ln x}$$

[33] For a more serious presentation of Finite Calculus, I suggest you start with *Finite Calculus: A Tutorial for Solving Nasty Sums*, by David Gleich.

[34] The falling power was defined on page 7 as $k^{\underline{0}} = 1$ and $k^{\underline{\alpha}} = k(k-1)^{\underline{\alpha-1}}$.

[35] They have to be closed on the left side, and open on the right side.

Binary Trees

Let us define a *binary tree* recursively as follows: a binary tree is either the empty tree $\emptyset$, or a pair of binary trees (L, R) where L is called the *left* child, while R is the *right* child.

As Figure 24 illustrates, a binary tree can be represented as a graph where each pair (L, R) is represented by a node connected to new nodes created for each of its non-empty children. These graphs are traditionally drawn going down, with the left and right children located on the corresponding side below their parent node. With this drawing convention the shape of the graph is enough to uniquely identify a binary tree, so we can forgo the mathematical notations and work only with pictures such as Figure 25.

A node that has only empty children (i.e., a node labeled by $(\emptyset, \emptyset)$) is called a *leaf* node. The other nodes are called *internal* nodes. These two sets of nodes are shown with two colors on Figures 24 and 25, but this coloration is purely cosmetic. The topmost node is called the *root* of the tree[36]. The *degree* of a node is the number of non-empty children: the leaves are the nodes with degree 0, while internal nodes have degree 1 or 2.

A *full* binary tree is a binary tree where all internal nodes have degree 2. The binary tree of Figure 25 is not full, while the one of Figure 26 is. Figure 19 on page 13 shows all possible full binary trees with 3 internal nodes.

The *depth* of a node is the number of edges between this node and the root. The root itself has depth 0; its children have depth 1; its grand children have depth 2; etc. The *height* of a tree is the maximum depth of its nodes.[37]

You should be able to prove the following properties[38] by yourself:

- A binary tree with n nodes has $n - 1$ edges.[39]
- A binary tree with ℓ leaves has exactly $\ell - 1$ internal nodes of degree 2.
- A binary tree of height h has at most 2^h leaves.
- The height of a binary tree with $\ell > 0$ leaves is at least $\lceil \log_2 \ell \rceil$.
- The number of nodes of a binary tree of height h is at most $2^{h+1} - 1$.
- The height of a binary tree with $n > 0$ nodes is at least $\lceil \log_2(n + 1) - 1 \rceil$, which we have shown on page 6 to be equal to $\lfloor \log_2 n \rfloor$.

A full binary tree of height h is *balanced* if the depth of each leaf is either h or $h - 1$. For instance, the full binary tree of Figure 26 is balanced because all its leaves have depth 2 or 3. A *balanced* full binary tree of height h necessarily has $\sum_{k=0}^{h-1} 2^k = 2^h - 1$ nodes of depth $h - 1$ or smaller and between 1 and 2^h nodes of depth h. So if we write n the number of nodes, we have $2^h \leq n < 2^{h+1}$, hence $h \leq \log_2(n) < h + 1$ and because h has to be an integer: $h = \lfloor \log_2(n) \rfloor$. The height of a balanced full binary tree of n nodes is therefore always $\lfloor \log_2(n) \rfloor$.

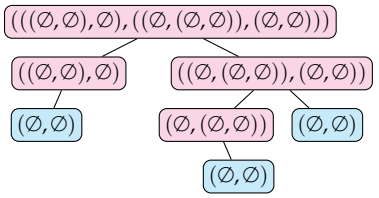


Figure 24: A graphical representation of the binary tree $(((\emptyset,\emptyset),\emptyset),((\emptyset,(\emptyset,\emptyset)),(\emptyset,\emptyset)))$.

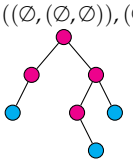


Figure 25: A nicer representation of the same binary tree. This tree is not full because it has two internal nodes of degree 1.

[36] In Figure 25 the root is an internal node. Can you build a tree where the root is a leaf?

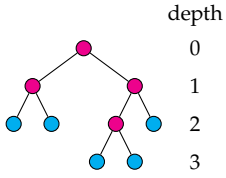


Figure 26: A full binary tree: each internal node has two non-empty children. The height of this tree is 3.

[37] We could also write "the maximum depth of its leaves", because for any internal node there exists a deeper leave.

[38] All of these are for binary trees, they do not assume the binary tree to be full.

[39] Hint: what do every node but the root have?

Computing Complexities for Algorithms

The *time complexity* of an algorithm, often noted $T(n)$, is a function that indicates how long the algorithm will take to process an input of size n. Similarly, the *space complexity* $S(n)$ measures the extra memory that the algorithm requires. These two definitions are vague[40] on purpose, as there are different ways to compute and express these quantities, depending on how we plan to use them:

[40] What would be the units of n, $T(n)$, and $S(n)$?

- A complexity can be given as a precise formula. For instance, we might say that SelectionSort requires $n(n-1)$ comparisons and n swaps to sort an array of size n. If these are the most frequent operations in the algorithm, this suggests that $T(n)$ can be expressed as a quadratic polynomial $T(n) = an^2 + bn + c$. Knowing this, we can compute the coefficients a, b, and c for a given implementation[41] of SelectionSort by measuring its run time on a few arrays of different sizes.

  If the implementation of two sorting algorithms A_1 and A_2 have for time complexity $T_1(n)$ and $T_2(n)$ as shown on Figure 27, we can see that A_1 is better for $n < 8$, and A_2 for $n > 8$.

[41] Two implementations of the same algorithm are likely to have different coefficients.

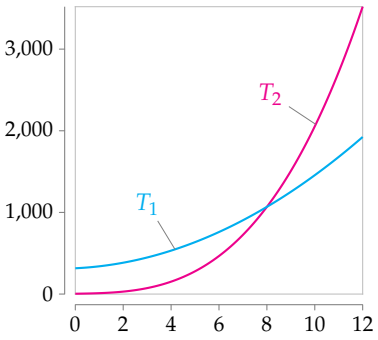


Figure 27: $T_1(n) = 10n^2 + 14n + 316$ and $T_2(n) = 2n^3 + 5n + 4$.

- A complexity can be given using Landau's notation, to give an idea of its order. For instance, we would say that SelectionSort has time complexity $T(n) = \Theta(n^2)$. This means that when n tends to ∞, $T(n)$ behaves like n^2 up to some multiplicative factor.

  This notation simplifies the derivations of complexities because it is only concerned about the asymptotic behavior of the algorithm. For instance, a $\Theta(n \log n)$ algorithm will be more efficient than a $\Theta(n^2)$ algorithm for large values of n. However, it tells us nothing about the behavior for small values of n.

  With this notation $10n^2 + 2n$ and $2n^2 + 10n$ would both be written $\Theta(n^2)$. Because of those *hidden constants*, we can hardly compare two algorithms that have the same order of complexity.

- In *computational complexity theory*, problems (not algorithms)[42] are classified according to their difficulty. For instance, the class *PTIME* (often abbreviated *P*) is the set of all problems that can be solved by some algorithm in polynomial time on a deterministic Turing machine. The class *EXPTIME* contains problems that can be solved in exponential time.[43] These classes are broad: *PTIME* doesn't distinguish between linear or cubic complexities, and *EXPTIME* does not distinguish between 2^n and 10^n, although these differences certainly do matter to us as programmers.

[42] *Sorting an array* is a problem that can be solved by many different algorithms.

[43] Obviously $PTIME \subset EXPTIME$.

In this lecture, we shall focus only on how to derive complexities of the first two kinds.

SELECTIONSORT

This is probably the simplest sorting algorithm to study.

Given an array A containing n values to sort in increasing order, SELECTIONSORT maintains the loop invariant depicted by Figure 28: at any iteration i, the values in the range $A[0..i-1]$ are the smallest i values of A in increasing order (i.e., this part is sorted and will not be changed), while the values in $A[i..n-1]$ are unsorted and larger than all the values in $A[0..i-1]$.

When $i = 0$ the array is completely unsorted, and when $i = n$ the array is fully sorted. Actually we can stop after $i = n-2$ because after this iteration the only "unsorted" value, $A[n-1]$, is necessarily the largest value in the array, so it is already at the right place.

To increase i while maintaining this invariant, all we need is to exchange $A[i]$ with the minimum value of $A[i..n-1]$. This gives us the following algorithm (illustrated by Figure 29):

	SELECTIONSORT(A, n)	(executions)
1	for $i \leftarrow 0$ to $n-2$ do	$n-1$
2	$min \leftarrow i$	$n-1$
3	for $j \leftarrow i+1$ to $n-1$ do	$(n-1)n/2$
4	if $A[j] < A[min]$	$(n-1)n/2$
5	$min \leftarrow j$	$\leq (n-1)n/2$
6	$A[min] \leftrightarrow A[i]$	$n-1$

Because line 1 iterates from 0 to $n-2$, we can easily tell that lines 1, 2, and 6 will be executed $n-1$ times. The other three lines are involved in two nested loops: for a given i the loop on line 3 will make $(n-1)+1-(i+1) = n-i-1$ iterations. We have to sum this for all i using for instance the formula from page 11:

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

Finally line 5 should have an execution count that is at most $(n-1)n/2$ since it is only executed if the previous comparison succeeds.

SELECTIONSORT performs $(n-1)n/2$ comparisons (line 4) and $n-1$ exchanges (line 6). Since the execution counts of all the other lines are also expressed using these quantities, we should be able to approximate[44] the total number of operations performed (or even the time to execute the algorithm) as a linear combination of these two quantities: a polynomial of the form $an^2 + bn + c$.

We can now predict the behavior of a SELECTIONSORT implementation after measuring it on a few arrays of different sizes. For instance, if an implementation gives the following timings:

size:	1,000	2,000	5,000	10,000	20,000	50,000	100,000	200,000
time:	0.001166	0.004356	0.018224	0.052226	0.173569	0.921581	3.678394	14.70667

we can use a least-square regression to fit these points to the following quadratic polynomial as shown on Figure 30:

$$\underbrace{3.676 \cdot 10^{-10}}_{a} \times n^2 \underbrace{- 4.422 \cdot 10^{-8}}_{b} \times n + \underbrace{9.810 \cdot 10^{-3}}_{c}$$

Now we can estimate we need 24.5 minutes to sort 2,000,000 values.

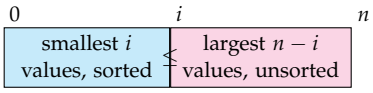


Figure 28: Loop invariant for SELECTIONSORT.

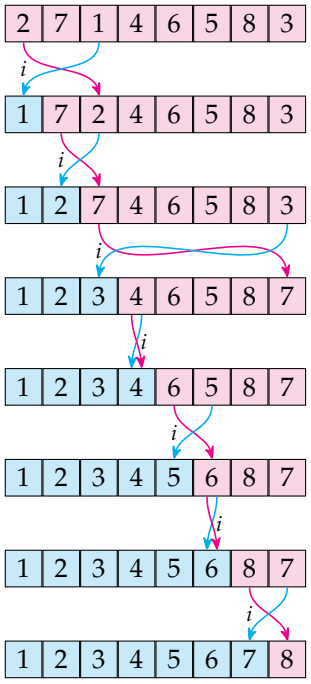


Figure 29: The colored arrows show the exchanges performed on line 6 by SELECTIONSORT. Exactly $n-1$ swaps are needed to sort an array of n elements.

[44] This is not *exactly* true, because line 5 may not be proportional to any of those. However, because line 5 is simple and execute less often than the comparisons, it should have little influence in practice.

```
>>> import numpy as np
>>> x=[1000,2000,5000,10000,
... 20000,50000,100000,200000]
>>> y=[0.001166,0.004356,
... 0.018224,0.052226,0.173569,
... 0.921581,3.678394,14.70667]
>>> p = np.polyfit(x, y, deg=2)
>>> print(p)
[ 3.67619503e-10 -4.42217128e-08
9.81022492e-03]
>>> np.polyval(p, 2000000)/60
24.506656289555412
```

Figure 30: Using `numpy` to find a quadratic polynomial that is a best fit (in a least-square sense) to our data set, and then predict the run time of our implementation on an input of size 2,000,000.

INSERTIONSORT

While INSERTIONSORT's loop invariant (Figure 31) looks similar to the invariant of SELECTIONSORT (page 19), it is actually more relaxed: there is no requirement for all the sorted values in $A[0..i-1]$ to be smaller than the unsorted values in $A[i..n-1]$. We can start with $i = 1$ because a sub-array of size 1 is always sorted. To increase i, it is necessary to *insert* $A[i]$ at the correct position in the sorted range, shifting some values to the right to make some room. The array will be sorted when we reach $i = n$, i.e., after the iteration for $i = n-1$.

There are actually a couple of ways to implement the shift-to-insert procedure.[45] The pseudo-code below (illustrated by Figure 32) scans the sorted values from right to left, shifting right all values greater than the one we want to insert (stored in the variable *key*), and until it finds a smaller value or the start of the array.

	INSERTIONSORT(A, n)	(executions)
1	for $i \leftarrow 1$ to $n-1$ do	$n-1$
2	$key \leftarrow A[i]$	$n-1$
3	$j \leftarrow i-1$	$n-1$
4	while $j \geq 0$ and $A[j] > key$ do	$\sum_i (t_i + 1)$
5	$A[j+1] \leftarrow A[j]$	$\sum_i t_i$
6	$j \leftarrow j-1$	$\sum_i t_i$
7	$A[j+1] \leftarrow key$	$n-1$

Lines 1, 2, 3, and 7 are obviously always executed $n-1$ times. However, we are not able to give a precise count for lines 4, 5, and 6. If we let t_i denote the number of iterations of the *while* loop for a given i, then we can write that lines 5 and 6 are both executed $\sum_{i=1}^{n-1} t_i$ times. Similarly line 4 is executed $\sum_{i=1}^{n-1}(t_i + 1)$ times because the condition has to be evaluated one more time before deciding to exit the *while* loop.

Our problem is that the actual value of t_i depends on the contents of the array A to sort, so we cannot compute a precise number of operations that is independent of A. Instead let us look at some extreme cases: what are the best and worst scenarios?

- The best case is when lines 5 and 6 are never executed.[46] In that case, $t_i = 0$ for all i, and line 4 is executed $\sum_{i=1}^{n-1}(t_i + 1) = n - 1$ times. The entire algorithm therefore executes a number of operations that is proportional to $n-1$, i.e., it is a linear function.

- The worst case is when t_i is maximal for each iteration.[47] In that case the *while* loop executes its body for all values between $j = i-1$ and $j = 0$, i.e., it performs $t_i = i$ iterations for a given i. The number of executions of lines 5 and 6 is therefore $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$ while lines 4 runs $\sum_{i=1}^{n-1}(i+1) = \frac{(n+2)(n-1)}{2}$ times. In this scenario, the total number of operations is a quadratic polynomial.

We conclude that INSERTIONSORT is quadratic in the worst case, and linear in the best case.[48]

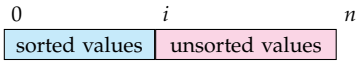


Figure 31: Loop invariant for INSERTIONSORT.

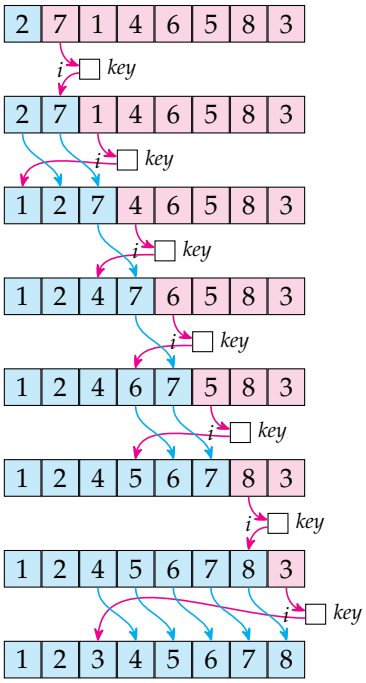


Figure 32: Running INSERTIONSORT on an example. For each iteration the purple arrows represent the assignments on lines 2 and 7, while the blue arrows are those from lines 5.

[45] Another option worth investigating is to locate the position to insert with a binary search, and then shift all values at once using `memmove()` or equivalent.

[46] For this to occur, *key* (which contains $A[i]$) must always be larger or equal to $A[i-1]$, i.e., A must already be sorted.

[47] This happens when *key* is smaller than all values in $A[0..i-1]$ and the *while* loop stops when $j < 0$.

[48] Can you guess how INSERTIONSORT behaves on the *average*? See page 21.

Average-Case Analysis

Knowing the worst and best case complexities of some algorithm is important, but it does not really tells us how it behaves *usually*. This is where an average analysis can be helpful: if possible we would like to consider all possible inputs of size n, compute the number of operations performed on each of them, and average the result. This procedure is not really practical, because we are usually not able (or willing) to compute a complexity for each individual case. Therefore, we resort to statistics and probabilities, making some hypothesis on the distribution of inputs. Instead of averaging on all possible inputs, we will also usually consider only different possible *shapes* of inputs, maybe with different probabilities.

Let us consider InsertionSort from page 20 again, and assume for simplicity that all the values in the array A are different. Although the two arrays of Figure 33 are different, from the point of view of the sorting algorithm they correspond the same input order and they can be sorted with the exact same operations.

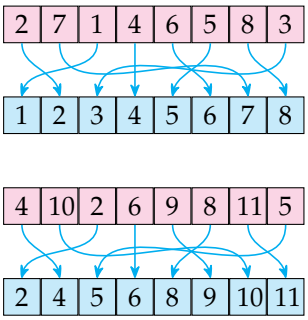


Figure 33: Two different arrays that have the same input order will be handled similarly by the sorting algorithm.

So instead of averaging InsertionSort over all inputs of size n, we will only consider all possible input orders. Each order can be given as a permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ of $\{1, 2, \ldots, n\}$, and there are $n!$ such permutations possible. For instance, the input order of the two arrays in Figure 33 corresponds to the permutation $(2, 7, 1, 4, 6, 5, 8, 3)$.

Given a permutation π, we say that (i, j) is an *inversion* if $i < j$ and $\pi_i > \pi_j$. For instance, the permutation $(2, 7, 1, 4, 6, 5, 8, 3)$ has 11 inversions: $(1, 3)$, $(2, 3)$, $(2, 4)$, $(2, 5)$, $(2, 6)$, $(2, 8)$, $(4, 8)$, $(5, 6)$, $(5, 8)$, $(6, 8)$, and $(7, 8)$. Note that the sorted permutation $(1, 2, 3, 4, 5, 6, 7, 8)$ contains no inversion, while the reverse permutation $(8, 7, 6, 5, 4, 3, 2, 1)$ contains $n(n-1)/2$ inversions[49]. At every iteration i of InsertionSort, when t_i values are shifted right to insert $A[i]$ to their left, exactly t_i inversions are canceled. The total number of executions of line 5 of InsertionSort, i.e., $\sum_{i=1}^{n-1} t_i$ is therefore equal to the number of inversions in the input array.[50]

Back to our average-case analysis. To count how many times line 5 will be executed on average[51] we only need to know the average number of inversions in a permutation of size n. For each permutation $(\pi_1, \ldots, \pi_i, \ldots, \pi_i, \ldots, \pi_n)$ that contains the inversion (i, j), there exists a permutation $(\pi_1, \ldots, \pi_j, \ldots, \pi_i, \ldots, \pi_n)$ that does not contain this inversion. This one-to-one mapping means that each inversion (i, j) has exactly $\frac{1}{2}$ chance to occur in a random permutation.[52] The expected number of inversions in a random permutation is therefore $\frac{1}{2}\binom{n}{2}$, that is the number of possible inversion multiplied by their probability to occur.

We conclude that on the average case, lines 5 and 6 are executed $\frac{1}{2}\binom{n}{2} = \frac{n(n-1)}{4}$ times[53], which is just half of our worst case scenario. The average number of operations of InsertionSort is therefore a quadratic function.

[49] This is the maximum number of permutations. Indeed, every permutation is a pair (i, j) satisfying $i < j$. There are $n(n-1)$ pairs, and half of them satisfies $i < j$.

[50] We counted 11 inversions for $(2, 7, 1, 4, 6, 5, 8, 3)$, and you can check that there are indeed 11 blue arrows on Figure 32 (page 20).

[51] We should write this as $E[\sum_{i=1}^{n-1} t_i]$, that is, the *expected* sum of all t_is.

[52] Because half of all the $n!$ existing permutations have the inversion (i, j), and the remaining half does not.

[53] The average number of executions of line 7 is left as an exercise to the reader.

BinarySearch

So far we have studied two iterative algorithms, but we should also know how to deal with recursive algorithms. As a very simple example, let us consider BinarySearch.

This takes a sorted array $A[b..e-1]$, a value v, and returns the index where v is in A, or where it should be in case it is not.

```
BinarySearch(A, b, e, v)
 1  if b < e then
 2      m ← ⌊(b + e)/2⌋
 3      if v = A[m] then
 4          return m
 5      else
 6          if v < A[m] then
 7              return BinarySearch(A, b, m, v)
 8          else
 9              return BinarySearch(A, m + 1, e, v)
10  else
11      return b
```

The algorithm first checks whether the middle value $A[m]$ is equal to v, otherwise it looks for v recursively in either $A[b..m-1]$ or $A[m+1..e-1]$.

Let us write $n = e - b$ for the size of the array, and $S(n)$ for the number of calls (including recursive calls) to BinarySearch needed to locate a value in the worst case. Clearly $S(0) = 1$ because calling BinarySearch with $b = e$ will return immediately. For $n \geq 1$, the worst-case scenario is when the value is never found, and the recursion always occurs on the larger of the two halves. Since one value has been removed, these halves have length $\lfloor (n-1)/2 \rfloor$ and $\lceil (n-1)/2 \rceil = \lfloor n/2 \rfloor$, and the latter is the larger one. Therefore, in the worst case, the number of calls to BinarySearch satisfies

$$S(n) = \begin{cases} 1 & \text{when } n = 0, \\ 1 + S(\lfloor n/2 \rfloor) & \text{when } n \geq 1. \end{cases}$$

You can actually solve this recursive equation (i.e., find a formula for $S(n)$) as if you had to replace a recursive implementation of this function (Figure 35) by an iterative version. Every time S is called recursively, its argument is divided by two, and 1 is added to the result. We could do this in a simple loop, as in Figure 36. So $S(n)$ is equal to 1 plus the number of times we need to perform an integer division of n by 2 to reach 0. This integer division is similar to a right shift by one bit, so $S(n)$ is equal to 1 plus the number of bits needed to represent n.[54] In other words:

$$S(n) = 2 + \lfloor \log_2(n) \rfloor \text{ if } n \geq 1, \text{ and } S(0) = 1$$

From this formula, we have a pretty good idea of the behavior of BinarySearch. Since the number of operations performed during each of these calls can be bounded by some constant c, the run time of BinarySearch cannot exceed $c \times S(n)$ in the worst-case scenario.[55]

BinarySearch$(A, 0, 8, 7)$:

	b			m				e
2	4	5	6	8	9	10	11	

BinarySearch$(A, 0, 4, 7)$:

	b		m		e			
2	4	5	6	8	9	10	11	

BinarySearch$(A, 3, 4, 7)$:

			bm	e				
2	4	5	6	8	9	10	11	

BinarySearch$(A, 4, 4, 7)$:

				be				
2	4	5	6	8	9	10	11	

return 4

Figure 34: Recursive call to BinarySearch showing the evolution of b and e (and the calculated m) when searching for the value 7 in the array.

```
unsigned s(unsigned n)
{
  if (n == 0) return 1;
  return 1 + s(n/2);
}
```

Figure 35: Straightforward, recursive implementation of $S(n)$.

```
unsigned s(unsigned n)
{
  unsigned res = 1;
  while (n != 0)
    {
      ++res;
      n /= 2;  // same as  n >>= 1
    }
  return res;
}
```

Figure 36: Iterative implementation of $S(n)$.

[54] The number $(110010)_2 = 2^5 + 2^4 + 2^1$, needs 6 bits, because its left-most 1-bit is the number 5 (counting from 0). We have $2^5 \leq m < 2^6$ hence $5 \leq \log_2(m) < 6$. More generally, the number of the left-most 1-bit in the binary representation of any non-negative integer m is $\lfloor \log_2(m) \rfloor$, and since bits are numbered from 0, the number of bits needed to represent m in base 2 is $1 + \lfloor \log_2(m) \rfloor$.

[55] Later, using notation introduced on page 23 we shall write that BinarySearch is a $O(\log n)$ algorithm for this reason.

Definitions for Big-Θ, Big-O, and Big-Ω Notations[56]

When we computed the number of operations performed by SELECTIONSORT (page 19) we concluded its run time should be a polynomial of the form $an^2 + bn + c$, and after running some experiments we even actually computed the values of a, b, and c. Of course these coefficients will be different if the same code is compiled differently, or executed on a different computer. However, the shape of the function $an^2 + bn + c$ is independent of these implementation details: the run time of SELECTIONSORT has to be a second-order polynomial. Most importantly, when n tends towards ∞ the most important term in this function will be an^2 and the $bn + c$ part will be negligible. We like to remember SELECTIONSORT as a *quadratic algorithm*, because n^2 is the dominant term in its complexity function.

The Θ, O, and Ω notations help making calculations using these dominant terms without bothering with all the implementation-related constants like a, b, and c.

- $f(n) \in \Theta(g(n))$ expresses the fact that $f(n)$'s asymptotic behavior[57] is comparable to $g(n)$, up to some multiplicative factor. For instance $an^2 + bn + c \in \Theta(n^2)$. We say that SELECTIONSORT's complexity is $\Theta(n^2)$.

  The formal definition of $f(n) \in \Theta(g(n))$ states that there must exist two positive constants c_1 and c_2 so that $f(n)$ is bounded below by $c_1 g(n)$ and bounded above by $c_2 g(n)$ for large values of n. This is illustrated by Figure 37.

$$\Theta(g(n)) = \left\{ f(n) \;\middle|\; \begin{array}{l} \exists c_1 > 0, \exists c_2 > 0, \exists n_0 \in \mathbb{N}, \\ \forall n \geq n_0,\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \end{array} \right\}$$

- $f(n) \in \mathrm{O}(g(n))$ expresses the fact that $f(n)$'s asymptotic behavior is dominated by $g(n)$, up to some multiplicative factor. For instance, INSERTIONSORT's complexity[58] can range from linear to quadratic depending on its input, so we can say it is in $\mathrm{O}(n^2)$, meaning its order is at most quadratic.

  $\mathrm{O}(g(n))$ can be defined as the set of all functions whose magnitude is bounded above by $cg(n)$ for some $c > 0$ and large n:

$$\mathrm{O}(g(n)) = \{f(n) \mid \exists c > 0,\ \exists n_0 \in \mathbb{N},\ \forall n \geq n_0, |f(n)| \leq cg(n)\}$$

- $f(n) \in \Omega(g(n))$ expresses the fact that $f(n)$'s asymptotic behavior dominates $g(n)$, up to some multiplicative factor. For instance, INSERTIONSORT's complexity is in $\Omega(n)$ since it is at least linear but may be larger.

  $\Omega(g(n))$ can be defined as the set of all functions bounded below by $cg(n)$ for some $c > 0$ and large n:

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0,\ \exists n_0 \in \mathbb{N},\ \forall n \geq n_0,\ 0 \leq cg(n) \leq f(n)\}$$

  These definitions imply that $\Theta(g(n)) = \mathrm{O}(g(n)) \cap \Omega(g(n))$.

[56] Those are sometimes called Landau's notations, although what Landau really invented was the small o notation. For some history about the notations, read "Big Omicron and Big Theta and Big Omega" by D. Knuth.

[57] i.e. when $n \to \infty$

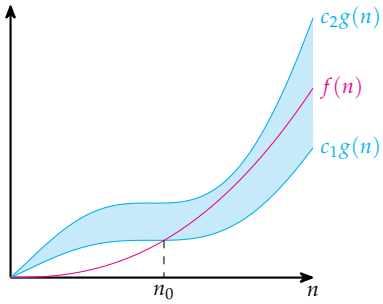


Figure 37: $f(n) \in \Theta(g(n))$: after some n_0 the function $f(n)$ is bounded by two copies of $g(n)$ with different scale factors.

[58] cf. page 20

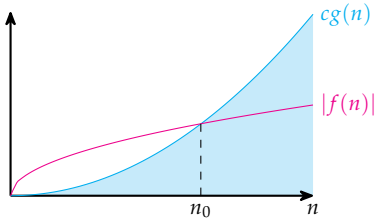


Figure 38: $f(n) \in \mathrm{O}(g(n))$: after some n_0 the function $|f(n)|$ is bounded above by $cg(n)$ for some constant c.

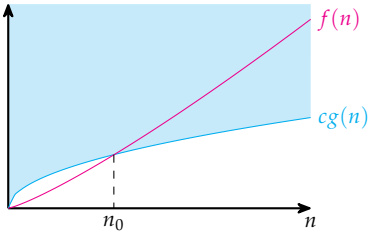


Figure 39: $f(n) \in \Omega(g(n))$: after some n_0 the function $f(n)$ is bounded below by $cg(n)$ for some constant c.

Properties of Big-Θ, Big-O, and Big-Ω Notations

Although $\Theta(g(n))$, $O(g(n))$, and $\Omega(g(n))$ are defined as sets of functions, we often abuse the notation to mean *one function in this set*. For instance, we would write $\Theta(n) + \Theta(n^2) = \Theta(n^2)$, which we can read as "*any linear function added to any quadratic function is a quadratic function*"[59], although a more rigorous way to write this would be $\{f(n) + g(n) \mid f(n) \in \Theta(n),\ g(n) \in \Theta(n^2)\} \subseteq \Theta(n^2)$.

With the above convention in mind, we have the following simplifications, where $f(n)$ and $g(n)$ are positive functions[60] and $\lambda > 0$ is a positive constant:

$$\begin{aligned}
\lambda &= \Theta(1) & \lambda &= O(1)\\
f(n) &= \Theta(f(n)) & f(n) &= O(f(n))\\
\Theta(f(n)) + \Theta(g(n)) &= \Theta(f(n) + g(n)) & O(f(n)) + O(g(n)) &= O(f(n) + g(n))\\
\Theta(f(n) + g(n)) &= \Theta(\max(f(n), g(n))) & O(f(n) + g(n)) &= O(\max(f(n), g(n)))\\
\Theta(f(n)) \cdot \Theta(g(n)) &= \Theta(f(n) \cdot g(n)) & O(f(n)) \cdot O(g(n)) &= O(f(n) \cdot g(n))\\
\Theta(\lambda f(n)) &= \Theta(f(n)) & O(\lambda f(n)) &= O(f(n))
\end{aligned}$$

These equalities, which can be proved[61] from the definitions of Θ and O given on page 23, hold for Ω as well. Following these rules we have that $4n^2 + 3n + 1 = \Theta(4n^2 + 3n + 1) = \Theta(4n^2) = \Theta(n^2)$, but we can generalize this to any polynomial: $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 = \Theta(n^k)$.

Things get a little fancier when we combine Θ, O and Ω. For instance, we have $\Theta(n^2) + O(n^2) = \Theta(n^2)$ because the sum of a quadratic function with a function that is *at most* quadratic will always be quadratic, and we have $\Theta(n^2) + \Omega(n^2) = \Omega(n^2)$ because the sum of a quadratic function with a function that is *at least* quadratic will be at least quadratic.

When $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \ell$ exists, we can use its value to decide whether $f(n)$ belongs to $\Theta(g(n))$, $O(g(n))$, or $\Omega(g(n))$:

$$\begin{aligned}
&\text{if } \lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0 && \text{then } f(n) = \Theta(g(n))\\
&\text{if } \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 && \text{then } f(n) = O(g(n)) \text{ and } f(n) \neq \Theta(g(n))\\
&\text{if } \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty && \text{then } f(n) = \Omega(g(n)) \text{ and } f(n) \neq \Theta(g(n))
\end{aligned}$$

Note that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ is the definition $f(n) = o(g(n))$. We actually have $o(g(n)) \subset O(g(n)) \setminus \Theta(g(n))$. Similarly, people occasionally write $f(n) = \omega(g(n))$ when $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, so that we have $f(n) = o(g(n)) \iff g(n) = \omega(f(n))$ just like we have $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$.

See Figure 40 for a Venn diagram showing how these different sets relate to each other.

Exercises. **1.** Show that $1 + \sin(n) + n$ is in $\Theta(n)$. **2.** Show that for any a and any $b > 0$, the function $(n + a)^b$ is in $\Theta(n^b)$. **3.** Show that $n + n\sin(n)$ is in $O(n)$ but is not in $\Theta(n)$. **4.** Show that $2n + n\sin(n)$ is in $\Theta(n)$. **5.** Prove $\Theta(\log_i n) = \Theta(\log_j n)$ for any $i > 1$ and $j > 1$.

[59] Note that this equality really goes one way only: in this context the notation "=" works like the word "*is*" in English. For instance, "$\Theta(n) = O(n^2)$" means that any function in $\Theta(n)$ is in $O(n^2)$, but the reverse does not hold.

[60] Since we are concerned with a number of operations performed by some algorithm, we will (almost) always have positive functions, and they will usually be increasing.

[61] Do not trust me, try it.

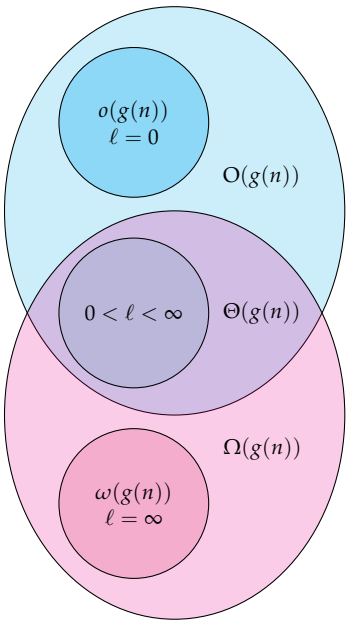


Figure 40: Relation between $o(g(n))$, $O(g(n))$, $\Theta(g(n))$, $\Omega(g(n))$, and $\omega(g(n))$. If the limit $\ell = \lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists, $f(n)$ belongs to one of the round classes.

Usage of Big-Θ, Big-O, and Big-Ω Notations

Let us consider again SELECTIONSORT[62] and show how to annotate it with these notations to derive its complexity.

[62] cf. page 19

SELECTIONSORT(A, n)	
1 for $i \leftarrow 0$ to $n-2$ do	$\Theta(n)$
2 $\quad min \leftarrow i$	$\Theta(n)$
3 $\quad$ for $j \leftarrow i+1$ to $n-1$ do	$\Theta(n^2)$
4 $\quad\quad$ if $A[j] < A[min]$	$\Theta(n^2)$
5 $\quad\quad\quad min \leftarrow j$	$O(n^2)$
6 $\quad\quad A[min] \leftrightarrow A[i]$	$\Theta(n)$
	$\Theta(n^2)$

For each line, we essentially make the same computations as before: we know that lines 1, 2 and 6 are executed $n-1$ times, which is a linear function, so we simply write $\Theta(n)$. Also, we know that lines 3 and 4 will be executed $\sum_{i=0}^{n-2} n-i-1$ times, but we need not compute this sum precisely. Summing a linear function between a constant and n is like integrating[63] a linear function between a constant and n: it will give a quadratic function, so we simply write $\Theta(n^2)$. Finally, line 5 can be executed as many times as line 4, but it could be executed less, so we write $O(n^2)$ to indicate that this is an upper bound. Now the complexity of the SELECTIONSORT is simply the sum of the complexity of all its lines: $\Theta(n) + \Theta(n) + \Theta(n^2) + \Theta(n^2) + O(n^2) + \Theta(n) = \Theta(n^2)$. We write that SELECTIONSORT *runs* in $\Theta(n^2)$, or that its *time complexity*[64] is $\Theta(n^2)$. We shall often write $T(n) = \Theta(n^2)$ instead of *the time complexity is* $\Theta(n^2)$.

[63] cf. page 14

[64] When people say just *complexity* they usually mean *time complexity*, i.e., a class of functions like $\Theta(n^2)$ or $O(n^3)$, into which that function giving the run time of the algorithm for an input of size n (or equivalently the number of operations performed) belongs. Another complexity that can be studied is the *space complexity*: how many extra space does the algorithm require to process an input of size n. SELECTIONSORT only needs a constant amount of additional memory (for the variables i, j, and min) regardless of n, so its state-space complexity is $S(n) = \Theta(1)$

We can use similar annotation on INSERTIONSORT[65] and conclude that its complexity is $O(n^2)$:

[65] cf. page 20

INSERTIONSORT(A, n)	
1 for $i \leftarrow 1$ to $n-1$ do	$\Theta(n)$
2 $\quad key \leftarrow A[i]$	$\Theta(n)$
3 $\quad j \leftarrow i-1$	$\Theta(n)$
4 $\quad$ while $j \geq 0$ and $A[j] > key$ do	$O(n^2)$
5 $\quad\quad A[j+1] \leftarrow A[j]$	$O(n^2)$
6 $\quad\quad j \leftarrow j-1$	$O(n^2)$
7 $\quad A[j+1] \leftarrow key$	$\Theta(n)$
	$O(n^2)$

Such annotations can also be used with recursive algorithms (such as our presentation of BINARYSEARCH), but they produce a recursive equation that the complexity function must satisfy, and we will explain how to deal with those later.[66]

[66] Starting on page 28.

A Bestiary of Common Complexity Functions

We will often compare algorithms with different time complexities, saying, for instance, that a $\Theta(n^2)$ algorithm is better than a $\Theta(n^3)$ algorithm.[67] To visualize how far apart different complexity functions are, consider Table 2 at the bottom of this page. It assumes we have a computer that can execute 3×10^9 operations per second[68] and considers many complexity functions we will encounter later. This table assumes a precise count of operations, like n, not a complexity class like $\Theta(n)$, so just keep in mind that an algorithm with complexity $\Theta(n)$ should have a run time more or less proportional to what the table gives in the n column.

Here are some algorithms that illustrate each complexity class:

$\Theta(n)$ is the cost of computing the minimum or maximum value in an array of size n. It is also the worst-case complexity of searching a value in an unsorted array.[69]

$\Theta(\log n)$ is the worst-case complexity of searching a value in a sorted array using BINARYSEARCH.[70] It is also the worst-case complexity of searching a value in a balanced binary search tree.

$\Theta(n \log n)$ is the typical complexity of a *good* sorting algorithm that relies on comparisons to sort values.[71]

$\Theta(n^2)$ is the complexity of SELECTIONSORT[72] on an array of size n, or the complexity of adding two matrices of size $n \times n$.

$\Theta(n^3)$ is the complexity for the naive[73] algorithm to multiply two matrices of size $n \times n$. You probably do not want to use it to multiply two $100\,000 \times 100\,000$ matrices.

$\Theta(n^{\log_2(7)})$ is the complexity of multiplying two $n \times n$ matrices using Strassen's algorithm[74]. Note that $\log_2(7) \approx 2.81$ so even if the difference between 3 and 2.81 is small, you can appreciate the difference between n^3 and $n^{2.81}$.

$\Theta(2^n)$ arises naturally in many problems that enumerate all subsets of n elements. For instance, the determinization of a n-state finite automaton is an $O(2^n)$ algorithm, because it constructs an automaton that contains 2^n states in the worst case.

[67] Note that as soon as we use the Θ, O, or Ω notations, we are discussing only about the asymptotic complexity, i.e., when $n \to \infty$. It would be wrong to assume that an $\Theta(n^2)$ algorithm is *always* better than a $\Theta(n^3)$ algorithm, especially for small values of n. See for instance Figure 27 on page 18.

[68] If we assume that one operation is executed in one CPU cycle, we can think of it as a 3GHz computer.

[69] Because it is $\Theta(n)$ in the worst case, we would write that the search of a value in an unsorted array can be implemented by a $O(n)$ algorithm.

[70] Likewise, we would write that BINARYSEARCH is a $O(\log n)$ algorithm. Note that we do not specify the base of the log when writing $O(\log n)$, $\Theta(\log n)$, or $\Omega(\log n)$ because all logarithm functions are equal up to a constant factor.

[71] e.g., MERGESORT, page 28.

[72] cf. page 19

[73] The one that implements $c_{ij} = \sum_k a_{ik} b_{kj}$ as a triple loop.

[74] a clever way to recursively express such a product using 7 products of sub-matrices of size $\frac{n}{2} \times \frac{n}{2}$

Table 2: An algorithm that requires $f(n)$ CPU cycles to process an input of size n will execute in $f(n)/(3 \times 10^9)$ seconds on a 3GHz CPU. This table shows run times for different f and n.

input size n	number $f(n)$ of operations to perform							
	$\log_2 n$	n	$n \log_2 n$	n^2	$n^{\log_2(7)}$	n^3	2^n	
10^1	1.1 ns	3.3 ns	11.1 ns	33.3 ns	0.2 µs	0.3 µs	0.3	ms
10^2	2.2 ns	33.3 ns	0.2 µs	3.3 µs	0.1 ms	0.3 ms	1.3×10^{13}	y
10^3	3.3 ns	0.3 µs	3.3 µs	0.3 ms	88.1 ms	0.3 s	1.1×10^{284}	y
10^4	4.4 ns	3.3 µs	44.2 µs	33.3 ms	56.5 s	5.5 min	6.3×10^{3002}	y
10^5	5.5 ns	33.3 µs	0.5 ms	3.3 s	10.1 h	3.8 d		
10^6	6.6 ns	0.3 ms	6.6 ms	5.5 min	0.7 y	10.6 y		
10^7	7.8 ns	3.3 ms	77.5 ms	9.3 h	473.8 y	10 570.0 y		
10^8	8.9 ns	33.3 ms	0.9 s	28.6 d	30 402.1 y			
10^9	10.0 ns	0.3 s	10.0 s	10.6 y				
10^{10}	11.0 ns	3.3 s	1.8 min	1057.0 y				

Merging two Sorted Sub-Arrays

The MERGE procedure will be used on next page to build MERGE-SORT, a better sorting algorithm than what we have seen so far. MERGE takes an array A and three indices i, j, and k, such that the values in the sub-array $A[i..j-1]$ are sorted (in increasing order), and the values in $A[j..k-1]$ are also sorted. The goal is to reorganize all these values so that $A[i..k-1]$ is sorted (Figure 41).

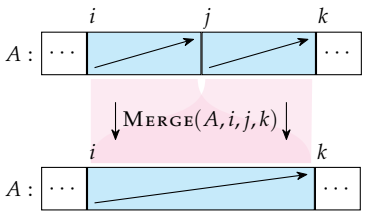


Figure 41: MERGE(A, i, j, k) takes two consecutive sorted sub-arrays $A[i..j-1]$ and $A[j..k-1]$ reorder the entire range.

```
MERGE(A, i, j, k)
 1  ℓ ← i                                  Θ(1)
 2  r ← j                                  Θ(1)
 3  for b ← i to k − 1 do                  Θ(n) for n = k − i
 4      if r = k or (ℓ < j and A[ℓ] ≤ A[r])  Θ(n)
 5          B[b] ← A[ℓ]                    O(n)
 6          ℓ ← ℓ + 1                      O(n)
 7      else
 8          B[b] ← A[r]                    O(n)
 9          r ← r + 1                      O(n)
10  A[i..k − 1] ← B[i..k − 1]              Θ(n)
                                           Θ(n)
```

The procedure works in two steps. First, lines 1–9, a temporary array B is filled with the sorted values, then, on line 10, the part of A that we had to sort is overwritten with the contents of B. This array B is supposed to be at least as large as A.

The actual merging, in lines 1–10, is done using three indices: ℓ (for *left*) points to the smallest unused value of $A[i..j-1]$, r (for *right*) points to the smallest unused value of $A[j..k-1]$, and b points to the current entry of B to fill. B is simply filled from left to right, with the smallest value between $A[\ell]$ and $A[r]$. Figure 42 shows an example with the various involved indices.

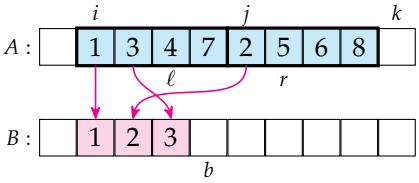


Figure 42: MERGE on an example, after the third iteration of its main loop. The arrows show previous executions of lines 5 or 8.

Of course at some point the value of one of the two sub-arrays will all be used: then either ℓ will reach j, or r will reach k. In these cases, the extra conditions on line 4 ensure that the remaining values will always be taken from the other sub-array.

If we use $n = k - i$ to denote the size of the range to sort, the complexity of MERGE is quite straightforward to establish. The loop on line 3 performs exactly n iterations, so lines 3 and 4 both account for $\Theta(n)$ operations. Lines 5, 6, 8, and 9 taken individually are each executed at most n times[75], so we write $O(n)$. Finally line 10 is a trap: it is actually copying n values from B to A, so it has to performs $\Theta(n)$ operations.

The total complexity is $\Theta(n)$: merging two sorted sub-arrays can be done in linear time.

[75] In fact lines 5 and 6 are necessarily executed $j - i$ times, while lines 8 and 9 are executed exactly $k - i$ times, so taken together these two groups of lines are executed n times. We could therefore lump all these four lines into a big $\Theta(n)$ but it would not change our result.

MERGESORT

"*Divide and conquer algorithms*"[76] are designed around the following idea: when faced with a complex problem, try to divide the problem in smaller sub-problems that are easier to solve (this is the *divide* step), and once these sub-problems are solved, use their solutions to construct a solution to the large problem (the *conquer* step). The division into smaller problems is usually done recursively until the problems are so small that their solutions are obvious.

The MERGESORT algorithm follows this idea: when given an unsorted array of size $n > 1$, it divides it into two unsorted arrays of size $n/2$ and recursively sorts those.[77] Once the two halves are sorted, the complete sorted array is built using the MERGE procedure described on page 27. Of course the recursive calls to sorts the arrays of size $n/2$ will probably divide the arrays into two arrays of size $n/4$. Eventually the recursion will stop on arrays of size 1: those are already sorted!

Here is the pseudo-code for MERGESORT. We assume that A, the array to be sorted between indices i (included) and j (excluded), will be modified in place. Figure 43 illustrates it.

MERGESORT(A, i, k)		$T(1)$	$T(n), n > 1$
1	if $k - i > 1$	$\Theta(1)$	$\Theta(1)$
2	$j \leftarrow \left\lfloor \frac{i+k}{2} \right\rfloor$		$\Theta(1)$
3	MERGESORT(A, i, j)		$T(\lfloor \frac{n}{2} \rfloor)$
4	MERGESORT(A, j, k)		$T(\lceil \frac{n}{2} \rceil)$
5	MERGE(A, i, j, k)		$\Theta(n)$

Let $n = k - i$ be the size of the array to sort, and let $T(n)$ denote the time complexity of MERGESORT. By looking at the pseudo-code, we can see that when $n = 1$, only the first line is executed in constant time, so $T(1) = \Theta(1)$. When $n > 1$, the first two lines cost $\Theta(1)$; then we have two recursive calls, one on an array of size $\lfloor \frac{n}{2} \rfloor$, and the other on an array of size $\lceil \frac{n}{2} \rceil$, those cost $T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil)$ operations; and finally we call MERGE on an array of size n, which we know costs $\Theta(n)$. The $\Theta(n)$ of line 5 dominates the $\Theta(1)$ of lines 1 and 2, so the complexity $T(n)$ is a function that satisfies

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

From these constraints, we can find what complexity class $T(n)$ belongs to. Can you guess the solution here? We will see different ways to solve this type of equations on the following pages.

Note that in practice we also have $T(2) = \Theta(1)$ and $T(3) = \Theta(1)$ because the number of operations needed to process a fixed-size input can always be bounded by a constant. So we usually write

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n)$$

without mentioning that $T(n) = \Theta(1)$.

[76] We will discuss this class of algorithms in more details later.

[77] Obviously this is a problem when n is odd, since the size of an array must be an integer. So in practice we have one sub-array of size $\lfloor \frac{n}{2} \rfloor$ and the other of size $n - \lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil$.

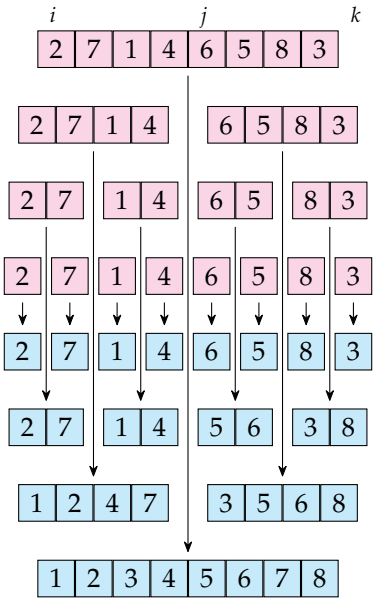


Figure 43: Running MERGESORT on an example. Each arrow represents one call to MERGESORT on the unsorted array above the arrow, and producing the sorted array at the bottom of the arrow. The two recursive calls are pictured on the sides of the arrow.

Exploring Recurrence Equations

Let us first consider recurrence equations that do not involve the Θ, O, Ω notations. For instance, let $M(n)$ denote the number of times line 4 of MERGE (page 27) is executed while running MERGESORT (page 28) on an array of length n. Since each call to MERGE on a sub-array of length n executes line 4 exactly n times, we have:

$$M(n) = \begin{cases} 0 & \text{if } n = 1 \\ M\left(\lceil \frac{n}{2} \rceil\right) + M\left(\lfloor \frac{n}{2} \rfloor\right) + n & \text{for } n \geq 2 \end{cases}$$

At first, the mix of $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$ might look intimidating. One can wonder if it would not be easier to solve equations such as

$$M_{floor}(n) = 2M_{floor}(\lfloor \tfrac{n}{2} \rfloor) + n \qquad \text{with } M_{floor}(1) = 0$$
$$\text{or} \qquad M_{ceil}(n) = 2M_{ceil}(\lceil \tfrac{n}{2} \rceil) + n \qquad \text{with } M_{ceil}(1) = 0$$

We can write a small program (Figure 44) to compute the first values from these functions and plot them (Figure 45 on this page, and Table 3 on next page). What can we make from this plot? First, we obviously have $M_{floor}(n) \leq M(n) \leq M_{ceil}(n)$ and this is easy to prove from our definitions. Then, these three functions coincide on values of n that are powers of 2: this should not be a surprise as $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$ are useless in this case. If $n = 2^m$, solving any of the these equations amounts to solving:

$$M(2^m) = 2M(2^{m-1}) + 2^m \text{ if } m \geq 1, \text{ and } M(2^0) = 0$$

Dividing everything by 2^m, we have $\frac{M(2^m)}{2^m} = \frac{M(2^{m-1})}{2^{m-1}} + 1$, and we can iterate this definition until we reach $M(2^0)$:

$$\frac{M(2^m)}{2^m} = \frac{M(2^{m-1})}{2^{m-1}} + 1 = \frac{M(2^{m-2})}{2^{m-2}} + 2 = \frac{M(2^{m-3})}{2^{m-3}} + 3 = \cdots = \frac{M(2^0)}{2^0} + m = m$$

So $M(2^m) = m2^m$ and since $m = \log_2 n$ it follows that $M(n) = n \log_2 n$, but only if n is a power of two. How far is $n \log_2 n$ from $M(n)$? After writing another small program, we can plot Figure 46: $M(n)$ appears closer to $n \log_2 n$ than M_{floor} and M_{ceil} are. From the same figure, we also easily see (this is not a proof) that all three functions satisfy $\frac{1}{2} n \log_2 n \leq M(n) \leq 2n \log_2 n$, which means that they are all in $\Theta(n \log n)$.

We will see later[78] that as long as all we want is a complexity class (such as $\Theta(n \log n)$), we can usually ignore the $\lceil \cdot \rceil$ or $\lfloor \cdot \rfloor$ functions in this type of recurrence equations.

However, if we need an exact solution these $\lceil \cdot \rceil$ or $\lfloor \cdot \rfloor$ functions do matter. Figure 45 leaves no doubt about that. On next page, we show how to compute an exact solution for $M(n)$.

```c
#include <stdio.h>

unsigned m(unsigned n)
{
  if (n == 1) return 0;
  return m(n / 2) + m(n - n/2) + n;
}

unsigned m_floor(unsigned n)
{
  if (n == 1) return 0;
  return 2 * m_floor(n / 2) + n;
}

unsigned m_ceil(unsigned n)
{
  if (n == 1) return 0;
  return 2 * m_ceil(n - n / 2) + n;
}

int main()
{
  for (unsigned n = 1; n <= 256; ++n)
    printf("%5u %5u %5u %5u\n",
        n, m_floor(n),
           m(n), m_ceil(n));
}
```

Figure 44: Computing $M(n)$, $M_{floor}(n)$, and $M_{ceil}(n)$ to draw Figure 45.

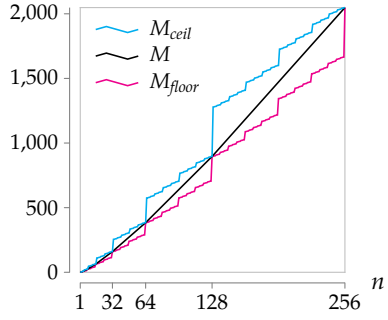


Figure 45: Plot of $M(n)$, $M_{floor}(n)$, and $M_{ceil}(n)$, as computed in Figure 44.

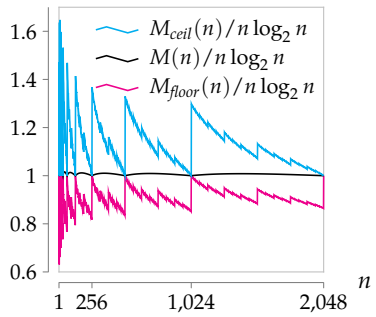


Figure 46: The ratio between the three M functions, and $n \log_2 n$.

[78] cf. page 31

❁ *Solving Recurrence Equations by Differentiation*

Let us consider the recurrence M from previous page:

$$M(n) = \begin{cases} 0 & \text{if } n = 1 \\ M\left(\lceil \frac{n}{2} \rceil\right) + M\left(\lfloor \frac{n}{2} \rfloor\right) + n & \text{for } n \geq 2 \end{cases}$$

We will solve this equation by calculating $U(n) = M(n+1) - M(n)$ and then realizing that $U(n)$ satisfies a recurrence equation we have already seen previously.

Notice first that $\lceil \frac{n}{2} \rceil = \lfloor \frac{n+1}{2} \rfloor$, so we can rewrite $M(n)$ using only $\lfloor \cdot \rfloor$:

$$\begin{aligned} M(n) &= M\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) + M\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \\ M(n+1) &= M\left(\left\lfloor \frac{n+2}{2} \right\rfloor\right) + M\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) + n + 1 \\ M(n+1) - M(n) &= M\left(\left\lfloor \frac{n+2}{2} \right\rfloor\right) - M\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 \\ M(n+1) - M(n) &= M\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) - M\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 \end{aligned}$$

Now if we let $U(n) = M(n+1) - M(n)$ we have

$$U(n) = \begin{cases} 2 & \text{if } n = 1 \\ U\left(\lfloor \frac{n}{2} \rfloor\right) + 1 & \text{if } n > 2 \end{cases}$$

Do you recognize this equation? For any $n \geq 1$ this definition of $U(n)$ is the same as the definition of $S(n)$ from page 22, so we can conclude that $U(n) = 2 + \lfloor \log_2 n \rfloor$.

Now, since we have $U(n) = M(n+1) - M(n)$, it follows that

$$\begin{aligned} M(n) &= M(1) + U(1) + U(2) + U(3) + \cdots + U(n-1) \\ M(n) &= 0 + \sum_{1 \leq i < n} U(i) \\ M(n) &= \sum_{1 \leq i < n} (2 + \lfloor \log_2 n \rfloor) \\ M(n) &= (n-1) + \sum_{1 \leq i < n} (1 + \lfloor \log_2 i \rfloor) \end{aligned}$$

And this is a sum we studied on page 15:

$$\sum_{i=1}^{n-1} (1 + \lfloor \log_2 i \rfloor) = n + 1 + n\lfloor \log_2 (n-1) \rfloor - 2^{\lfloor \log_2 (n-1) \rfloor + 1}$$

So we can conclude[79] that

$$M(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2n + n\lfloor \log_2 (n-1) \rfloor - 2^{\lfloor \log_2 (n-1) \rfloor + 1} & \text{if } n \geq 2 \end{cases}$$

n	$M_{floor}(n)$	$M(n)$	$M_{ceil}(n)$
1	0	0	0
2	2	2	2
3	3	5	7
4	8	8	8
5	9	12	19
6	12	16	20
7	13	20	23
8	24	24	24
9	25	29	47
10	28	34	48
11	29	39	51
12	36	44	52
13	37	49	59
14	40	54	60
15	41	59	63
16	64	64	64

Table 3: The first values of $M_{floor}(n)$, $M(n)$, $M_{ceil}(n)$, as defined on page 29.

[79] It is very easy to forget a $\lfloor \cdot \rfloor$ or a little -1 somewhere while making this kind of calculation. To detect such mistakes, I usually evaluate both formulas (here, the definition of $M(n)$ at the top of the page, and the one at the bottom) on a handful of values, and check that they are the same (here, the values should be those given by Table 3).

Master Theorem for Recurrence Equations

The following theorem can be used to solve many (but not all) recurrence equations derived from recursive algorithms, typically those obtained from divide-and-conquer algorithms.[80]

Theorem 1 (Master Theorem). *Consider a recurrence equation such as*

$$T(n) = \Theta(1) \qquad \text{for } n < n_0$$
$$T(n) = aT\left(\frac{n}{b} + O(1)\right) + f(n) \qquad \text{for } n \geq n_0$$

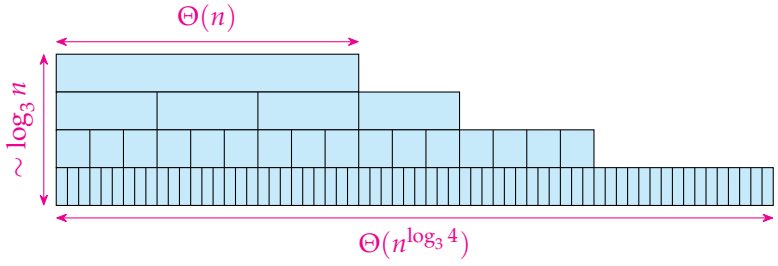
with $a \geq 1$, $b > 1$, *and* $n_0 > 0$.

1. *If* $f(n) = O(n^{(\log_b a)-\varepsilon})$ *for some* $\varepsilon > 0$, *then* $T(n) = \Theta(n^{\log_b a})$.
2. *If* $f(n) = \Theta(n^{\log_b a})$, *then* $T(n) = \Theta(n^{\log_b a} \log n)$.
3. *If* $f(n) = \Omega(n^{(\log_b a)+\varepsilon})$ *for some* $\varepsilon > 0$, ***and*** *if* $af\left(\frac{n}{b}\right) \leq cf(n)$ *for some* $c < 1$ *and all large values of* n, *then* $T(n) = \Theta(f(n))$.
4. *In other cases*[81], *the theorem does not apply.*

Figures 47–49 illustrate this theorem by picturing the work $f(n)$ performed by each recursive call as a rectangle.

Examples:

- $T(n) = T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) + \Theta(n)$. This is the recursive equation for the complexity of MergeSort, as established on page 28. We can rewrite it as[82]: $T(n) = 2T\left(\frac{n}{2} + O(1)\right) + \Theta(n)$.
  So we have $a = b = 2$ and $f(n) = \Theta(n)$. We compute $n^{\log_b a} = n^{\log_2 2} = n^1$, and we now have to check which case of the theorem applies. Is $f(n)$ in $O(n^{1-\varepsilon})$ (case 1), in $\Theta(n^1)$ (case 2), or in $\Omega(n^{1+\varepsilon})$ (case 3)? Obviously we are in the second case since $f(n) = \Theta(n) = \Theta(n^1)$. We therefore conclude immediately that MergeSort has complexity $T(n) = \Theta(n \log n)$.
- $T(n) = T(\lfloor \frac{n}{2} \rfloor) + \Theta(1)$. This is the worst case complexity of BinarySearch (page 22). We have $a = 1$, $b = 2$, and $f(n) = \Theta(1)$. $n^{\log_2 1} = n^0 = 1$. Again in case 2, we conclude that $T(n) = \Theta(n^0 \log n) = \Theta(\log n)$ for the *worst case* of BinarySearch.[83]
- $T(n) = \sqrt{n} + 3T(n/4)$. We have $b = 4$, $a = 3$, and $\log_4 3 \approx 0.792$. We have $\sqrt{n} = n^{1/2} = O(n^{(\log_4 3)-\varepsilon})$ if we take for instance $\varepsilon = 0.2$. So this is the first case of the theorem, an $T(n) = \Theta(n^{\log_4 3})$.
- $T(n) = n^2 + 3T(n/4)$. Same constants, different $f(n)$. This times, $n^2 = \Omega(n^{(\log_4 3)+\varepsilon})$ if we take for instance $\varepsilon = 1$. Furthermore, the function $f(n) = n^2$ verifies $4f(n/3) \leq cn^2$ if we take for instance $c = 1/2$, so $T(n) = \Theta(n^2)$.



[80] Divide-and-conquer algorithms will typically perform a recursive calls on sub-problems of size $\frac{n}{b}$, and use $f(n)$ operations to split the problem and merge the sub-solutions.

[81] These are the cases where ε or c cannot be found. For instance, if you consider $T(n) = 2T(n/2) + n \log_2 n$, you can show that $n \log_2 n = \Omega(n^1)$ but you cannot find any $\varepsilon > 0$ such that $n \log_2 n = \Omega(n^{1+\varepsilon})$, so the theorem does not apply.

[82] Note how the $\frac{n}{b} + O(1)$ in the theorem accommodates any terms like $\frac{n}{b}$, or $\lfloor \frac{n}{b} \rfloor$, or even $\lceil \frac{n+5}{b} \rceil$. This is great news: no need to worry about integer parts anymore!

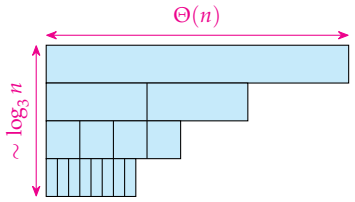


Figure 47: If $T(n) = 2T(n/3) + \Theta(n)$ we are in the third case of the theorem: the work performed by recursive calls diminishes exponentially fast, so only the initial $\Theta(n)$ matters.

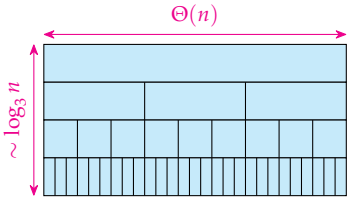


Figure 48: If $T(n) = 3T(n/3) + \Theta(n)$ we are in the second case of the theorem: the total work performed at each level of the recursion is the same, so the complexity $\Theta(n)$ has to be multiplied by $\Theta(\log n)$.

[83] So we can say that BinarySearch is $O(\log n)$ in general.

Figure 49: If $T(n) = 4T(n/3) + \Theta(n)$ we are in the first case of the theorem: the total work performed at each level increases exponentially, and the work performed on the last level dominates everything else.

Establishing Upper Bounds by Mathematical Induction

Here we review a way to prove that $T(n) = O(f(n))$ when you have some recursive equation for $T(n)$ but do not want to (or cannot) use the master theorem. However, like all inductive proofs, you need to know[84] the solution (i.e., $f(n)$) before you can start the proof.

To prove the $T(n) = O(f(n))$, we need to show that there exists some constant $c > 0$ and some $n_0 \in \mathbb{N}$ such that for values of n larger than n_0 we have $T(n) \leq cf(n)$.[85] Note that once we have a constant c that works, we can decide to use a larger c and the proof will still work. In fact, the ability to pick a c large enough is often necessary to complete the proof.

Ideally, we make our inductive proof as follows:

1. We write our inductive hypothesis, H_n : $\quad T(n) \leq cf(n)$.
2. We show that H_{n_0} holds, from some n_0 and c we supply.
3. We show that H_n holds for any $n > n_0$, if we assume H_{n_0}, H_{n_0+1}, ..., and H_{n-1}.
4. We conclude that $T(n) = O(f(n))$.

Sometimes we fail at step 3 and we may have to revise our definition of $f(n)$ before giving up.

Example: Assume we have $T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + \Theta(1) & \text{if } n > 1 \end{cases}$ and we want to prove that $T(n) = O(\log n)$.

Our inductive hypothesis H_n is that $T(n) \leq c \log_2 n$. Clearly we cannot prove H_1 because $T(1)$ is a constant necessary larger than $c \log_2 1 = 0$. However $T(2) = T(1) + \Theta(1) = \Theta(1)$ is a constant as well, so we can choose c large enough so that $T(2) \leq c \log_2 2 = c$, and H_2 holds. Similarly, $T(3) = T(2) + \Theta(1) = \Theta(1)$ is also a constant, so clearly H_3 holds if we keep c larger than these two constants.

Now for any $n \geq 4$ let us assume that $H_2, H_3, \ldots, H_{n-1}$ holds and let us prove H_n. By definition of $T(n)$ we have

$$T(n) = T(\lfloor n/2 \rfloor) + \Theta(1)$$

Since $n \geq 4$ we have $\lfloor n/2 \rfloor \geq 2$, so we use hypothesis $H_{\lfloor n/2 \rfloor}$:

$$T(n) \leq c \log_2 \left\lfloor \frac{n}{2} \right\rfloor + \Theta(1) \leq c \log_2 \frac{n}{2} + \Theta(1)$$
$$T(n) \leq c \log_2 n - (c - \Theta(1))$$

Now we argue that since we can pick c as large as we want, we can make sure that $c - \Theta(1)$ is positive. Therefore

$$T(n) \leq c \log_2 n$$

We have proved H_n, and by mathematical induction we conclude that $T(n) = O(\log n)$.[86]

It would be nice if it was always that easy!

[84] Guessing the solution is also an option: sometimes the recurrence equation looks like some equation you have already solved, or some equation that you would know how to solve, and it seems legitimate to estimate that the solution should be be similar. You would then use mathematical induction to confirm your guess.

[85] This is the definition of $T(n) = O(f(n))$, as seen on page 23.

[86] **Exercise**: Using the same recursive definition of $T(n)$, adapt this method to demonstrate that $T(n) = \Omega(\log n)$.

When Mathematical Induction on Recurrence Fails

The example from the previous page is a case where the proof goes well. This is not always the case and sometimes we have to adapt our induction hypothesis in order to complete the proof.

Consider the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\lfloor n/2 \rfloor) + 1 & \text{if } n > 1 \end{cases}$$

From our experience, we *guess* that $T(n) = \mathrm{O}(n)$, so let's try to prove the hypothesis $H_n : T(n) \leq cn$ for some constant c. Clearly H_1 is true. So for any $n > 1$, let us assume that $H_1, \ldots, H_{n-1}$ hold, and use that to prove H_n:

$$T(n) = 2T(\lfloor n/2 \rfloor) + 1$$

since $n > 1$, we have $1 \leq \lfloor n/2 \rfloor < n$ and can apply $H_{\lfloor n/2 \rfloor}$:

$$\begin{aligned} T(n) &\leq 2c\lfloor n/2 \rfloor + 1 \\ T(n) &\leq 2cn/2 + 1 \\ T(n) &\leq cn + 1 \end{aligned}$$

Unfortunately, this last equation does not imply H_n, so we cannot conclude our inductive proof.

The trick is to realize that we are just off by a constant, i.e., something negligible in front of the $\mathrm{O}(n)$ bound we are trying to establish. In order to get rid of this constant, we can introduce one in our hypothesis. Let us attempt the same proof with $H'_n : T(n) \leq cn - 1$. Hypothesis H'_1 still hold for a c large enough. Again, for any $n > 1$, let us assume that $H'_1, \ldots, H'_{n-1}$ hold, and use that to prove H'_n:

$$T(n) = 2T(\lfloor n/2 \rfloor) + 1$$

we apply $H'_{\lfloor n/2 \rfloor}$:

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor - 1) + 1 \\ T(n) &\leq 2c\lfloor n/2 \rfloor - 1 \\ T(n) &\leq 2cn/2 - 1 \\ T(n) &\leq cn - 1 \end{aligned}$$

Now, this is exactly H'_n, so by mathematical induction we have proved that $T(n) \leq cn - 1$ holds for all $n \geq 1$. This of course implies that $T(n) \leq cn$ for all $n \geq 1$ and hence that $T(n) = \mathrm{O}(n)$.

More Examples of Complexities

Let us apply the techniques we learned so far to different algorithms and operations on data structures. The presentation of those algorithms and data structures, which you should probably already know, is just a pretext to practice the computation of complexities.

The next sorting algorithm we study is HEAPSORT. It uses a data structure called *heap*, which is a *nearly complete binary tree* (see below) with some additional constraints.

Nearly Complete Binary Trees

A *nearly complete binary tree* is a binary tree in which all levels, except possibly the last, are fully filled, and furthermore, the nodes from the last level are filled from left to right. Figure 50 gives an example.

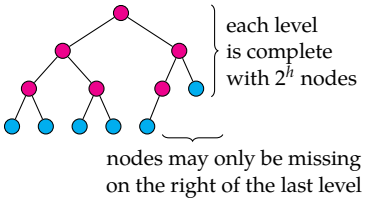


Figure 50: A nearly complete binary tree has all its levels complete, except maybe the last one where all nodes are flush left.

A nearly complete binary tree with n nodes can be efficiently represented as an array of n elements, as illustrated by Figure 51: the array is simply filled by reading the values of the tree one level after the other, i.e., from top to bottom and from left to right. The requirement that a nearly complete binary tree can only have missing nodes at the end of its last level stems from this array-based representation: we do not want any hole in the array.

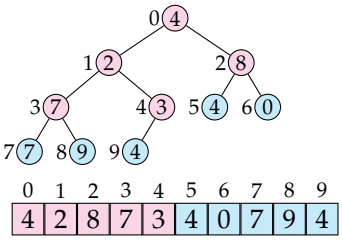


Figure 51: A nearly complete binary tree storing integers, and its representation as an array of integers.

This array-based representation is very space efficient since it does not need to store any pointer to parent and children. A node can be referred to by its index i in the array, and the index of its parent and children can be computed from i. Assuming the number of nodes (i.e., the size of the array) is known to be n, we have the following formulas[87]:

$$\begin{aligned} \textsc{LeftChild}(i) &= 2i+1 && \text{if } 2i+1 < n \\ \textsc{RightChild}(i) &= 2i+2 && \text{if } 2i+2 < n \\ \textsc{Parent}(i) &= \left\lfloor \frac{i-1}{2} \right\rfloor && \text{if } i > 0 \end{aligned}$$

[87] The formulas are different if array indices start at 1 instead of 0.

Furthermore, if a nearly complete binary tree has n nodes, we know it has exactly $\lfloor \frac{n}{2} \rfloor$ internal nodes and $\lceil \frac{n}{2} \rceil$ leaves. These leaves are necessarily stored at positions $\lfloor \frac{n}{2} \rfloor$ to $n-1$ in the array. This fact will be used in BUILDHEAP[88] to work on all subtrees but the leaves.

[88] cf. p. 35

Heaps

A *max-heap* is a *nearly complete binary tree* storing elements in an order that satisfies the following *heap constraint*: the value of any node must be greater than (or equal to) that of its children. A *min-heap* can be defined similarly (each node has a value less than that of its children), but we will only focus on max-heaps from now on.

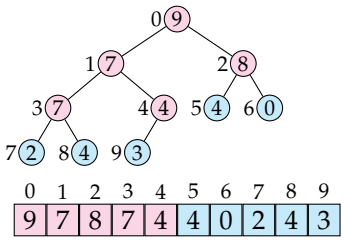


Figure 52: A (max-)heap storing the same set of values as in Fig. 51.

For instance, the heap of Figure 52 was built from the nearly complete binary tree of Figure 51 by applying the algorithm BUILDHEAP.

Max-heaps have the important property that the maximum value can always be found at its root. This can be used for sorting.

HEAPIFY and BUILDHEAP

The HEAPIFY function is the main building block for the BUILDHEAP algorithm. Let A be an array of size m storing a nearly complete binary tree. HEAPIFY takes the index i of a node whose left and right children are already known to be subtrees that satisfy the heap property, and it rearranges the values of i and its children so that the subtree rooted in i has the heap property. These conditions are illustrated by Figure 53.

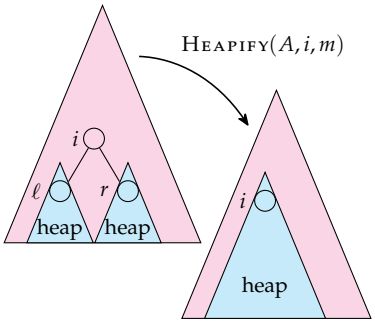
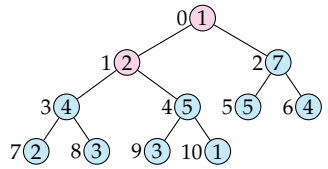


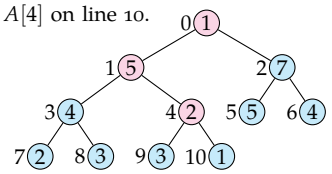
Figure 53: Pre- and post-conditions of Heapify. The input is a node i whose children subtrees are already known to satisfy the heap property. In the output the entire subtree rooted in i satisfies the heap property. This implies that $A[i]$ in the output should be equal to $\max(A[i], A[\ell], A[r])$ in the input.

Note that if the left child ℓ of i satisfies the heap property, its value $A[\ell]$ is necessarily the maximum of the left subtree. Similarly, $A[r]$ is the maximum of the right subtree. If $A[i]$ is already greater than $A[\ell]$ and $A[r]$, then the subtree rooted in i already satisfies the heap property. Otherwise, two of these three values have to be swapped: bringing the maximum at the top, and possibly destroying the heap property of one of the children (but this can be fixed recursively).

```
HEAPIFY(A, i, m)
 1   ℓ ← LEFTCHILD(i)                    Θ(1)
 2   r ← RIGHTCHILD(i)                   Θ(1)
 3   if ℓ < m and A[ℓ] > A[i]            Θ(1)
 4       g ← ℓ                           O(1)
 5   else
 6       g ← i                           O(1)
 7   if r < m and A[r] > A[g]            Θ(1)
 8       g ← r                           O(1)
 9   if g ≠ i                            Θ(1)
10       A[i] ↔ A[g]                     O(1)
11       HEAPIFY(A, g, m)                ?
```



When running HEAPIFY$(A, 1, 11)$ on the above tree, $A[1]$ is swapped with $A[4]$ on line 10.



The subtree of 4 is then corrected by calling HEAPIFY$(A, 4, 11)$ recursively.

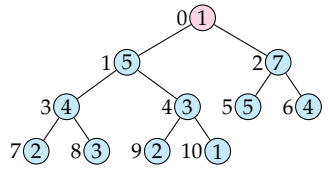


Figure 54: Execution of HEAPIFY$(A, 1, 11)$ on an example. States colored in blue are roots of subtrees with the heap property.

Figure 54 illustrates this algorithm on an example. Using HEAPIFY to turn a complete binary tree into a heap is now quite easy: notice that all leaves already satisfy the heap property, so all we need is to call HEAPIFY on the internal nodes, in a bottom-up way. Remember that the first leave is at position $\lfloor n/2 \rfloor$ in the array, so the last internal node is just before.

```
BUILDHEAP(A, n)
1   for i from ⌊n/2⌋ − 1 down to 0:   Θ(n)
2       HEAPIFY(A, i, n)              ?
```

Figure 55 runs BUILDHEAP on the nearly complete binary tree used as example on the previous page.

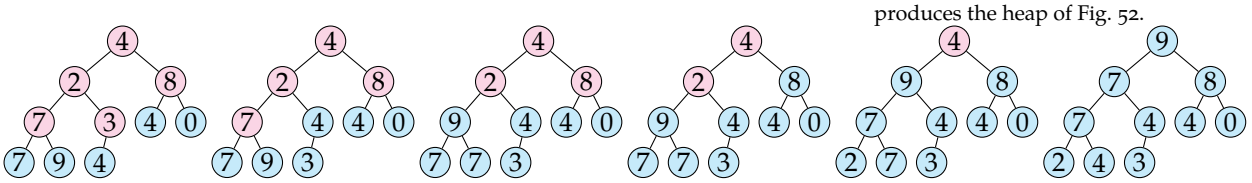


Figure 55: Running BUILDHEAP on the nearly complete binary tree from Fig. 51 produces the heap of Fig. 52.

The Complexity of HEAPIFY

Page 35 presents HEAPIFY and BUILDHEAP, but does not give their complexity.

HEAPIFY contains different execution branches. The most efficient scenario is obviously when $g = i$ on line 9, because then no recursion occurs. In this case, HEAPIFY executes in constant time.

For the recursive case, it is instructive to consider different ways to *measure* the size of the input.

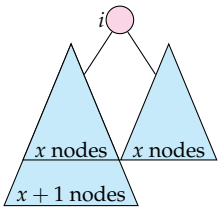


Figure 56: Worst case for the recursion of HEAPIFY: the left subtree has slightly more than twice the number of nodes of the right subtree. If $s = 3x + 2$, the left subtree has $2x + 1 = (2s - 1)/3$ nodes.

- HEAPIFY(A, i, m) will only work on nodes that belong to the subtree rooted in i. So we could use $T_H(s)$ to denote the time complexity of HEAPIFY on a subtree of s nodes. When HEAPIFY recurses into one of the two children of i, how many nodes are left in the worst case? To answer that, look at Figure 56: because the last level of a heap is not necessarily full, the left subtree can actually have twice the numbers of nodes of the right one. The left subtree can actually have up to $\lceil (2s-1)/3 \rceil = 2s/3 + \mathrm{O}(1)$ nodes. We therefore have the following recurrence:

$$T_H(s) \leq \begin{cases} \Theta(1) & \text{if } s = 1 \\ T_H(2s/3 + \mathrm{O}(1)) + \Theta(1) & \text{if } s > 1 \end{cases}$$

  This is not exactly the form of the Master theorem[89] because of the inequality. However, we can use the Master theorem to find that $U(s) = U(2s/3 + \mathrm{O}(1)) + \Theta(1)$ has for solution $U(s) = \Theta(\log s)$, and from that we conclude:

$$T_H(s) \leq U(s) = \Theta(\log s) \qquad \text{hence} \qquad T_H(s) = \mathrm{O}(\log s).$$

- Another option is to express the complexity $T_H(h)$ of HEAPIFY working on a subtree of height h. Each recursive call reduces the height by one, so we have $T_H(h) \leq T_H(h-1) + \Theta(1)$ until we handle a leaf with $T_H(0) = \Theta(1)$. By iterating this definition, we easily find that $T_H(h) = \mathrm{O}(h)$:

$$\begin{aligned} T_H(h) &\leq T_H(h-1) + \Theta(1) \\ T_H(h) &\leq T_H(h-2) + \Theta(1) + \Theta(1) \\ &\vdots \\ T_H(h) &\leq T_H(0) + \underbrace{\Theta(1) + \ldots + \Theta(1)}_{h \text{ terms}} \\ T_H(h) &\leq (h+1)\Theta(1) \\ T_H(h) &\leq \Theta(h) \\ T_H(h) &= \mathrm{O}(h) \end{aligned}$$

Note that these two results, $T_H(s) = \mathrm{O}(\log s)$ and $T_H(h) = \mathrm{O}(h)$, are compatible because $h = \Theta(\log s)$ for complete binary trees.[90]

We will use both expressions for T_H on next page, to compute the complexity of BUILDHEAP.

[89] cf. p. 31

[90] **Exercise**: Prove that any complete binary tree of s nodes has a height of exactly $h = \lfloor \log_2 s \rfloor$.

The Complexity of BUILDHEAP

BUILDHEAP(A, n)	
1 for i from $\lfloor n/2 \rfloor - 1$ down to 0:	$\Theta(n)$
2 HEAPIFY(A, i, n)	?

Having established the complexity of HEAPIFY on page 36, we only need to answer one question before we can give complexity $T_{BH}(n)$ of running BUILDHEAP: *"what is the cost of line 2?"*

- We can consider that in the worst case, HEAPIFY runs on a subtree of n nodes. This is the case when called with $i = 0$ and the HEAPIFY call then costs $T_H(n) = \mathrm{O}(\log n)$. It costs less in the other iterations, but $\mathrm{O}(\log n)$ already gives an upper bound anyway. Since there are $\lfloor n/2 \rfloor$ iterations, the total complexity can be expressed as follows:

$$T_{BH}(n) = \underbrace{\Theta(n)}_{\text{line 1}} + \underbrace{\lfloor n/2 \rfloor \mathrm{O}(\log n)}_{\text{line 2}}$$
$$T_{BH}(n) = \Theta(n) + \Theta(n)\mathrm{O}(\log n)$$
$$T_{BH}(n) = \mathrm{O}(n \log n)$$

  However, that is a crude upper bound, because we considered that all calls to HEAPIFY cost as much as the last one.

- In practice, HEAPIFY is called on many small subtrees where it has constant cost. For instance, on all subtrees of height 1, HEAPIFY costs $T_H(1) = \Theta(1)$. A more precise evaluation of line 2 would therefore account for the different sizes of each subtree considered. Let $S(h, n)$ be the number of subtrees of height h in a heap of size n. We can express the complexity of BUILDHEAP as:

$$T_{BH}(n) = \underbrace{\Theta(n)}_{\text{line 1}} + \underbrace{\sum_{h=1}^{\lfloor \log n \rfloor} S(h, n) T_H(h)}_{\text{line 2}} \qquad (10)$$

  Indeed: we have $S(h, n)$ subtrees of height h, the call to BUILDHEAP costs $T_H(h)$ for each of them, and we are running BUILDHEAP on all subtrees with heights ranging from 1 (the node just above the leaves) to $\lfloor \log n \rfloor$ (for the root[91]).

  Finding an exact formula for $S(h, n)$ is tricky, but we can establish the upper bound $S(h, n) \leq n/2^h$ as shown in Figure 57. From that we have:

$$\sum_{h=1}^{\lfloor \log n \rfloor} S(h, n) T_H(h) \leq \sum_{h=1}^{\lfloor \log n \rfloor} \frac{n\mathrm{O}(h)}{2^h} = n\,\mathrm{O}\left(\sum_{h=1}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = \mathrm{O}(n)$$

  The trick is to recognize the sum as the start of a series that converges[92], so it can be reduced to $\mathrm{O}(1)$. Plugging this in equation (10), we get:

$$T_{BH}(n) = \Theta(n) + \mathrm{O}(n) = \Theta(n)$$

  A complexity that is both lower (n versus $n \log n$) and more precise (Θ versus O) than our first attempt!

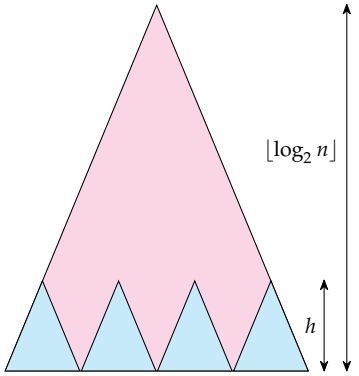


Figure 57: The number of subtrees of height h in a *complete binary tree* of n nodes *without missing nodes on the last level*, can be expressed as the number of nodes at depth $d = \lfloor \log_2 n \rfloor - h$, that is $2^{\lfloor \log_2 n \rfloor - h}$. This value is smaller or equal to $2^{\log_2(n) - h} = n/2^h$.

Now if the *binary tree* is *nearly complete* (i.e., it has missing nodes), $n/2^h$ is still an upper bound of the number of subtrees with height h.

So we conclude that $S(h, n) \leq n/2^h$.

[91] See the remark 90 on p. 36.

[92] Start from $\sum_{k=0}^{\infty} r^k = \frac{1}{1-r}$ which can be established for any $|r| < 1$ from eq. 7 p. 12. Differentiate both sides w.r.t. r, then multiply by r to obtain $\sum_{k=0}^{\infty} k r^k = \frac{r}{(1-r)^2}$. In our case we have $r = \frac{1}{2}$ and $\sum_{k=0}^{\infty} k \left(\frac{1}{2}\right)^k$ converges to 2.

HeapSort

Sorting an array in ascending order using a max-heap is easy: once the heap has been built, its topmost value (i.e., the first value of the array) is the maximum. This maximum should be therefore moved to the end of the array. If we do that with an exchange, and new consider only the first $n-1$ values to be part of the tree, we are in the situation depicted on Figure 58: calling HEAPIFY on the root of this (restricted) tree is all we need to sift up its maximum value. This can be iterated to sort the entire array: each iteration places one new value at its correct place, and reorder the remaining heap.

HEAPSORT(A, n)	
1 BUILDHEAP(A, n)	$\Theta(n)$
2 for i from $n-1$ down to 1	$\Theta(n)$
3 $\quad A[0] \leftrightarrow A[i]$	$\Theta(n)$
4 $\quad$ HEAPIFY$(A, 0, i)$	$O(n \log n)$?

The complexity of the first three lines of HEAPSORT, should be quite obvious: we know the cost of BUILDHEAP from page 37, and line 3 is a constant-time operation repeated $n-1$ times. That leaves us with the cost of line 4.

The first call to HEAPIFY is done on an array of size $n-1$, so its cost should be $T_H(n-1) = O(\log(n-1)) = O(\log n)$ according to what we established on page 36. The following iterations will call HEAPIFY on smaller arrays, so we can still use $O(\log n)$ as an upper bound, and claim that the sum of all the these calls will cost $(n-1)O(\log n) = O(n \log n)$.

It would be legitimate to ask whether we could get a better complexity bound by being more precise when summing the costs of the different calls to HEAPIFY like we did for BUILDHEAP on page 37. Here the total work performed by all iterations of line 4 is

$$\sum_{i=1}^{n-1} T_H(i) = \sum_{i=1}^{n-1} O(\log i) = O\left(\sum_{i=1}^{n-1} \log i\right) = O\left(\log \prod_{i=1}^{n-1} i\right)$$
$$= O\big(\log((n-1)!)\big) \qquad (11)$$

Stirling's formula is a powerful tool to simplify expressions involving factorials, if you can remember it. We have

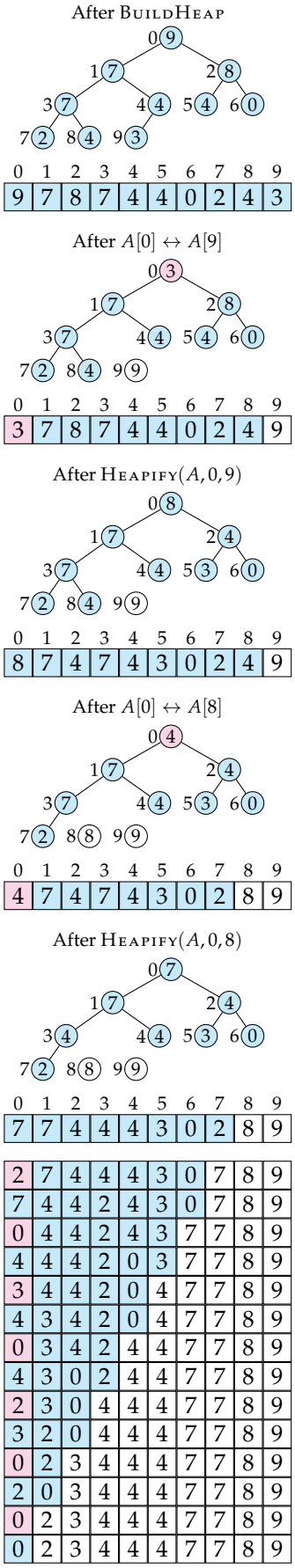
$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \qquad \text{hence} \qquad \log_2(n!) \sim n \log_2 n.$$

(For another way to obtain the equivalence on the right, see page 14.) We can therefore return to equation (11) and simplify it:

$$\sum_{i=1}^{n-1} T_H(i) = O\big((n-1)\log(n-1)\big) = O(n \log n)$$

Unfortunately, this result is not better than our original approximation. We conclude that HEAPSORT(A, n) runs in $O(n \log n)$.

Can you explain the fundamental difference between the loops of BUILDHEAP and HEAPSORT? Why is one $O(n)$ and the other $O(n \log n)$?



2	7	4	4	4	3	0	7	8	9
7	4	4	2	4	3	0	7	8	9
0	4	4	2	4	3	7	7	8	9
4	4	4	2	0	3	7	7	8	9
3	4	4	2	0	4	7	7	8	9
4	3	4	2	0	4	7	7	8	9
0	3	4	2	4	4	7	7	8	9
4	3	0	2	4	4	7	7	8	9
2	3	0	4	4	4	7	7	8	9
3	2	0	4	4	4	7	7	8	9
0	2	3	4	4	4	7	7	8	9
2	0	3	4	4	4	7	7	8	9
0	2	3	4	4	4	7	7	8	9
0	2	3	4	4	4	7	7	8	9

Figure 58: Progression of HEAPSORT, starting from the entire heap.

PARTITION

The PARTITION algorithm is a building block for QUICKSORT. PARTITION reorders a given range of elements in an array, such that all the elements in the left-hand side of the range are smaller than those in the right-hand side as pictured by Figure 59. The resulting range does not need to be sorted.[93]

One way to implement PARTITION is to choose a value, let's say $x \leftarrow A[\ell]$ and use it as a threshold to decide whether an element $A[v]$ can belong to the left-hand part (if $A[v] \leq x$) or to the right-hand part (if $A[v] \geq x$).[94] The following implementation of the reordering is often described as the "*collapse the walls*" technique. The walls are in fact two indices i and j starting at both ends of the range, and moving towards each other, exchanging values along the way.

```
PARTITION(A, ℓ, r)
1  x ← A[ℓ]                                          Θ(1)
2  i ← ℓ − 1; j ← r                                  Θ(1)
3  repeat forever                                    O(n)
       // find a value that can go to the right-hand side
4      do i ← i + 1 until A[i] ≥ x                   Θ(n)
       // find a value that can go to the left-hand side
5      do j ← j − 1 until A[j] ≤ x
       // swap the two values unless the walls collapsed
6      if j ≤ i                                      O(n)
7          return i + (i = ℓ)                        Θ(1)
8      A[i] ↔ A[j]                                   O(n)
```

The "repeat forever" loop might look daunting, but since lines 4 and 5 necessarily update i and j at each iteration of the main loop, it is guaranteed that eventually $j \leq i$ and the algorithm will terminate.

What is less obvious is that there are exactly two ways in which the algorithm may terminate. Either $i = j$ (in this case $A[i] = x$), or $i = j + 1$ as in Figure 60. It is not possible for i to be larger than $j + 1$, because all the values to the left of i are less than or equal to x, so the loop decrementing j will stop as soon as it passes i.

The algorithm assumes that the range contains at least two values ($r - l \geq 2$). To argue that the returned value p satisfies $\ell < p < r$, consider what it would take for this to be violated: To have $p = \ell$, line 4 should be executed only once, which means that line 5 will execute until $j = i = \ell$. However, in this case line 7 will return $i + 1$ so not ℓ.

Finally, the $\Theta(n)$ complexity of PARTITION should be obvious after we realize that because of the "collapsing walls" strategy, the sum of the executions of lines 4 and 5 is at least $n + 1$ (if we end with $i = j$) and at most $n + 2$ (if we end with $i = j + 1$).

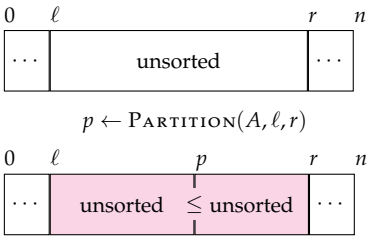


Figure 59: Overview of the PARTITION algorithm. The range $A[\ell..r-1]$ is reordered so that any value in $A[\ell..p-1]$ is less than or equal to any value in $A[p..r-1]$. The value p should be such that $\ell < p < r$, ensuring that each part is non-empty. Note that the two parts may have different lengths.

[93] Sorting $A[\ell..r-1]$ would be one way to implement PARTITION(A, ℓ, r), but it would be less efficient.

[94] Note that elements equal to x can go to either side; this is on purpose.

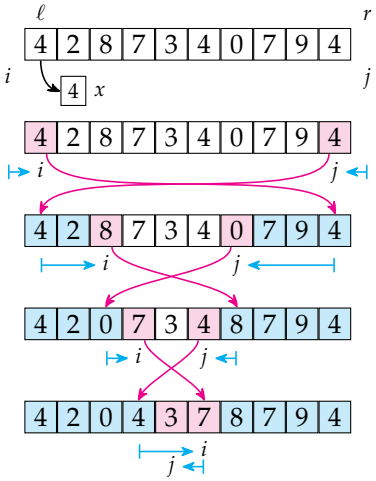


Figure 60: Execution of PARTITION on an example. In this case, the index returned is i, and the algorithm has (by chance!) reordered the range in two equal partitions.

QUICKSORT

QUICKSORT consists in recursively calling PARTITION on the two parts created by PARTITION, until we reach an array of length 1 (that does not need to be sorted).

```
QUICKSORT(A, ℓ, r)
1  if r − ℓ > 1
2      p ← PARTITION(A, ℓ, r)
3      QUICKSORT(A, ℓ, p)
4      QUICKSORT(A, p, r)
```

$$\frac{T_{QS}(1)}{\Theta(1)} \quad \frac{T_{QS}(n)}{\Theta(1)} \text{ for } n = r - \ell > 1$$
$$\Theta(n)$$
$$T_{QS}(L)? \text{ for } L = p - \ell$$
$$T_{QS}(n - L)?$$

Figure 61 shows the effects of the different calls to PARTITION occurring while sorting an example array with QUICKSORT.

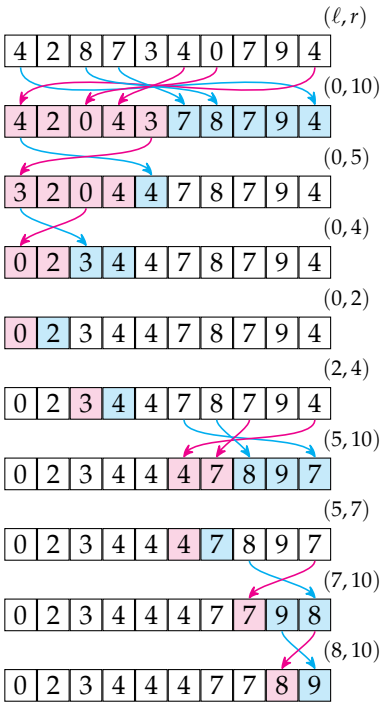


Figure 61: Effect of the successive calls to PARTITION(A, ℓ, r) during the recursion of QUICKSORT$(A, 0, 10)$. The pairs displayed on the side give the value of ℓ and r passed to PARTITION.

The proof that QUICKSORT actually sorts the array can be done by induction on the length of the considered range. The induction hypothesis H_n is "for any range $[\ell..r-1]$ of length $r - \ell = n$, calling QUICKSORT(A, ℓ, r) will sort all the elements in $A[\ell..r-1]$".

Clearly H_1 is true, because a range of length 1 is already sorted and QUICKSORT does not modify the array in this case. Consider some arbitrary $n > 1$, and assume that H_i is true for all $i < n$. Running QUICKSORT on a range of length $n > 1$ will execute lines 2–4:

- The result of line 2 is that all values in the range $A[\ell..p-1]$ are smaller than all values in the range $A[p..r-1]$.
- Furthermore, we have $\ell < p < r$, which implies that the two ranges $[\ell..p-1]$ and $[p..r-1]$ have lengths smaller than n and by hypothesis we can therefore state that after running lines 3 and 4, the values in $A[\ell..p-1]$ and $A[p..r-1]$ are sorted.

Combining these two points, it follows that $A[\ell..r-1]$ is sorted after executing lines 2–4.

Evaluating the complexity of QUICKSORT is less easy, because the recursive calls on lines 3 and 4 are not necessarily done on ranges of equal length. The PARTITION function could return any p that satisfies $\ell < p < r$. So if the size of the input range is $n = r - \ell$, then after calling PARTITION, the left part may have a length $L = p - \ell$ anywhere between 1 and $n - 1$, and the right part would have the remaining length $n - L$ (Fig. 62).

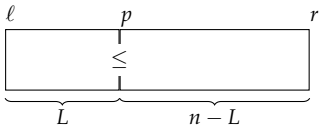


Figure 62: Let us assume that PARTITION always puts L elements in the left part, and $n - L$ elements in the right one.

It would therefore be tempting to express the complexity as the solution of

$$T_{QS}(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ \Theta(n) + T_{QS}(L) + T_{QS}(n - L) & \text{if } n > 1 \end{cases}$$

Unfortunately that is incorrect, because the above assumes that L would have the same value in every recursive call: i.e., PARTITION would *always* produce a left part of size L. Clearly that is not true. However, solving this equation can give us some clues about the possible behaviors of QUICKSORT.

Worst and Best Cases for QuickSort

Page 40 ended with a recursive expression of the complexity $T_{QS}(n)$ of sorting an array of size n, but that equation assumed that the Partition always created a left part of length L. Let us evaluate scenarios with different values of L.

- The case where L is always equal to 1 occurs when running QuickSort on a sorted array. In this case $T_{QS}(L) = \Theta(1)$, and the recursive equation can be simplified to

$$T_{QS}(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ \Theta(n) + T_{QS}(n-1) & \text{if } n > 1 \end{cases}$$

  Solving $T_{QS}(n) = \Theta(n) + T_{QS}(n-1)$ iteratively[95], we find that

$$\begin{aligned} T_{QS}(n) &= \Theta(n) + T_{QS}(n-1) \\ &= \Theta(n) + \Theta(n-1) + T_{QS}(n-2) \\ &= \Theta(n) + \Theta(n-1) + \Theta(n-2) + T_{QS}(n-3) \\ &= \Theta(n) + \Theta(n-1) + \Theta(n-2) + \cdots + \Theta(1) \\ &= \sum_{i=1}^{n} \Theta(i) = \Theta\left(\sum_{i=1}^{n} i\right) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2) \end{aligned}$$

  So QuickSort needs $\Theta(n^2)$ operations to sort a sorted array...[96] Note that the result is the same if L is replaced by *any* constant.

- Another interesting case would be when $L = \lfloor n/2 \rfloor$, i.e., when Partition always cuts the range in its middle. Then we have

$$T_{QS}(n) = \Theta(n) + T_{QS}(\lfloor n/2 \rfloor) + T_{QS}(\lceil n/2 \rceil).$$

  This is the same equation as for MergeSort (page 28), so we know the solution is $T_{QS}(n) = \Theta(n \log n)$.

- What if $L = n/10$? For this 10%-90% scenario the equation is

$$T_{QS}(n) = \Theta(n) + T_{QS}(\lfloor n/10 \rfloor) + T_{QS}(\lceil 9n/10 \rceil).$$

  Figure 63 shows the shape of the recursion tree: each node is labeled by the length of the array passed to Partition. The shortest branch of the tree is the left one, where the range is always divided by 10: the height of this branch is $\log_{10} n$. The longest branch is the right one, with height $\log_{10/9} n$ since the range is (slowly) divided by 10/9 at each recursive call. The work performed by Partition is proportional to the value displayed on each node of this tree, therefore the total cost of QuickSort is proportional to the sum of all the nodes of this tree. The sum of each line of the first $\log_{10} n$ lines if this tree is necessarily n, so these sum to $n \log_{10} n$. But the algorithm processes more than that. The total for each remaining line is less than n, so the sum of the whole tree is less than $n \log_{10/9} n$. We therefore have

  $\Theta(n \log_{10} n) \leq T_{QS}(n) \leq \Theta(n \log_{10/9} n)$ hence $T_{QS}(n) = \Theta(n \log n)$.

  The same result holds if $L = n/10000$ or any other ratio.[97]

[95] Be careful when doing this type of iterative computations. It would be tempting to simplify $\Theta(n) + \Theta(n-1)$ as $\Theta(n)$ (this is true), then simplify $\Theta(n) + \Theta(n-2)$ as $\Theta(n)$ (this is true as well), and continue until we obtain that $T_{QS}(n) = \Theta(n)$ (which is incorrect). What did we do wrong? The number of terms we summed is not constant: we can only perform these reductions a constant number of times.

If you are unsure, it is better to replace $\Theta(n)$ by some representative function of the class, like cn, and solve $F(n) = cn + F(n-1)$ instead. Then you have $T_{QS}(n) = \Theta(F(n))$

[96] This is bad, and it also implies that this implementation of QuickSort behaves badly for "nearly sorted" arrays. We discuss some mitigating techniques on page 43.

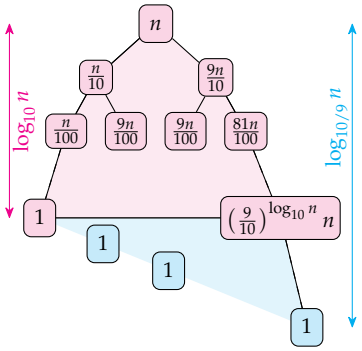


Figure 63: Shape of the tree of the recursive calls to QuickSort in a scenario where the Partition always makes a 10%-90% split.

[97] The difference between the bad cases and the good cases discussed on this page is whether L is constant or whether it is proportional to n. The actual constant or ratio does not affect the resulting complexity class.

Average Complexity of QuickSort

Let us start again from the equation[98]

$$T_{QS}(n) = \Theta(n) + T_{QS}(L) + T_{QS}(n-L).$$

[98] The fact that $T_{QS}(1) = \Theta(1)$ is implicit here, but it implies that later down the page we also have $F(1) = c$ and $Y(1) = c$.

On page 41, we considered some arbitrary (but fixed) expressions for L to establish the complexity of QuickSort on some particular scenarios. However, in practice, L may take a different value in each recursive call. All we know is that $0 < L < n$ because Partition guarantees that the left and right sides may not be empty.

To derive an *average* complexity, assume that L is a random variable taking its value uniformly in $\{1, 2, ..., n-1\}$. We can therefore compute $\hat{T}_{QS}$, the average complexity of QuickSort by averaging the complexity we could obtain for each of these $n-1$ different values, recursively:

$$\hat{T}_{QS}(n) = \frac{1}{n-1}\sum_{L=1}^{n-1}\left(\Theta(n) + \hat{T}_{QS}(L) + \hat{T}_{QS}(n-L)\right)$$

$$\hat{T}_{QS}(n) = \Theta(n) + \frac{1}{n-1}\left(\sum_{L=1}^{n-1}\hat{T}_{QS}(L) + \sum_{L=1}^{n-1}\hat{T}_{QS}(n-L)\right)$$

$$\hat{T}_{QS}(n) = \Theta(n) + \frac{2}{n-1}\sum_{L=1}^{n-1}\hat{T}_{QS}(L)$$

To avoid any errors[99], let's replace $\Theta(n)$ by some representative function cn. The new function $F(n)$ is such that $\hat{T}_{QS}(n) = \Theta(F(n))$:

[99] cf. remark 95 on p. 41

$$F(n) = cn + \frac{2}{n-1}\sum_{L=1}^{n-1}F(L)$$

To get rid of the sum, we first multiply both sides by $n-1$ to get rid of the non-constant factor in front of the sum, and then subtract the same expression for $F(n-1)$:

$$(n-1)F(n) = (n-1)cn + 2\sum_{L=1}^{n-1}F(L)$$

$$(n-2)F(n-1) = (n-2)c(n-1) + 2\sum_{L=1}^{n-2}F(L)$$

$$(n-1)F(n) - (n-2)F(n-1) = 2c(n-1) + 2F(n-1)$$

$$(n-1)F(n) = 2c(n-1) + nF(n-1)$$

Let's divide both sides by $n(n-1)$ and then set $Y(n) = F(n)/n$:

$$\frac{F(n)}{n} = \frac{2c}{n} + \frac{F(n-1)}{n-1}$$

$$Y(n) = \frac{2c}{n} + Y(n-1) = 2c\sum_{i=1}^{n}\frac{1}{i}$$

From this harmonic series[100], we conclude that $Y(n) = \Theta(\log n)$, hence $F(n) = \Theta(n \log n)$. The average complexity of QuickSort is therefore $\hat{T}_{QS} = \Theta(n \log n)$.

[100] The fact that $\sum_{i=1}^{n}\frac{1}{i} = \Theta(\log n)$ can be derived from Euler's formula ($\sum_{i=1}^{n}\frac{1}{i} = \ln n + \gamma + o(1)$), or easily proven by bounding the sum with integrals as done on page 14:

$$\int_2^{n+1}\frac{1}{i}\mathrm{d}i \leq \sum_{i=2}^{n}\frac{1}{i} \leq \int_1^{n}\frac{1}{i}\mathrm{d}i$$

$$\ln(n+1) - \ln(2) \leq \sum_{i=2}^{n}\frac{1}{i} \leq \ln(n)$$

$$\Theta(\log n) \leq \sum_{i=2}^{n}\frac{1}{i} \leq \Theta(\log n)$$

$$\Theta(\log n) \leq \sum_{i=1}^{n}\frac{1}{i} \leq \Theta(\log n)$$

QuickSort Optimizations

Typical QuickSort optimizations include:

- Selecting a different pivot value in the Partition procedure from page 39. The ideal value would be the median of the range as it would ensure equal size for both sides. However, the median is not really easy to compute without sorting the range already.[101] The usual strategy is to pick the median of the three values $A[\ell]$, $A[r-1]$ and $A[\lfloor (r+\ell)/2 \rfloor]$. Line 1 of Partition is replaced by $x \leftarrow \text{MedianOf3}(A[\ell], A[r-1], A[\lfloor (r+\ell)/2 \rfloor])$. With this change, QuickSort deals nicely with nearly-sorted arrays.[102]

- The last recursive call to QuickSort is a *tail call*, so it can be optimized as a loop. Compare these equivalent implementations:

```
QuickSort(A, ℓ, r)
1  if r − ℓ > 1
2      p ← Partition(A, ℓ, r)
3      QuickSort(A, ℓ, p)
4      QuickSort(A, p, r)
```

```
QuickSort(A, ℓ, r)
1  while r − ℓ > 1
2      p ← Partition(A, ℓ, r)
3      QuickSort(A, ℓ, p)
4      ℓ ← p
```

  Any decent compiler would already do this kind of *tail call elimination* automatically: this saves memory, because the value of the local variables have to be saved on the stack before each call.

  However, what the compiler cannot guess is that the order of the two recursive calls to QuickSort does not matter: we can actually *choose* which of the two calls should be turned into a loop. Here, we want to always recurse on the smaller part, to keep the recursion as shallow as possible.

```
QuickSort(A, ℓ, r)
1  while r − ℓ > 1
2      p ← Partition(A, ℓ, r)
3      if p − ℓ ≤ r − p
4          QuickSort(A, ℓ, p)
5          ℓ ← p
6      else
7          QuickSort(A, p, r)
8          r ← p
```

  While this does not change the time complexity of the algorithm, it changes its memory complexity[103]. Indeed the memory complexity was $O(n)$ in our first implementation of QuickSort because the recursion could be n-deep in the worst case; it is now $O(\log n)$ because there is no way to recurse on an sub-array larger than $n/2$.

- Use InsertionSort when the array has a small length (like 10 values — the precise bound has to be found empirically). Even if InsertionSort has $O(n^2)$ complexity, it usually performs a lot better than QuickSort for small-sized input, because it does not have all the overhead of running Partition and making recursive calls.

[101] It is possible to find the median of an array with only $\Theta(n)$ operations using an algorithm sometimes called "median of medians". However this would be very inconvenient here: firstly the constant hidden behind the $\Theta(n)$ notation is quite large, and secondly this algorithm is itself based on a recursive procedure similar to QuickSort.

[102] Input arrays that trigger the worst-case $\Theta(n^2)$ complexity still exist (see Fig. 64 page 44), but are harder to come by.

[103] I.e., the number of additional memory an algorithm requires to process its input — this includes the stack in case of recursive algorithms.

IntroSort, or How to Avoid QuickSort's Worst Case

Even with the usual optimizations described on page 43, it is possible to construct an input that triggers QuickSort's worst case. Figure 64 shows an example where the pivot selected by MedianOf3 is the second smallest value, so that Partition does a single swap and creates a left part with $L = 2$ elements, and a right part with $n - 2$ elements. The left part can be sorted recursively in constant time, but when sorting the right part recursively, a similar $(2, n-2)$ partition is produced again. This unfortunate situation occurs until half of the array is sorted: if we sum just the costs of running Partition on the largest parts up to this point we have $\Theta(n) + \Theta(n-2) + \Theta(n-4) + \cdots + \Theta(n/2) = \Theta(n^2)$, so the worst case of QuickSort is necessarily reached. Any input where the recursion depth[104] of QuickSort is linear will lead to the worst case complexity of $\Theta(n^2)$. On the contrary, if the recursion depth is in $O(\log n)$, then we obtain a best case complexity of $\Theta(n \log n)$.

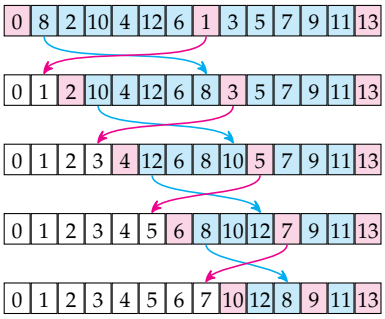


Figure 64: A worst case for QuickSort when the pivot is the median of $A[\ell]$, $A[r-1]$ and $A[\lfloor (r+\ell)/2 \rfloor]$.

An easy way to avoid the worst case of QuickSort is to not use QuickSort, and resort instead to HeapSort or MergeSort, that ensure a $\Theta(n \log n)$ worst case. However practice shows that QuickSort's is much faster on the average, so it is usually preferred.

The idea of IntroSort[105] is to execute a variant of QuickSort that will keep track of the depth of the recursive calls in order to ensure it is sub-linear. If that depths exceeds $c\lfloor \log n \rfloor$ for some constant c, then IntroSort stops the recursion and sorts the current sub-array with HeapSort instead. Doing so ensure that the entire array will be sorted in $\Theta(n \log n)$, either by the variant of QuickSort, or with the help of HeapSort in the difficult cases. The constant c can be chosen arbitrarily, but it should not be too small to give QuickSort a chance to finish and limit the number of calls to HeapSort. In practice $c = 2$ is often used.

The pseudo-code below includes the tail call optimization from page 43. It could be combined with InsertionSort as well.[106]

```
IntroSort(A, ℓ, r)
 1  IntroSortRec(A, ℓ, r, 2⌊log(r − ℓ)⌋)

IntroSortRec(A, ℓ, r, depth_limit)
 1  while r − ℓ > 1
 2      if depth_limit = 0
 3          HeapSort(A, ℓ, r)
 4          return
 5      else
 6          depth_limit ← depth_limit − 1
 7          p ← Partition(A, ℓ, r)
 8          if p − ℓ ≤ r − p
 9              IntroSortRec(A, ℓ, p, depth_limit)
10              ℓ ← p
11          else
12              IntroSortRec(A, p, r, depth_limit)
13              r ← p
```

[104] This should be understood without the tail call optimization discussed on page 43

[105] IntroSort was invented by David Musser, one of the original authors (with Alexander Stepanov) of the C++ Standard Template Library (STL). It is now the default sorting algorithm behind most (if not all) `std::sort()` implementations, because the STL requires `std::sort()` to run in $\Theta(n \log n)$.

[106] Here we assume that HeapSort(A, ℓ, r) executes HeapSort on the sub-array $A[\ell..r-1]$. This does not exactly matches to prototype of page 38.

QuickSelect

Given an array A of n values, the value of rank k is the k^{th} smallest value of A. An algorithm that takes $\mathcal{A}$ and k as input and returns the rank-k value is called a *selection* algorithm.[107] In what follows, we assume $0 \leq k < n$. The value of rank $k = 0$ is the minimal value, and the value of rank $n - 1$ is the maximal value.

QuickSelect can be seen as a modification of QuickSort to implement a selection without sorting the entire array. The trick is that after Partition has run, we know (from the sizes of the two parts) in which part we need to search our value recursively.

```
QuickSelect(A, n, k)                                          T(1)   T(n)
1  return QuickSelectRec(A, 0, n, k)
QuickSelectRec(A, ℓ, r, k)                                    Θ(1)   Θ(1)
1  if r − ℓ = 1 then return A[ℓ]
2  p ← Partition(A, ℓ, r)                                            Θ(n)
3  L ← p − ℓ                                                         Θ(1)
4  if k < L                                                          Θ(1)
5     then return QuickSelectRec(A, ℓ, p, k)                         T(L)
6     else return QuickSelectRec(A, p, r, k − L)                     T(n − L)
```

Figure 65 shows an example.

The complexity differs from QuickSort, because at most one of the two possible recursive calls on lines 5–6 may be executed. In the worst case, we have to assume that Partition performed poorly ($L = 1$), and that we always have to recurse into the largest part of $n - 1$ elements. This gives us the following recursive equation:

$$T(1) = \Theta(1),$$
$$T(n) = T(n-1) + \Theta(n) \text{ for } n > 1$$

We conclude that the worst case complexity is $T(n) = \Theta(n^2)$.

Let us now consider a case where Partition behaves ideally, always splitting the array of n elements in two equal halves. The equation becomes:

$$T(1) = \Theta(1),$$
$$T(n) = T(n/2) + \Theta(n) \text{ for } n > 1$$

The Master Theorem (p. 31) tells us the solution to this equation is $T(n) = \Theta(n)$. While this is not the best case scenario[108], this seems like a favorable case. If we imagine a similar scenario where we recurse into 90% of the array (or any other ratio), the complexity will remain linear.

On page 46 we will prove that the average complexity of QuickSelect is actually $\Theta(n)$, despite the quadratic worst case.[109]

Do you believe it is possible to write a selection algorithm that would always run in $\Theta(n)$? Jump to page 47 for an answer.

[107] One easy (and inefficient) way to build a *selection* algorithm is to first sort A in increasing order, and then pick its k^{th} value. What sorting algorithm would you use, and what would be the complexity of that *selection* algorithm?

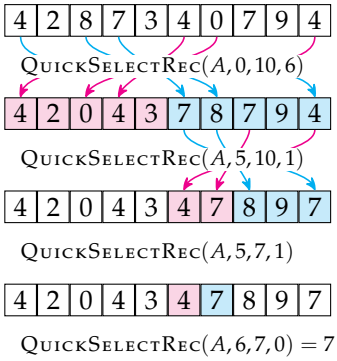


Figure 65: An example execution of QuickSelect$(A, 10, 6)$. The figure shows the state of the array before the next recursive call to QuickSelectRec.

[108] **Exercise:** Devise a scenario where calling QuickSelect on an array of size n runs in $\Theta(1)$.

[109] **Exercise:** Using the idea presented page 44, write another selection algorithm, called IntroSelect, with a worst case of $\Theta(n \log n)$.

Average complexity of QUICKSELECT

To study the average complexity $\hat{T}(n)$ for QUICKSELECT, we consider (as we did on page 42) that the value of L can be anything between 1 and $n-1$ with equal probability, and we average over all these possibilities. The problem in our case, is that we do not know if the algorithm will recurse in a part of size L or $n-L$, so let us compute an *upper bound* of $\hat{T}$ by assuming we recurse into the largest of these two parts.

$$\hat{T}(1) = \Theta(1),$$
$$\hat{T}(n) \leq \frac{1}{n-1}\sum_{i=1}^{n-1}\hat{T}(\max(i, n-i)) + \Theta(n) \text{ if } n > 1$$

We can simplify this a bit because $\max(i, n-i) = \begin{cases} i & \text{if } i \geq \lceil n/2 \rceil \\ n-i & \text{if } i \leq \lfloor n/2 \rfloor \end{cases}$

If n is even, all the terms between $\hat{T}(\lceil n/2 \rceil)$ and $\hat{T}(n)$ appear twice in the sum. If n is even, $\hat{T}(\lfloor n/2 \rfloor)$ additionally occurs once, but it is OK to count it twice since we are establishing an upper bound.

$$\hat{T}(n) \leq \frac{2}{n-1}\sum_{i=\lfloor n/2 \rfloor}^{n-1}\hat{T}(i) + \Theta(n)$$

It is not clear how to simplify this, however from the last scenario of page 45, it seems *likely* that $\hat{T}(n) = \Theta(n)$. Since we are working on an upper bound, let us prove that $\hat{T}(n) = \mathrm{O}(n)$ by induction[110].

Our hypothesis $H(n)$ is that $\hat{T}(n) \leq cn$ from some c we can pick arbitrarily large. Clearly $H(1)$ is true, because $\hat{T}(n) = \Theta(1)$, so we can find some c larger than this $\Theta(1)$ constant. Now let us assume that $H(i)$ holds for $1 \leq i < n$ and let us establish $H(n)$:[111]

$$T(n) \leq \frac{2}{n-1}\sum_{i=\lfloor n/2 \rfloor}^{n-1} ci + \Theta(n) \leq \frac{2c}{n-1}\frac{3n^2}{8} + \Theta(n)$$
$$T(n) \leq \frac{3cn(n-1)+3cn}{4(n-1)} + \Theta(n) \leq \frac{3cn}{4} + \frac{3c(n-1)+3c}{4(n-1)} + \Theta(n)$$
$$T(n) \leq cn - \frac{cn}{4} + \frac{3c}{4} + \frac{3c}{4(n-1)} + \Theta(n) \leq cn + \left(\Theta(n) - \frac{cn}{4}\right)$$

We can take c large enough so that $cn/4$ dominates $\Theta(n)$

$$T(n) \leq cn$$

Conclusion: $H(n)$ holds for all $n \geq 1$ and therefore $\hat{T}(n) = \mathrm{O}(n)$.

However since QUICKSELECT has to call PARTITION for $n > 1$, it necessarily performs at least $\Theta(n)$ operations, so we can claim that its average complexity is in fact $\hat{T}(n) = \Theta(n)$.

[110] cf. pp. 32–33

[111] In the first line, the sum over ci can be removed by showing that

$$\sum_{i=\lfloor n/2 \rfloor}^{n-1} i \leq \frac{3}{8}n^2.$$

$$\sum_{i=\lfloor n/2 \rfloor}^{n-1} i = \frac{(n-1+\lfloor n/2 \rfloor)(n-\lfloor n/2 \rfloor)}{2}$$
$$= \frac{n^2 - n + \lfloor n/2 \rfloor - \lfloor n/2 \rfloor^2}{2}$$

if n is even:

$$= \frac{4n^2 - 4n + 2n - n^2}{8}$$
$$= \frac{3n^2 - 2n}{8} \leq \frac{3n^2}{8}$$

if n is odd:

$$= \frac{4n^2 - 4n + 2(n-1) - (n-1)^2}{8}$$
$$= \frac{3n^2 - 3}{8} \leq \frac{3n^2}{8}$$

Linear Selection

While QUICKSELECT has an average complexity of $\Theta(n)$, it still has a $\Theta(n^2)$ worst case when we arrange the data so that PARTITION behaves badly (for instance using an arrangement similar to Fig. 64 on page 44). However if we could make sure that PARTITION would always allow us to eliminate a number of values that is a fraction of n (as opposed to a constant), then the selection would be linear.

This is actually the key of the LINEARSELECT algorithm[112]: at any step of the recursion, it removes a quarter of the values. LINEARSELECT can be seen as a variant of QUICKSELECT where PARTITION is changed to select its pivot as the median of medians (steps 1–3).

LINEARSELECT(A, n, k)	$T(n)$
1 consider the n input values as $\lceil \frac{n}{5} \rceil$ groups of 5 values (with maybe the last group having less than 5 values) and compute the median of each group	$\Theta(n)$
2 compute the median x of all those $\lceil \frac{n}{5} \rceil$ medians	$T(\lceil \frac{n}{5} \rceil)$
3 use x as the pivot of PARTITION (i.e., replacing line 1 of the algorithm on page 39) to reorganize A	$\Theta(n)$
4 recursively call LINEARSELECT on one of the two parts (depending on k), as done in QUICKSELECT.	$\leq T(\frac{3n}{4})$

Computing the median of 5 values can be done in constant time, so repeating this operation $\lceil \frac{n}{5} \rceil$ times in step 1 requires $\Theta(n)$ operations. Computing the median of these $\lceil \frac{n}{5} \rceil$ medians however cannot be done in constant time since the number of values depends on n; however it correspond to the value of rank $\lfloor \lceil \frac{n}{5} \rceil / 2 \rfloor$ among these $\lceil \frac{n}{5} \rceil$ values, so we can compute it by calling LINEARSELECT recursively on an array of size $\lceil \frac{n}{5} \rceil$. Hence, assuming that running LINEARSELECT on n values takes $T(n)$ operations, then step 2 needs $T(\lceil \frac{n}{5} \rceil)$ operations. Calling PARTITION costs $\Theta(n)$, as seen on page 39, but thanks to our pivot, we are sure that at each of the two parts contains at least 25% of the array (see Fig. 66), so in the worst case, the recursive call of step 4 will cost $T(\frac{3n}{4})$. Thus, we have:

$$T(1) = \Theta(1)$$
$$T(n) \leq \Theta(n) + T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{3n}{4}\right)$$

Let us prove the following induction hypothesis[113], H_n: "$T(n) \leq cn$". Clearly H_1 holds, because $T(1) = \Theta(1)$ and we can find c large enough to dominate that $\Theta(1)$. Let us assume that H_1, H_2, ..., H_{n-1} hold, and inject this knowledge into our recursive inequality:

$$T(n) \leq \Theta(n) + c\left\lceil \frac{n}{5} \right\rceil + \frac{3cn}{4} \leq \Theta(n) + \frac{c(n+4)}{5} + \frac{3cn}{4}$$
$$T(n) \leq \Theta(n) + \frac{19cn}{20} + \frac{4c}{3} = cn + \left(\Theta(n) + \frac{4c}{3} - \frac{cn}{20}\right)$$

We can chose c large enough so that $cn/20$ dominates $\Theta(n) + \frac{4c}{3}$. Then $T(n) \leq cn$, which means that H_n holds for any $n \geq 1$, and this allows us to conclude that $T(n) = O(n)$.

We can strengthen this result to $T(n) = \Theta(n)$ because of step 3.

[112] This algorithm is also known as "*median of medians*" for reasons that will be soon obvious.

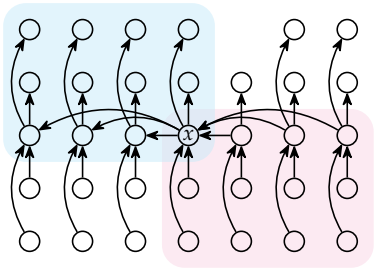


Figure 66: The circles represent an array of 34 values organized in 7 groups (6 groups of 5 values, and one group of 4 values). We interpret $(u) \rightarrow (v)$ as meaning $u \leq v$. Inside each group (i.e., each column), values have been moved so that the median is on the center line, values above it are greater, and values below it are lesser. Similarly, columns have been ordered such that the median x of all medians has three greater medians to its left, and three lesser medians to its right.

We can see that there are more than $n/4$ values that are greater than x (the top left quarter), and more than $n/4$ values that are lesser than x. Therefore, if x is used as a pivot in PARTITION, both parts are guaranteed to have more than $n/4$ values.

[113] See pp. 32–33 for the technique. One way to guess a probable solution for this equation is to consider that $T(\lceil \frac{n}{5} \rceil)$ seems much smaller than $T(\frac{3n}{4})$. So as a first guess, we neglect it and solve $T(n) \leq \Theta(n) + T(\frac{3n}{4})$. In this simplified case, we find using the master theorem that $T(n) = O(n)$. Then we use induction to verify if this solution is still correct for the complete problem.

Space Complexity

As mentioned on page 18 the space complexity of an algorithm, often noted $S(n)$, is the amount of additional memory required to process an input of size n.

The space complexity can be computed using the same mathematical tools we used for computing time complexities. We simply sum the sizes of all local variables, and if there are function calls (even recursive calls), we add the maximum space complexities of all these calls.

For a simple iterative algorithm like SelectionSort (p. 19) or InsertionSort (p. 20), there is only a couple of local variables used to store values or indices, so $S(n) = \Theta(1)$.[114]

A recursive algorithm like BinarySearch (p. 22) has to store one local variable m per recursive call. Its space complexity is the solution of $S(n) \leq \Theta(1) + S(n/2)$ which is $S(n) = O(\log n)$. However if we account for the fact that BinarySearch is *tail recursive* and assume the compiler will perform *tail calls elimination*, changing the recursion into a simple loop, then the space complexity drops to $\Theta(1)$.

Similarly, the space complexity of HeapSort (p. 38) satisfies $S(n) \leq S(2n/3 + O(1)) + \Theta(1)$, and this can be solved as for $T_H(n)$ on page 36: $S(n) = O(\log n)$.

For an algorithm like MergeSort (p. 28), we can assume that the array B of size $\Theta(n)$ used in Merge (p. 27) is allocated once before the recursion, and the space requirement of the recursion is $\Theta(\log n)$. We conclude that $S(n) = \Theta(n)$.

Finally, the space complexity of QuickSort depends on how it has been implemented. If we use the implementation described on pages 39–40, each call to QuickSort requires $\Theta(1)$ local variables (call to Partition included) but we can have $n - 1$ recursive calls in the worst case. Therefore $S(n) = O(n)$. However if the trick of page 43 is used to only recurse on the smaller of the to sides of the partition, then the recursive depth is at most $\log_2 n$ and the space complexity drops to $S(n) = O(\log n)$.

In-place Algorithms

An algorithm is *in-place* if its space complexity belong to $O(\log n)$. In other words, the additional memory it requires to process an input of size n is at most logarithmic in n.

This definition is a bit more practical than what we would intuitively allow: limiting the memory consumption to $O(\log n)$ instead of $O(1)$ allows some recursive algorithms like HeapSort to qualify as in-place while still disallowing algorithms that would use recursion to allocate enough local variables to duplicate the input on the stack.

[114] Here, and in all our examples, we are making the practical assumption that values and indices are integers stored in a fixed amount of memory. Some people who are interested in bit-level complexity may have different expectation. For instance they could say that if we want to work with array that are arbitrarily large, we need $\Theta(\log n)$ bits to store an index. Similarly, if we want be able to store n distinct values in the input array, it needs $\Theta(n \log n)$ bits.

More Sort-Related Topics

Stable Sorting Techniques

While we have illustrated all sorting algorithms over arrays of integers, in real-life it is often the case that we want to sort records of values according to some key. The only change required to the algorithms is that instead of comparing array elements ($A[i] < A[j]$) we compare some fields of these elements (e.g., $A[i].grade < A[j].grade$). Let call this field the *sort key*.

input		output 1		output 2		output 3		output 4	
name	grade	name	grade	name	grade	name	grade	name	grade
Robert	10	John	5	Michael	5	John	5	Michael	5
William	15	Michael	5	John	5	Michael	5	John	5
James	10	Robert	10	Robert	10	James	10	James	10
John	5	James	10	James	10	Robert	10	Robert	10
Michael	5	William	15	William	15	William	15	William	15

Figure 67: A table listing students and their grades. Sorting this table in ascending-grade order may return any of the four displayed outputs. For instance SELECTIONSORT produces output 3, INSERTIONSORT and MERGESORT both produce output 1, HEAPSORT produces output 4, and the output of QUICKSORT depends on how the pivot is selected in PARTITION.

A sorting algorithm is *stable* if it preserves the order of elements with equal keys. An *unstable* sorting algorithm may turn the input of Fig 67 into any of the four possible output. A *stable* sort will necessarily produce output 1: John and Michael have equal grade so their relative order has to be preserved; likewise for Robert and James.

The algorithms INSERTIONSORT and MERGESORT, as presented on pages 20 and 27–28 are stable.[115] The following sorts are unstable: SELECTIONSORT, HEAPSORT, QUICKSORT.

Stable algorithms are mostly useful when records are sorted multiple times, one key at a time. For instance to obtain output 3 of Fig 67, where people with equal grades are sorted alphabetically, we could proceed in two steps as illustrated by Fig. 68.

name	grade		name	grade		name	grade
Robert	10		James	10		John	5
William	15	(1) →	John	5	(2) →	Michael	5
James	10		Michael	5		James	10
John	5		Robert	10		Robert	10
Michael	5		William	15		William	15

[115] However some very subtle changes to these algorithms will lose the *stable* property. In INSERTIONSORT, change $A[i] > key$ into $A[i] \geq key$ and the algorithm will still sort the array, but in an unstable way. Changing $A[\ell] \leq A[r]$ into $A[\ell] < A[r]$ in MERGE will have a similar effect on MERGESORT.

Figure 68: (1) Sort by names, then (2) use a stable sort to order by grades. In this case, we would get the same output using a single sorting procedure by modifying the comparison function to order by grades and then by names in case of equal grades, however there are situations (e.g., using a cheap spreadsheet) where we have less control on the comparison function.

Finally, we can turn any unstable sort into a stable one by appending the input order to the key (Fig. 69), or equivalently, by adding a new column with the input order and using it for tie breaking.

name	grade		name	grade′		name	grade′		name	grade
Robert	10		Robert	1001		John	504		John	5
William	15	(1) →	William	1502	(2) →	Michael	505	(3) →	Michael	5
James	10		James	1003		Robert	1001		Robert	10
John	5		John	504		James	1003		James	10
Michael	5		Michael	505		William	1502		William	15

Figure 69: Making any sort stable: (1) modify the key to include the input order, (2) sort against the new key, (3) revert to old keys.

Further Reading

We recommend the following books (ordered by relevance):

- *Introduction to Algorithms* (*Third Edition*) by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press, 2009.

  This book covers most of the topics touched in our lecture. Focus on chapters: 1–4, 6–13, and 15. This books also has chapters on graphs that intersect with the lecture on graph theory you will get next semester.

- *Concrete Mathematics: A Foundation for Computer Science* (*Second Edition*) by Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. Addison-Wesley, 1994.

  An introduction to mathematical tools useful to the computer scientist, and presented very nicely.

- *Advanced Data Structures* by Peter Brass. Cambridge University Press, 2008.

  This book presents a wide range of data structures. It is well illustrated, and it gives actual C code for implementing each data structure.

- *Analysis of Algorithms* (*Second Edition*) by Robert Sedgewick and Philippe Flajolet. Addison-Weysley, 2013.

  This book focuses on the mathematical tools needed for studying the complexity of algorithm, but it goes very fast into powerful techniques (such as generating functions) that are beyond the scope of the current lecture. The first two chapters contains material discussed in this lecture. In particular, our illustration of the master theorem (page 31) is inspired from this book.