Notes 1: Finite precision arithmetic, algorithms and computational complexity

1.1 Numerical arithmetic and precision

This module is about using digital computers to do calculations and process data. As a prelude it is worth learning a little bit about how digital computers do arithmetic because all is not always as it seems. Arithmetic is here taken to mean the operations of addition/subtraction and multiplication/division.

Let us begin by asking a simple question: what is a number? In the familiar (decimal) representation of integers, the string of digits $d_2d_1d_0$ is really shorthand for

$$d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0,$$

where the digits, $d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Likewise a real number, having decimal representation $d_2d_1d_0.d_{-1}d_{-2}d_{-3}$, is shorthand for

$$d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0 + d_{-1} \times 10^{-1} + d_{-2} \times 10^{-2} + d_{-3} \times 10^{-3}.$$

There is no fundamental reason why we should work in base 10. In a base-b representation, the number $(d_2d_1d_0.d_{-1}d_{-2}d_{-3})_b$ would be shorthand for

$$d_2 \times b^2 + d_1 \times b^1 + d_0 \times b^0 + d_{-1} \times b^{-1} + d_{-2} \times b^{-2} + d_{-3} \times b^{-3},$$

where the digits $d_i \in \{0, 1, 2, \ldots b-1\}$.

Digital computers represent numbers using base 2. A single binary digit is called a "bit". Physically, a bit is an electrical circuit which can be off or on thereby representing the two binary digits, 0 and 1. A string of 8 bits is called a "byte". Binary arithmetic, although convenient for computers, is very unwieldy for humans. Hence the nerdy joke: there are 10 types of people in the world - those who understand binary arithmetic and those who don't.

Numbers are represented in a computer by a string of bits of a fixed length. Most commonly used are 32-bit and 64-bit representations. As a result of this fixed length, the set of possible numbers is finite. At this level, we see that computer arithmetic cannot be equivalent to conventional arithmetic.

1.1.1 Representation of integers

Setting aside this finiteness, integers can be represented exactly provided they are not too large. For example an unsigned n-bit integer is represented as $x = (d_{n-1} \ldots d_1 d_0)_2$ with $d_i \in \{0, 1\}$. The largest and smallest integers representable in this format are

$$x_{max} = \sum_{i=0}^{n-1} 2^i = 2^n - 1 \qquad x_{min} = 0.$$

To represent signed integers, we can use one of the bits to encode the sign.

1.1.2 Representation of real numbers - fixed and floating point

Real numbers generally require an infinite string of digits and so cannot be represented exactly using a finite string of bits. Computers therefore *simulate* real arithmetic. Note that numbers which have a terminating expansion for one choice of base may have a non-terminating expansion in another. For example $(0.1)_{10} = (0.0001100110011001100110011 \ldots)_2$. One approach is to store a fixed number of integer and fractional digits. This is called a fixed-point representation. For example, we can use $2n$ bits to represent a real number with the following signed fixed-point representation:

$$x = \pm(d_{n-1} \ldots d_1 d_0 . d_{-1} d_{-2} \ldots d_{-n+1})_2,$$

with $d_i \in \{0, 1\}$. We necessarily introduce an error in representing numbers whose binary expansions contain more fractional digits than $n-1$. This error is called *round-off error*. It is a feature of hardware and cannot be avoided. Hardware can implement different protocols for rounding:

- Round to zero
- Round half away from zero
- Round half to even

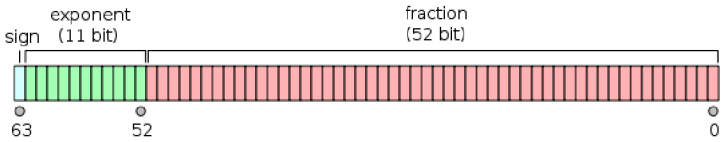
The advantage of fixed point representation is that the representable numbers are uniformly spaced. The big disadvantage is that they cannot represent very big or very small numbers as commonly arise in science and engineering. For this latter reason, fixed point representations are rarely if ever used in practice. A better approach is the so-called floating-point representation in which a number consists of two parts:

- a significand: contains the number's digits. Negative numbers have negative significands.
- an exponent: says where the decimal (or binary) point is placed relative to the beginning of the significand. Negative exponents represent numbers that are very small (i.e. close to zero).

IEEE 754 provides the standard for floating point representation of real numbers. The standard defines a 32-bit format known as a single precision and a 64 bit format known as double precision. The bits are used as follows:

Format	Total bits	Significand bits	Exponent bits
single precision	32	23 + sign	8
single precision	64	52 + sign	11

In memory, a double precision number looks like the following:



This format has the attractive properties:

- it can represent numbers having very different magnitudes. This is limited by the length of the exponent.
- the *relative* accuracy is the same at all magnitudes. This is limited by the length of the significand. The *absolute* accuracy depends on the scale. The representable numbers are not uniformly spaced.
- multiplying a very large and a very small number preserves the accuracy of both in the result.

For more detail than you every wanted to know about floating point arithmetic see [1] and the references therein.

1.1.3 Rounding errors

Rounding errors mean that floating point arithmetic can have properties which seem bizarre if one forgets the fact that its purpose is to simulate conventional arithmetic, not to reproduce it. Consider for example the following piece of C code:

```
double a,b,c;
a=0.1;
b=0.2;
c=0.1+0.2;
fprintf(stdout, "a=%.16e\n",a);
fprintf(stdout, "b=%.16e\n",b);
fprintf(stdout, "c=%.16e\n",c);
```

which gives the output

```
research@cleo:~$ ./a.out
a=1.0000000000000001e-01
b=2.0000000000000001e-01
c=3.0000000000000004e-01
```

Round-off error can mean that statements which seem mathematically correct can be false. Here is another example:

```
double a,b;
a=0.15 + 0.15;
b=0.1+0.2;
if(a==b)
```

```
{fprintf(stdout, "True\n");}
else
{fprintf(stdout, "False\n");}
```

which gives the output

```
research@smudge:~$ ./a.out
False
```

The rounding errors in these examples are small. The machine accuracy, ϵ_m, is the smallest floating point number which when added to the floating point number 1.0 produces a result which is different from 1.0. IEEE 754 single precision format has $\epsilon_m \approx 1.19 \times 10^{-7}$ whereas IEEE 754 double precision format has $\epsilon_m \approx 2.22 \times 10^{-16}$. Any operation on floating point numbers should be thought of as introducing an error of at least ϵ_m. When the results of such operations are fed into other operations to form an algorithm, these errors propagate through the calculations. One might hope that random errors of opposite sign cancel each other so that the error in an n-step calculation scales as $\epsilon \sim \sqrt{n}\,\epsilon_m$. In practice,

- some algorithms contain regularities which mean that $\epsilon \sim n\,\epsilon_m$.
- certain individual floating point operations can hugely amplify the error.

A particularly important example of the latter is known as "loss of significance". This occurs when two nearly equal numbers are subtracted. Loss of significance is best illustrated with an example from finite precision decimal arithmetic. Let us represent numbers, x, to 5-digit decimal precision with the notation $\mathrm{fl}(x)$. Consider the following exact arithmetic calculation:

$$x = 0.123456789 \qquad y = 0.1234 \qquad z = x - y = 0.000056789.$$

Repeating the calculation in 5-digit precision gives

$$\mathrm{fl}(x) - \mathrm{fl}(y) = 0.12345 - 0.12340 = 0.00005.$$

Note that the result contains only a single significant digit. This has terrible consequences for the relative accuracy of the calculation. Although the relative size of the rounding errors in x and y are very small

$$\frac{x - \mathrm{fl}(x)}{x} \approx 0.0000549 \qquad \frac{y - \mathrm{fl}(y)}{y} = 0,$$

the relative size of the rounding error in $x - y$ is almost 12%:

$$\frac{z - (\mathrm{fl}(x) - \mathrm{fl}(y))}{z} \approx 0.1195!$$

1.1.4 Stability of algorithms

Certain operations can amplify rounding errors. This can sometimes lead to catastrophic failure when algorithms which are exact in conventional arithmetic are executed in floating point. Such algorithms are said to be numerically unstable. The branch of mathematics which studies how errors propagate in algorithms is called numerical analysis. Although somewhat artificial, the following example from [2, chap. 1] clearly illustrates the concept of numerical instability. Consider the problem of calculating the series, $\{a_n = \phi^n,\ n = 0, 1, 2, \ldots\}$, of integer powers of the golden mean, $\phi = (\sqrt{5} - 1)/2$. This can be done in two ways:

- **Method 1:**
  Set $a_0 = 1$ and calculate
  $$a_n = \phi\, a_{n-1} \qquad \text{for } n = 1, 2, \ldots \tag{1.1}$$
- **Method 2:**
  Set $a_0 = 1$, $a_1 = \phi$ and calculate
  $$a_n = a_{n-2} - a_{n-1} \qquad \text{for } n = 2, 3, \ldots \tag{1.2}$$

The second method is less obvious but it is easily demonstrated by direct substitution that ϕ satisfies this recurrence relation. Both are mathematically equivalent and correct. Fig. 1.1 shows the implementation of these two methods on my laptop computer with $\phi = 0.6$ (8 digits of precision). Clearly method 2 starts to go seriously wrong by about $n = 20$

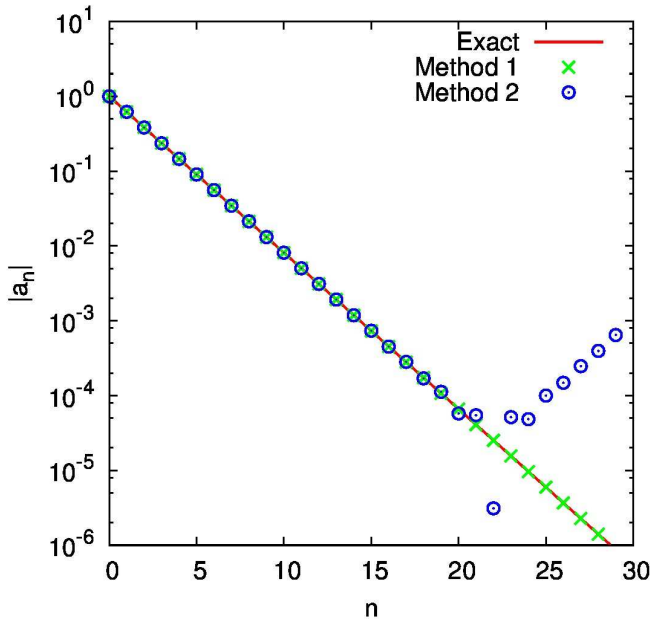


Figure 1.1: Numerical implementation of algorithms (1.1) and (1.2)

and gives entirely wrong results for larger values of n. The reason is that the algorithm (1.2) is unstable and amplifies the rounding error in the value of ϕ.

The reason for this can be seen by noting that the computer is not solving Eq. (1.2) with the starting conditions $a_0 = 1$ and $a_1 = \phi$ but rather with the starting conditions $a_0 = 1$ and $a_1 = \phi + \varepsilon$. If we make the ansatz

$$a_n = x^n,$$

we find by direct substitution that Eq. (1.2) is satisfied provided that

$$x = x_+ \qquad \text{or} \qquad x = x_-$$

where

$$x_\pm = \frac{-1 \pm \sqrt{5}}{2}.$$

$x_+ = \phi$ corresponds to the solution we want. Since Eq. (1.2) is a linear recurrence relation, the general solution is

$$a_n = C_1\, x_+^n + C_2\, x_-^n,$$

where $C_{1,2}$ are constants. If we use the starting conditions $a_0 = 1$ and $a_1 = \phi + \varepsilon$ to evaluate these constants we obtain

$$a_n = \left(1 - \frac{\varepsilon}{\sqrt{5}}\right) \phi^n + \frac{\varepsilon}{\sqrt{5}}\, x_-^n.$$

The second term grows (in a sign-alternating fashion) with n. Therefore any small amount of error, ε, introduced by finite precision is amplified by the iteration of the recurrence relation and eventually dominates the solution. Avoiding such instabilities is a key priority in designing good numerical algorithms.

1.2 Computational complexity of algorithms

1.2.1 Counting FLOPS and Big O notation

The performance speed of a computer is usually characterised by the number of FLoating-point OPerations per Second (FLOPS) which it is capable of executing. Computational complexity of an algorithm is characterised by the rate at which the total number of arithmetic operations required to solve a problem of size n grows with n. We shall denote this number of operations by $F(n)$. The more efficient the algorithm, the slower the $F(n)$ grows with n. The computational complexity

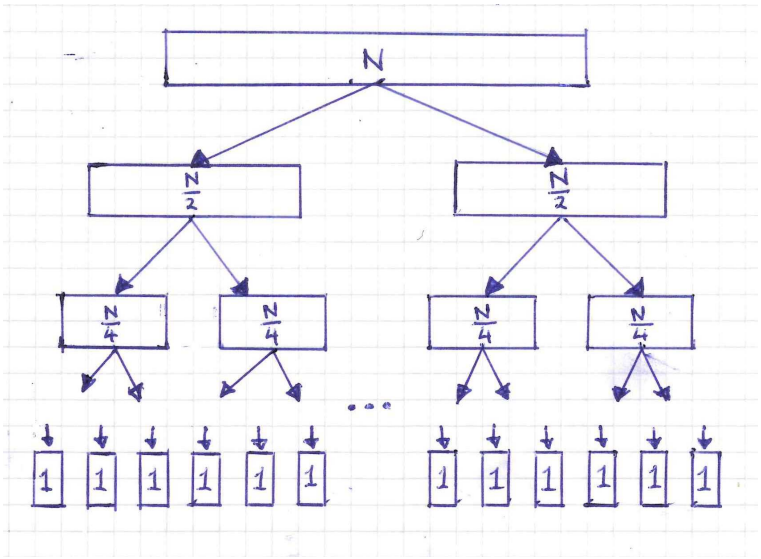


Figure 1.2: Conceptualisation of the recursive structure of a divide-and-conquer algorithm.

of an algorithm is usually quantified by quoting an asymptotic bound on the behaviour of $F(n)$ as $n \to \infty$. The common notation, $F(n) \sim O(g(n))$, means that for sufficiently large values of n,

$$|F(n)|\,| \leq k \cdot g(n)$$

for some positive k. This says that as n gets large, the absolute value of $F(n)$ is bounded from above by a positive multiple of the function $g(n)$. There is a proliferation of more precise notation in use to express lower bounds and degrees of tightness of bounds which we will not really need here but which you can read about on Wikipedia [3]

Questions of computational complexity and deciding whether a particular algorithm is optimal or not can by highly nontrivial. For example, consider the problem of multiplying two matrices of size $n \times n$:

$$\mathbf{A}\,\mathbf{B} = \mathbf{C}.$$

If we use textbook matrix multiplication, we must compute n^2 entries to arrive at the matrix $\mathbf{C}$. Each of these entries requires taking the scalar product of a row from $\mathbf{A}$ with a column from $\mathbf{B}$. Since each is of length n, the scalar product requires n multiplications and n additions, or $2n$ FLOPs. The total number of operations required to multiply the two matrices is then $n^2 \times 2n = 2n^3$. So textbook matrix multiplication is a $O(n^3)$ algorithm.

1.2.2 Algorithmic complexity of matrix multiplication and Strassen's algorithm

It is interesting to ask could we do any better than $2n^3$? Suppose for simplicity that $n = 2^m$ for some integer m. We could then go about the multiplication differently by dividing the matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ into smaller matrices of half the size,

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{pmatrix} \quad \mathbf{C} = \begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix}$$

up the problem into smaller matrix multiplications of half the size:

$$\begin{aligned} \mathbf{C}_{11} &= \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} \\ \mathbf{C}_{12} &= \mathbf{A}_{11}\mathbf{B}_{21} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{C}_{21} &= \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} \\ \mathbf{C}_{22} &= \mathbf{A}_{21}\mathbf{B}_{21} + \mathbf{A}_{22}\mathbf{B}_{22}. \end{aligned} \tag{1.3}$$

This is an example of a type of recursive algorithm known as a "divide and conquer" algorithm. The operation of multiplication of matrices of size $n \times n$ is defined in terms of the multiplication of matrices of size $\frac{n}{2} \times \frac{n}{2}$. It is implicitly understood in Eq. (1.3) that when we get to matrices of size 1×1, we do not recurse any further but simply use scalar multiplication (this is called the "base case" in the literature on recursive algorithms). The conceptual structure of a divide-and-conquer

algorithm is illustrated in Fig. 1.2.It may not be immediately clear whether we gained anything by reformulating the textbook matrix multiplication algorithm in this way. Eq. (1.3) involves 8 matrix multiplications of size $\frac{n}{2}$ and 4 matrix additions of size $\frac{n}{2}$ (matrix addition clearly requires n^2 FLOPs). Therefore the computational complexity, $F(n)$, must satisfy the recursion

$$F(n) = 8\,F\left(\frac{n}{2}\right) + 4\,\left(\frac{n}{2}\right)^2,$$

with the starting condition $F(1) = 1$ to capture the fact that we use scalar multiplication at the lowest level of the recursion. The solution to this recursion is

$$F(n) = 2n^3 - n^2.$$

Although we have reduced the size of the calculation slightly (by doing fewer additions?) the divide-and-conquer version of matrix multiplication is still an $O(n^3)$ algorithm and even the prefactor remains the same. Therefore we have not made any significant gain for large values of n. Consider, however, the following instead of Eq. (1.3):

$$\begin{aligned}
\mathbf{C}_{11} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{12} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{C}_{21} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{C}_{22} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6,
\end{aligned} \tag{1.4}$$

where the matrices $\mathbf{M}_1$ to $\mathbf{M}_7$ are defined as

$$\begin{aligned}
\mathbf{M}_1 &= (\mathbf{A}_{11} + \mathbf{A}_{22})\,(\mathbf{B}_{11} + \mathbf{B}_{22}) \\
\mathbf{M}_2 &= (\mathbf{A}_{21} + \mathbf{A}_{22})\,\mathbf{B}_{11} \\
\mathbf{M}_3 &= \mathbf{A}_{11}\,(\mathbf{B}_{12} - \mathbf{B}_{22}) \\
\mathbf{M}_4 &= \mathbf{A}_{22}\,(\mathbf{B}_{21} + \mathbf{B}_1) \\
\mathbf{M}_5 &= (\mathbf{A}_{11} + \mathbf{A}_{12})\,\mathbf{B}_{22} \\
\mathbf{M}_6 &= (\mathbf{A}_{21} - \mathbf{A}_{11})\,(\mathbf{B}_{11} + \mathbf{B}_{12}) \\
\mathbf{M}_7 &= (\mathbf{A}_{12} - \mathbf{A}_{22})\,(\mathbf{B}_{21} + \mathbf{B}_{22})\,.
\end{aligned} \tag{1.5}$$

These crazy formulae were dreamt up by Strassen [4]. See also [2, chap. 2]. Since the sub-matrices of $\mathbf{A}$ and $\mathbf{B}$ are nowhere commuted in these formulae everything really works and this is a perfectly legitimate way to calculate the product $\mathbf{A}\,\mathbf{B}$. Notice that compared to the 8 multiplications in Eq. (1.3), Eqs. (1.4) and (1.5) involve only 7 multiplications. Admittedly the price to be paid is a far greater number of additions - 18 instead of 4. However the savings from the lower number of multiplications propagates through the recursion in a way which more than compensates for the (subleading) contribution from the increased number of additions. The computational complexity satisfies the recursion

$$F(n) = 7\,F\left(\frac{n}{2}\right) + 18\,\left(\frac{n}{2}\right)^2,$$

with the starting condition $F(1) = 1$. The solution to this recursion is

$$F(n) = n^{\frac{\log(7)}{\log(2)}} - 6n^2. \tag{1.6}$$

Strassen's algorithm is therefore approximately $O(n^{2.81})$! I think this is astounding.

For large problems, considerations of computer performance and algorithmic efficiency become of paramount importance. When the runtime of a scientific computation has reached the limits of what is acceptable, there are two choices:

- get a faster computer (increase the number of FLOPS)
- use a more efficient algorithm

Generally one must trade off between algorithmic efficiency and difficulty of implementation. As a general rule, efficient algorithms are more difficult to implement than brute force ones. This is an example of the "no-free-lunch" theorem, which is a fundamental principle of computational science.

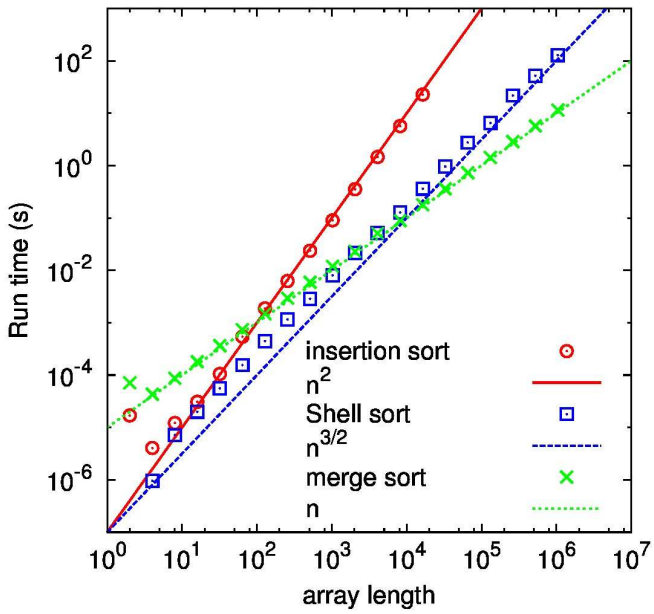


Figure 1.3: Performance measurements of 3 different sorting algorithms: insertion sort (red circles), Shellsort with powers of two increments (blue squares) and mergesort (green crosses). The solid lines are guides show different theoretical scalings.

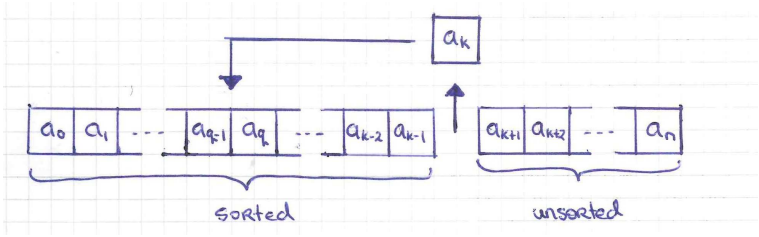


Figure 1.4: General idea of the insertion sort algorithm. At step k of the algorithm, the elements $a_1 \ldots a_{k-1}$ are already sorted. We need to take the next list element a_k, find the value of q in the range $0 \leq q < k-1$ such that $a_{q-1} \leq a_k < a_q$ and insert a_k between a_{q-1} and a_q.

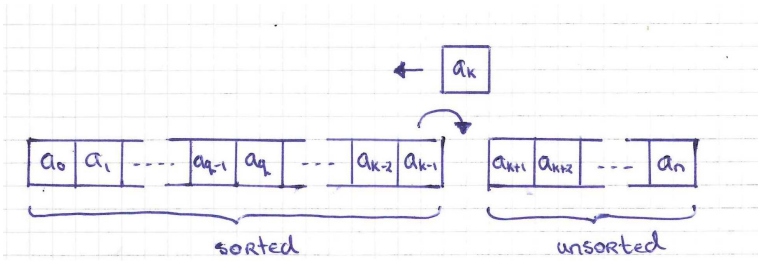
1.3 Sorting

In its simplest form, the sorting problem is the task of designing an algorithm which can transform a list of integers, $\{n_1, n_2, \ldots n_N\}$ into ascending order with the fewest possible number of comparisons. There are many sorting algorithms. We consider a few of them here (see [5] for many others) to illustrate the point that putting some thought into how to solve a problem efficiently can deliver enormous efficiency savings compared to a brute force approach. Figure 1.3 shows compares performance measurements on the three sorting algorithms known as insertion sort, Shellsort and mergesort for a range of list lengths. It is clear that the choice of algorithm has a huge impact on the execution time when the length of the list is large.

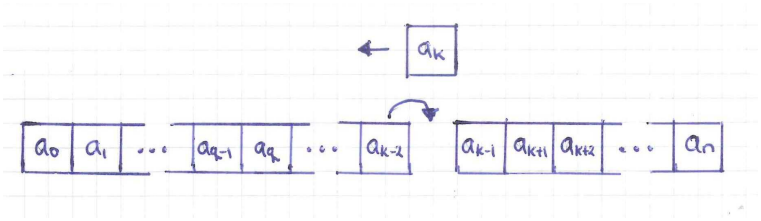
1.3.1 Insertion sort

Insertion sort is one of the more "obvious" way of sorting a list of numbers. It is often quoted as the algorithm of choice for card players sorting their hand. It is illustrated in Fig. 1.4. We begin by placing the first two list elements in order. We then take the third list element, step through the first two (which are already in order) and insert it when the correct location is found. We then repeat this process with the fourth, fifth and all remaining elements of the list. At step n of the process, the preceding n-1 elements are already sorted so insertion of element n in the correct place results in a list in which the first n are sorted. This procedure can be implemented "in place" meaning that we do not need to allocate additional memory to

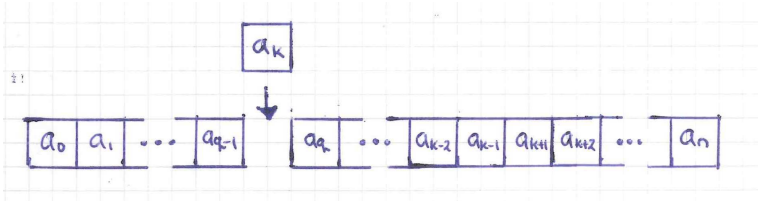
sort the array (this is not true of some other sorting procedures). This is achieved as follows. Assuming that the first $k-1$ elements are sorted, we first remove a_k from the array opening up a gap into which other elements can move:



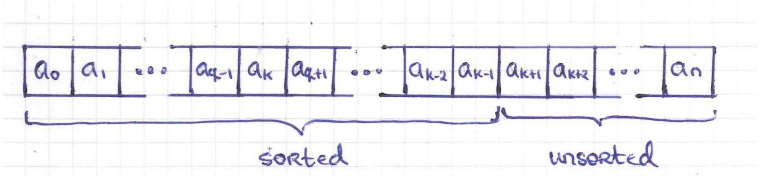
We check if $a_k < a_{k-1}$. If not, a_k is already in the right place and we put it back where it was and proceed to the next step. Otherwise we copy a_{k-1} into the gap and compare a_k to a_{k-2}.



We repeat this process, moving the sorted elements to the right until we either find a_{q-1} such that $a_k > a_{q-1}$ or we reach the beginning of the array. In either case we insert a_k into the gap and proceed to the next step.



The resulting array now has the first k elements sorted.



We repeat this process until the entire array is sorted. For each of the n elements of the array we have to make up to $n-1$ comparisons. The complexity of the insertion sort algorithm is therefore $O(n^2)$.

1.3.2 Partial sorting and Shell's method

A *partial sort* with increment q of a list of numbers is an ordering of the array in which all subsequences with interval q are sorted. The concept is best illustrated with an example. Fig. 1.5 shows a partial sort of the array $\{a_0, a_1, \ldots a_{18}\}$ with increment 4. The array itself is not sorted globally but the subarrays $\{a_0, a_4, a_8, a_{12}, a_{16}\}$, $\{a_1, a_5, a_9, a_{13}, a_{17}\}$, $\{a_2, a_6, a_{10}, a_{14}, a_{18}\}$ and $\{a_3, a_7, a_{11}, a_{15}\}$ are each individually sorted. A small modification to the insertion sort algorithm described above allows one to produce partial sorts: at each step of the algorithm we step through the sorted section of the array in increments of q instead of increments of 1. Obviously a partial sort with increment 1 is a full sort.

While the usefulness of a partial sort may not be immediately obvious, partial sorts can be used to significantly speed up the insertion sort while retaining the "in-place" property. An improvement on the insertion sort, known as ShellSort (after its originator [6]), involves doing a series of partial sorts with intervals, q, taken from a pre-specified list, Q. We start from a large increment and finish with increment 1 which produces a fully sorted list. As as simple example we could take the intervals to be powers of 2: $Q = \{2^i : i = i_{\max}, i_{\max}-1, \ldots, 2, 1, 0\}$ where $i_{\max}$ is the largest value of i such that $2^i < n/2$. The performance of the ShellSort algorithm depends strongly on the choice of the sequence Q but it is generally faster on average than insertion sort. This is very counter-intuitive since the sequence of partial sorts always includes a full insertion sort as its final step! The solution to this conundrum is to realise that each partial sort allows

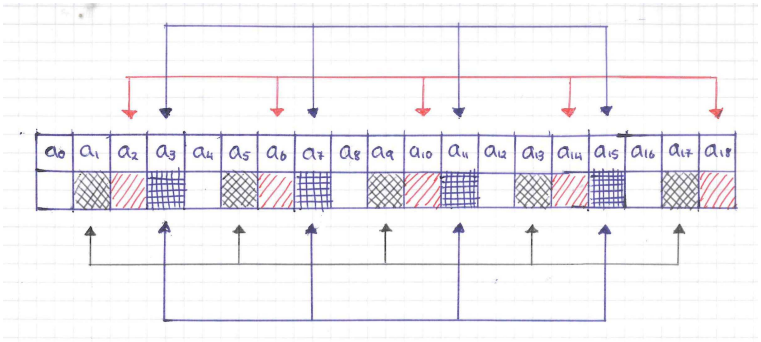


Figure 1.5: Graphical representation of a partial sort of the array $\{a_0, a_1, \ldots a_{18}\}$ with increment 4.

elements to move relatively long distances in the original list so that elements can quickly get to approximately the right locations. The initial passes with large increment get rid of large large amounts of disorder quickly, and leaving less work for subsequent passes with shorter increments to do. The net result is less work overall - the final (full) sort ends up sorting an array which is almost sorted so it runs in almost $O(n)$ time.

How much of an overall speed-up can we get? Choosing Q to be powers of 2 for example, scales on average as about $O(n^{3/2})$ (but the worst case is still $O(n^2)$). See Fig. 1.3. Better choices include $Q = \{(3^i - 1)/2 \ : \ i = i_{\max}, i_{\max} - 1, \ldots, 2, 1, 0\}$ for which the average scales as about $O(n^{5/4})$ and the worst case is $O(n^{3/2})$ [5]. The general question of the computational complexity of ShellSort is still an open problem.

1.3.3 Mergesort

A different approach to sorting based on a divide-and-conquer paradigm is the mergesort algorithm which was originally invented by Von Neumann. Mergesort is based on the merging of two arrays *which are already sorted* to produce a larger sorted array. Because the input arrays are already sorted, this can be done efficiently using a divide-and-conquer approach. Here is an implementation of such a function in Python:

```
def merge(A,B):
    if len(A) == 0:
        return B
    if len(B) ==0:
        return A
    if A[0] < B[0]:
        return [A[0]] + merge(A[1::],B)
    else:
        return [B[0]] + merge(A,B[1::])
```

This function runs in $O(n)$ time where n is the lenght of the output (merged) array. Can you write down the recursion satisfied by $F(n)$ and show that this function executes in $O(N)$ time? Armed with the ability to merge two sorted arrays together, a dazzlingly elegant recursive function which performs a sort can be written down:

```
def mergeSort(A):
    n=len(A)
    if n == 1:
        return A    # an array of length 1 is already sorted
    else:
        m=n/2
        return merge(mergeSort(A[0:m]), mergeSort(A[m:n]))
```

How fast is this algorithm? It turns out to be $O(n \log(n))$ which is a significant improvement. We can see this heuristically as follows. Suppose that n is a power of 2 (this assumption can be relaxed at the expense of introducing additional technical arguments which are not illuminating). Referring to Fig. 1.2, we see that the number of levels, L, in the recursion is $n = 2^L$ from which we see that $L = \log(n)/\log(2)$. At each level, m, in the recursion, we have to solve 2^m sub-problems, each of size $n/2^m$. The total work at each level of the recursion is therefore $O(n)$. Putting these two facts together we can see

that the total work for the sort is $O(n\ \log(n))$. The performance of mergesort is shown in Fig. 1.3. It clearly becomes competitive for large n although the values of n reached in my experiment don't seem to be large enough to clearly see the logarithmic correction to linear scaling. Note that mergesort is significantly slower for small arrays due to the additional overheads of doing the recursive calls. There is an important lesson here too: the asyptotically most efficient algorithm is not necessarily the best choice for finite sized problems!

Bibliography

[1] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991. `http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`.

[2] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes: The art of scientific computing (3rd ed.)*. Cambridge University Press, New York, NY, USA, 2007.

[3] Big o notation, 2014. `http://en.wikipedia.org/wiki/Big_O_notation`.

[4] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[5] D. E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Professional, Reading, Mass, 2 edition edition, 1998.

[6] D. L. Shell. A high-speed sorting procedure. *Commun. ACM*, 2(7):30–32, 1959.