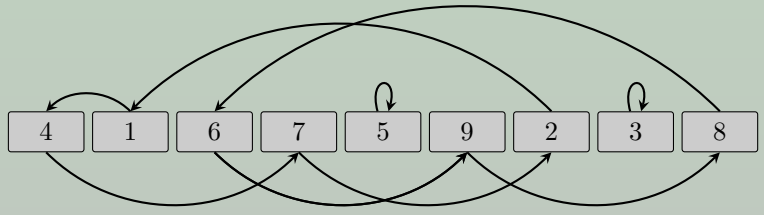
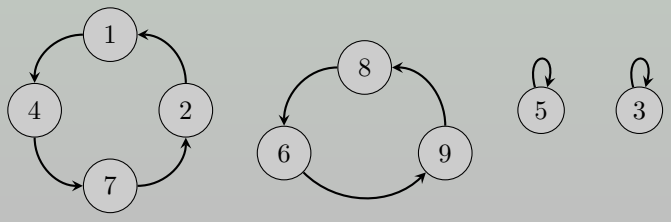
Fundamental Algorithms

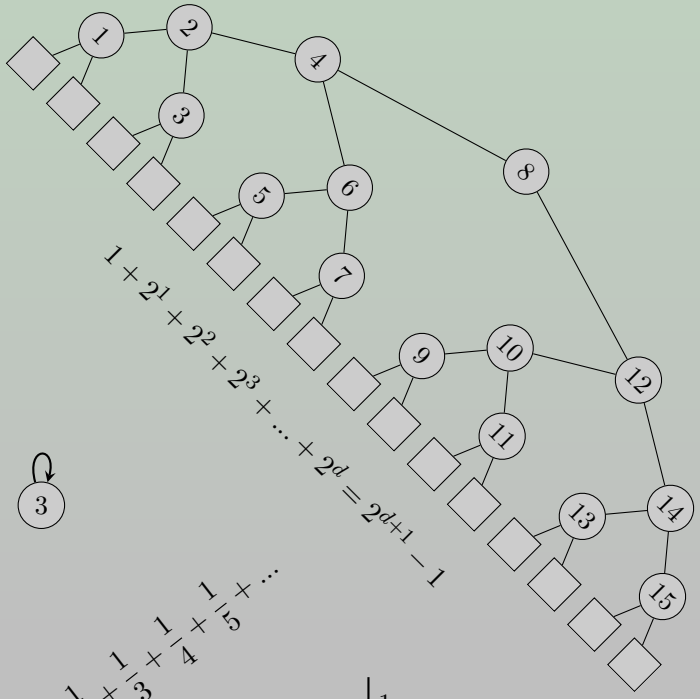
A Tour for scientists and engineers



$$c(n,k) = c(n-1,k-1) + (n-1)c(n-1,k)$$

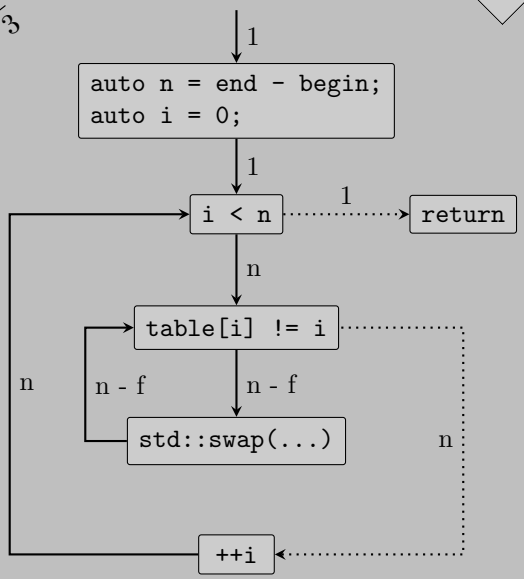


$$p = 1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \frac{1}{4!} - \ldots$$



$$1 + 2^1 + 2^2 + 2^3 + \cdots + 2^d = 2^{d+1} - 1$$

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots$$




```
template <class Iter>
void unpermute(Iter begin, Iter end, Iter table)
{
    auto n = end - begin;
    for (auto i = 0; i < n; ++i) {
        while (table[i] != i) {
            std::swap(begin[i], begin[table[i]]);
            std::swap(table[i], table[table[i]]);
        }
    }
}
```

Marcelo Zimbres (mzimbres@gmail.com)

Preface

A large amount of positions in the software industry require professionals with a good understanding of the foundations of physics, mathematics, engineering and computer science. As a result of this demand it is not uncommon for scientists and engineers to pursue a career in software development, a discipline that they did not have the opportunity to learn, most notably in regard to its fundamental topics such as data structures and algorithms.

On the other hand, the background in mathematics that most scientists and engineers have put them in a position where they can learn this new field without much difficult. Data structures and algorithms is extensive in its contents and can become quite heavy if covered in detail, however by focusing on its fundamental topics it is possible to go quite far, solve problems that are complex and use most libraries and programming languages with confidence. The 80-20 rule suggests that more than 80% of the problems are covered by only 20% of the theory, therefore when programming we will be likely using no more than 0.8% of the theory, more than half of the time!

This book was written with that in mind and covers background material that is considered fundamental, including an in depth mathematical treatment. It is meant for those people described above with some background in mathematics and programming that want to either learn or improve their understanding of the building blocks of software. I also pay special attention to people that are preparing for job interviews by providing special problem sections with focus on questions that arise frequently. I have tried my best to organize these problems according to the chapters they correlate the most, however the reader shouldn't be surprised if they have little connection with the material covered in the text, often times *coding interview* problems have little *direct* overlap with fundamental topics in data structures and algorithms.

The language that was chosen to present the subject is `C++` given its widespread use in engineering and scientific programming. Only average knowledge however is assumed from the reader and any advanced feature that is needed will be reviewed before being used. The level of mathematics necessary to follow the text corresponds to the first couple semesters in undergraduate courses in natural sciences.

Marcelo Zimbres
2018, Karlsruhe

Prologue

The daily life of a programmer consists of using simple building blocks called data structures and algorithms to build complex things called programs. The limit on how complex a program can get seems to be bound on the ability of its authors to understand it. Experience shows however that the complexity of many programs out there in the real world grow beyond that limit, to the point where small changes can have unpredictable side effects and lead to disaster, profilers cannot spot the performance bottleneck anymore and a complete rewrite of the system is a wish whispered far too often on the corridors of corporations. Instead, those changes are most often appended to existing interfaces, contributing to even more incomprehensible and unpredictable systems. To address these problems we see a proliferation of coding guidelines and good practices being proposed on a yearly basis, with software teams around the globe claiming to follow them rigorously, often times to no avail.

Writing programs that are simple and elegant is far from trivial and requires years of experience, hard training and above all, a solid knowledge of the foundations of computer science. The reward for all this effort is however high since simple code lends itself to being grasped in one single act of mental apprehension. If on one side there is no limit on how complex a program can grow on the other there does exist a limit on how simple it can be, where there is nothing more to add or remove, with an emphasis on the last one. Simplicity, often times associated with beauty, has been influencing humanity since its birth and comes up most often in connection with arts, however the role it plays in diverse fields such as mathematics and physics do not second that of art and given the importance it also has in software it is instructive to trace some parallels.

Physicists have been witnessing their theories that describe the world getting simpler and simpler along the centuries and many important physicists consider in fact simplicity and beauty to belong to a fundamental theory of nature. Examples of these phenomena are for instance the simplification in the description of planetary motion with the introduction of the heliocentric model and its later incorporation in the theory of gravitation, when Sir Issac Newton discovered that the force that pulls us down to the earth is also responsible for keeping the moon in orbit. The unification of electricity and magnetism into one single phenomena called electromagnetism, described by a single set of equations. The massive simplification brought by the principles of the special theory of relativity stating that all physical laws are independent of the translational motion between observers in iner-

tial frames. The further extension of these principles to encompass gravity and observers that are accelerating relative to each other in the so called general theory of relativity. The list goes on and on and the trend is though to proceed even further, specially in fundamental theories like the standard model of particle physics where there seems to be a consensus among physicists that in the end of all this massive simplification we will be left with a single theory of everything.

Why did it take us all these centuries to understand the simplicity behind nature? The answer to this question is that simplicity is not obvious at all and we may be sometimes even fooled by our senses about it. A famous example occurred to one of the most important astronomers of all time, Johannes Kepler, when he proposed a model for the solar system where the distances between planets could be understood in terms of the platonic solids, a model that had a strong aesthetic appealing to him.

In the realm of software we experience a similar thing, some time after writing a program we notice that the design once though to be simple and elegant is not that good after all and that there are simpler ways of doing it that were not obvious when we wrote it. This process repeats itself typically many times along the lifetime of a program. But to begin with, what criteria shall we use the differentiate simple and beautiful from complex and ugly in software? As we saw above in our analogy with physics, we may be fooled by our senses about beauty! Additionally, people have different opinions on these matters and since everyone can claim to be writing simple and elegant code we need something more concrete than mere opinion. This concern is perhaps best justified in view of the popular adage *beauty is on the eyes of the beholder*. In physics, a theory that is claimed to be simple and elegant can be tested on an experiment and be eventually falsified if it is incorrect. If we have to decide between two theories that predict correct results it is customary to use Occan's razor to prefer the one that is considered simpler in some sense, for example, by making less assumptions. In the case of software we also test if our code produces correct values but the test itself is just a prerequisite and not a final criterion for the simplicity we want to achieve.

There does not seem to be a consensus to what the answers to this question really are, opinions varying widely. In the case of software, we usually do not aim only for simplicity but also for flexibility and scalability, since it is common to have to add new functionality on existing software, for which it may not have been planned for in first place. In the repertoire of techniques that are supposed to help us keeping complexity under control and to guarantee high quality code we see, object orientation, functional, generic,

extreme, literate, meta etc. programming, unit testing, agile development, code review, design patterns, pair programming, UML, test driven development, clean code etc, not to mention all the tools, libraries and programming languages to support these techniques.

One point however where there is a strong agreement among the experts, is that there is no hope for anyone in writing great code without a solid knowledge of the its fundamental building blocks, the grounds on which it grows, namely, the data structures and algorithms used by the programmer. Getting them right in first place plays definitely a major role in our final criterion for simple and elegant code. A well trained programmer has less difficult in recognizing the most suitable data structures and algorithms for his problem at hand and most importantly, to reason about a problem in terms of them.

In this book we will be concerned with algorithms that have a more or less exact mathematical treatment. The intuition gained while studying them is fundamental to approach complex problems that cannot be understood in an exact way and gives the programmer the ability to see the consequences of his design decisions early in development. When we refer to an algorithm that is *simple* we are not necessarily meaning non-advanced as simple problems may require concepts that are considered advanced in some sense, specially in regard to the mathematics they may require to be understood. This is not any different than in physics where we see very clean and simple theories such as the general theory of relativity requiring quite advanced math to be formulated.

Contents

Part I

Fundamental topics

In this introductory chapter we will review some fundamental algorithms that arise in connection with arrays, analysing them carefully and using them as an invitation to introduce some mathematical tools that will be needed in later chapters. An array is perhaps the simplest data structure used on software, it is just a collection of elements placed side by side in computer memory so that to reach the next element in a sequence we have to go to the next memory address. It is important to recognize this fact from the beginning since as we will see, more complex data structures have to either store some information about how to reach the next element, resulting in a greater memory consumption or develop a rule so that the location of an element in an array can be inferred. By the end of this chapter the need of more complex data structures than arrays should become clear and wished for.

Chapter 1

Searching

A fundamental operation on programming is that of searching for a specific element in a set of things. Search algorithms varies widely according to the data structure that is used to store the elements and we can say with confidence that both concepts are in fact bound to each other in this regard. The choice for a specific data structure in turn comes up most often with the need of improving the performance of a program, sometimes it is even possible to do completely away with the need of searching with a suitable choice of the data structure, some of them will appear in the next chapter. Before we get to abstract, let us see some algorithms that arise frequently when working with arrays.

1.1 Linear search

A good starting point in the realm of search algorithms is the linear search given the simplicity of the problem it tries to solve: search for a specific element in a set given that their arrangement in the array is unknown. It seems in fact too simple to be worth the effort but as we shall see, many interesting techniques arise from its analysis. All it does is to traverse a range of elements returning either successfully or unsuccessfully depending on whether the element has been found or not. A graphical illustration of this procedure is shown in figure 1.1 and a typical STL implementation appears in code fragment 1.1.

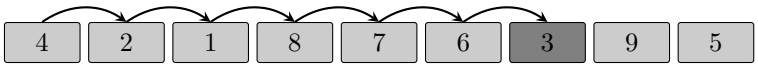


Figure 1.1: Graphical illustration of the linear search. The element found is shown in the darker box.

```
1 template <class Iter, class T>
2 auto find(Iter begin, Iter end, T const& k)
3 {
4   while (begin != end && *begin != k)
5     ++begin;
6
7   return begin;
8 }
```

Listing 1.1: Typical STL implementation of the linear search.

The practice of passing begin and end iterators to specify a range is convenient very often but not always optimal. In this case we can improve performance and simplify the function by introducing a sentinel at the end of the array so that we either hit the desired element or the sentinel, freeing ourselves from having to continuously check whether the range end has been reached, code fragment 1.2 illustrates this procedure.

```
1 template <class Iter, class T>
2 auto find_with_sentinel(Iter begin, const T& v)
3 {
4   while (*begin != v)
5     ++begin;
6
7   return begin;
8 }
```

Listing 1.2: Linear Search with sentinel

It is not difficult to see that the execution time of this algorithm is determined by the number of comparisons performed in the condition `if (*begin != k)`. In the best possible scenario it performs only one comparison to find the element and in the worst it may have to check them all, hitting the last element on a successful search or the end of the array in an unsuccessful one. We will be also often interested in the average execution time of algorithms, for example, we see that to find the first element in the range we need one comparison, for the second two, for the third three and so on. If we have n

elements we need a total of

$$1 + 2 + 3 + ... + n = \frac{n(n+1)}{2}, \tag{1.1}$$

comparisons to find each one of them, concluding that its average is given by $\overline{k} = (n+1)/2$. We have to be careful however since for some input data we may be looking for some elements more often than others, above we assumed for example that all elements were equally interesting. That is one reason why it is convenient to work with the general definition of average

$$\overline{k} \;=\; \sum_{k=1}^{n} k p_k, \tag{1.2}$$

where p_k is the probability of making k comparisons. If all elements in the range are equally likely we have for example $p_k = 1/n$.

Sometimes we are also interested in how close successful searches are to the average and the quantity that is used most often to express it is the so called *standard deviation* σ, defined by

$$\sigma^2 \;=\; \sum_{k=1}^{n} (k - \overline{k})^2 p_k = \sum_{k=1}^{n} k^2 p_k - \overline{k}^2 \tag{1.3}$$

(see exercise 1.1). It must be clear at this point that the form of the p_k determine the runtime behaviour of the linear search so let us work out some important cases.

1.1.1 Uniform distribution

As previously said, if all elements in the array are equally interesting then $p_k = 1/n$, a result that follows from the normalization of the probability $p_1 + p_2 + ... + p_n = 1$. Inserting this into (1.2) results in

$$\overline{k} \;=\; \frac{1 + 2 + 3 + ... + n}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}, \tag{1.4}$$

namely, the average number of comparisons is as most people would expect approximately half of the total number of elements. This value of $\overline{k}$ can now be used to calculate the standard deviation σ. Using (1.3) we have

$$\sigma^2 \;=\; \frac{1 + 2^2 + 3^2 + ... + n^2}{n} - \frac{(n+1)^2}{4}. \tag{1.5}$$

At this point if we happen to know the formula

$$1 + 2^2 + 3^2 + ... + n^2 = \frac{n(n+1)(2n+1)}{6}, \tag{1.6}$$

we could insert it into equation (1.5) and obtain the desired result

$$\sigma = \sqrt{\frac{(n+1)(n-1)}{12}}. \tag{1.7}$$

However, in most cases a closed expression for summations like that are either unknown or difficult to guess. That provides us with a good opportunity to introduce the so called *generating function* that is a function used to encode the information contained in a sequence like the p_k in a power series.

1.1.2 Generating function

Consider the following function

$$G(z) = p_1 z + p_2 z^2 + p_3 z^3 + ... + p_n z^n = \sum_{k=1}^{n} p_k z^k. \tag{1.8}$$

From the fact that the probability must be normalized $G(z)$ has the following property $G(1) = p_1 + p_2 + p_3 + ... + p_n = 1$. Other interesting properties arise from the derivatives of $G(z)$, for example

$$G'(z) = \sum_{k=1}^{n} k p_k z^{k-1}, \tag{1.9}$$

with the special case

$$G'(1) = \overline{k} = \sum_{k=1}^{n} k p_k, \tag{1.10}$$

namely, $G'(1)$ represents the average $\overline{k}$. One can also show that the standard deviation is given by

$$\sigma^2 = G''(1) + G'(1) - G'(1)^2 \tag{1.11}$$

(see exercise 1.2). We have now all the machinery necessary to evaluate (1.5) in a more general way. By inserting $p_k = 1/n$ into (1.8) we see that the generating function has a closed form

$$G(z) = \frac{z}{n} + \frac{z^2}{n} + \frac{z^3}{n} + ... + \frac{z^n}{n} = \frac{1}{n}\frac{z^{n+1} - z}{z - 1} \tag{1.12}$$

(see exercise 1.3). To evaluate $G'(z)$ and $G''(z)$ we can either derive (1.12) or read off the coefficients from its Taylor expansion. Let us do the later, the Taylor expansion of $G(z)$ has the general form

$$G(1+z) = G(1) + G'(1)z + G''(1)\frac{z^2}{2!} + ..., \tag{1.13}$$

whereas its expansion in powers of z reads

$$\begin{aligned} G(1+z) &= \frac{1}{n}\frac{(1+z)^{n+1} - 1 - z}{z} \\ &= 1 + \frac{n+1}{2}z + \frac{(n+1)(n-1)}{6}z^2 + ..., \end{aligned} \tag{1.14}$$

where we have used the approximation $(1+z)^n \approx 1 + nz$. By comparing the two we see that $G'(1) = (n+1)/2$, agreeing with (1.4) and $G''(1) = (n+1)(n-1)/6$, inserting this into (1.11) results in (1.7). A summary of the number of comparisons is shown on table 1.1.

Min	Max	Mean	Variance
1	n	$\frac{n+1}{2}$	$\frac{(n+1)(n-1)}{12}$

Table 1.1: Summary of the number of comparisons for a uniform distribution.

1.1.3 Zipf's law and harmonic numbers

The arrangement of the elements in the array analysed above where all elements are equally probable is not the most likely to occur in practice. For example, we may want to organize the elements such that those with the most frequent access are located first in the array. In such case a good fit for the probability p_k is given by the Zipf's law, that states that the first element is searched for twice as often as the second, three times as often as the third and so on. In other words, the probability of performing k comparisons is inversely proportional to the element position in the array, resulting in a probability of the form $p_k = c/k$ where c is a constant to be determined. From the normalization of the probability $p_1 + p_2 + ... + p_n = 1$ it follows that

$$\frac{1}{c} = H_n = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}. \tag{1.15}$$

The quantity H_n, called *harmonic number*, is well known in mathematical circles and arises frequently in analysis of algorithms. Its value can be approximated by the following formula

$$H_n \approx \log n + \gamma, \tag{1.16}$$

where $\gamma = 0.44742...$ is the Euler constant. This relation will be very useful to derive the asymptotic behaviour of some formulas that will appear later in this book.

We can now use $p_k = 1/kH_n$ to obtain the generating function

$$G_n(z) = \frac{1}{H_n}\left(\frac{z}{1} + \frac{z^2}{2} + ... + \frac{z^n}{n}\right) = \frac{1}{H_n}\sum_{k=1}^{n}\frac{z^k}{k}. \tag{1.17}$$

From equations (1.10) and (1.11) we can now evaluate $\overline{k}$ and σ, obtaining

$$\overline{k} = \frac{n}{H_n} \tag{1.18}$$

$$\sigma = \sqrt{\frac{n(n+1)}{2H_n} - \frac{n^2}{H_n^2}}. \tag{1.19}$$

A summary of the number of comparisons for this case appears on table 1.2, figure 1.2 shows the behaviour of $\overline{k}$ and σ for some typical values of n.

Min	Max	Mean	Variance
1	n	$\frac{n}{H_n}$	$\frac{n(n+1)}{2H_n} - \frac{n^2}{H_n^2}$

Table 1.2: Summary of the number of comparisons for Zipf's law.

1.1.4 Counting instructions

. So far we have been focused in understanding the execution of the linear search considering the number of comparisons performed. It is now possible to go one step further and calculate the exact number of instructions executed, for example, if we assume the execution of all instructions to take the same unit of time u, we see that the total time needed for a successful search of algorithm 1.1 will be

$$t = (3k + 1)u, \tag{1.20}$$

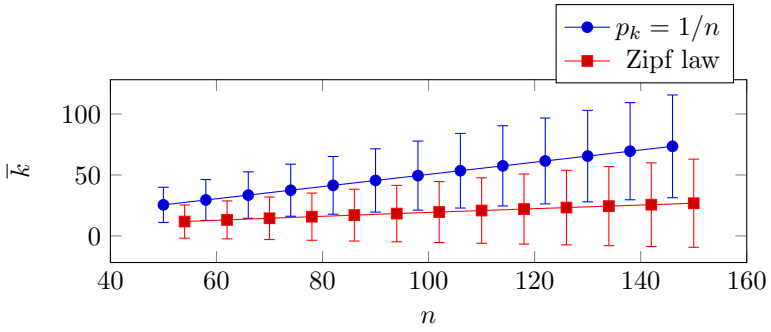


Figure 1.2: Behaviour of $\overline{k}$ and σ (represented by the error bar) for a linear search.

where k denotes again the number of comparisons performed. Algorithm 1.2 on the other hand will take

$$t = (2k + 1)u, \tag{1.21}$$

being a clear winner. To obtain these equations more easily it is a good idea to draw a diagram that shows the execution flow in the algorithm. The exact number of times a given instruction is executed can then be calculated by means of the Kirchhoff's law that states that the flow of execution entering an instruction must be equal to the flow leaving it. Figure 1.3 shows such a digram for a successful search with algorithm 1.1.

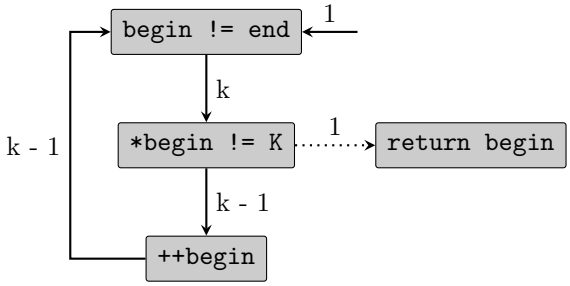


Figure 1.3: The execution flow for algorithm 1.1. Dotted arrows represent a boolean condition that evaluates to false.

In real life however, different instructions take different amounts of time to be executed so that our assumption becomes unjustifiable, specially when dealing with moderately large amounts of data, where the memory hierarchy

begins to play a significant role in the performance of instructions that read or write into RAM memory.

There are also some other reasons on why we will not prioritize such detailed analysis in this book. The first is that `C++` is a high level programming language and as such its statements do not have a clear correspondence with the machine instructions that are generated when its code is compiled, so that counting the number of instruction executed may become hard. On the other hand if we constrain ourselves to count only `C++` statements without bothering about how these statements are translated into machine code we may end up with a quantity that does not represent the program execution time very well. The second reason is that even if we could translate a `C++` statement into specific machine instructions, we would have to adopt a specific computer architecture to do so, such as x86, x64, arm, mips etc, and the reader would hardly profit anything from such a detailed analysis.

1.2 Binary search

The binary search is an algorithm that, just like the linear search, is used to search for an element in a range. The difference between the two is that the binary search is designed for ranges that are sorted, allowing us to adopt a completely different strategy, resulting in huge performance gains. In the linear search for example when two elements are compared and concluded to be different we can eliminate only one element from the remaining ones, in the case of a binary search after one comparison we can to throw away half of the elements.

The algorithm works as follows, we begin by comparing the element we want to search for with the range middle element. If the middle element is greater than the element at hand we immediately discard the second half of the range since all those elements are also greater. If however it happens to be less we do the opposite, discarding the first half of the range. By continuing this procedure with the remaining halves we eventually find an element that is neither less nor greater and that is the element we are searching for or we run out of elements, in which case we conclude that the search was unsuccessful. The graphical illustration of this procedure is shown in figure 1.4.

The procedure described above is easier said than done as there are many corner cases that are easy to get wrong. The implementation presented here uses two auxiliary variables to mark the range being searched.

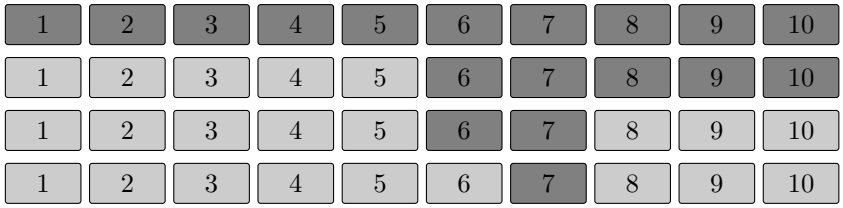


Figure 1.4: Illustration of the binary search. Successive iterations are shown from top to bottom with the discarded halves depicted in the lighter boxes. The number 7 is found after four iterations.

```
template<class Iter, class T>
auto binary_search(Iter begin, Iter end, const T& K)
{
  if (begin == end) return false;

  auto low = 0;
  auto high = end - begin - 1;
  while (low <= high) {
    auto mid = (low + high) / 2;
    if (K < begin[mid])
      high = mid - 1;
    else if (begin[mid] < K)
      low = mid + 1;
    else
      return true;
  }
  return false;
}
```

Listing 1.3: Binary search

1.2.1 The extended binary search tree

The number of comparisons performed by algorithm 1.3 is again the most important quantity related to its execution time. To analyse its behaviour properly it is convenient to introduce the *binary decision tree* associated with the input data. To construct it we use the first middle point calculated by algorithm 1.3 as the tree root node and apply this procedure in recursive fashion to the other two halves. For example, the middle point of the first half will become the root node of the left subtree. An example binary search tree can be seen on figure 1.5.

To simplify the analysis we will assume that the binary decision tree is

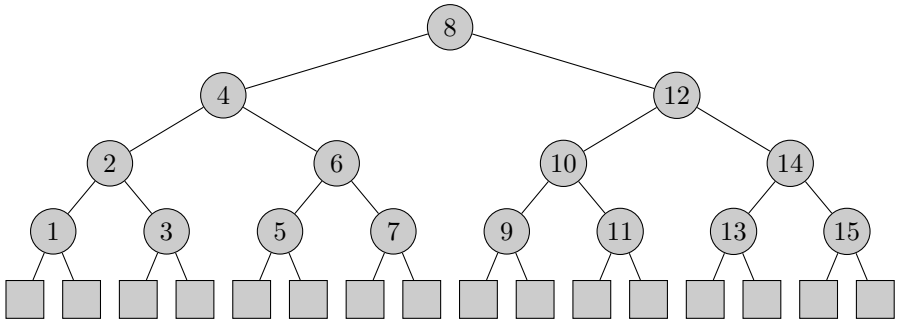


Figure 1.5: Illustration of an extended binary search tree for an array that contains the elements $1, 2, ..., 15$ in sorted order. The square nodes represent the null subtree of a given node.

perfect, meaning that it has no missing nodes at any depth. In this case we have a simple expression for the total number of nodes in the tree, namely

$$n = 1 + 2^1 + 2^2 + 2^3 + ... + 2^{d-1} = 2^d - 1, \tag{1.22}$$

where d is the total depth of the tree. This assumption does not make any important difference in our final conclusion since the execution time of a tree with a missing node in its last level is bounded by this one.

With this representation of the input data it is easy to see that in the best possible scenario we may find the element right after comparing it with the root node. In the worst case we may have to go downwards in the tree until we find the element at the bottom, and there are 2^{d-1} elements that can trigger this case, or end up in one of the 2^d nodes depicted as squares and this case represents an unsuccessful search.

The analysis of trees is so important that we will have an entire chapter dedicated to them, where the runtime behaviour of searches in trees will be treated in detail. However, given the simplicity of the three in figure (1.5) we will not wait until there and will use the opportunity to put into practice some of the methods introduced in the last section.

We have seen that if the probability p_k of performing k comparisons is known we can use it to calculate the generating function and with that its mean and standard deviation. That number is given by $p_k = C_k/n$, where C_k is the number of elements that can be reached after k comparisons. In the linear search for example we can reach only one element after k comparisons and therefore we have seen that $p_k = 1/n$. Let us keep our previous assumption that all elements are equally interesting, in that case we see that we have to perform two comparisons to find the root node, the

first to check it is not less and the second to check it is not greater, concluding that the minimum value of k to be 2, resulting in $C_2 = 1$. For the nodes directly below the root node we have to perform $k = 3$ comparisons, that is, two to identify the node plus the one we made in the root where we decided to go either left or right. Since there are two nodes at that depth we have $C_3 = 2$. It is not difficult to see that the general form of C_k is

$$C_k = 2^{k-2} \quad \text{where} \quad 2 \leq k < d. \tag{1.23}$$

We can now write the generating function for the number of comparisons, from equation (1.8) we have

$$G_n(z) \quad = \quad \frac{1}{n}\sum_{k=2}^{d+1} 2^{k-2} z^k = \frac{1}{4n}\sum_{k=2}^{d+1}(2z)^k = \frac{1}{4n}\sum_{k=1}^{d+1}(2z)^k - 2z. \tag{1.24}$$

The final summation above has a close form similar to equation (1.12), so that the final form of the generating function reads

$$G_n(z) = \frac{1}{n}\frac{((2z)^d - 1)z^2}{2z - 1}. \tag{1.25}$$

To calculate the mean and the standard deviation we can either expand equation (1.25) in powers of z and read off the coefficients as we did for the linear search or derive it, let us do the later this time, we have

$$G_n'(z) = 2\left(\frac{1}{z} + \frac{1}{2z-1}\right) G_n(z) + \frac{d2^d}{n}\frac{z^{d+1}}{2z-1}. \tag{1.26}$$

From this equation we see that

$$\overline{k} \quad = \quad \frac{d2^d}{n}. \tag{1.27}$$

Since even for small values of d we have $n = 2^d - 1 \approx 2^d$ we can approximate the value of equation (1.27) to

$$\overline{k} \quad \approx \quad d \approx \log n, \tag{1.28}$$

concluding that the total number of comparisons grows linearly with the tree depth and logarithmically with the total number of elements. The next important quantity we need is the standard deviation σ. From equation (1.26) we see that

$$G_n''(1) \quad = \quad 2 + (d-1)\overline{k}. \tag{1.29}$$

Inserting the equation above into (1.11) results in

$$\sigma = \sqrt{2 + (d - \overline{k})\overline{k}} \approx \sqrt{2}, \tag{1.30}$$

where we used $\overline{k} \approx d$ in the last approximation. That result shows that the standard deviation is constant as the number of elements grows. A summary of the number of comparisons is shown on table 1.3.

Min	Max	Mean	Variance
1	$\log n$	$\log n$	2

Table 1.3: Approximate values for the number of comparisons of the binary search.

1.2.2 Lower bound

Here we provide a solution that is suitable only for random access iterators, the STL version requires only forward iterators.

Returns an iterator pointing to the first element in the range that is not less than the value provided or the range end if no such element is found. This algorithm is part of the STL and is called `std::lower_bound`

```
1  template<class Iter, class T>
2  auto lower_bound(Iter begin, Iter end, const T& K)
3  {
4    if (begin == end)
5      return end;
6
7    auto low = 0;
8    auto high = end - begin - 1;
9
10   while (low <= high) {
11     int mid = (low + high) / 2;
12
13     if (K < begin[mid])
14       high = mid - 1;
15     else if (begin[mid] < K)
16       low = mid + 1;
17     else
18       return begin + mid;
19   }
20
21   return begin + high + 1;
```

```
22 }
```

Listing 1.4: Lower bound

1.3 Max element

The last algorithm we will review in this section is how to search for the max element in an array. It is useful on its own right and also as part of other algorithms, like for example in straight selection and straight insertion sort, that will be treated in the next section. The implementation is quite simple (see figure 1.6), it traverses the array from begin to end keeping track of the maximum element encountered so far, a `C++` implementation of this procedure comes in code fragment 1.5.

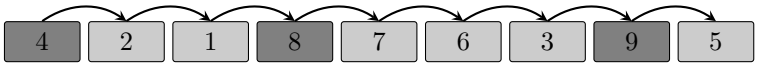


Figure 1.6: The darker boxes represent the maximum value encountered so far.

```
1  template <class Iter>
2  constexpr auto max_element(Iter begin, Iter end)
3  {
4    if (begin == end) return end;
5
6    auto max = begin;
7    while (++begin != end)
8      if (*max < *begin)
9        max = begin;
10
11   return max;
12 }
```

Listing 1.5: Max element

The first important thing to notice about this algorithm is that it must always test all elements to make sure the maximum has been found. That means its asymptotic behaviour must be $O(n)$ in both the best and worst case scenarios, this contrasts with the linear search for example where the best case is $O(1)$.

1.3.1 Left-to-right maxima

The runtime behaviour of this algorithm is related to the number of times the condition `*max < *begin` evaluates to true resulting in the assignment in the next line. Once this quantity is known all other properties can be expressed in terms of it, this can be seem more clearly in figure 1.7.

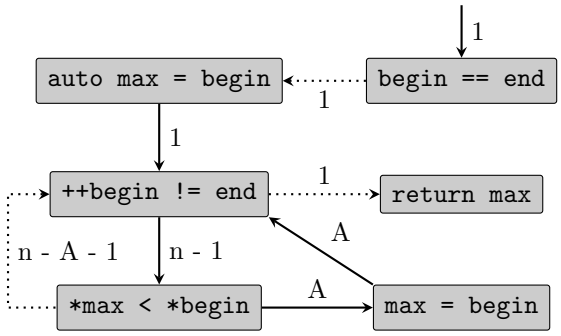


Figure 1.7: Flow chart for `max_element` expressed in terms of the number of elements n in the input array and its number of left-to-right maxima A.

To analyse it we will assume that the array elements belong to the set $\{1, 2, 3, ..., n\}$ since all that matters here are their sizes relative to each other and not their specific values.

We have seen with the analysis of the linear search that if the probability p_k of performing k assignments is the same for all k, then the average number of assignments $\overline{k}$ and the standard deviation σ would be given by equations (1.4) and (1.7) respectively. This assumption is however not a good one here, even intuitively we expect more and more assignments to occur before the maximum value is found as the array grows. It is more realistic to consider all permutation of the n elements to be equally likely.

1.3.2 Signless Stirling numbers

To determine p_k we need the numbers $c(n, k)$ that count the number of permutations of n elements that require k assignments. We see for example from figure 1.8 that $c(3, 1) = 2$, $c(3, 2) = 3$ etc, and $c(4, 1) = 6$, $c(4, 2) = 11$, etc. A closed form of these numbers is not obvious at a first glance, a successful approach to determine them is to find the recurrence relation they obey.

We see that whatever the permutation is, say $x_1, x_2, ..., x_n$, if its last element x_n is equal to n than this permutation has exactly one more assign-

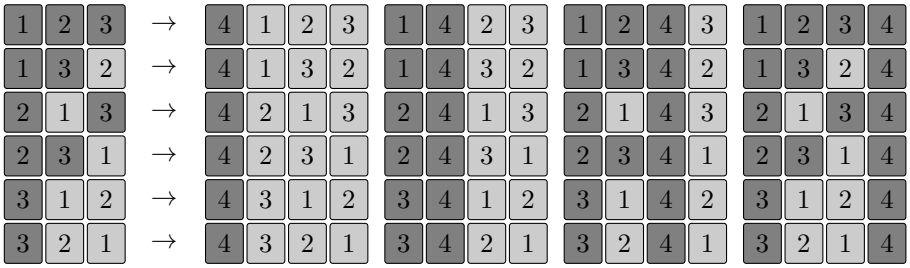


Figure 1.8: List of all permutations with size 3 (left) and 4 (right) respectively, with dark boxes representing the elements that require assignment when processed by algorithm 1.5. The permutations on the right have been obtained from that on the left by inserting a 4 into all possible positions.

ment than $x_1, x_2, ..., x_{n-1}$, contributing $c(n-1, k-1)$ permutations to the final number (if $x_n = n$ it will certainly trigger an assignment increasing k by one). On the other hand, if x_n is not n it will certainly not trigger an assignment, since this can occur in $n-1$ ways (x_n can be $1, 2, ..., n-1$), the number of permutations will be $n-1$ times the number of permutations that occurred in $x_1, x_2, ..., x_{n-1}$, namely $(n-1)c(n-1, k)$. Since we have exhausted the possible values of x_n we only have to sum up both contributions to get the final number

$$c(n, k) = c(n-1, k-1) + (n-1)c(n-1, k), \tag{1.31}$$

with the initial condition $c(k, 0) = \delta_{k0}$, where δ_{ij} is the *Kronecker Delta* symbol whose value is 1 if $i = j$ and zero otherwise. The numbers $c(n, k)$ are known as *signless Stirling numbers* and we will meet them again in this book in the study of cycles in permutations. We still do not have a closed formula for $c(n, k)$, but we actually do not need one if we manage to determine the generating function. If we multiply equation (1.31) by z^k and sum over k we get

$$g_n(z) = z \sum_{k=1}^{n} c(n-1, k-1) z^{k-1} + (n-1) \sum_{k=1}^{n} c(n-1, k) z^k. \tag{1.32}$$

By using the initial conditions for the numbers $c(n, k)$ and renaming indices in the summation it follows that

$$g_n(z) = (z + n - 1) g_{n-1}(z). \tag{1.33}$$

Expanding the recurrence relation above we see that the final form of the generating function for the $c(n,k)$ is given by

$$g_n(z) = z(z+1)(z+2)...(z+n-1). \tag{1.34}$$

Now, from the fact that the probability of having k assignments is given by

$$p_k = \frac{c(n,k)}{n!}, \tag{1.35}$$

we can divide both sides of equation (1.33) by $n!$ to obtain the generating function

$$G_n(z) = \frac{(z+n-1)}{n} G_{n-1}(z). \tag{1.36}$$

With this equation it is already possible to calculate the average number of comparisons $\overline{k}$ and σ. We see that

$$G'_n(z) = \frac{G_{n-1}(z)}{n} + \frac{(z+n-1)}{n} G'_{n-1}(z). \tag{1.37}$$

For $z = 1$ it takes the simple form

$$G'_n(1) = \frac{1}{n} + \frac{G'_{n-1}(1)}{n}. \tag{1.38}$$

Expanding the recurrence relation and inserting its final value in equation (1.10) results in

$$\overline{k} = H_n = 1 + \frac{1}{2} + ... + \frac{1}{n}. \tag{1.39}$$

That means, the average number of assignments in this algorithm is given by the harmonic number H_n. To calculate the standard deviation we need $G''(1)$. From equation (1.37) it follows that

$$G''_n(1) = \frac{2H_n}{n} + G_{n-1}(z), \tag{1.40}$$

from which we get

$$G''_n(1) \quad = \quad 2\sum_{k=1}^{n} \frac{H_k}{k} = H_n^2 + H_n^{(2)}, \tag{1.41}$$

(see exercise (1.6)) where $H_n^{(r)}$ is a generalization of the harmonic series defined by

$$H_n^{(r)} = 1 + \frac{1}{2^r} + \frac{1}{3^r} + ... + \frac{1}{n^r}. \tag{1.42}$$

When $r > 1$ the summation above has a finite value and when $n = \infty$ it is known as the *Riemann Zeta function*. We have now all pieces to calculate the standard deviation. It follows from equation (1.11) that

$$\sigma = \sqrt{H_n + H_n^{(2)}}. \tag{1.43}$$

Figure (1.9) shows some values of $\overline{k}$ and σ for some values of n.

Min	Max	Mean	Variance
n	n	H_n	$H_n + H_n^{(2)}$

Table 1.4: Summary of the number of assignments for the max element.

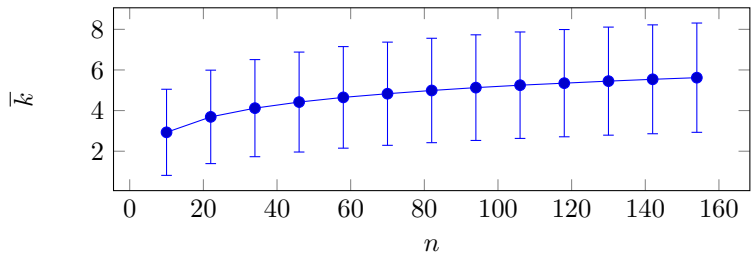


Figure 1.9: Average and standard deviation (represented by error bars) for the maximum element algorithm.

1.3.3 Products of generating functions.

It took us some laborious work to derive $\overline{k}$ and σ from the generating function (1.36), we will use the rest of this section to show that there is simpler way to do this by using some properties of the generating function. Suppose that $g(z)$ and $h(z)$ are two generating functions and let their product be

$$f(z) = g(z)h(z). \tag{1.44}$$

Deriving this equation and taking its value for $z = 1$, results in the important relation

$$f'(1) = g'(1) + h'(1). \tag{1.45}$$

In other words, to calculate the average of a generating function that can be expressed as a product of generating functions we only have to sum the individual averages, namely

$$\overline{k}_f = \overline{k}_g + \overline{k}_h. \tag{1.46}$$

A similar relation exists for the standard deviation σ. If we derive (1.44) twice we get

$$f''(1) = g''(1) + 2g'(1)h'(1) + h''(1). \tag{1.47}$$

using this equation into (1.11) produces

$$\sigma_f^2 \;\; = \;\; \sigma_g^2 + \sigma_h^2. \tag{1.48}$$

Now, back to equation (1.36) we see it can be expressed as a product of generating functions that have the following form $g_n(z) = (z+n-1)/n$ with $g'(1) = 1/n$ and $g''(1) = 0$. It follows from equations (1.46) that

$$\overline{k} = \sum_{k=1}^{n} \frac{1}{n} = H_n, \tag{1.49}$$

agreeing with equation (1.39). Similarly, for the standard deviation we have

$$\sigma^2 = \sum_{k=1}^{n} \left(\frac{1}{n} - \frac{1}{n^2} \right) = H_n - H_n^{(2)}, \tag{1.50}$$

agreeing with equation (1.43).

Exercises

1.1. Show how to obtain equation (1.3).

1.2. Show that

$$\sigma^2 = G''(1) + G'(1) - G'(1)^2.$$

1.3. Show that

$$z + z^2 + z^3 + ... + z^n = \frac{z^{n+1} - z}{z - 1}. \tag{1.51}$$

1.4. Convince yourself with a geometrical argument why the following important relation holds

$$\sum_{i=1}^{n} \sum_{j=1}^{i} a_{ij} = \sum_{j=1}^{n} \sum_{i=j}^{n} a_{ij}.$$

1.5. Use exercise (1.4) to show the important relation

$$\sum_{i=0}^{n} \sum_{j=0}^{i} a_i a_j = \frac{1}{2} \left(\left(\sum_{i=0}^{n} a_i \right)^2 + \left(\sum_{i=0}^{n} a_i^2 \right) \right).$$

1.6. Use the result obtained in exercise 1.5 to show equation (1.41).

1.7. Show that the generating function (1.5) can be compactly expressed as

$$G(z) = \frac{1}{z+n} \binom{z+n}{n}.$$

1.8. (*Upper bound*). Modify algorithm 1.4 so that it calculates the upper and not the lower bound. This algorithm is also part of the STL under the name `std::upper_bound`

Interview Problems

1.9. (*Reverse*). Write an algorithm to reverse the elements in an array.

1.10. Elaborate an algorithm that performs a binary search on an array that has bee rotated. For example

10 9 8 7 6 5 1 2 3 4

1.11. Implement a binary search recursively.

Solutions

1.1.

$$\begin{aligned}\sigma^2 &= \sum_{k=1}^{N}(k-\overline{k})^2 p_k = \sum_{k=1}^{N}(k^2 - 2k\overline{k} + \overline{k}^2)p_k \\ &= \sum_{k=1}^{N} k^2 p_k - 2\overline{k}\sum_{k=1}^{N} k p_k + \overline{k}^2 \sum_{k=1}^{N} p_k = \sum_{k=1}^{N} k^2 p_k - \overline{k}^2.\end{aligned}$$

1.4. We can write the summation in the following form

$$\begin{aligned}&a_{11} + \\ &a_{21} + a_{22} + \\ &a_{31} + a_{32} + a_{33} + \\ &a_{41} + a_{42} + a_{43} + a_{44} \\ &\ldots,\end{aligned}$$

and notice that summing over rows or columns should produce the same result.

1.7. It follows from a simple expansion of the recursive relation

$$\begin{aligned}G(z) &= \frac{(z+n-1)(z+n-2)...z}{n(n-1)...1} = \frac{(z+n-1)!}{n!z!} \\ &= \frac{1}{z+n}\binom{z+n}{n}.\end{aligned}$$

1.8.

1.9. A solution that works for random access iterators.

```
template<class Iter>
void reverse(Iter begin, Iter end) noexcept
{
  auto i = 0;
  auto j = end - begin;
  while (i < --j)
    std::swap(begin[i++], begin[j]);
}
```

1.10. One way to solve this is by noting that if we divide the array in two parts we get one part that is sorted and the other that is not. If the element in is the sorted range you can use normal binary search to search it, otherwise you eliminate that range a apply the same procedure in the unsorted range, for example

```
template <class Iter, class T>
bool binary_search_rotated(Iter begin, Iter end, const T& K)
{
  auto is_sorted = [&](auto a, auto b)
  { return !(begin[b - 1] < begin[a]); };

  auto in_range = [&](auto a, auto b)
  { return !(K < begin[a] || begin[b - 1] < K); };

  auto low = 0;
  auto high = end - begin;
  while (low < high) {
    auto mid = (low + high) / 2;
    if (is_sorted(mid, high)) {
      if (in_range(mid, high))
        return binary_search(begin + mid, begin + high, K);
      high = mid;
    } else {
      if (in_range(low, mid))
        return binary_search(begin + low, begin + mid, K);
      low = mid;
    }
  }
  return false;
}
```

1.11.

```
template<class Iter, class T>
auto bs(Iter begin, Iter end, T const& K)
{
  if (begin == end) return false;

  auto mid = (end - begin) / 2;
  if (K < begin[mid]) return bs(begin, begin + mid, K);
  else if (begin[mid] < K) return bs(begin + mid + 1, end, K);
  else return true;
}
```

Chapter 2

Sorting

The next class of algorithms that arises very frequently in programming is that of sorting a set of things according to some criterion. Programmers usually have little to do with the programming of sorting algorithms since general purpose implementations like quick sort perform very well on a wide range of scenarios, but just like the analysis of searching in the last section, the analysis of even the most simple sorting algorithms teaches us many lessons and techniques without which it is virtually impossible to understand more complex ones.

We will be concerned in this chapter with three main sorting strategies: *counting*, *inserting* and *selecting* and all algorithms that we will treat have a runtime complexity $O(n^2)$. Other sorting algorithms with better complexities and based of other strategies will appear in a future chapter dedicated to sorting.

2.1 Comparison counting

Comparison counting sort is a very simple sorting algorithms and our interest on it lies entirely on the mathematical problems that arise from its analysis and not on its performance. The main idea behind it is to calculate an address table composed of counters that inform the position where every element should be moved to in order to end up with a sorted array. To do so we compare every element to every other element in the array keeping track of how many they exceed overall. Code fragment 2.1 shows how this works.

```cpp
template <class Iter>
auto calc_address_table(Iter begin, Iter end)
{
  const auto n = end - begin;
  std::vector<int> count(n, 0);

  for (auto i = 1; i < n; ++i)
    for (auto j = 0; j < i; ++j)
      if (begin[i] < begin[j])
        ++count[j];
      else
        ++count[i];

  return count;
}
```

Listing 2.1: Address table

As a first step towards the analysis of this algorithm let us understand the information the address table conveys and for that let us use as input the following numbers

$$4\ 1\ 6\ 7\ 5\ 9\ 2\ 3\ 8.$$

By feeding algorithm 2.1 with them we obtain the following output

$$3\ 0\ 5\ 6\ 4\ 8\ 1\ 2\ 7.$$

The meaning of these numbers is clear, the first input number 4 shall be moved to position 3, 1 to 0, 6 to 5 and so on. This fact characterizes the address table as the *inverse* of a permutation as it informs where a given element should be moved to. A permutation on the other hand is usually interpreted as providing the information about which element shall replace the element at hand. It is perhaps easier to understand this by looking into figure 2.1 where arrows are used to indicate the position the elements should be moved to in order to obtain a sorted array. If we were to interpret the address table as a permutation and not the inverse thereof, the arrows would be pointing in the opposite direction.

With these clarifications it is easy to finish the sorting procedure, but first let us understand some important quantities related to algorithm 2.1.

2.1.1 Comparisons

The most important quantity related to the runtime complexity of comparison counting is the number of comparisons performed in `if (begin[i] <`

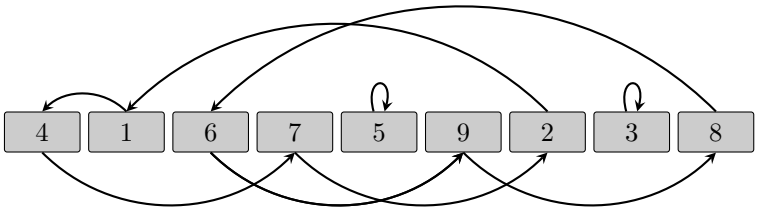


Figure 2.1: Address table calculated by algorithm 2.1 represented as arrows indicating the sorted position of the elements. Elements whose arrows point to themselves are called fixed points.

`begin[j])`, where elements are compared with each other. By noticing that the comparisons happen only once for every pair of elements we see that all we have to do is to calculate the total number of pairs we can form with the input numbers disregarding their order, that means (a, b) and (b, a) should be counted only once. This problem is more generally stated as: *In how many ways we can choose k among n?* and this is readily recognized as being given by the binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{2.1}$$

In our case we have therefore

$$\binom{n}{2} = \frac{n(n-1)}{2} = \frac{n^2 - n}{2}. \tag{2.2}$$

The equation above indicates a quadratic growth of the number of comparisons. It is important to notice that the number of comparisons is fixed and does not depend on the size of the input data, resulting in a time complexity that is equal for both the best and worst case with zero variance. Table 2.1 summarizes this fact.

Min	Max	Mean	Variance
$\frac{n^2-n}{2}$	$\frac{n^2-n}{2}$	$\frac{n^2-n}{2}$	0

Table 2.1: Summary of the number of comparisons performed by comparison counting.

2.1.2 Inversions

The next quantity we have to analyse is the number of times the test on the boolean condition in `if (begin[i] < begin[j])` evaluates to true. This happens for all pairs of elements satisfying $x_i < x_j$ when $i > j$, where x_i denotes an arbitrary array element. Such a pair of elements is said to be an *inversion* in the input array, for example, in the permutation 3142 we have three inversions, 31, 32 and 42.

Let us use $I_n(k)$ to denote the number of permutations of n elements that have k inversions and let $x_1x_2...x_n$ arbitrarily denote such a permutation. We see that if $x_n = 1$ then $x_1x_2...x_{n-1}$ must have $k-n+1$ inversions since all $n-1$ elements in front of it are bigger than itself. If on the other hand $x_n = 2$, then only $n-2$ elements will be bigger and from that it follows that $x_1x_2...x_{n-1}$ must have $k-n+2$ inversions. Similar considerations apply for all possible values of x_n. If we sum up all these contributions we obtain the recurrence relation satisfied by $I_n(k)$

$$\begin{aligned} I_n(k) &= I_{n-1}(k-n+1) + I_{n-1}(k-n+2) + ... + I_{n-1}(k) \\ &= \sum_{i=0}^{n-1} I_{n-1}(k-i). \end{aligned} \tag{2.3}$$

Multiplying both sides by z^k and summing over k results in the generating function of the number of inversions

$$g_n(z) = \sum_{k=0}^{M} \sum_{i=0}^{n-1} I_{n-1}(k-i) z^k, \tag{2.4}$$

where M is the maximum number of inversions (see exercise 2.10) and where we assumed that $I_n(k) = 0$ when $k < 0$. By reverting the summation order and making $k - i = j$ we obtain

$$g_n(z) = \sum_{i=0}^{n-1} \sum_{j=0}^{M} I_{n-1}(j) z^{j+i}. \tag{2.5}$$

Therefore, the generating function for $I_n(k)$ has the following form

$$g_n(z) = (1 + z + z^2 + ... + z^{n-1}) g_{n-1}(z). \tag{2.6}$$

Like in section 1.3 we will assume here that each of the $n!$ permutations of the input data is equally likely to occur resulting in a probability for each k with the following form $p_k = I_n(k)/n!$. Again, we do not need a close

expression for $I_n(k)$ to proceed, it is enough to divide equation (2.6) by $n!$ to obtain the generating function for the average number of comparisons

$$G_n(z) = h_1(z)h_2(z)...h_n(z), \tag{2.7}$$

where

$$h_n(z) = \frac{1+z+z^2+...+z^{n-1}}{n}. \tag{2.8}$$

Since in this case the generating function can be expressed as a product of functions and since these functions have the property $h_n(1) = 1$, we can use formula (1.46) to find the average number of inversions. We have $h_i'(1) = (i-1)/2$ and $h_i''(1) = (k^2-1)/12$. Summing up over all i results in the final values for the mean and standard deviation

$$\overline{k} = \sum_{i=2}^{n} \frac{i-1}{2} = \frac{n^2-n}{4}, \tag{2.9}$$

$$\sigma^2 = \sum_{k=1}^{n} \frac{k^2-1}{12} = \frac{n(n-1)(2n+5)}{72}, \tag{2.10}$$

where we have used $\overline{k}_1 = 0$ to obtain the equations above. A summary of the number of inversions can be seen on table 2.2, a graphical representation is found on figure 2.2.

Min	Max	Mean	Variance
0	$\frac{n^2-n}{2}$	$\frac{n^2-n}{4}$	$\frac{n(n-1)(2n+5)}{72}$

Table 2.2: Summary the statistics for the number of inversions in a permutation.

2.1.3 Unpermuting

Now that we have a fairly good understanding of the address table we can proceed to finalize the sorting procedure initiated when we built it. The simplest way to do so is by using some kind of temporary storage (see exercise 2.2), to avoid that however we can explore the cycle structure depicted by the arrows in figure 2.1. The key point to achieve that is to notice that by following the arrows we eventually come back to the starting point, for

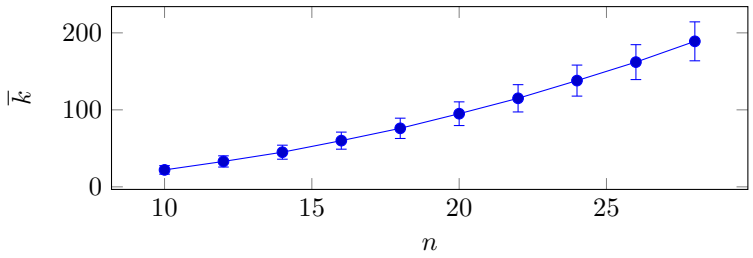


Figure 2.2: Average and standard deviation (represented by error bars) for the number of inversions in a permutation.

example, if we begin at the first element in that figure we have $4 \to 7 \to 2 \to 1$ and then 1 points back to 4 closing the cycle. In the literature such cycles are usually denoted in the following form (4721), the other cycles in that figure become therefore (698), (5) and (3) where the cycles with size one are called fixed points since they lead to nowhere. With these observations it is not difficult to elaborate an algorithm to *un*permute an array according the information contained in the address table, the code fragment below shows such an implementation.

```
1  template <class Iter>
2  void unpermute(Iter begin, Iter end, Iter table)
3  {
4    auto n = end - begin;
5    for (auto i = 0; i < n; ++i) {
6        while (table[i] != i) {
7            std::swap(begin[i], begin[table[i]]);
8            std::swap(table[i], table[table[i]]);
9        }
10   }
11 }
```

Listing 2.2: Unpermute

The last step to sort the input array using the `unpermute` is straightforward and has been left to exercise 2.3, here we will derive the runtime properties of the `unpermute` algorithm.

Even though this algorithm is short and quite simple aesthetically, the analysis of its runtime behaviour is not straightforward and requires some non-trivial math. The first step in that direction is related to the evaluation of the boolean condition `table[i] != i` that determines when the `std::swap`'s in the innermost loop will be executed. In particular we are interested in the elements in the array `table` that causes that condition to

evaluate to false, namely, the fixed points introduced some paragraphs above when we talked about the address table. The flow chart in figure 2.3 shows how all other properties of `unpermute` can be expressed in terms of the size of the input array and its number of fixed points.

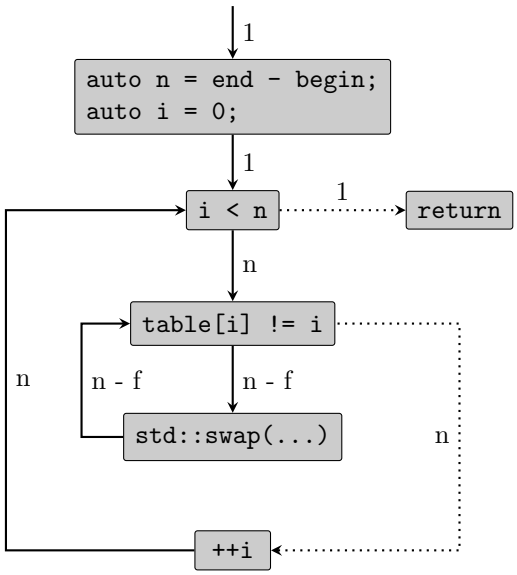


Figure 2.3: The execution flow for the unpermute algorithm expressed in terms of the number of fixed points f contained in the input array.

2.1.4 Fixed points

Let us denote the number of permutations of n elements that have k fixed points by $f(n,k)$, for example, in figure 2.4 we that $f(4,0) = 9$, $f(4,1) = 8$, $f(4,2) = 5$, $f(4,3) = 0$ and $f(4,0) = 1$.

To determine the general form $f(n,k)$ we can proceed with the following observations, there are $\binom{n}{k}$ ways we can choose k fixed points among the n elements in the permutation. Therefore, if we multiply this number by the number of ways we can arrange the remaining $n-k$ elements so that they do not contain any fixed points we get the following relation

$$f(n,k) = \binom{n}{k} f(n-k,0). \tag{2.11}$$

The second important observation concerns the normalization of $f(n,k)$ and is quite obvious: If we sum up the f's in equation (2.11) over all possible

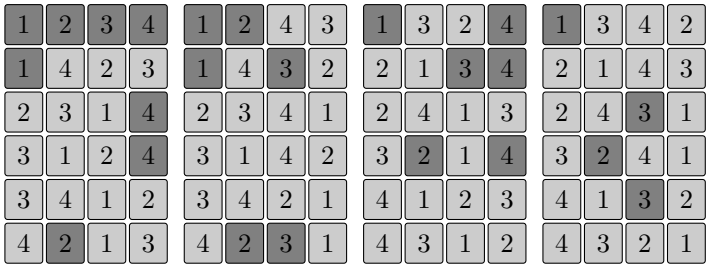


Figure 2.4: Illustration of all permutations with $n = 4$ elements with fixed points depicted in the darker boxes.

values of k it must result in $n!$ since the whole can be written as a sum of its parts, therefore

$$n! = \sum_{k=0}^{n} f(n,k) = \sum_{k=0}^{n} \binom{n}{k} f(n-k,0). \tag{2.12}$$

The last summation above can be simplified using the fact that the binomial coefficients satisfies the relation $\binom{n}{k} = \binom{n}{n-k}$, therefore

$$n! = \sum_{k=0}^{n} \binom{n}{k} f(k,0). \tag{2.13}$$

If we manage now to remove the summation on the right-hand side we can insert the resulting formula for $f(n,0)$ in equation (2.11) to obtain the final form of $f(n,k)$. This can be achieved by multiplying both sides of equation (2.14) by $\binom{m}{n}(-1)^{m-n}$ and sum over n, obtaining thus

$$\begin{aligned}
\sum_{n=0}^{m} n! \binom{m}{n} (-1)^{m-n} &= \sum_{n=0}^{m}\sum_{k=0}^{n} \binom{n}{k}\binom{m}{n} (-1)^{m-n} f(n,0) \\
&= \sum_{k=0}^{m} f(k,0) \sum_{n=k}^{m} \binom{n}{k}\binom{m}{n} (-1)^{m-n} \\
&= \sum_{k=0}^{m} f(k,0)\delta_{km} = f(m,0),
\end{aligned} \tag{2.14}$$

where we have used exercise 1.4 to simplify the summation in the second equality and 2.9 to obtain the third equality from the second. Rearranging this formula slightly we get

$$\frac{f(n,0)}{n!} = 1 - \frac{1}{1!} + \frac{1}{2!} - \ldots + \frac{(-1)^n}{n!}. \tag{2.15}$$

When n tends to infinity the right-hand side of this equation approaches e^{-1} very rapidly so that the $f(n,0) \approx n!/e$ becomes a good approximation. The final form of the f's can be now obtained by inserting (2.15) into (2.11).

$$f(n,k) \quad = \quad \frac{n!}{k!}\left(1 - \frac{1}{1!} + \frac{1}{2!} - ... + \frac{(-1)^{n-k}}{(n-k)!}\right). \tag{2.16}$$

If we assume once again that all $n!$ permutations of the input array are equally likely, the probability that such a permutation has k fixed points is given by $p_k = f(n,k)/n!$ and with this it is not difficult to see that the generating function $G_n(z) = p_0 + p_1 z + p_2 z^2 + ...$ associated with p_k can be written as

$$G_n(z) = \sum_{i=0}^{n} \frac{(-1)^{n-i}}{(n-i)!} \sum_{k=0}^{i} \frac{z^k}{k!}, \tag{2.17}$$

where exercise 1.4 has been used to invert the summations. Equation (2.17) can be written in a more convenient way by noting that the first summation in that equation seems to be related to the expansion of the constant e as observed in the paragraph after equation (2.15). The second summation on the other hand is clearly the Taylor expansion of e^z around $z = 0$ and truncated on the i'th term. A function that suggests itself is $e^{-1}e^z = e^{z-1}$, exercise 2.15 shows that by expanding it in powers of z up to the n'th term results in 2.17. With these observations we can write $G_n(z)$ in the somewhat simpler form

$$G_n(z) = \sum_{k=0}^{n} \frac{(z-1)^k}{k!}. \tag{2.18}$$

We see more clearly now that the generating function satisfies the following relation $G_n^{'}(z) = G_{n-1}(z)$ and from that the average and variance associated with with p_k can be easily calculated. Table 2.3 summarizes the results.

Min	Max	Mean	Variance
0	n	1	1

Table 2.3: Summary of of fixed points statistics.

2.1.5 Cycles

We have seen so far in this section that some properties of permutations like inversions, cycles and fixed points were very important into deriving the runtime behaviour of comparison counting sort. To make our study complete we will dedicate the remainder of this section to establish a connection between the number of cycles in a permutation with the number of left-to-right maximum that appeared in section 1.3.

We want to answer the following question: What is number of permutations with size n that have k cycles? Like in section 1.3 let us denote these numbers by $c(n, k)$ and stablish the recursive relation they satisfy. If we have one permutation with size $n-1$ and want to create a new one with size n we have two options

- Insert n in the n'th position making it a new fixed point. This will result in an increase of the number of cycles by one and incur no change in any of the $c(n-1, k-1)$ previous cycles.
- Insert n into one of the $c(n-1, k)$ cycles available and this can made in $n-1$ ways. This can be more easily seem seems in figure 2.5 where insert operation corresponds to replacing one arrow with a new node and two arrows.

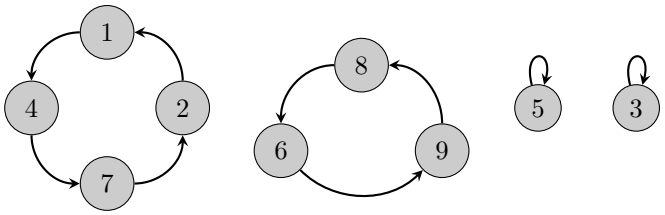


Figure 2.5: Individual representations of the cycles in figure 2.1.

By summing up both contribution we obtain the final form of the recursive relation obeyed by $c(k, n)$, we have

$$c(n, k) = c(n-1, k-1) + (n-1)c(n-1, k). \tag{2.19}$$

This equation is not new to us as it appeared in our study of left-to-right maxima in section 1.3, therefore all results regarding the average and variance obtained in that section are valid here too.

2.2 Distribution counting

An important variation of the comparison counting sort algorithm is possible when sorting an array of integers whose values are bounded to a specific range, say $A \leq x_i \leq B$ and the span range $B - A$ is reasonably small. In that case we do not have to compare elements to determine how many they exceed overall, we can use a table with size $B - A + 1$ and count how many elements there are with a given size by the element value as a table index, say `table[array[i] - A]`. Write a program that uses this idea to sort an array.

```
template <class Iter>
void dist_counting_sort( Iter begin, Iter end
                       , int min, int max)
{
  auto n = end - begin;
  auto m = max - min + 1;

  std::vector<int> count(m, 0);
  for (auto i = 0; i < n; ++i)
    ++count[begin[i] - min];

  for (auto i = 1; i < m; ++i)
    count[i] += count[i - 1];

  std::vector<int> out(n, 0);
  for (auto i = 0; i < n; ++i)
    out[--count[begin[i] - min]] = begin[i];

  std::copy(std::begin(out), std::end(out), begin);
}
```

2.3 Straight insertion

The sorting algorithm we are about to see is based on the idea of insertion and differently from the algorithm analysed in the previous section its performance is quite good for small array sizes, to the point that it can be safely considered in production code. It works by processing the array sequentially, say from left to right, keeping the elements that have already been processed in sorted order. As we advance with the unsorted elements we perform a search in the sorted range to find out where they should be inserted. Since the elements in an array lie on neighbouring memory ad-

dresses, after finding the correct insert position we have to make room for it by rotating those elements that are greater than itself to the next position. The procedure described above is illustrated on figure 2.6. The implementation provided in code fragment 2.3 merges the search and the rotation into one single operation.

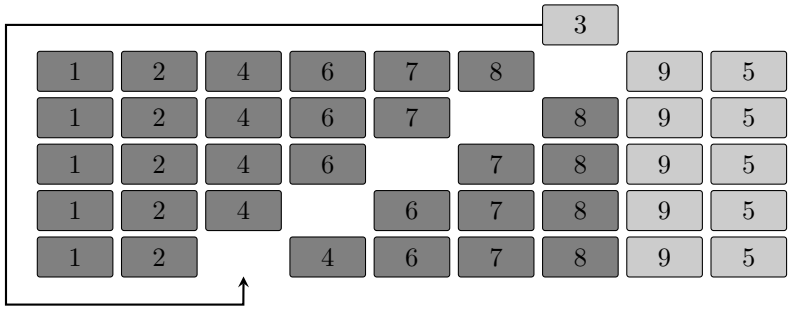


Figure 2.6: Illustration of the search and rotate operation for straight insertion sort.

```
1  template <class Iter>
2  void straight_insertion(Iter begin, Iter end)
3  {
4    if (begin == end) return;
5
6    auto n = end - begin;
7    for (auto j = 1; j < n; ++j) {
8      auto i = j - 1;
9      auto k = begin[j];
10     while (k < begin[i]) {
11       begin[i + 1] = begin[i];
12       if (--i < 0) break;
13     }
14     begin[i + 1] = k;
15   }
16 }
```

Listing 2.3: Straight insertion sort

At first glance the analysis of this algorithm seems to be more complicated than the previous ones. It is not obvious for example how many times the condition on the innermost loop will be evaluated and more specifically what characteristics of the input array determine that. A careful look however reveals that all quantities needed to express its execution time are already known. For example, when we search (backwards) for the insert po-

sition of an element we go only as far as there are elements bigger than itself but since the elements in that range have already been sorted, this number is equal to the number of inversions associated with that element. Inversions have been carefully analysed in section 2.1 and table 2.2 summarizes the important results regarding them.

We also notice that the `break` statement inside the innermost loop can be hit only if the element being processed is less than all elements in the sorted range or in other words, if we have found a left-to-right minimum. In section 1.3 we analysed the runtime of the algorithm that searches for the maximum element in a range and by symmetry all results obtained there apply also for the minimum number of elements.

With these observations, we can calculate the number of times each `C++` statement is executed. We see for example from the execution flow of straight insertion shown in figure 2.7 that the number of times the boolean condition is evaluated in the innermost loop is given by

$$k = n - 1 + I - M, \tag{2.20}$$

where I is the number of inversions and M the number of left-to-right minimum elements. The average value of this quantity can be easily calculated with the formulas provided by table 2.2 and 1.4. Straight insertion has a fairly good performance for small values of n, figure 2.8 compares it with `std::sort`.

2.4 Binary insertion

We can slightly improve the performance of straight insertion by using the methods of the binary search studied in section 1.2. Such a change would result in a number of comparisons that grows as $O(log(n))$ (see exercise 2.1). Even though the net speed up would be non negligible, the number of rotations required to make room for each new element in the sorted range would still remain $O(n^2)$. In section **??** we will see that with a change in the data structure the need for a rotation of the elements can be completely removed (though at the expense of not being able to perform a binary search anymore).

2.5 Straight selection

Here we come to the last sorting algorithm we will review in this section, straight selection sort is based on a very simple idea that many people would

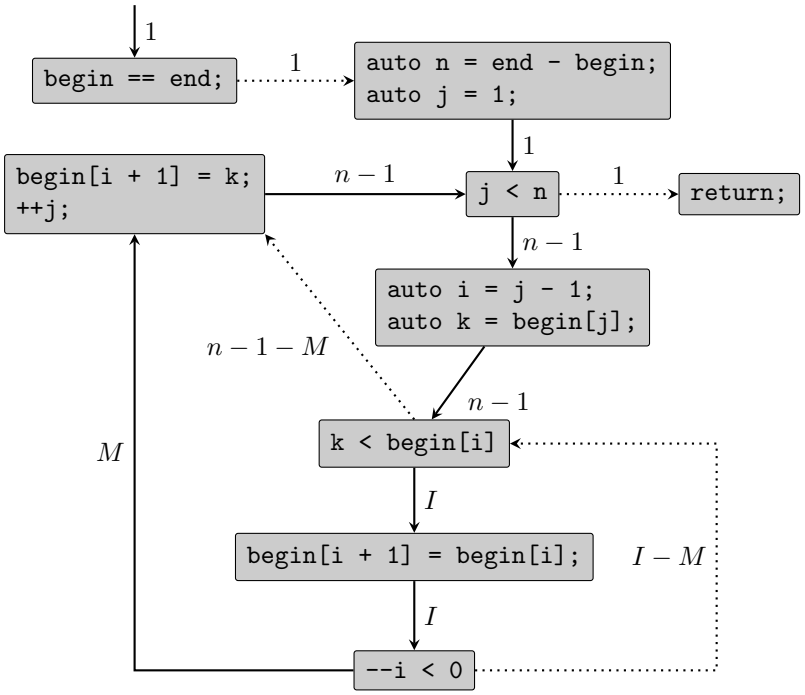


Figure 2.7: Execution flow for straight insertion sort for a non-empty array.

naturally come up with, select the minimum element in the input array and make it the first element in the output range, select the second minimum and do likewise, until all elements have been moved to the output range. Since we are working with arrays moving elements to the output range may lead to inconveniences as it would generate a gap in the input range that would have to be skipped in future searches. To work around that we could for example rotate all elements by one to fill the gap or mark an element as already selected to ignore it on future searches. Of course the procedure described above is much more complex than it has to be, we can for example perform a swap of the minimum element with the one occupying its sorted position, this procedure is illustrated in figure 2.9 and an implementation comes in code fragment 2.4.

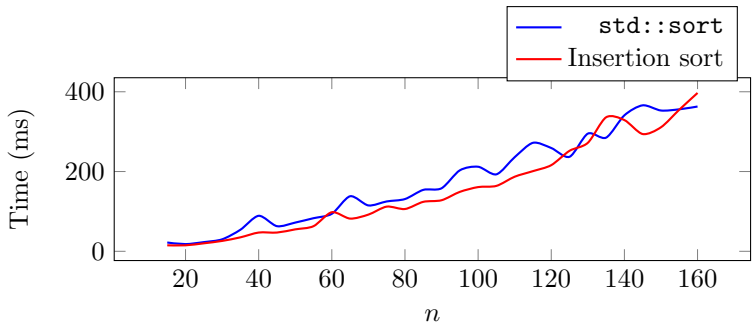


Figure 2.8: Performance of straight insertion compared to an `std::sort` that is implemented with quick sort.

```
1  template <class Iter>
2  void straight_selection(Iter begin, Iter end)
3  {
4    if (begin == end)
5      return;
6
7    for (; begin != std::prev(end); ++begin)
8      std::iter_swap( begin
9                    , min_element(begin, end));
10 }
```

Listing 2.4: Straight selection sort

The reader may have noticed that we can easily reformulate the procedure above to work with `max_element` instead of `min_element` and by symmetry we can employ here the results obtained in section 1.3.

2.5.1 Comparisons

The first result that we can use from that analysis regards the number of comparisons performed in straight selection. We have seen that `max_element` has to check all elements in its input range to be able to conclude that maximum element has been found, from that we conclude that all elements will be checked against each other and only once, therefore we can use here that same arguments here that we used when we computed the number of comparisons for *comparison counting* is section 2.1. If we denote by A the number of assignments we can write

$$A = \binom{n}{2} = \frac{n^2 - n}{2}. \tag{2.21}$$

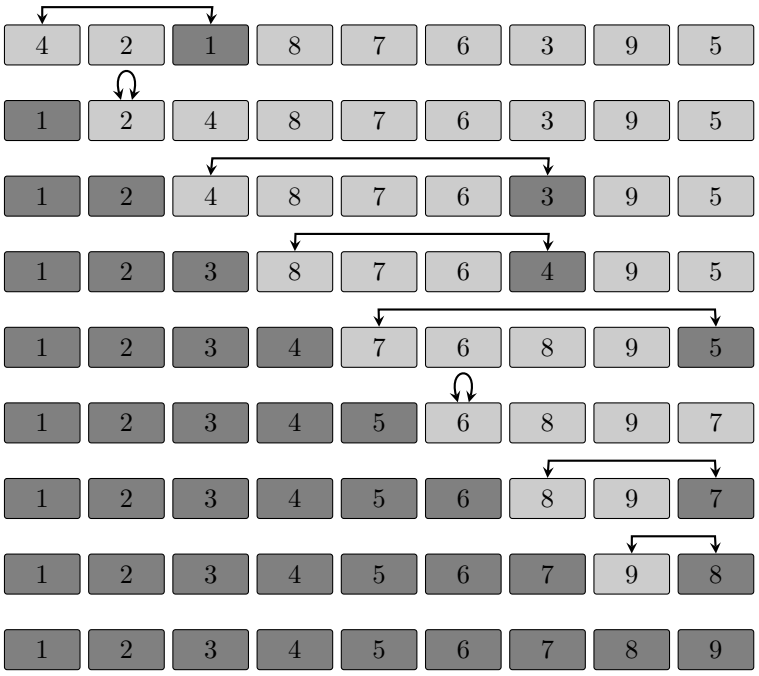


Figure 2.9: Illustration straight selection sort.

2.5.2 Assignments

The second result we can make use here regards the average number of assignments made inside each call to `min_element`, which was shown to be given by the harmonic number H_n. Since we have to perform a total of $n-1$ searches, we conclude that the total number of assignments B is given by

$$B = \sum_{i=2}^{n} H_n = \left(\sum_{i=1}^{n} \sum_{j=1}^{i} \frac{1}{j} \right) - 1 = \left(\sum_{j=1}^{n} \frac{1}{j} \sum_{i=j}^{n} 1 \right) - 1, \tag{2.22}$$

where the first equality follows from the definition of the harmonic number and the second from exercise 1.4. Expanding it further we obtain the final form for of B

$$B = (n+1)(H_n - 1), \tag{2.23}$$

where we have used the fact that after the swap with each minimum element the unsorted range retains its randomness, namely all $(n-1)!$ permutations of the input numbers remain equally likely. The leading dependency on

n grows as $O(n \log n)$ using approximation (1.16). We will not derive the variance of B here as that would takes far afield.

2.6 Bubble sort

```
1  template <class Iter>
2  void bubble_sort(Iter begin, Iter end)
3  {
4    if (begin == end) return;
5
6    auto B = end - begin - 1;
7    while (B != 0) {
8      auto t = 0;
9      for (auto j = 0; j < B; ++j) {
10       if (begin[j] > begin[j + 1]) {
11         std::swap(begin[j], begin[j + 1]);
12         t = j + 1;
13       }
14     }
15     B = t;
16   }
17 }
```

Exercises

2.1. (*Binary insertion*). Use exercise **??** to implement insertion sort as described in the end of section 2.3.

2.2. (*Comparison counting sort*). Write a program to finish the sorting procedure of section 2.1 using an auxiliary buffer.

2.3. (*Inplace comparison counting sort*). Finish the sorting procedure of section 2.1 using the `unpermute` algorithm shown in code fragment 2.2.

2.4. (*Upermute on the fly*). Sometimes it is desirable to unpermute an array with the inverse of a permutation that is generated on the fly. Adapt algorithm 2.2 for that case and assume (a) that you can use one bit in each array element to accomplish your task, (b) that you are not allowed to modify the elements (set or unset bits). Use your code to transpose a matrix (c) assuming that its elements are passed to the function in row-major-order (natural to the `c` language). In that case, if we denote by r the number of

rows and by c the number of columns, then an element that is located at index i in the array should be moved to index

$$i' = ri \bmod (rc-1), \quad \text{where} \quad 0 \le i < rc-1. \tag{2.24}$$

2.5. Explain the meaning of the fundamental relation satisfied by the binomial coefficients below.

$$\binom{r}{k} = \binom{r-1}{k} + \binom{r-1}{k-1}. \tag{2.25}$$

2.6. Show that by successive applications of equation (2.25) we can prove the following important relation

$$\sum_{k=0}^{n} \binom{r+k}{k} = \binom{r+n+1}{n}. \tag{2.26}$$

2.7. When r is not a positive integer we can define the binomial coefficient by the following formula

$$\binom{r}{k} = \frac{r(r-1)...(r-k+1)}{k(k-1)...(1)} = \prod_{j=1}^{n} \frac{r++-j}{j}. \tag{2.27}$$

Use this formula to show that

$$\binom{r}{k} = (-1)^k \binom{k-r-1}{k}. \tag{2.28}$$

2.8. Use the result obtained in exercise 2.7 to show that

$$\sum_{k \le n} \binom{r}{k} (-1)^k = (-1)^n \binom{r-1}{n}. \tag{2.29}$$

2.9. Use the result obtained in exercise 2.8 to show that

$$\sum_{k} \binom{r}{k} \binom{k}{n} (-1)^{r-k} = \delta_{rn}. \tag{2.30}$$

2.10. Determine the value of M in equation 2.4.

2.11. (*God's party for dogs*). If this book ever gets published this exercise will be removed, it is an old joke my father used to tell us: *Do you know why the dogs go smell each other's asses when they meet on the street?* No ... *It is because in the creation of the world God made a party where all Dogs where required to leave their asses in the entrance hall to be allowed to enter the party. But when the party terminated they were too drank to be able to find their asses back and got whatever they found. As of today they are still looking for their correct asses.* What is the probability that no dog was able to find his own ass?

2.12. (*Permute*) Write an algorithm to *permute* an array according to a given permutation. This algorithm should have the exact opposite effect of algorithm the *unpermute* algorithm presented in the text.

2.13. What address table should we use in algorithm 2.2 to unpermute the word *logarithm* into algorithm? What would it be if would process it with the *permute* algorithm of exercises 2.12?

2.14. (*Inverse of a permutation*) So far in the text and exercises we have seen how to *permute* or *unpermute* an array according to a permutation or inverse thereof. The following algorithm can be used to calculate the inverse of a permutation, try to understand how it works.

```
1  template <class Iter>
2  void inplace_inverse_perm(Iter begin, Iter end, bool
       begin_at_zero)
3  {
4    auto n = end - begin;
5    auto m = n;
6    auto j = -1;
7
8    auto k = begin_at_zero ? 1 : 0;
9    do {
10     auto i = begin[m - 1] + k;
11     if (i > -1) {
12       do {
13         begin[m - 1] = j - k;
14         j = -m;
15         m = i;
16         i = begin[m - 1] + k;
17       } while (i > 0);
18       i = j;
19     }
20     begin[m - 1] = -i - k;
```

```
21      --m;
22    } while (m > 0);
23  }
```

2.15. Show that the expansion of e^{z-1} up to the n'th term leads to equation 2.17.

Interview Problems

2.16. (*Rotate image*). Write a program that rotates an image by 90 degrees counter and clockwise. (*Hint*: use the `unpermute` algorithm.)

Solutions

2.2.

```
1  template <class Iter>
2  void comparison_counting_sort(Iter begin, Iter end)
3  {
4    const auto n = end - begin;
5    auto count = calc_address table(begin, end);
6
7    std::vector<int> tmp(n, 0);
8    for (auto i = 0; i < n; ++i)
9      tmp[count[i]] = begin[i];
10
11   std::copy(std::begin(tmp), std::end(tmp), begin);
12 }
```

2.3.

```
1  template <class Iter>
2  void inplace_comparison_counting_sort(Iter begin, Iter end)
3  {
4    auto table = calc_address_table(begin, end);
5    unpermute(begin, end, std::begin(table));
6  }
```

2.4.a.

```
template <class Iter>
void unpermute_on_the_fly_bit( Iter begin, Iter end, auto index
                             , auto set_bit, auto unset_bit
                             , auto is_set)
{
  const auto n = end - begin;
  for (auto i = 0; i < n; ++i) {
    auto k = index(i);
    if (!is_set(begin[i])) {
      while (k != i) {
        std::swap(begin[i], begin[k]);
        set_bit(begin[k]);
        k = index(k);
      }
    }
    unset_bit(begin[i]);
  }
}
```

If the input array has an integer data type we can set/unset/check with the following mask and bitwise operations

```
constexpr auto mask = 1 << 30;
const auto set_bit = [=](auto& o) {o |= mask;};
const auto unset_bit = [=](auto& o) {o &= ~mask;};
const auto is_set = [=](auto& o) {return o & mask;};
```

2.4.b.

```
template <class Iter, class Index>
void unpermute_on_the_fly( Iter begin, Iter end
                         , Index index)
{
  const auto n = end - begin;
  for (auto i = 0; i < n; ++i) {
    auto k = index(i);
    while (k > i)
      k = index(k);
    if (k == i) {
      k = index(i);
      while (k != i) {
        std::swap(begin[i], begin[k]);
        k = index(k);
      }
    }
  }
```

```
}
```

2.5. The term $\binom{r-1}{k}$ corresponds to all combination of k elements among r that do not contain the r'th element, so that we are missing only those combinations the contain the r'th element. To construct them we can choose $k-1$ elements among the first $r-1$ elements and put the r'th in each one of them.

2.10. The maximum number of inversions happens when all possible pairs of elements in the array are inversions and this number is given by $\binom{n}{2}$.

2.12.

```
template <class Iter>
void permute(Iter begin, Iter end, Iter permutation)
{
  auto n = end - begin;
  for (auto i = 0; i < n; ++i) {
    if (permutation[i] != i) {
      auto t = begin[i];
      auto j = i;
      do {
        auto k = permutation[j];
        begin[j] = begin[k];
        permutation[j] = j;
        j = k;
      } while (permutation[j] != i);
      begin[j] = t;
      permutation[j] = j;
    }
  }
}
```

Listing 2.5: Permute and array

Chapter 3

Combinatorial algorithms

In the analysis of the algorithms covered so far we had to answer many questions regarding the *counting of the number of ways a certain pattern is present in some input data*, a branch of mathematics called *enumerative combinatorics*. Some examples were

- Permutations: How many ways there are to arrange n numbers without repetition?
- Combinations: In how many ways I can choose k elements from a set with size n?
- How many permutations of n elements have exactly k cycles, left-to-right minima, fixed points and inversions?

In this section we will be concerned with how can we list the diffrent patterns. Most of this section is based of the book *The Art of Computer Programming.* Vol. 4.

3.1 Generating all tuples

Let us begin with the quite interesting problem, how can we list all passwords with length k from an alphabet with size n were the repetition of digits is allowed, for example 34831 should be considered a valid passwork. Since there are n choices for each of the k entries it is easy to see that the total number of passwords is given by n^k.

The following algorithm calculates the next password in lexicografical order from a given input password and allows the user to provide a range of

values for each dividual entry. The first element in the range is used only for convenience and is not part of the tuple.

```
template <class Iter>
auto next_tuple( Iter begin, Iter end
                , Iter min, Iter max)
{
    auto j = end - begin - 1;
    while (begin[j] == max[j]) {
      begin[j] = min[j];
      --j;
    }
    ++begin[j];
    return j != 0;
}
```

Listing 3.1: Tuples

It is easy now to iterate over all passible passwords. In code fragment 3.2 the first digit is constrained to the range $[1, 3]$, the second to $[5, 7]$ and so on. The `visit` function is meant to perform whatever operation we interested in.

```
void iterative()
{
  std::vector<int> min {0, 1, 5, 1};
  std::vector<int> max {2, 3, 7, 3};
  auto arr = min;

  do {
    visit(arr);
  } while (next_tuple( std::begin(arr), std::end(arr)
                     , std::begin(min), std::begin(max)));
}
```

Listing 3.2: Iteration over tuples

3.1.1 Binary tuples

An easy way to generate tuples whose digits assume only two values is by making use of the binary representation of an integer inside a computer, where we associate the values according to a bit that is either true or false. To produce all the tuples it is enough to increasingly add one to a variable beginning with 0. By the time we reach the k'th bit all possible bit patterns of the 2^k tuples will have been generated and therefore all tuples as well.

This technique is specially usefull to generate all subsets of a string where we include each element in the set if its corresponding bit is set or not. An

example is shown in code fragment 3.3.

```
1 void binary_tuples()
2 {
3     std::string str {"zrkg"};
4
5     auto v = (1 << str.size()) - 1;
6     do {
7       visit_int(v, str);
8     } while (v-- != 0);
9 }
```

Listing 3.3: String subsets

3.2 Generating all permutations

Suppose now that instead of generating all passwords like in section 3.1, we want to generate all $n!$ permutations associated with a given password of length n. For a password like abc we want therefore

$$abc, bac, bca, acb, cab, cba.$$

If we were to write them in a piece of paper, we could begin with one element and insert subsequent elements in all possible ways until we have inserted them all. For example for the string abc we begin with c and insert b and a one after the other in all possible ways we get

c
bc, cb
abc, bac, bca, acb, cab, cba.

The interative approax presented here suitable for a computer implementation can be divide in three main steps

1) Search backwards in the array for the first element that is less than its successor (i.e. $a_i < a_{i+1}$).

2) Search backwards again for the first element that is greater than the element found in (1) swap them.

3) Reverse the range $[a_{i+1}, a_n]$.

```
template <class Iter>
bool next_permutation(Iter begin, Iter end)
{
    auto b = end;
    auto a = std::prev(b);

    while (!(*--a < *--b));

    if (a == begin) {
      rt::reverse(++a, end);
      return false;
    }

    b = end;
    while (!(*a < *--b));

    std::iter_swap(a, b);

    rt::reverse(++a, end);

    return true;
}
```

For example, let us apply this algorithm to the permutation 2543, we have

$$2543,\ \mathbf{25}43,\ \mathbf{2}54\mathbf{3},\ \mathbf{3}542,\ 3\mathbf{245},\ 3245.$$

The program that outputs the permutations reads

```
void main()
{
  std::vector<int> v {0, 1, 2, 3};
  do {
    visit(v);
  } while (rt::next_permutation(std::begin(v), std::end(v)));
}
```

3.3 Random permutations

Shuffling. An important question that arises when studying permutations is how to generate a random on one. The algorithm below randomize its input array given a random number generator that produces values uniformly distributed in the interval $[0, 1)$

```
template <class Rand>
void shuffle(std::vector<int>& v, Rand& rand)
{
  auto j = v.size();
  do {
    auto k = std::floor(j * rand());
    std::swap(v[k], v[--j]);
  } while (j != 0);
}
```

Random permutation. If all we need is a random permutation of the elements 1, 2, ..., n the following algorithm is recommended.

```
template <class Rand>
auto random_permutation(int n, Rand& rand)
{
  std::vector<int> ret(n);
  for (auto j = 0; j < n; ++j) {
    auto k = std::floor((j + 1) * rand());
    ret[j] = ret[k];
    ret[k] = j + 1;
  }

  return ret;
}
```

3.4 Generating all combinations

How difficult it is to win a lottery where one chooses 6 among 60 numbers? To answer this question we have to calculate how many outcomes are possible with this configuration.

To generate all combinations of k items out of n.

```cpp
bool next_combination(std::vector<int>& v)
{
  int t = v.size() - 2;
  auto i = 0;
  while ((v[i] + 1) == v[i + 1]) {
    v[i] = i;
    ++i;
  }
  if (i == t)
    return false;
  ++v[i];
  return true;
}
```

And to generate all combinations

```cpp
int main()
{
  int n = 5;
  std::vector<int> v {0, 1, 2, n, 0};

  do {
    visit(v);
  } while (next_combination(v));
}
```

3.5 Generating all compositions

3.6 Generating all partitions

The partitions of a number refers to the ways of representing those numbers as a sum os smaller non-negative numbers disregarding order. For example.

An algorithm to generate all partitions in reverse lexicographical order can be seem below.

```
auto kdelta(int a, int b) { return a == b ? 1 : 0; }

struct next_partition {
  int last, q;
  std::vector<int> a;

  next_partition(int n)
  : last(1), q(last - kdelta(n, 1)), a(n + 1, 1)
  {
    a[0] = 0;
    a[1] = n;
  }

  auto next()
  {
    if (a[q] == 2) {
      a[q--] = 1;
      ++last;
      return true;
    };

    if (q == 0) return false;

    int x = a[q] - 1;
    a[q] = x;
    auto n = last - q + 1;
    last = q + 1;

    while (x < n) {
      a[last++] = x;
      n -= x;
    }

    a[last] = n;
    q = last - kdelta(n, 1);
    return true;
  }
};
```

Listing 3.4: All partitions of n

To visit all partitions one can use for example

```
rt::next_partition part(6);
do {
  rt::visit_part(part.a, part.last);
} while (part.next());
```

Exercises

Interview Problems

3.1. (*Counting bits*). Write a function that counts the number of bits that are set in a variable of type `unsigned`.

3.2. (*Reversing bits*). Write a function that reverse the bits in an integer.

3.3. Write a function that adds two integers without using the arithmetic operators `+` and `-`.

3.4. Write a function that returns true if an integer is a power of two.

3.5. Write a function to align a pointer either on the previous or next aligned address.

3.6. (*Bit image*). You are given a `std::vector<T>` with size `rows x cols` where each bit in the array represents a pixel in an image. Assume that `T` is an unsigned integral type. Write a function that sets the bits inside a rectangular region defined by a begin/end row/column bit. Provide an answer (a) where bits are set individually and one (b) where you do it efficiently by setting many bits at once.

3.7. Elaborate an algorithm to shuffle a deck of cards recursively. Assume you have an appropriate random number generator.

3.8. A well-ordered password with length n is defined as a sequence of digits that obey the rule $a_i \leq a_{i+1} \leq a_{i+2} \leq ... \leq a_n$. Explain how can you generate all passwords with algorithms studied in this section.

3.9. Formulate an algorithm to generate all tuples with length n recursively.

3.10. Write an algorithm to generate all permutation of a string recursively.

3.11. In the Brazilian lottery it is also possible to choose 7, 8, etc numbers in the same card. What should be the price of these cards in comparison the card with six numbers?

3.12. Assume you have a sorted deck of cards and you shuffle it a couple of seconds. What is the probability that at least one person in mankind history has had that very same permutation. Assume your shuffles were perfect.

Solutions

3.1. Subtracting one from the value is equivalent to unsetting the higher order bit and and'ing it with itself recovers its bit pattern (with the exception of the higher order bit of course).

```
int count_bits(int v)
{
  int c = 0;
  for (; v; ++c)
    v &= v - 1;

    return c;
}
```

3.2. One efficient way to invert bits is to operate on neighbouring blocks in parallel, for example in a byte we would have

$$\begin{aligned} &x_0x_1x_2x_3x_4x_5x_6x_7 \\ &x_1x_0x_3x_2x_5x_4x_7x_6 \\ &x_3x_2x_1x_0x_7x_6x_5x_4 \\ &x_7x_6x_5x_4x_3x_2x_1x_0 \end{aligned} \tag{3.1}$$

All that we need to do this are suitable mask and bitwise operations. For example, to invert neighbouring bits one needs a pattern like 01010101, to invert pairs of bits we use 00110011 and for the last operation we use 00001111.

```
auto op(unsigned o, unsigned m, unsigned s)
{ return ((o >> s) & m) | (o & m) << s; }

auto reverse(unsigned o)
{
  o = op(o, 0x55555555,  1);
  o = op(o, 0x33333333,  2);
  o = op(o, 0x0F0F0F0F,  4);
  o = op(o, 0x00FF00FF,  8);
  o = op(o, 0x0000FFFF, 16);

  return o;
}
```

3.3. In this case we `xor` to add bits without carry and get the carries with an `and` operation and add them again until there is nothing more to be added.

```
auto add(unsigned a, unsigned b)
{
  while (b != 0) {
    auto sum = a ^ b
    b = (a & b) << 1;
    a = sum;
  }

  return a;
}
```

3.4.

```
constexpr auto is_power_of_two(std::size_t N) noexcept
{ return N > 0 && (N & (N - 1)) == 0; }
```

3.5.

```
constexpr
auto is_aligned(std::size_t a, std::size_t N) noexcept
{ return (a & (N - 1)) == 0; }

constexpr
auto align_previous(std::size_t a, std::size_t N) noexcept
{ return (a & ~(N - 1)); }

constexpr
auto align_next(std::size_t a, std::size_t N) noexcept
{ return align_previous(a, N) + N; }
```

3.6.a. This is not the most efficient solution but perhaps the simplest. Each bit is set individually. All that is needed to accomplish this task is some care with the boundaries and knowledge of bitwise operations.

```
template <class T>
void set_bit(T& o, int i) { o |= 1 << i; }

template <class T>
void set_bits_slow( std::vector<T>& img, int cols
                  , int row_bit_begin, int row_bit_end
                  , int col_bit_begin, int col_bit_end)
{
  constexpr int s = 8 * sizeof T {};

  for (auto i = row_bit_begin; i < row_bit_end; ++i)
```

```
12      for (auto j = col_bit_begin; j < col_bit_end; ++j)
13        set_bit(img[i * cols + j / s], j % s);
14  }
```

3.6.b. Instead of setting bits individually we can use one single instruction to set 8, 16, etc. at once.

```
1   template <class T>
2   void set_bits_fast( std::vector<T>& img, int cols
3                     , int row_bit_begin, int row_bit_end
4                     , int col_bit_begin, int col_bit_end)
5   {
6     constexpr int s = 8 * sizeof T {};
7
8     auto begin = col_bit_begin / s;
9     T begin_mask = T(-1) << col_bit_begin % s;
10
11    auto end = col_bit_end / s;
12    T end_mask = (1 << col_bit_end % s) - 1;
13
14    if (begin == end) {
15      for (auto i = row_bit_begin; i < row_bit_end; ++i)
16        img[i * cols + begin] |= begin_mask & end_mask;
17
18      return;
19    }
20
21    for (auto i = row_bit_begin; i < row_bit_end; ++i) {
22      img[cols * i + begin] |= begin_mask;
23
24      for (auto j = begin + 1; j < end; ++j)
25        img[i * cols + j] = T(-1);
26
27      if (end != cols)
28        img[i * cols + end] |= end_mask;
29    }
30  }
```

3.7.a. We can solve this exercise with a small change in the recursive permutation algorithm of section 3.2. The idea is to add a new item in the front of the deque and swap it with another randomly chosen item.

```
1   template <class Rand>
2   void shuffle_recursive( std::stack<int> s, Rand& rand
3                         , std::deque<int> d = {})
4   {
```

```
5     if (s.empty()) {
6       visit(d);
7       return;
8     }
9
10    d.push_front(s.top());
11    s.pop();
12
13    auto i = rand() % d.size();
14    std::swap(d.front(), d[i]);
15    shuffle_recursive(s, rand, d);
16  }
```

3.7.b. Instead of constructing the permutations recursively, we can randomize it recursively. For example

```
1  template <class Rand>
2  void shuffle_recursive2( std::vector<int>& v, int i
3                         , Rand& rand)
4  {
5    if (i == 0)
6      return;
7
8    shuffle_recursive2(v, i - 1, rand);
9
10   auto j = rand() % (i + 1);
11   std::swap(v[i], v[j]);
12   visit(v);
13 }
```

3.9. A possible solution can be seem below.

```
1  template <class T, std::size_t N>
2  void recursive(std::array<T, N> arr, int idx)
3  {
4    if (idx == arr.size()) {
5      visit(arr);
6      return;
7    }
8
9    for (auto i = min[idx]; i < max[idx]; ++i) {
10     arr[idx] = i;
11     recursive(arr, idx + 1);
12   }
13 }
```

3.10. The idea of the following algorithm is to begin with the elements in a stack and on each recursive call we pop one elements of the stack until it gets empty.

```
void recursive( std::stack<int> s
              , std::deque<int> d = {})
{
  if (s.empty()) {
    visit(d);
    return;
  }
  d.push_front(s.top());
  s.pop();

  unsigned i = 0, j = 1;
  for (;;) {
    recursive(s, d);
    if (j == d.size())
      break;

    std::swap(d[i++], d[j++]);
  }
}
```

This function can be called in the following way

```
int main()
{
  std::stack<int> s {{1, 2, 3}};
  recursive(s);
}
```

Part II

Algorithms and Data Structures

Chapter 1

Linked lists

The STL provides data structures and algorithms. It introduces the concept of iterators, which we will see in the next section.

It is also useful to define an iterator type that can be used with STL algorithms. A minimal example of such iterator can be seems below.

```
1  template <class Successor>
2  class bst_iter {
3    private:
4    Successor s;
5    public:
6    using value_type = int;
7    using pointer = bst_node*;
8    using reference = bst_node&;
9    using difference_type = std::ptrdiff_t;
10   using iterator_category = std::forward_iterator_tag;
11   bst_iter(bst_node* root = nullptr) noexcept
12   : s(root) {}
13   auto& operator++() noexcept { s.next(); return *this; }
14   auto operator++(int) noexcept
15   { auto tmp(*this); operator++(); return tmp; }
16
17   const auto& operator*() const noexcept {return s.p->info;}
18   friend auto operator==( const bst_iter& rhs
19                         , const bst_iter& lhs) noexcept
20   { return lhs.s.p == rhs.s.p; }
21   friend auto operator!=( const bst_iter& rhs
22                         , const bst_iter& lhs) noexcept
23   { return !(lhs == rhs); }
24 };
```

Based on this one can quite easily implement `operator--` changing all oc-

currences of left with right. This iterator template shown above can be also used with the other type of traversal we will see below. Printing the binary search tree is now quite simple

```
1  using iter = rt::bst_iter<rt::inorder_successor>;
2  std::for_each( iter {root.left}, iter {}
3               , [](auto o){std::cout << o << std::endl;});
4
5  std::copy( iter {root.left}, iter {}
6           , std::ostream_iterator<int>(std::cout, " "));
```

Let us see how can we adapt some algorithms from previous sections so that they work with iterators.

Binary search. We see that algorithm 1.3 requires random access i.e. we jump from one memory location to the other, however the STL implementation of a binary search requires only forward iterators. Why would one use a binary search on a singly linked list instead of a simple linear search? Linear search uses $O(n)$ comparisons i.e. for every element whereas a binary search will perform $O(log(n))$ comparisons. If `operator++` is cheap for the type, than a binary search may be worth.

An implementation that requires only forward iterator can be seem below

```cpp
template<class Iter, class T>
bool binary_search_stl(Iter begin, Iter end, const T& K)
{
  auto l = std::distance(begin, end);

  while (l > 0) {
    auto half = l / 2;
    auto mid = begin;
    std::advance(mid, half);

    if (K < *mid) {
      l = half;
    } else if (*mid < K) {
      begin = mid;
      ++begin;
      l = l - half - 1;
    } else {
      return true;
    }
  }
  return false;
}
```

1.1 List insertion sort

1.2 Merge sort

1.2.1 Two-way merge

```
template <class Iter, class Iter2>
auto merge( Iter begin1, Iter end1
          , Iter begin2, Iter end2
          , Iter2 output)
{
    if (begin1 == end1) return output;

    for (;;) {
        if (*begin1 <= *begin2) {
            *output++ = *begin1++;
            if (begin1 == end1) {
                std::copy(begin2, end2, output++);
                return output;
            }
        } else {
            *output++ = *begin2++;
            if (begin2 == end2) {
                std::copy(begin1, end1, output++);
                return output;
            }
        }
    }

    return output;
}
```

1.2.2 List merging

1.3 Memory allocation in C++

The data structures we will be dealing with in this book are assumed to fit on fast access memory. It is therefore a good idea to make a short review of memory allocation in C++. A programmer has three main ways to allocate memory inside a program which differ basically in their lifetime in the program.

The first type is the static allocation. The lifetime of variables with this kind of storage is the same as that of the `main` function and it cannot grow in

size along program execution. The amount of memory needed for all static variable in a program is determined at compile time and they all go into a place called *data segment*. Variables declared globally or with the `static` specifier belong in this category.

```
1 // Both a and b will live as long as the main function.
2 int a;
3
4 void foo()
5 {
6   static int b;
7 }
8
9 int main() { }
```

The second type of allocation occurs when a variable is declared inside a scope e.g. inside a function. Such allocations occur at program runtime e.g. when execution enters a function, but its size must be known at compile time. Allocation and deallocation also occurs automatically when execution enters and exits the scope respectively. Variables with this storage are said to be on the *stack*. For example

```
1 void foo()
2 {
3   int b; // Existense tied to the scope of foo.
4 }
```

When the memory size us unknown at compile time or it is too big to fit into a function frame we have to make a trip to the heap also called *free storage*. Allocation of memory on the Heap happen at program runtime and is supported at language level by means of the `operator new`. For example

```
1 auto* p1 = new int; // Allocates an int
```

The term *dynamic allocation* is used for such allocations. While memory allocated statically is automatically freed upon function return, memory allocated dynamically on the heap has to be released by the programmer as only he knows when it is not needed anymore. That is achieved at language level with the `operator delete`.

```
delete p1;
```

In C++ however there is often no need of managing dynamic memory allocation manually as it is very prone to errors. A much better idea is to use smart pointers like `std::unique_ptr`

Array allocation The allocation of arrays is supported at language and library level. For example

```
// Built-in arrays supported at language level.
int buffer[32];

// Library solution.
std::array<int, 32> buffer;
```

The later has the similar properties as built-in array but fits better in the Standard Template Library (more on chapter **??**).

```
auto* p2 = new int[32];  // Allocates an array of int's
delete [] p2;  // Deallocates arrays.
```

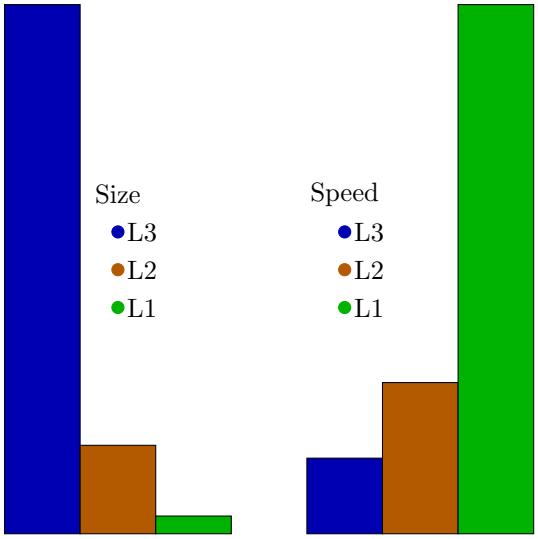
A better idea or a container when it makes sense. For example

```
std::vector<int> buffer;
```

will use dynamic memory allocation internally in a safe way. In STL containers memory management has been abstracted with the introduction of *allocators* that will be reviewed later on, until there we will use `operator new`.

1.3.1 Cache locality

When working with linked data structures as the linked list presented obove, it is a good idea to consider memory fragmentation. It is actually so important that it is difficult to have a C++ conference where the topic is not touched at least by some speakers.



Exercises

1.1. Modify algorithm (3.1) so that is requires only bidirectional iterators instead of random access iterators.

Interview Problems

Solutions

1.1

```
template <class Iter>
auto next_tuple_stl( Iter begin, Iter end
                   , Iter min, Iter max)
{
    auto size = std::distance(begin, end);
    std::advance(min, size);
    std::advance(max, size);

    while (*--end == *--max)
      *end = *--min;

    *end += 1;
    return end != begin;
}
```

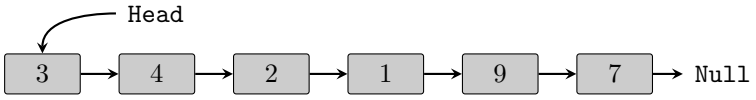
Chapter 2

Stacks

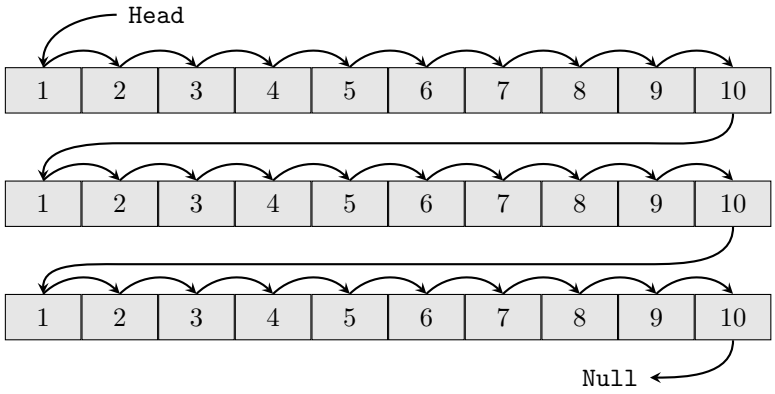
2.1 Quick sort

2.2 Node allocation

We have seen so far how to allocate contiguous blocks of memory, many data structures however use what we call linked allocation where blocks of contiguous memory have pointers to other blocks of contiguous memory elsewhere. The simplest example of such data structure is perhaps the a linked list as illustrated in the figure below.



When the blocks of memory have the same size it is custumary to use a technique called node allocation where a block containing e.g. N nodes is allocated at once and all nodes are linked together by using a field inside each node to store the address of the next block. We only have to keep track of the first node, usually called `HEAD`, so that when a new node is needed in the programm we can use the node the `HEAD` is pointing to and make the `HEAD` point to the next one. Such a data structure is called a stack or *FIFO* that stands for *first in first out*. The figure below illustrate the proceedure



Exercises

Interview Problems

2.1. Suppose you have the following string

```
const std::string s {"1 13 - 6 + 9 8 +"};
```

Your task is to process the string applying the + or - operator that you meet during traversal to the last two elements if appropriate. For example

```
" 1 13 - 6 + 9 8 +"// Input string
"12  6 + 9 8 +"    // After -
"18  9 8 +"        // After +
"18 17"            // End result
```

Solutions

2.1 Iterative solution. It is not stated in the problem description but you are expected to convert those numbers to integers to work with them properly. As a first step in solving this problem we can write the function that processes the + and - signs.

```
template <class C>
void handle_op(std::stack<int>& stack, C c)
{
  if (stack.size() < 2)
    return;
  auto a = stack.top();
  stack.pop();
  auto b = stack.top();
  stack.pop();
  stack.push(c(a, b));
}
```

The callable parameter `c` will be either a `std::plus` or `std::minus`. The iterative procedure can then be written

```
void iterative(const std::string& s)
{
  using iter_type = std::istream_iterator<std::string>;
  std::stringstream ss(s);
  std::stack<int> stack;

  auto f = [&](const auto& o)
  {
    if (o == "+")
      handle_op(stack, std::plus<int>{});
    else if (o == "-")
      handle_op(stack, std::minus<int>{});
    else
      stack.push(std::stoi(o));
  };

  std::for_each(iter_type{ss}, iter_type{}, f);
  visit(stack);
}
```

Where we use a `std::istream_iterator` to traverse the string using a space as a separator.

Recursive solution. One can also write a similar but recursive solution

```
1
2  void recursive( std::stack<int> stack
3                , std::istream_iterator<std::string> iter)
4  {
5    using iter_type = std::istream_iterator<std::string>;
6    if (iter == iter_type {}) {
7      visit(stack);
8      return;
9    }
10
11   auto s = *iter;
12   if (s == "+")
13     handle_op(stack, std::plus<int>{});
14   else if (s == "-")
15     handle_op(stack, std::minus<int>{});
16   else
17     stack.push(std::stoi(s));
18
19   recursive(std::move(stack), ++iter);
20 }
```

Chapter 3

Queues

3.1 Priority queues

3.1.1 Heaps

Heap sort

3.2 Deques

Exercises

Interview Problems

3.1. Design a program that logs messages asynchronously into a file. The program may have many threads producing the messages.

Solutions

3.1

A typical solution to this problem is to write the messages in a queue in one thread and write them to a file in the logging thread. To inform there is a message awaiting to be logged one can use a `std::condition_variable`. Basically the data needed is

```
std::mutex mutex;
std::condition_variable cv;
bool has_new_msg;
std::queue<std::string> msg_queue;
```

Where `mutex` is needed to synchronize the access to the queue. Note that `std::queue` is not guaranteed to be thread safe. The boolean `has_new_msg` is necessary in order to handle spurious wakes of the condition variable `cv`. We can design the functions that runs in the logging thread to write all enqueued message when it is notified. For example

```
void logger()
{
  std::ofstream ofs("log.txt");
  for (;;) {
    std::unique_lock<std::mutex> lock(mutex);
    cv.wait(lock, [&] { return has_new_msg; });

    if (msg_queue.empty())
      break;

    while (!msg_queue.empty()) {
      ofs << msg_queue.front() << std::endl;
      msg_queue.pop();
    }
    has_new_msg = false;
  }
}
```

In the loop above, if the condition variable is notified with an empty queue the function exits allowing the applicaiton to join the logging thread. The function that posts individual messages in the queue reads

```
void post_msg(std::string msg = {})
{
    std::unique_lock<std::mutex> lock(mutex);
    msg_queue.push(msg);
    if (msg.empty())
      std::queue<std::string>{}.swap(msg_queue);
    has_new_msg = true;
    lock.unlock();
    cv.notify_one();
}
```

Any message that happens to be on the logging queue will be discarded when an empty message is posted. This function can still be extended to post more than one message at once in the queue.

An example main function that posts some messages and exits can be seem below

```
1  int main()
2  {
3    using namespace std::literals::chrono_literals;
4    std::thread logging_thread(logger);
5
6    for (auto i = 0; i < 10; ++i) {
7      std::this_thread::sleep_for(20ms);
8      post_msg("Message: " + std::to_string(i));
9    }
10   post_msg();
11   logging_thread.join();
12 }
```

Comment. I was asked this problem in an interviewer and additionally the interviewer asked me what could be a cause of delays in this logging mechanism? He was meaning that as the queue grows larger it may have to reallocate memory to fit new items. I told him a queue uses a deque as its underlying data structure and therefore allocations of more memory does not affect existing items. He was thinking a queue uses continuous blocks of memory like a vector.

Chapter 4

Binary search trees

4.1 Fundamental operations

It is impossible to say enough about binary trees. Int this section we will see the most important algorithms and its applications. To begin with we need a node

```
1 struct bst_node {
2   int info;
3   bst_node* left;
4   bst_node* right;
5 };
```

4.1.1 Search and insertion

The next algorithm inserts a new element in the binary search tree

```
1  void bst_insert(bst_node& root, int o)
2  {
3    if (!root.left) {
4      root.left = new bst_node {o, nullptr, nullptr};
5      return;
6    }
7
8    auto* p = root.left;
9    while (p) {
10     if (o < p->info) {
11       if (!p->left) {
12         p->left = new bst_node {o, nullptr, nullptr};
13         return;
14       }
15       p = p->left;
16     } else if (p->info < o) {
17       if (!p->right) {
18         p->right = new bst_node {o, nullptr, nullptr};
19         return;
20       }
21       p = p->right;
22     }
23   }
24 }
```

4.1.2 Tree insertion sort

```
void bst_insertion_sort_impl(bst_node& head, int key)
{
  if (!head.left) {
    head.left = new bst_node {key, nullptr, nullptr};
    return;
  }

  auto* p = head.left;
  while (p) {
    if (key < p->info) {
      if (!p->left) {
        p->left = new bst_node {key, nullptr, nullptr};
        return;
      }
      p = p->left;
    } else {
      if (!p->right) {
        p->right = new bst_node {key, nullptr, nullptr};
        return;
      }
      p = p->right;
    }
  }
}
```

```
void tree_insertion_sort(std::vector<int>& input)
{
  bst_node root {};
  for (auto o : input)
    bst_insertion_sort_impl(root, o);

  using iter = rt::bst_iter<inorder_successor>;

  std::copy(iter {root.left}, iter {}, std::begin(input));
}
```

4.1.3 Deletion

4.2 Depth first traversal

There are two main ways to traverse a tree, breadth first or depth first. There are three main ways to perform a depth first traversal each one with their own uses.

4.2.1 Preorder

The code below shows the recursive definition of the preorder traversal

```
1  void preorder_recursive(bst_node* p)
2  {
3    if (!p)
4      return;
5
6    visit(p);
7    preorder_recursive(p->left);
8    preorder_recursive(p->right);
9  }
```

In words we have, visit the root, visit the left subtree and visit the right subtree. An iterative implementation of the recursive definition can be seem below. It uses a stack to keep track of nodes that still have to be visited

```
1  void preorder_traversal(const bst_node* p)
2  {
3    std::stack<const bst_node*> s;
4    while (p) {
5      visit(p);
6      if (p->right)
7        s.push(p->right);
8
9      p = p->left;
10     if (!p && !s.empty()) {
11       p = s.top();
12       s.pop();
13     }
14   }
15 }
```

Preorder traversals are used to copy a binary tree to another memory location, this is used for example in `C++` ordered containers to implement copy

constructors. It is convenient to factor the code that calculates the preorder successor out of the loop in the iterative implementation above. For example

```
struct preorder_successor {
  std::stack<bst_node*> s;
  bst_node* p;
  void next()
  {
    if (p->right)
      s.push(p->right);

    p = p->left;
    if (!p && !s.empty()) {
      p = s.top();
      s.pop();
    }
  }
  preorder_successor(bst_node* root) : p(root) {}
};
```

The usage of preorder traversal to copy a binary tree to another memory location can be seen below.

```
void copy(bst_node* from, bst_node* to)
{
  preorder_successor p(from);
  preorder_successor q(to);

  for (;;) {
    if (p.p->left)
      q.p->left = new bst_node {{}, nullptr, nullptr};

    p.next();
    q.next();

    if (p.p == from)
      return;

    if (p.p->right)
      q.p->right = new bst_node {{}, nullptr, nullptr};

    q.p->info = p.p->info;
  }
}
```

In section **??** we will define iterators that make it easy to traverse binary

search trees.

4.2.2 Inorder traversal

The recursive definition of inorder traversal is given by

```
1  void inorder_recursive(bst_node* p)
2  {
3    if (!p)
4      return;
5
6    inorder_recursive(p->left);
7    visit(p);
8    inorder_recursive(p->right);
9  }
```

That means, first visit the left subtree, then visit the root and then visit the right subtree. The inorder traversal visits the elements in the binary search tree in sorted order. The iterative solution uses a stack

```
1  void inorder_traversal(const bst_node* p)
2  {
3    std::stack<const bst_node*> s;
4    for (;;) {
5      while (p) {
6        s.push(p);
7        p = p->left;
8      }
9      if (s.empty())
10       break;
11     p = s.top();
12     s.pop();
13     visit(p);
14     p = p->right;
15   }
16 }
```

The inorder traversal is used to implement iterators in unordered unordered `C++` containers like `std::map` and `std::set`. The code below rearranges the iterative code above into an iterator like class except that the iterator implementation uses a `operator++` instead of `next` among other details that are not important now.

```
struct inorder_successor {
  std::stack<const bst_node*> s;
  const bst_node* p;
  void left_most()
  {
    while (p) {
      s.push(p);
      p = p->left;
    }
    if (s.empty())
      return;

    p = s.top();
    s.pop();
  }
  void next()
  {
    p = p->right;
    left_most();
  }
  inorder_successor(const bst_node* root)
  : p(root) {left_most();}
};
```

4.2.3 Post order traversal

The recursive definition is

```
void postorder_recursive(bst_node* p)
{
  if (!p)
    return;

  postorder_recursive(p->left);
  postorder_recursive(p->right);
  visit(p);
}
```

The iterative solution is

```
1  void postorder_traversal(const bst_node* p)
2  {
3    std::stack<const bst_node*> s;
4    const bst_node* q = nullptr;
5    for (;;) {
6      while (p) {
7        s.push(p);
8        p = p->left;
9      }
10
11     for (;;) {
12       if (s.empty())
13         return;
14
15       p = s.top();
16       s.pop();
17
18       if (!p->right || p->right == q) {
19         visit(p);
20         q = p;
21         continue;
22       }
23       s.push(p);
24       p = p->right;
25       break;
26     }
27   }
28 }
```

Deleting a BST.

4.3 Breadth first traversal

Exercises

4.1. Suppose you are going to interpret consecutive entries in a map as intervals. For example, the consecutive entries (K_i, V_i) and (K_{i+1}, V_{i+1}) associates all $K_i \leq K < K_{i+1}$ with the value V_i, where the index i refers to an element in the map. Consecutive intervals should also have different values associated to them i.e. the condition $V_i \neq V_{i+1}$ must hold. The map is initialized with the entry (`std::numerical_limits<K>::min()`, V_0). The assign function has the following signature

```
void assign(const K& a, const K& b, const V& v);
```

As an example supose the we have the map `std::map<std::uint8_t, char>` and insert the following entries one by one `(3, 8, 'b')`, `(5, 9, 'c')`, `(2, 10, 'd')`. This will produce the following maps

0	1	2	3	4	5	6	7	8	9	10	11	12	...	255
a	a	a	a	a	a	a	a	a	a	a	a	a	...	a
a	a	a	**b**	b	b	b	b	**a**	a	a	a	a	...	a
a	a	a	**b**	b	**c**	c	c	c	**a**	a	a	a	...	a
a	a	**d**	d	d	d	d	d	d	d	**a**	a	a	...	a

Where only the entries in bold are items in the map. Other entries are only mapped values. Previous values inside the range should be overwriten and outside the range should not be touched. If the interval is incompatible you should do nothing.

The key type K is less-than comparable (operator $<$ is available) but not equality comparable. The value type V on the other hand is equality comparable (operator $==$) but does not implement any other operator.

Solutions

1) Solution. To properly solve this you have to sketch on paper what a maps looks like. Assume a struct in the form

```
template<class K, class V>
struct interval_map {
  std::map<K,V> map {{std::numeric_limits<K>::min(), {}}};
  void assign(const K& a, const K& b, const V& v);
};
```

The solution provided below performs only two $O(\log(n))$ operations. We first check if the input range is valid and return appropritely if not. Second we check insertion position and check if the insertion values do not produce equal consecutive values. Last we either insert the value or update a previous value.

```
void assign(const K& a, const K& b, const V& v)
{
  auto comp = map.key_comp();

  if (!comp(a, b))
    return;

  auto lbb = map.lower_bound(b);
  auto vb = lbb == std::end(map) ? map.rbegin()->second
          : comp(b, lbb->first)  ? std::prev(lbb)->second
          :                        lbb->second;

  if (v == vb)
    return;

  auto lba = map.lower_bound(a);
  auto va = lba == std::end(map)   ? map.rbegin()->second
          : lba == std::begin(map) ? lba->second
          :                          std::prev(lba)->second;

  if (v == va && lba != std::begin(map))
    return;

  map.erase( std::next(map.insert_or_assign(lba, a, v))
           , map.insert_or_assign(lbb, b, vb));
}
```

If you are asked this question on an interview you will almost certainly be asked how can you test your solution. There are many corner cases to test, like when the begin or the end keys are equal to an element already in the map. It is useful to write some randomized intervals an check whether the condition $V_i \neq V_{i+1}$ always holds. A function to test this could be

```
auto check_valid_map(const std::map<K, V>& map)
{
  auto func = [](const auto& a, const auto& b)
  {return a.second == b.second;};

  return std::adjacent_find( std::cbegin(map), std::cend(map)
                           , func) == std::cend(map);
}
```

Chapter 5

Hashing

5.1 Open addressing

5.2 Linear probing

Chapter 6

Bibliography

[1] The art of computer programming. Vol. 1, 2, 3 and 4A.

[2] A walk through combinatorics.