FORMAL DEFINITIONS OF ASYMPTOTIC NOTATION

Overview of Asymptotic Notation

\(O(g(n))\), \(o(g(n))\), \(\Omega(g(n))\), and \(\Theta(g(n))\) are sets of functions.

Intuitively:

- \(O(g(n))\) contains functions whose dominant term is at most that of \(g(n)\).
- \(o(g(n))\) contains functions whose dominant term is strictly smaller than that of \(g(n)\).
- \(\Omega(g(n))\) contains functions whose dominant term is at least that of \(g(n)\).
- \(\Theta(g(n))\) contains functions whose dominant term is equal to that of \(g(n)\).

Definitions

Definitions of "Big Oh"

\(O(g(n))\) is a set of functions.

\(f(n)\) is in \(O(g(n))\) if

\[ \exists c \in \mathbb{R}, \forall n \geq n_0, f(n) \leq c \cdot g(n) \]

This is read as "There exists a \(c\) such that, for all but a finite number of \(n\)s, \(f(n)\) is bounded above by \(c \cdot g(n)\)."

In other words,

\[ \exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, f(n) \leq c \cdot g(n) \]

where \(n_0\) is the threshold above which this is true. This is shown in the figure, where \(c = 2\):
For example, consider

\[ T(n) = 10n^3 + 100n^2 + 55 \]

Since \( T(n) \leq 11n^3 \) once \( n \) gets large enough,

\[ T(n) \in \mathcal{O}(n^3). \]

Let's be more rigorous: We must find a \( c \) and \( n_0 \) such that

\[ \forall n \geq n_0, T(n) \leq c \cdot n^3. \]

From the equation, guess that \( c = 11 \). Then

\[ T(n) \leq 11n^3 \]
\[ \iff 10n^3 + 100n^2 + 55 \leq 11n^3 \]
\[ \iff 100n^2 + 55 \leq n^3 \]
\[ \iff 100 + \frac{55}{n^2} \leq n. \]

Since \( \frac{55}{n^2} \leq 1 \) when \( n \geq 8 \), \( n \) must be greater than 7 and (from the last line above) \( n \)

must be greater than 100.

Therefore, if \( c = 11 \) and \( n_0 = 101 \),

\[ \forall n \geq n_0, T(n) \leq c \cdot n^3 \]

and that proves that \( T(n) \in \mathcal{O}(n^3) \).
Definition of "Little Oh"

\( f(n) \) is in "little-oh" of \( g(n) \) if

\[
\forall c, \forall n \in \mathbb{N}, c \cdot f(n) \leq g(n).
\]

In other words, no matter how big \( c \) is, \( c \cdot f(n) \) is eventually bounded above by \( g(n) \).

In other words, \( f(n) \) eventually is strictly smaller than \( g(n) \), regardless of the coefficient in front of the dominant term of \( f(n) \).

The Usefulness of "Little Oh"

Given Algorithm A with time \( T_A(n) \) and Algorithm B with time \( T_B(n) \):

- If \( T_A(n) \in o(T_B(n)) \) then A is eventually faster than B.

For example, one sorting algorithm, \( A \), with \( T_A(n) = 10000n \log n \) will eventually be faster than another sorting algorithm, \( B \), with \( T_B(n) = 0.001n^2 \), since \( T_A(n) \in o(T_B(n)) \).

Definition of "Omega"

\( f(n) \) is in \( \Omega(g(n)) \) if

\[
\exists c, \exists n \in \mathbb{N}, c \cdot f(n) \geq g(n).
\]

In other words, \( f(n) \) is eventually greater than \( g(n) \).

\( g(n) \) is a lower bound of \( f(n) \).

Note that \( f(n) \in \Omega(g(n)) \) if and only if \( g(n) \in O(f(n)) \).

Definition of "Theta"

\( f(n) \) is in \( \Theta(g(n)) \) if

\[
f(n) \in O(g(n)) \text{ and } g(n) \in O(f(n)).
\]

In other words, \( f(n) \) and \( g(n) \) are eventually within a constant factor of each other.
Tight Bounds

\( n^3 \in O(n^{100}) \) but this isn’t a "tight bound".

\( O(g(n)) \) is a tight bound for \( f(n) \) if and only if

\[ \forall \delta(n) \text{ such that } f(n) \in O(\delta(n)), g(n) \in O(\delta(n)) \] also.

In other words, if there’s no function, \( \delta(n) \), that lies in the gap between \( f(n) \) and \( g(n) \), then \( O(g(n)) \) is a tight bound for \( f(n) \).

For example, \( O(n^{100}) \) is not a tight bound for \( n^3 \) because the function \( n^4 \) lies in the gap between \( n^3 \) and \( n^{100} \). That is,

\( n^3 \in O(n^4) \) but \( n^{100} \notin O(n^4) \).

Summary

- Always use \( \Theta \) and \( \omega \) for upper bounds.
- Always use \( \Omega \) for lower bounds.
- Never use \( \Omega \) for lower bounds.

A word on notation: In some texts, like CLR, you may see the notation

\[ f(n) = O(g(n)). \]

This is equivalent to our notation,

\[ f(n) \in O(g(n)). \]

Always use our notation.

Copyright 1998    James Stewart
Big-O Notation

Motivation

This lecture is about measuring the performance of computer programs. Our approach is based on the asymptotic complexity measure. That means that we don’t try to count the exact number of steps of a program, but how that number grows with the size of the input to the program. That gives us a measure that will work for different operating systems, compilers and CPUs. The asymptotic complexity is written using big-O notation. We will describe big-O notation, and show how big-O bounds can be computed for some common algorithms.

Running Time of Some Algorithms

Let’s consider a short piece of MATLAB code:

\[ x = 3*y + 2; \]
\[ z = z + 1; \]

If y, z are scalars, this piece of code takes a constant amount of time, which we write as O(1). In terms of actual computer instructions or clock ticks, it’s difficult to say exactly how long it takes. But whatever it is, it should be the same whenever this piece of code is executed. Because it’s a constant, we don’t need to be specific as to whether its computer cycles or instructions, or the time on our watch. O(1) means some constant, it might be 5, or 1 or 1000.

Now let’s consider a simple for loop:

\[
\text{for } i = 1:N \\
\quad v(i) = v(i) + 1; \\
\text{end}
\]

This loop will run exactly N times, and because the inside of the loop takes constant time, the total running time is proportional to N. We write it as O(N). The actual number of instructions might be 50N, while the running time might be 17N microseconds. It might even be 17N+3 microseconds because the loop needs some time to start up. The big-O notation allows a multiplication factor (like 17) as well as an additive factor (like 3). As long as it’s a linear function which is proportional to N, the correct notation is O(N) and the code is said to have linear running time.

Now let’s look at a more complicated example, a nested for loop:

\[
\text{for } i = 1:N \\
\quad \text{for } j = 1:N \\
\quad \\
\quad \quad a(i, j) = b(i, j) * x; \\
\quad \text{end} \\
\text{end}
\]
The outer \texttt{for} loop executes $N$ times, while the inner loop executes $N$ times for every execution of the outer loop. That is, the inner loop executes $N \times N = N^2$ times. The assignment statement in the inner loop takes constant time, so the running time of the code is $O(N^2)$ steps. This piece of code is said to have \textit{quadratic} running time.

\textbf{Danger!}

In MATLAB, do not assume that simple statements take constant time. Many do not!

\begin{verbatim}
    a = b;
\end{verbatim}

will take $O(1)$ time if $b$ is a scalar. But if $b$ is a $1 \times N$ vector, it will take $O(N)$ time, because MATLAB will have to copy every element. If $b$ is an $N \times N$ matrix, the running time is $O(N^2)$.

\textbf{Rules for using big-O}

Big-O bounds, because they ignore constants, usually allow for very simple expressions for running time bounds. Below are some properties of big-O that allow bounds to be simplified. The most important property is that big-O gives an upper bound only. If an algorithm is $O(N^2)$, it doesn’t have to take $N^2$ steps (or a constant multiple of $N^2$). But it can’t take more than $N^2$. So any algorithm that is $O(N)$, is also an $O(N^2)$ algorithm. If this seems confusing, think of big-O as being like “<”. Any number that is $< N$ is also $< N^2$.

1. Ignoring constant factors: $O(c \ f(N)) = O(f(N))$, where $c$ is a constant; e.g. $O(20 \ N^3) = O(N^3)$

2. Ignoring smaller terms: If $a < b$ then $O(a+b) = O(b)$, for example $O(N^2+N) = O(N^2)$

3. Upper bound only: If $a < b$ then an $O(a)$ algorithm is also an $O(b)$ algorithm. For example, an $O(N)$ algorithm is also an $O(N^2)$ algorithm (but not vice versa).

4. $N$ and $\log N$ are “bigger” than any constant, from an asymptotic view (that means for large enough $N$). So if $k$ is a constant, an $O(N + k)$ algorithm is also $O(N)$, by ignoring smaller terms. Similarly, an $O(\log N + k)$ algorithm is also $O(\log N)$.

5. Another consequence of the last item is that an $O(N \ \log N + N)$ algorithm, which is $O(N(\log N + 1))$, can be simplified to $O(N \ \log N)$.

\textbf{Estimating running time}

The graph below shows the running time for an $O(N)$ algorithm (the straight line) and an $O(N^2)$ algorithm. Note that in this case, the $O(N)$ algorithm takes more time than the $O(N^2)$ algorithm for small inputs. That is because the big-O notation pays no attention to constant factors. The $O(N)$ algorithm must have a larger constant factor than the $O(N^2)$ algorithm. When the input size ($N$) is small, the constant factor is important. As $N$ grows, the $O(N)$ algorithm takes less and less time relative to the $O(N^2)$ algorithm. So for large $N$, constant factors don’t matter, just the big-O bounds.
Log-log plots

The \( O(N) \) and \( O(N^2) \) algorithms have exponents of 1 and 2 respectively. By exponents, we mean the power of \( N \) appearing in the big-O bound. On the graph above, it's difficult to determine the exponent from the graph. The straight line graph obviously has exponent 1, but the curved line could be 2 or 2.5 or 3, and it would be difficult to tell. If we plot the log of the running time versus the log of \( N \), the relationship becomes much clearer.

The graph below shows a log-log plot of the \( O(N) \) and the \( O(N^2) \) algorithm again. Both of them are now straight lines. Notice that one line has slope 1, and the other has slope 2, corresponding to the exponents of the algorithms.

Log-log plots provide a convenient way to determine asymptotic bounds from some running time data. It turns out that the slope of a log-log plot gives the running time exponent. That is, an \( O(N) \) algorithm has slope 1, an \( O(N^2) \) algorithm has slope 2, etc. To see this we can take the log
of the expression for the running time bound. Let $T$ be the running time in some units, let $N$ be
the problem size, and let $k$ be the running time exponent. Let $c$ be the constant hidden by the
big-O bound. Then:

$$T < c N^k$$

$$\log_{10} T < \log_{10} c + k \log_{10} N$$

Which makes it clear that the slope of the log-log plot is $k$. If you plot the running time of an
unknown algorithm versus $N$, and you find that the plot has slope 3, then you know that the
algorithm is $O(N^3)$. It doesn’t matter if the curve is not perfectly straight. There will often be
deviations from the straight line for small values of $N$. But as long as the curve seems to be
converging to a straight line, then the running time bound should be valid.

**Properties of logs**

The log function shows up so often in running time bounds that it’s worth reviewing some of its
properties. We list some important ones below:

- $\log_b x = p$ if and only if $b^p = x$ (definition)
- $\log_b x * y = \log_b x + \log_b y$
- $\log_b x / y = \log_b x - \log_b y$
- $\log_b x^p = p \log_b x$ which implies that $(x^p)^q = x^{(pq)}$
- $\log_b x = \log_a x * \log_b a$

The last of these rules is particularly important for big-O bounds. It says that the log to the base
$b$ and the log to the base $a$ are related by a constant factor, $\log_b a$. If a log appears in a big-O
bound, for example $O(N \log_b N)$, then it is the same as $O(N \log_a N)$ because the big-O
bound hides the constant factor between the logs. The base doesn’t make any difference, so it is
usually left out of big-O bounds. The bound above would normally be written $O(N \log N)$.

**Analysis of matrix multiply**

Let’s start with an easy case. Multiplying two $N \times N$ matrices. The code to compute the matrix
product $C = A \times B$ is given below.

```plaintext
for i = 1:N
  for j = 1:N
    C(i, j) = 0
    for k = 1:N
      C(i, j) = C(i, j) + A(i, k) * B(k, j);
    end
  end
end
```
There are 3 nested for loops, each of which runs N times. The innermost loop therefore executes N*N*N = N^3 times. The innermost statement, which contains a scalar sum and product takes constant O(1) time. So the algorithm overall takes O(N^3) time.

**Analysis of bubble sort**

In the previous chapter we looked at bubble sort and binary search. We will give a simple analysis of both algorithms here. First, let’s look at bubble sort. The main body of the code looks something like this:

```plaintext
for i = (N-1) : -1 : 1
  for j = 1 : i
    if (a(j) > a(j+1)) swap a(j) and a(j+1); end
  end
end
```

This looks like the double loop we looked at earlier. The innermost statement, the if, takes O(1) time. It doesn’t necessarily take the same time when the condition is true as it does when it is false, but both times are bounded by a constant. But there is an important difference here. The outer loop executes N times, but the inner loop executes a number of times that depends on i. The first time the inner for executes, it runs i = N-1 times. The second time it runs N-2 times, etc. The total number of times the inner if statement executes is therefore:

\[ (N-1) + (N-2) + \ldots + 3 + 2 + 1 \]

This is the sum of an arithmetic series. The value of the sum is N(N-1)/2. So the running time of bubble sort is O(N(N-1)/2), which is O((N^2-N)/2). Using the rules for big-O given earlier, this bound simplifies to O((N^2)/2) by ignoring a smaller term, and to O(N^2), by ignoring a constant factor. Thus, bubble sort is an O(N^2) algorithm.

**Analysis of binary search**

Binary search is a little harder to analyze because it doesn’t have a for loop. But it’s still pretty easy because the search interval halves each time we iterate the search. The sequence of search intervals looks something like this:

\[ N, \frac{N}{2}, \frac{N}{4}, \ldots, 8, 4, 2, 1 \]

It’s not obvious how long this sequence is, but if we take logs, it is:

\[ \log_2 N, \log_2 N - 1, \log_2 N - 2, \ldots, 3, 2, 1, 0 \]

Since the second sequence decrements by 1 each time down to 0, its length must be \( \log_2 N + 1 \). It takes only constant time to do each test of binary search, so the total running time is just the number of times that we iterate, which is \( \log_2 N + 1 \). So binary search is an \( O(\log_2 N) \) algorithm. Since the base of the log doesn’t matter in an asymptotic bound, we can write that binary search is \( O(\log N) \).
Exercises

1. Give a simplified big-O bound for the following running times:
   
a) $20N^2$  
b) $10N^3 + 6N$  
c) $5N \log N + 30N$  
d) $N + 3N/\log N$

2. You have an $O(N^2)$ algorithm which takes 5 minutes to run on a problem of size $N = 1000$. How long will it take to solve a problem of size $N = 3000$? What about $N = 10000$?

3. Suppose we make the following change to binary search: Instead of setting $\text{mid} = \lfloor \text{left} + \text{right} \rceil /2$, we set $\text{mid} = \text{left} + 1$. So this version breaks the array into a one element subarray on the left, and an $N-1$ element subarray on the right. Then in the worst case, the recursive calls always go to the larger subarray (the one with $N-1$ elements). Give a big-O bound for this version of binary search.
Bounding Summations

Mathematical Induction:

To Prove:

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

when \( n = 1 \), the sum equals \( \frac{(1(1+1))}{2} = 1 \) so this checks out.

Assume true for \( n \), and prove \( n + 1 \)

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

\[ = \frac{(n+1)(n/2+1)}{2} \]

\[ = (n+1)(\frac{n+2}{2}) \] done!

Induction for bounds:

Prove

\[ \sum_{k=0}^{n} 3^k = \mathcal{O}(3^n) \]

recall

\[ \sum_{k=0}^{n} 3^k = \frac{(3^{n+1}-1)}{3 - 1} = \frac{(3^{n+1}-1)}{2} \]

must show that
\[
\sum_{k=0}^{n} 3^k \leq c 3^n \quad \text{for some } c \geq 0 \text{ and } n \geq n_0
\]

First

\[
\sum_{k=0}^{0} 3^k = 1 \leq c 3^0 = c, \quad c = 1 \text{ works for } n = 0
\]

Assume

\[
\sum_{k=0}^{n} 3^k \leq c 3^n
\]

Then \( n + 1 \)

\[
\sum_{i=0}^{n+1} 3^k = \sum_{i=0}^{n} 3^k + 3^{n+1} \leq c 3^n + 3^{n+1} = (1/3 + 1/c) c 3^{n+1} \leq c 3^{n+1}
\]

Provided that \((1/3 + 1/c) \leq 1 \Rightarrow 1/c \leq 1 - 1/3 \Rightarrow c \geq 3/2\)

Some Tricks:

Bound each term

\[
\sum_{k=1}^{n} k \leq \sum_{k=1}^{n} n = n^2
\]
for \( \sum_{i=1}^{n} a_k \), if \( a_k \leq a_{\text{max}} \forall k = 1 \ldots n \), \( \leq \sum_{k=1}^{n} a_{\text{max}} = n a_{\text{max}} \)

bound with Geometric Series
if \( a_{k+1} / a_k \leq r \forall k \) then \( a_{k+1} \leq r a_k \leq r^2 a_{k-1} \ldots \leq a_0 r^{k+1} \) therefore

\[
\sum_{k=0}^{n} a_k \leq \sum_{k=0}^{n} a_0 r^k = a_0 (\frac{r^{k+1} - 1}{1 - r})
\]

if \( r < 1 \) then \( a_0 (\frac{1}{r - 1}) = \sum_{k=0}^{\infty} a_0 r^k \)

Example

\[
\sum_{k=0}^{\infty} \frac{k}{3^k} = \sum_{k=0}^{\infty} a_k
\]

\( \frac{a_{k+1}}{a_k} = \frac{(k + 1) / 3^{k+1} \ast 3^k}{k} = \frac{(k + 1) / k \ast 1/3 \leq 1/3 \ast 2/1 = 2/3}{k} \)

as \( (k + 1) / k \leq 2 \) (equality when \( k = 1 \))

\[
\sum_{k=1}^{\infty} \frac{k}{3^k} \leq \sum_{k=1}^{\infty} 1/3 (2/3)^k = 1/3 (1 / (1 - 2/3)) = 1
\]

Harmonic Series (as a counter example)

\[
\sum_{k=1}^{\infty} k^{-1} = \lim_{n \to \infty} \sum_{k=1}^{n} k^{-1} = \lim_{n \to \infty} \Theta(\ln n) = +\infty
\]

So this Diverges

\( \frac{a_{k+1}}{a_k} = (1 / (k + 1)) / 1/k = k / (k + 1) < 1 \) but not!! \( \leq c < 1 \)
Splitting Sums

A lower bound for

\[ \sum_{k=1}^{n} k \geq \sum_{k=1}^{n} 1 = n \]

is

\[ \sum_{k=1}^{n} k = \sum_{k=1}^{n/2} k + \sum_{k=n/2+1}^{n} k \geq 0 + (n/2)^2 = \Omega(n^2/4) \]

ignoring initial terms

\[ \sum_{k=0}^{k_0-1} a_k + \sum_{k=k_0}^{n} a_k = O(1) + \sum_{k=k_0}^{n} a_k \]

\[ \sum_{k=0}^{\infty} \frac{k^2}{2^k} = \sum_{k=0}^{\infty} \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k} \leq O(1) + \frac{9/8}{2^0} \sum_{k=0}^{(8/9)^k} \]

\[ \frac{(k + 1)^2}{2^{k+1}} * \frac{2^k}{k^2} = (k + 1)^2 / 2k^2 \leq 8/9 \]

with \( k \leq 3 \Rightarrow O(1) \)

<table>
<thead>
<tr>
<th>( H_n = \sum_{k=0}^{n} \frac{1}{k} \leq \sum_{i=0}^{\lceil \lg n \rceil} \frac{2^i - 1}{2^i - 1} \leq \sum_{j=0}^{\lceil \lg n \rceil} \frac{1}{2^i} )</th>
<th>( \sum_{i=0}^{\lceil \lg n \rceil} 2^i - 1 )</th>
<th>( \sum_{j=0}^{\lceil \lg n \rceil} \frac{2^i - 1}{2^i} \leq \sum_{j=0}^{\lceil \lg n \rceil} \frac{1}{2^i} = \sum_{j=0}^{\lceil \lg n \rceil} 2^i - 1 )</th>
<th>( \sum_{i=0}^{\lceil \lg n \rceil} \frac{1}{2^i} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sum_{i=0}^{\lceil \lg n \rceil} \frac{2^i - 1}{2^i} \leq \sum_{j=0}^{\lceil \lg n \rceil} \frac{1}{2^i} )</td>
<td>( \sum_{i=0}^{\lceil \lg n \rceil} 2^i - 1 )</td>
<td>( \sum_{j=0}^{\lceil \lg n \rceil} \frac{1}{2^i} )</td>
<td>( \sum_{i=0}^{\lceil \lg n \rceil} \frac{1}{2^i} = \sum_{j=0}^{\lceil \lg n \rceil} \frac{1}{2^i} )</td>
</tr>
</tbody>
</table>
Approximation by integrals

\[
\int_{m-1}^{n} f(x) \, dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m}^{n+1} f(x) \, dx \leq \log(n) + 1
\]

\[
\int_{m}^{n+1} f(x) \, dx \leq \sum_{k=m}^{n} f(k) \leq \int_{n-1}^{n} f(x) \, dx
\]

\[
\sum_{k=1}^{n} \frac{1}{k} \geq \int_{1}^{n+1} \frac{1}{x} \, dx = \log(n + 1)
\]

\[
\sum_{k=2}^{n} \frac{1}{k} \leq \int_{1}^{n} \frac{1}{x} \, dx = \log n
\]

\[
\sum_{k=1}^{n} \frac{1}{k} \leq \log n + 1
\]
Figure 3.1 Approximation of $\sum_{k=m}^{n} f(k)$ by integrals. The area of each rectangle is shown within the rectangle, and the total rectangle area represents the value of the summation. The integral is represented by the shaded area under the curve. By comparing areas in (a), we get $\int_{m-\frac{1}{2}}^{n-\frac{1}{2}} f(x) \, dx \leq \sum_{k=m}^{n} f(k)$, and then by shifting the rectangles one unit to the right, we get $\sum_{k=m}^{n} f(k) \leq \int_{m}^{n+1} f(x) \, dx$ in (b).
Chapter 1

Recurrence Relations

1.1 Introduction

Many algorithms are recursive — a large problem is broken up into sub-problems, each a smaller instance of the original problem. The time-complexity analysis of a recursive algorithm is usually determined by solving a recurrence relation, which is a recursive definition of a function.

We introduce recurrence relations by means of an example: the Towers of Hanoi problem. In this problem, we start off with three rods or towers. On one of the rods is a pile of some number of disks, say, \( N \). Each disk in the stack is a little smaller than the one underneath it. The challenge is to transfer the tower of disks to another rod, subject to a couple of constraints:

1. only one disk may be moved at a time;
2. a larger disk may never be placed on top of a smaller disk.

As it turns out, there is an easy recursive solution to this problem. Clearly, if \( N = 1 \) we just move the unique disk and we are done. Now, let \( N \geq 2 \), and assume that we know how to solve the problem when there are \( N - 1 \) disks, i.e., we know how to transfer \( N - 1 \) disks subject to the above constraints. Our strategy for \( N \geq 2 \) disks consists of three phases:

1. Transfer the top \( N - 1 \) disks to a free rod.
2. Move the bottom (largest) disk on to the other free rod.
3. Move the stack of \( N - 1 \) disks on to the rod with the largest disk.

Hence, to solve the problem for \( N \) disks, we solve it for \( N - 1 \) disks, move the bottom disk, then solve the problem for \( N - 1 \) disks again. How can we analyze the efficiency of this recursive strategy?

Let \( T(N) \) be the number of moves performed to transfer \( N \) disks. Clearly, \( T(1) = 1 \). For \( N \geq 2 \), we have that Phases 1 and 3 make \( T(N - 1) \) moves each, while Phase 2 makes exactly one move, so that \( T(N) = 2T(N - 1) + 1 \). We can write this more formally as a recurrence relation:

\[
T(1) = 1 \\
T(N) = 2 \cdot T(N - 1) + 1 \quad \text{for } N \geq 2
\]

The above recurrence relation allows us to compute \( T(N) \) by repeated evaluations of the above formula. For example, we can compute \( T(5) \) as follows:

\[
T(1) = 1 \\
T(2) = 2T(1) + 1 = 3 \\
T(3) = 2T(2) + 1 = 7 \\
T(4) = 2T(3) + 1 = 15 \\
T(5) = 2T(4) + 1 = 31
\]
This method of evaluating the function \( T(N) \) is rather inefficient; think, for example, how long it would take to compute \( T(100) \). We would like to “solve” the recurrence relation, i.e., find a \textit{closed form} for \( T(N) \) that allows us to compute \( T(N) \) with a constant number of arithmetic operations, for any value of \( N \).

To solve the above recurrence relation, we will use a technique called \textit{unfolding} (also known as “expansion,” or “telescoping”, or “repeated substitution”). We start by observing that the recurrence relation applied to \( N - 1 \) gives:

\[
T(N - 1) = 2T(N - 2) + 1
\]

Hence, we have

\[
T(N) = 2(2T(N - 2) + 1) + 1
\]

By repeated substitutions we obtain:

\[
\begin{align*}
T(N) & = 2T(N - 1) + 1 & \text{Step 0} \\
& = 2(2T(N - 2) + 1) + 1 & \text{Step 1} \\
& = 4T(N - 2) + 2 + 1 \\
& = 8T(N - 3) + 4 + 2 + 1 & \text{Step 2} \\
& \vdots \\
T(N) & = 2^i T(N-i) + 2^{i-1} + 2^{i-2} + \cdots + 1 & \text{Step } i \\
& = 2^i T(N-i) + 2^i - 1 \\
& \vdots
\end{align*}
\]

The substitution process terminates at Step \( i \) such that \( N - i = 1 \), i.e., at Step \( N - 1 \). We conclude that

\[
T(N) = 2^{N-1}T(1) + 2^{N-1} - 1 = 2^{N-1} + 2^{N-1} - 1 = 2^N - 1
\]

Hence, the solution of the recurrence relation for \( T(N) \) is

\[
T(N) = 2^N - 1
\]

Using the asymptotic notation we can also say

\[
T(N) = \Theta(2^N)
\]

\subsection*{1.2 Fundamental Recurrence Relations}

In this section we examine several important recurrence relations and present techniques to solve them. There are three general strategies for solving recurrence relations. One is intuition, the second, unfolding, and the third, a combination of guessing and induction.

\subsubsection*{1.2.1 \( T(N) = T(N/2) + 1 \); \( T(1) = 1 \)}

This recurrence relation describes the worst-case time complexity of binary search in a sorted array. Binary search compares the search value \( x \) to the middle element of the array. If they are equal, we are done. If \( x \) is less than the middle element, then binary search is recursively called on the first half of the array. If \( x \) is greater, then the second half of the array is recursively searched.
Let $T(N)$ be the time complexity of binary search on an array with $N$ elements. In the worst case (when the $x$ is not found or is found at the very last step), binary search spends some constant amount of time to compare $x$ with the middle element, and then takes time $T(N/2)$ to search a subarray of size $N/2$. Hence, we have

$$
T(1) = 1 \\
T(N) = T(N/2) + 1 \quad \text{for } N \geq 2
$$

By unfolding the recurrence, we obtain:

$$
\begin{align*}
T(N) & = T(N/2) + 1 \quad \text{Step 1} \\
& = T(N/4) + 1 + 1 \quad \text{Step 2} \\
& = T(N/8) + 1 + 1 + 1 \quad \text{Step 3} \\
& \vdots \\
T(N) & = T(N/2^i) + i \quad \text{Step } i \\
& \vdots
\end{align*}
$$

The substitution process terminates at step $i$ such that $N/2^i = 1$, or $i = \log_2 N$. Thus, $T(N) = 1 + \log_2 N$, and the worst-case time complexity of binary search is $\Theta(\log N)$.

Notice the abstraction we used. The inner workings of the algorithm were not used in the evaluation of the recurrence relation. The only piece of information we used was $T(N) = T(N/2) + 1$.

A reasonable question is: why were we allowed to pick an integer $i$ such that $N = 2^i$? What happens if $N$ isn’t a power of 2? If our ultimate goal is to get an asymptotic expression for $T(N)$, it turns out that we can fudge a little. Suppose that $N$ is not a power of 2. We can take the integer $n$ such that that $2^n \leq N < 2^{n+1}$, i.e., $n = \lfloor \log_2 N \rfloor$, where $\lfloor x \rfloor$ denotes the largest integer less than or equal to $x$. (This is equivalent to “padding” the array with dummy values so that the size of the array is a power of 2.) We have

$$
T(2^n) \leq T(N) < T(2^{n+1})
$$

Hence

$$
n + 1 \leq T(N) < n + 2$$

$$
\lfloor \log_2 N \rfloor + 1 \leq T(N) < \lfloor \log_2 N \rfloor + 2
$$

from which we derive $T(N) = \Theta(\log N)$

Note that we assumed $T(N)$ is nondecreasing. In other words, if $M \leq N$, then $T(M) \leq T(N)$. This is a reasonable assumption for time complexity functions since it should take more time to process more input data.

Finally, although $T(N) = \log_2 N + 1$, we write that it is $\Theta(\log N)$ instead of $\Theta(\log_2 N)$. Why can we ignore the subscript, i.e., the base of the logarithm? Recall that in general

$$
\log_b x = \frac{\log_a x}{\log_a b}
$$

which means that changing the base of the logarithm from $a$ to $b$ corresponds to multiplying by the constant $\frac{1}{\log_a b}$. But remember that in asymptotic notation we ignore constant factors, so that the base of the logarithm is irrelevant and we write simply $T(N) = \Theta(\log N)$.
1.2.2 Induction

The mathematical technique of \textit{induction} can be used to prove the correctness of a solution for a recurrence relation. Here is how induction works. Suppose we’re trying to prove that some property \( P \) is true for all positive integers \( n \). We denote the assertion that “\( P \) is true for \( n \)” by writing \( P(n) \). How can we show \( P(n) \) for all integers \( n \geq 1 \)? The inductive argument has two steps. First, we must prove the \textit{basis case}, showing that \( P(1) \) is true. (Usually this step is pretty straightforward.) Second, we need to prove the \textit{inductive step}: that if \( P(n) \) is true for all integers \( n \) such that \( 1 \leq n < k \), then \( P(n) \) is also true for \( n = k \).

It may not be obvious why proving \( P(1) \), and that if \( P(n) \) for \( n < k \) then \( P(k) \), should prove anything at all. Proof by induction is fairly subtle. Well, suppose we were trying to verify a particular case, say, \( P(5) \). From the basis step we know \( P(1) \). By the inductive step, if \( P(1) \) is true, then so is \( P(2) \). Invoking the inductive step again, since \( P(1) \) and \( P(2) \) are true, so is \( P(3) \), and so on up to \( P(5) \).

At any rate, we shall use induction to prove that \( T(N) = \log_2 N + 1 \).

\textit{Basis step:} In this case, it’s easy, because \( \log_2 1 + 1 = 1 \) and we are given that \( T(1) = 1 \).

\textit{Inductive step:} By the inductive hypothesis, we assume that \( T(n) = \log_2 n + 1 \) for all \( n < N \). So what can we say about \( T(N) \)? By definition, \( T(N) = T(N/2) + 1 \). But from the inductive hypothesis, we know that \( T(N/2) = \log_2 (N/2) + 1 = \log_2 N \). (This is because \( N/2 < N \).) So \( T(N) = \log_2 N + 1 \), and the inductive step is complete.

Since we were able to prove the basis step and the inductive step, we have formally proved by induction that \( T(N) = \log_2 N + 1 \) for all \( N \geq 1 \).

One slightly tricky thing about such proofs is knowing where to go. Unless you have a good idea of what the answer is, induction may not help you prove much. Typically, one makes an “educated guess” of the answer, using techniques such as telescoping. Then, this guess is (hopefully) proved correct using induction.

1.2.3 \( T(N) = T(N/2) + N; \quad T(1) = 1 \)

Initial Attack

This recurrence relation describes a hypothetical algorithm which looks at every element \( (N) \), then recurses on half of them \( (T(N/2)) \). We’ll use unfolding again:

\[
T(N) = T(N/2) + N \\
= T(N/4) + N/2 + N \\
= T(N/8) + N/4 + N/2 + N \\
\vdots \\
= T(N/2^i) + N/2^{i-1} + N/2^{i-2} + \cdots + N/2 + N \quad \text{Step } i \\
\vdots
\]

We stop substituting at step \( n \) such that \( N/2^n = 1 \), i.e., at step \( n = \log_2 N \), and we obtain

\[
T(N) = N \left( 1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^n} \right)
\]

The above expression contains a sum which occurs fairly often.

\footnote{In fact, this important technique was completely unknown to Euclid, so most of his theorems were only proven up to \( n = 3 \) and merely vigorously asserted for the general case.}
A Brief Detour

Consider the sum

\[ G_{\frac{1}{2}}(n) = \sum_{i=0}^{n} \left( \frac{1}{2} \right)^i = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^n} \]

This sum is a special case of a geometric sum, which is of the type

\[ G_{\alpha}(n) = \sum_{i=0}^{n} \alpha^i \quad (\alpha \neq 1) \]

From the theory of geometric sums we have that

\[ G_{\alpha}(n) = \frac{\alpha^{n+1} - 1}{\alpha - 1} \]

Hence

\[ 1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^n} = 2 - \frac{1}{2^n} < 2 \quad \text{for all } n \geq 1 \]

Back to \( T(N) = T(N/2) + N \)

Anyways, we have found that

\[ N < T(N) = N \left( 1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^n} \right) < 2N \]

We happily ignore the constants, and conclude that

\[ T(N) = \Theta(N) \]

Notice that we have not found an exact solution for the recurrence, but only an asymptotic solution. This is enough for the purpose of time complexity analysis that motivated our study of recurrences. A more accurate analysis shows that

\[ T(N) = 2N - 1. \]

Even if we don’t know how this answer was arrived at, we can prove this time bound using induction.

Basis step: This is simple; \( T(1) = 1 = 2 \cdot 1 - 1. \)

Inductive step: Assume that \( T(n) = 2n - 1 \) for \( n < N. \) Then \( T(N) = T(N/2) + N, \) by definition.

Now, \( N/2 < N, \) so by the inductive hypothesis, assume \( T(N/2) = 2 \cdot \frac{N}{2} - 1 = N - 1. \) Thus, \( T(N) = N - 1 + N = 2N - 1, \) and the proof is complete.

1.2.4 \( T(N) = T(\alpha N) + N; \quad T(1) = 1 \quad (0 < \alpha < 1) \)

We now solve this slightly more general recurrence relation. It corresponds to an algorithm which looks at every element, then recursively operates on some fraction \( \alpha \) of the original elements. Note that \( 0 < \alpha < 1. \) If \( \alpha \) weren’t positive, the expression would be meaningless. If \( \alpha \) were greater than one, the size of the array would increase with every recursion — it would never terminate. So, we’ll make the assumption that \( 0 < \alpha < 1. \)

As usual, we unfold the recurrence.
\[ T(N) = T(\alpha N) + N \quad \text{Step 1} \]
\[ = T(\alpha^2 N) + \alpha N + N \quad \text{Step 2} \]
\[ = T(\alpha^3 N) + \alpha^2 N + \alpha N + N \quad \text{Step 3} \]
\[ \vdots \]
\[ T(N) = T(\alpha^i N) + \alpha^{i-1} N + \alpha^{i-2} N + \cdots + \alpha^2 N + \alpha N + N \quad \text{Step } i \]
\[ \vdots \]

The substitution process terminates at Step \( m \) such that \( \alpha^m N = 1 \), or \( m = \log_\alpha N \), and we have:

\[ T(N) = N(1 + \alpha + \alpha^2 + \cdots + \alpha^{m-1} + \alpha^m) \]

We encounter again a geometric sum so that

\[ T(N) = NG_\alpha(m) = N \frac{\alpha^{m+1} - 1}{\alpha - 1} \]

Since \( \alpha < 1 \), then \( \alpha^m < 1 \) and we have

\[ N < T(N) < \frac{1}{1-\alpha} N \]

Again, we ignore constants and simply write

\[ T(N) = \Theta(N) \]

1.2.5 \( T(N) = 2T(N/2) + N; \quad T(1) = 1 \)

This recurrence relation describes a typical “divide and conquer” algorithm (such as merge-sort) that divides the input into two halves, processes recursively each half, and then combines the solutions of the two subproblems in linear time.

As usual, we unfold the recurrence

\[ T(N) = 2T(N/2) + N \quad \text{Step 1} \]
\[ = 2(2T(N/4) + N/2) + N \quad \text{Step 2} \]
\[ = 4T(N/4) + 2N \]
\[ = 4(2T(N/8) + N/4) + 2N \quad \text{Step 3} \]
\[ = 8T(N/8) + 3N \]
\[ \vdots \]
\[ = 2^iT(N/2^i) + iN \quad \text{Step } i \]
\[ \vdots \]

The substitution process terminates at step \( n \) such that \( N/2^n = 1 \), i.e., \( n = \log_2 N \), and we obtain:

\[ T(N) = 2^n T(1) + nN = N + N \log_2 N \]

Remembering that the asymptotic notation allows us to discard asymptotically smaller terms and constant factors, we conclude that
\[ T(N) = \Theta(N \log N). \]

Using the technique of induction, we can prove the claim that \( T(N) = N + N \log_2 N \).

**Basis step:** Since \( \log_b 1 = 0 \), \( T(1) = 1 = 1 + 1 \log_2 1 \).

**Inductive step:** Assume that \( T(n) = 2T(n/2) + n \) for \( n < N \). Then

\[
\begin{align*}
T(N) &= 2T(N/2) + N \\
&= 2 \left[ (\frac{N}{2})^2 + \frac{N}{2} \log_2 \frac{N}{2} \right] + N \\
&= N + N \log_2 \frac{N}{2} + N \\
&= N + N \log_2 N - N \log_2 N \\
&= N + N \log_2 N.
\end{align*}
\]

This completes the proof.

1.2.6 \( T(N) = 2T(N/2) + N^\alpha; \quad T(1) = 1; \quad \alpha \neq 1 \)

Without much commentary, we proceed as always:

\[
\begin{align*}
T(N) &= 2T(N/2) + N^\alpha & \text{Step 1} \\
&= 2(2T(N/4) + (\frac{N}{2})^\alpha) + N^\alpha & \text{Step 2} \\
&= 4T(N/4) + N^\alpha(1 + \frac{1}{2}) \\
&= 4(2T(N/8) + (\frac{N}{4})^\alpha) + N^\alpha(1 + \frac{1}{2^2}) \\
&= 8T(N/8) + N^\alpha(1 + \frac{1}{2^2} + \frac{1}{2^3}) \\
& \vdots \\
&= 2^iT(N/2^i) + N^\alpha(1 + \frac{1}{2^i} + \frac{1}{4^i} + \cdots + \frac{1}{(2^{i-1})^\alpha}) & \text{Step } i \\
& \vdots
\end{align*}
\]

At this point, we (hopefully!) have enough experience to recognize that the substitution process terminates at step \( n = \log_2 N \). It now remains to evaluate the sum

\[
1 + \frac{1}{2^\alpha} + \frac{1}{4^\alpha} + \cdots + \frac{1}{(2^{n-1})^\alpha} = \sum_{i=0}^{n-1} \frac{1}{(2^i)^\alpha}
\]

We replace \( \frac{1}{(2^i)^\alpha} \) with \( \beta^i \), where \( \beta = \left( \frac{1}{2} \right)^{\alpha} \). Note that \( \beta < 1 \). This gives us

\[
\sum_{i=0}^{n-1} \frac{1}{(2^i)^\alpha} = \sum_{i=0}^{n-1} \beta^i = G_\beta(n-1) = \frac{\beta^n}{\beta - 1} < \frac{1}{1 - \beta}
\]

Hence, we have

\[
2^n + N^\alpha < T(N) < 2^n + \frac{1}{1 - \beta} N^\alpha
\]

\[
N + N^\alpha < T(N) < N + \frac{1}{1 - \beta} N^\alpha
\]

To find an asymptotic expression for \( T(N) \) we need to distinguish two cases. If \( \alpha < 1 \) then \( N^\alpha \) is asymptotically smaller than \( N \) and we have \( T(N) = \Theta(N) \). If instead \( \alpha > 1 \) then \( N^\alpha \) is asymptotically larger than \( N \) so that \( T(N) = \Theta(N^\alpha) \). Therefore

\[
T(N) = \begin{cases} 
\Theta(N) & \text{if } \alpha < 1 \\
\Theta(N^\alpha) & \text{if } \alpha > 1
\end{cases}
\]
1.3 Conclusion

We have shown how to solve several recurrence relations. For some recurrences we have found exact solutions, while for some others we have determined asymptotic solutions. It probably wouldn’t hurt to summarize our results:

<table>
<thead>
<tr>
<th>Recurrence Relation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(N) = 2T(N - 1) + 1$</td>
<td>$\Theta(2^N)$</td>
</tr>
<tr>
<td>$T(N) = T(N/2) + 1$</td>
<td>$\Theta(\log N)$</td>
</tr>
<tr>
<td>$T(N) = T(N/2) + N$</td>
<td>$\Theta(N)$</td>
</tr>
<tr>
<td>$T(N) = T(\alpha N) + N$</td>
<td>$\Theta(N)$ if $\alpha &lt; 1$</td>
</tr>
<tr>
<td>$T(N) = 2T(N/2) + N$</td>
<td>$\Theta(N \log N)$</td>
</tr>
</tbody>
</table>
| $T(N) = 2T(N/2) + N^\alpha$ | $\Theta(N)$ if $\alpha < 1$
$\Theta(N^\alpha)$ if $\alpha > 1$ |
Data Structures and Algorithms

1 Data Structures

Abstract Data Type is a well-defined set of operations on a mathematical model. Data Structure is the way an ADT is implemented. We look at simple data structures for representing dynamic sets. These data structures are often implemented using dynamically allocated objects and pointers. The main operations on these data structures are the insertion and the deletion of an element, searching for an element, finding the minimum or maximum element, finding the successor or predecessor of an element, etc.

1.1 Stacks

Objects can be inserted into a stack at any time, but only the most recently inserted (“last”) object can be removed at any time. E.g., Internet Web browsers store the address of recently visited sites on a stack. A stack is a container of objects that are inserted according to the last in first out (LIFO) principal.

Stack Abstract Data Type

A stack is an abstract data type (ADT) supporting the following two methods
- $\text{push}(o)$ : insert object $o$ at the top of the stack
- $\text{pop}()$ : remove from the stack and return the top object on the stack; an error occurs if the stack is empty

Stack supporting methods

The stack supporting methods are:
- $\text{size}()$ : return the number of objects in the stack
- $\text{isEmpty}()$ : return a Boolean indicating if the stack is empty
- $\text{top}()$ : return the top object on the stack, without removing it; an error occurs if the stack is empty

Applications of stacks

Direct applications
- Page-visited history in a Web browser
- Undo sequence in a text editor
- Chain of method calls in the Java Virtual Machine or C++ runtime environment

Indirect applications
- Auxiliary data structure for algorithms
- Component of other data structures

Array implementation of stack

A stack can be implemented with an $N$ element array $S$, with elements stored from $S[0]$ to $S[t]$, where $t$ is an integer that gives the index of the top element in $S$. 
Stack Methods Complexity

Each of the stack methods executes a constant number of statements. All supporting methods of the Stack ADT can be easily implemented in constant time thus, in array implementation of stack ADT each method runs in $O(1)$ time.

In a push operation, when the array is full, we can replace the array with a larger one. How large should the new array be?

- Incremental strategy: increase the size by a constant $c$
- Doubling strategy: double the size

Comparison of the Strategies

We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of $n$ push operations. We assume that we start with an empty stack represented by an array of size 1. We call amortized time of a push operation the average time taken by a push over the series of operations, i.e., $T(n)/n$.

Analysis of the Incremental Strategy

We replace the array $k = n/c$ times. The total time $T(n)$ of a series of $n$ push operations is proportional to

\[ n + c + 2c + 3c + 4c + \ldots + kc \]
\[ n + c(l + 2 + 3 + \ldots + k) \]
\[ n + ck(k + 1)/2 \]

Since $c$ is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$. The amortized time of a push operation is $O(n)$.

Direct Analysis of the Doubling Strategy

We replace the array $k = \log_2 n$ times. The total time $T(n)$ of a series of $n$ push operations is proportional to

\[ n + 1 + 2 + 4 + 8 + \ldots + 2^k \]
\[ n + 2^{k+1} - 1 = 2n - 1 \]
$T(n)$ is $O(n)$. The amortized time of a push operation is $O(1)$.

**Stack implementation using a linked list**

Stack Implementation Using a Linked List:

```
Stack Implementation Using a Linked List:

Pop():
if isEmpty()
    stop
    tmp <- top
top <- top.next
tmp2 <- tmp.data
delete tmp
return tmp2
```

Push(x):
```
    tmp <- new ListItem
    tmp.data <- x
    return TRUE
```

IsEmpty():
```
if top = 0
    return TRUE
else
    top <- tmp
    return FALSE
```

One of the benefits of using stack to implement method invocation is that it allows programs to use recursion. Recursion is a powerful method, as it often allows designing simple and efficient programs for fairly difficult problems.

### 1.2 Queues

A queue is a container of objects that are inserted according to the first in first out (FIFO) principle. Objects can be inserted into a queue at any time, but only the element that was in the queue the longest can be removed at any time. We say that elements enter the queue at the rear and are removed from the front.

![Queue Diagram](image)

**Queue ADT**

The queue ADT supports the following two fundamental methods:
- `enqueue(o)` : insert object `o` at the rear of the queue
- `dequeue(o)` : remove and return from the queue the object at the front; an error occurs if the queue is empty

**Queue supporting methods**
- `size()` : return the number of objects in the queue
- `isEmpty()` : return a Boolean value indicating whether the queue is empty
- `front()` : return, but do not remove, the front object in the queue; an error occurs if the queue is empty
Applications of queues

Direct applications
- Waiting lines
- Access to shared resources (e.g., printer)
- Multiprogramming

Indirect applications
- Auxiliary data structure for algorithms
- Component of other data structures

Array implementation of Queue

A queue can be implemented an N element array Q, with elements stored from S[f] to S[r] (mod N). f is an index of Q storing the first element of the queue (if not empty), r is an index to the next available array cell in Q (if Q is not full).

Normal (f ≤ r) configuration (a) and wrap around (f > r) configuration (b)

Queue Methods Complexity

Each of the queue methods executes a constant number of statements. All supporting methods of the queue ADT can be easily implemented in constant time. Thus, in array implementation of queue ADT each method runs in O(1) time.

Queue and Multiprogramming

Multiprogramming is a way of achieving a limited form of parallelism. It allows running multiple tasks or computational threads at the same time. E.g., one thread can be responsible for catching mouse clicks while others can be responsible for moving parts of animation around in a screen canvas. When we design a program or operating system that uses multiple threads, we must disallow an individual thread to monopolise the CPU, in order to avoid application or
applet hanging. One of the solutions is to utilise a queue to allocate the CPU time to the running threats in the round-robin protocol.

1.3 Singly Linked List

A singly linked list is a concrete data structure consisting of a sequence of nodes. Each node stores
- element
- link to the next node

Traversing in only one direction is possible.

![Singly Linked List Diagram]

We can implement a queue with a singly linked list. The front element is stored at the first node. The rear element is stored at the last node. The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time.

List ADT (sequence of elements)

List ADT supports the referring methods:
- first() : return the position of the first element; error occurs if list S is empty
- last() : return the position of the last element; error occurs if S is empty
- isFirst() : return a Boolean indicating whether the given position is the first one
- isLast() : return a Boolean indicating whether the given position is the last one
- before() : return the position of the element in S preceding the one at position p; error if p is first
- after() : return the position of the element in S preceding the one at position p; error if p is first

List ADT supports the following update methods:
- replaceElement(p,e) : p – position, e - element
- swapElements(p,q) : p,q - positions
- insertFirst(e) : e - element
- insertLast(e) : e - element
- insertBefore(p,e) : p – position, e - element
- insertAfter(p,e) : p – position, e - element
- remove(p) : p – position

1.4 Doubly Linked List

A node in a doubly linked list stores two references – a next link, and a prev link which points to the previous node in the list (traversing in two two directions is possible).
Doubly linked list with two sentinel (dummy) nodes header and trailer.

List Update (element insertion)
The pseudo-code for insertAfter(p,e)

```
Algorithm insertAfter(p,e):
    Create a new node v
    v.element ← e
    v.prev ← p  \{link v to its predecessor\}
    v.next ← p.next  \{link v to its successor\}
    (p.next).prev ← v  \{link p's old successor to v\}
    p.next ← v  \{link p to its new successor, v\}
    return v  \{the position for the element e\}
```

List Update (element removal)
The pseudo-code for remove(p)

```
Algorithm remove(p):
    t ← p.element  \{a temporary variable to hold the return value\}
    (p.prev).next ← p.next  \{linking out p\}
    (p.next).prev ← p.prev
    p.prev ← null  \{invalidating the position p\}
    p.next ← null
    return t
```

List Update complexity
What is the cost (complexity) of both insertion and removal update? If the address of element at position p is known, the cost of an update is O(1). If only the address of a header is known, the cost of an update is O(p) (we need to traverse the list from position 0 up to p).

Searching a linked list
The following procedure finds the first element with key \( k \) in list \( L \). It returns a pointer to that element. If no element with key is found, the special pointer NIL is returned.

```
List_Search(L,k)
x := head[L]
while x!=NIL and key[x]!=k do
    x := next[x]
return x
```
It takes $\Theta(n)$ time at most to search a list of $n$ objects (linear search).

### 1.5 Tree

A tree is a structure composed by a collection of interconnected elements. Each element in the tree is called node. A link between two nodes is called edge. Each edge links two nodes. The edges are assumed to be directed, which means that they go from one node to the other. If we have an edge going from node $a$ to node $b$, then we say that the edge is an incoming edge for node $b$; similarly for an edge from node $a$ to node $b$ we say that the edge is outgoing for node $a$.

The nodes are organized into a hierarchical structure. At the top of the tree lies a distinguished node called root of the tree. The root is the only node, which does not have incoming edges. Each other node in the tree has exactly one incoming edge. Each node can have an arbitrary number of outgoing edges.

![Diagram of a tree](image)

If $n$ is a node, and this is connected to the nodes $n_1 \ldots n_k$ via outgoing edges, then $n_1 \ldots n_k$ are all children of $n$. Nodes $N_1; N_2; N_3$ are children of node $N$. In turn, $N$ is called the parent of $N_1; N_2; N_3$. $N$ is also in this case the root of the tree (as $N$ does not have any incoming edge).

Observe that, since each node (except for the root) has exactly one incoming edge, then this means that each node in the tree (with the exception of the root) has exactly one parent. The nodes in the tree, which do not have any outgoing edges, are called leaves of the tree. Node $N_2$ is a leaf, as it does not have any edge leaving the node. A node, which is neither a root nor a leaf, is called internal. Thus internal nodes are all those nodes that have a parent and at least one child.

Nodes in the tree can be arranged in levels. The root is the only node at level 0. Level 1 contains all the children of the root node. Level 2 contains all the children of the nodes at level 1, and so on.
The height of a tree is the number of different levels that are present in the tree. Any node \( n \) in a tree can be reached starting from the root and following the edges. The sequence of edges traversed to go from the root to a given node \( n \) represent a path in the tree. In each tree there is exactly one path from the root to any given node. According to this definition, it is easy to see that if a node \( n \) belongs to the level number \( i \) of a tree, then this means that the unique path from the root to node \( n \) contains exactly \( i \) edges. Also, according to this we can redefine the notion of height of a tree as follows: the height of a tree is the length of the longest path from the root present in the tree. Children of a given node are ordered from left to right; for example \( N_1 \) is the first child, \( N_2 \) is the second child, and \( N_3 \) is the third child of \( N \).

Another important concept is that of subtree. Given a tree and a node \( n \) in the tree, the set of all nodes which have \( n \) as ancestor represents a tree structure; this is called the subtree rooted in \( n \). This is very important, as it allows to provide a recursive construction of trees: a tree is either composed by zero nodes or it is composed by a node (the root) which has a number of children, and these children in turn represent other (smaller) trees.

![Diagram](image)

### 1.6 Binary Trees

Given a tree, in the definition given before we did not place any restriction on the number of children each node can have. For an arbitrary tree, the degree of the tree is the maximum number of children that each node is allowed to have in the tree. A commonly used type of tree is the Binary Tree. A Binary tree is characterized by the fact that each node can have at most two children (i.e., the degree of the tree is two). When dealing with binary trees, we identify the first child of a node as the left child and the second child as the right child. Alternatively, if we use the recursive decomposition of trees described earlier, we can view a binary tree as either an empty tree (i.e., a tree with zero nodes), or as composed by a root element and the remaining nodes are partitioned in two binary trees (which are called the left and right subtree of the original tree). For example, if we consider the tree representing an arithmetic expression, this is commonly a binary tree. In the figure, the root is the node +, and the figure identifies the left and right subtree. These are, in turn, binary trees as well.

![Binary Tree Diagram](image)
Properties

- If we consider a binary tree containing \( n \) nodes, then this tree contains exactly \( n - 1 \) edges;
- A binary of height \( h \) has a number of nodes which is greater or equal than \( h \) and less or equal than \( 2^h - 1 \);
- If we have a binary tree containing \( n \) nodes, then the height of the tree is at most \( n \) and at least \( \lceil \lg_2(n+1) \rceil \);
- If we consider a binary tree of height \( h \) which has been completely filled (i.e., each node not at level \( h \) has two children), then such tree contains exactly \( 2^h - 1 \) nodes; this tree is called the Complete binary tree of order \( h \).

Implementation Using Linked Structures

The intuition behind this implementation is to organize the elements in the tree as a collection of structures linked to each other’s. Each node in the tree is represented by a separate structure, and each node contains pointer to its children (two pointers at most, since we are dealing only with binary trees). If one desires to have access to the parent of a node, then an additional pointer can be maintained in each node, which points to the parent of that node. An empty tree is represented by a NULL pointer.

Implementation using Arrays

Representing linear structures (lists, stacks, etc.) using arrays was a quite natural operation, considering that the array itself is an inherently linear organization (a sequence of variables). Trees are not linear. Thus representing them using arrays requires a proper mapping from the tree structure into a linear organization. In general representing trees using arrays is feasible (without excessive complications) only if we know a priori what is the maximum number of edges coming out of each node. Binary trees fall in this class, as we know that each node is going to have at most two successors. Consider the tree in Figure. The nodes in the tree has been numbered consecutively, starting from 1 (root), 2 and 3 have been assigned to the children of the root, 4 . . . 7 have been used for the grandchildren of the root and so on.

The intuition is that since

- the first \( k \) levels of the tree can contain at most \( 1 + 2 + 4 + \ldots + 2^{k-1} \) nodes, i.e., \( 2^k - 1 \) nodes;
- the level \( k \) of the tree will contain at most \( 2^k \) nodes;
then we can conclude that by assigning a consecutive number to each node, moving down level by level, the nodes at level $k$ will use the numbers from $2^k$ to $2^{k+1} - 1$.

This is suggesting an easy idea for representing trees as arrays; we consider the above numbering scheme, and if a node is assigned node $j$, then we just simply store that node in the location $a[j]$ of the array. Thus, if we are positioned in a node which is lying in position $i$ of the array, then we can easily identify:

- the left child will be in position $2 \times i$;
- the right child will be in position $2 \times i + 1$;
- its parent (if the node is not the root) is in position $i / 2$;
- the root is in position 1

We assume that if a node does not exist, then the corresponding entry in the array is filled with a default value. The major problem is that the array required to store even a relatively small tree can be very large. If we consider the tree containing $k$ nodes $n_1; n_2; \ldots; n_k$ organized as follows: $n_1$ is the root, $n_2$ is the right child of $n_1$, $n_3$ is the right child of $n_2$, $\ldots$, $n_k$ is the right child of $n_{k-1}$, then we can easily see that we will need an array of size $2^k - 1$ (e.g., for $k = 11$ we need an array with $2^{10} = 1024$ positions—thus we have 1024 positions used to store a tree which contains only 11 nodes...).

**Traversing Trees**

A typical problem is that of traversing a binary tree, i.e., scanning all the nodes of the tree in a predefined order. We can start by distinguishing two forms of tree traversal:

1. **Depth-first traversal**: the processing proceeds along a path from the root to one child to the most distant descendant of that first child before processing a second child. I.e., all successors of a child are processed before moving to another child.

2. **Breadth-first traversal**: the processing proceeds horizontally from the root to all of its children, then to its children’s children and so forth until all nodes have been processed.

**Depth-First Traversal**

There are three typical orders for (depth-first) traversing a tree:

- **PreOrder**;
- **PostOrder**;
- **InOrder**

Consider the example of the Figure. The tree contains the description of an arithmetic expression. Let us consider the binary tree to have base type char (i.e., the value in each node of the tree is a single character).

PreOrder: Visiting a tree in preorder involves exploring the tree as follows:

1. first of all visit the root of the tree (and read its content);
2. visit in PreOrder the left subtree of the root;
3. visit in PreOrder the right subtree of the root;

In the example of Figure, a PreOrder visit will thus produce the following sequence of values:

```
/    ^ b 2 * * 4 a c * 2 a
```
PostOrder: Visiting a tree in postorder involves exploring the tree as follows:
1. first visit the whole left subtree in PostOrder;
2. then visit the whole right subtree in PostOrder;
3. the visit the root of the tree;

In the example of Figure, a PostOrder visit will thus produce the following sequence of values:
\[ b \ 2 \ ^\ 4 \ a \ * \ c \ * \ 2 \ a \ * \ / \]

InOrder: Visiting a tree in inorder involves exploring the tree as follows:
1. first visit the whole left subtree in InOrder;
2. the visit the root of the tree;
3. then visit the whole right subtree in InOrder;

In the example of Figure, an InOrder visit will thus produce the following sequence of values:
\[ b \ \ 2 \ - \ 4 \ * \ a \ * \ c \ / \ 2 \ * \ a \]

Breadth-First Traversal

In the breadth-first traversal of a binary tree, we process all of the children of a node before proceeding with the next level. In other words, given a root at level \( n \), we process all nodes at level \( n \) before proceeding with the nodes at level \( n + 1 \). While depth-first traversal are made according to the recursive structure of a tree (i.e., a tree is a root with two other trees attached below), breadth-first is instead ignoring this structuring and cutting the tree horizontally by levels. This implies that recursive solutions are not going to help in writing a breadth-first procedure. The order in which nodes are considered matches well with the FIFO order provided by a queue (in which nodes at level \( n \) are inserted in the queue before any node of level \( n + 1 \); as consequence all nodes of level \( n \) are going to be encountered, while dequeueing the queue before encountering any node of level \( n + 1 \)). The breadth-first traversal of the tree in figure will produce
\[ \ * \ ^\ 2 \ a \ b \ 2 \ * \ c \ 4 \ a \]

1.7 Binary Search Trees

A binary search tree is a data structure, which is used to store information identified by unique keys. Each piece of information is uniquely associated to a key. You can think about keys as non-negative integer numbers. The problem is to maintain the set of data in such a way to allow fast retrieval of information; given a key we would like to quickly discover whether such key has
been used, and in this case we want to extract the corresponding piece of information. Binary search trees are commonly used to solve this problem. The keys are stored in the nodes of a binary tree (we can assume the existence of an external table which maps each key to the corresponding data). Keys are inserted in the tree according to the following rule:

for each node $N$ (containing the key $k$) in a binary search tree:

- all the keys in the left subtree of $N$ have value smaller than $k$;
- all the keys in the right subtree of $N$ have value bigger than $k$.

In order to search a key in the tree, we need to move left or right depending on the result of the comparison between the key and the root of the current subtree.

```c
void Search(Tree t, int Key) {
    if (EmptyTree(t))
        printf(`Key not present``);
    else
        if (GetRoot(t) == Key)
            printf(`FOUND!!!``);
        else
            if (RootInfo(t) > Key)
                Search(GetLeft(t),Key);
            else
                Search(GetRight(t),Key);
}
```

The more complex operation on binary search trees is the deletion of a key. Since we do not place any restrictions on the insertion of elements (any key can be inserted in the tree), then we should also not place any restriction on which key can be removed. This may potentially lead to complications, since we may be trying to remove a node which has successors, thus leaving a “hole” in the tree. E.g., if we try to remove node 15 from the tree in previous Figure, we need to make sure that we are not losing access to node 17.
There are four possible cases, which may occur:

1. The node to be removed does not have any child; in this case the node can be safely removed without endangering access to other nodes in the tree; we should only make sure that the connection between the node and its parent is removed.

2. If the node has a right subtree but no left subtree, then we can simply attach the right subtree to the node’s parent (see next Figure (i)).

3. If the node has a left subtree but no right subtree, then we can simply attach the left subtree to the node’s parent;

4. If the node to be deleted has both left and right subtrees, then the simple way to proceed is the following (see also next Figure (ii)):
   - Find the largest node in the left subtree of the node to be deleted;
   - Copy such value in the node to be deleted;
   - Delete the largest node from the left subtree;

Note that the last step is a recursive call to the procedure which deletes nodes from a binary search tree. Nevertheless, in this recursive case we are guaranteed that the node to be removed does not have a right subtree (i.e., it will be either case 1 or case 3 above).

Cost of Searching

If we consider searching one key in a given tree:

Best Case: The best case is the one in which we search for a given key and this key is present in the root of the tree. One comparison is sufficient to produce the result.

Worst Case: the tree is completely unbalanced and is reduced to a straight sequence of nodes. In this case, if there are \( n \) nodes in the tree, then we may take up to \( 2 \times n \) comparisons to produce an answer.
2 Hashing

Hashing is the transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find the item using the shorter hashed key than to find it using the original value. It is also used in many encryption algorithms.

As a simple example of the using of hashing in databases, a group of people could be arranged in a database like this:

Abernathy, Sara
Epperdingle, Roscoe
Moore, Wilfred
Smith, David
(and many more sorted into alphabetical order)

Each of these names would be the key in the database for that person's data. A database search mechanism would first have to start looking character-by-character across the name for matches until it found the match (or ruled the other entries out). But if each of the names were hashed, it might be possible (depending on the number of names in the database) to generate a unique four-digit key for each name. For example:

7864   Abernathy, Sara
9802   Epperdingle, Roscoe
1990   Moore, Wilfred
8822   Smith, David
(and so forth)

A search for any name would first consist of computing the hash value (using the same hash function used to store the item) and then comparing for a match using that value. It would, in general, be much faster to find a match across four digits, each having only 10 possibilities, than across an unpredictable value length where each character had 26 possibilities.

The hash function is used to index the original value or key and then used later each time the data associated with the value or key is to be retrieved. Thus, hashing is always a one-way operation. There's no need to "reverse engineer" the hash function by analysing the hashed values. In fact, the ideal hash function can't be derived by such analysis. A good hash function also should not produce the same hash value from two different inputs. If it does, this is known as a collision. A hash function that offers an extremely low risk of collision may be considered acceptable.

Here are some relatively simple hash functions that have been used:

- The division-remainder method: The size of the number of items in the table is estimated. That number is then used as a divisor into each original value or key to extract a quotient and a remainder. The remainder is the hashed value. (Since this method is liable to produce a number of collisions, any search mechanism would have to be able to recognize a collision and offer an alternate search mechanism.)
- Folding: This method divides the original value (digits in this case) into several parts, adds the parts together, and then uses the last four digits (or some other arbitrary number of digits that will work) as the hashed value or key.
- Radix transformation: Where the value or key is digital, the number base (or radix) can be changed resulting in a different sequence of digits. High-order digits could be discarded to fit a hash value of uniform length.
• Digit rearrangement: This is simply taking part of the original value or key such as digits in positions 3 through 6, reversing their order, and then using that sequence of digits as the hash value or key.

A hash function that works well for database storage and retrieval might not work as for cryptographic or error-checking purposes.

Implementation of Hashing

1. Direct addressing
2. Hashing with chaining
3. Open addressing

2.1 Direct Addressing

Direct addressing is one of the simpler and quicker methods of look up used. We can search for an element in \( O(1) \) time. One position for each element in the array for every possible value of keys. In direct addressing, \( x \) is stored at position \( \text{key}[x] \) in a table \( T \). The table \( T \) has indices from 0 to \( m-1 \), where \( m-1 \) is the maximal key value. When the key doesn't exist, a nil is assigned to that position in the table.

The standard operations insert, delete and search take one time unit.

Dictionary operations for example:

- Direct-address search \( (T, k) \) return \( T[k] \)
- Direct-address Insert \( (T, x) \) \( T[\text{key}[x]] \leftarrow x \)
- Direct Address delete\( (T, x) \) \( T[\text{key}[x]] \leftarrow \text{nil} \).

If all operations are so efficient, why is direct addressing not used all the time? There are fundamentally two problems:

1. In some applications, \( \text{key}[x] = \text{key}[y] \) even though \( x \neq y \). This is called a collision, and warrants special treatment.
2. Sometimes, \( n \), the number of data items, is much less than \( m \). A case in point is a database of phone numbers with 100 entries, where the key is the phone number.
Clearly, there are no collisions, yet $m = 10,000,000 >> 100 = n$, so most of remainders unused.

### 2.2 Hashing with Chaining

Hashing with chaining is an application of linked lists and gives an approach to collision resolution. In hashing with chaining, the hash table contains linked lists of elements or pointers to elements. The lists are referred to as chains, and the technique is called chaining. This is a common technique where a straightforward implementation is desired and maximum efficiency isn't required. Each linked list contains all the elements whose keys hash to the same index. Using chains minimizes search by dividing the set to be searched into lots of smaller pieces. There's nothing inherently wrong with linear search with small sequences; it's just that it gets slower as the sequences to be searched get longer. In this approach to resolving collisions, each sequence of elements whose keys hash to the same value will stay relatively short, so linear search is adequate.

This Hash Table is an array $[0 .. m-1]$ of linked list. The table entries are called buckets or slots, and the linked lists are called chains. $x$ is placed in linked_list $h[x]$ (number of hash function).

**Analysis**

Here we analyse the number of probes in a table for doing "search". A probe is a check of a table entry. All other components of the algorithm are ignored, but as they are collectively bounded by the number of probes, it suffices to just consider probes.

$n =$ input size, $m=$ table size, and $a =$ LOAD FACTOR = $n / m$

- **WORST CASE** : (Number of probes for search /insert) 
  $T=$ Theta(n) (all elements are in 1 linked list)

- **AVERAGE CASE** : Assume $h[key[x]]$ spreads input independently and randomly over $m$ buckets.
  1. "UNSUCCESSFUL SEARCH": When the element searched is not in the table. 
     Number of probes = length of the linked list $h[k]$. Expected time = $n/m = a (+1)$. 
     Where 'a' is $n/m$ and '+1' is to check NIL. And $k$ is uniformly and randomly distributed over all values in the table.
  2. SPACE: Space is relative to input size. 
     SPACE / Input size = $(m+n)/n = 1 + 1/a$
3. "SUCCESSFUL SEARCH": When the element searched figures in the table.

\[ ti = \frac{1 + (i - 1)}{m} \]

Number of probes = 1 + (i - 1) / m. Where (i - 1) / m comes from the time needed to insert the i-th element in a table with i-1 element. By the unsuccessful search part, the expected time is (i-1) / m.

On average, the time to find an element in the table is

\[
\frac{1}{n} \sum(1 + \frac{i - 1}{m}) = 1 + \frac{n(n-1)}{2nm} \leq 1 + \frac{a}{2}.
\]

Provided that \( \frac{1}{2} < a < 2 \), O(1) is the average time required for search/insert/delete operations.

### 2.3 Hashing by Open Addressing

Open addressing hashing works on the same principle as other types of hashing; that is, it uses a hashing function \( h \) and the key of a datum \( x \) to map \( x \) to \( h[\text{key}(x)] \), and the values are stored in a table. The difference is in what is stored in the table. Instead of maintaining a pointer to a linked list, the table contains the actual values of \( \text{key}(x) \).
Implementation

The table is $T[0..m-1]$ where $m$ is the table size, and the input is $x_1,\ldots,x_n$. The hash function $h$ is given.

We will insert $x_i$ in slot $h[\text{key}[x_i]]$ in $T$. However, if the slot is occupied, we will try the next slot, until an empty slot is found.

**Example:** Say we are given a table with 10 slots. Let our hash function be the last digit of $X$. Now we input the values (in order) 19, 33, 43, 53, 21.

**INSERT:**

We apply our hash function to the first value: the last digit is 9, so 19 goes in slot 9. Next is 33: this goes in slot 3. Now we come to a problem: the next number, 43 should also go into slot 3. We have no linked lists, so where does it go? In open addressing, a value that is slated for an occupied slot goes into the next empty slot. 43 then goes into slot 4. 53: slot 3 is filled, go to slot 4. Slot 4 is filled, go to slot 5. Slot 5 is empty, so 53 goes into slot 5.

The table is usually made circular; so that a value meant to be inserted into the last slot (which is occupied) is sent around to the front of the list. So if we were to insert 99 into the table, we see that slot 9 is occupied, and 99 is sent to slot 0, which is empty.

Note that if we have more values entered than slots, we can run out of room in the table, and be unable to make any more insertions. For this reason, the load factor $\alpha$ in open addressing can never exceed 1.0.

**DELETE:**

Say we want to delete 43. First we go to slot 3: It's occupied, but not by 43. So we go on to the next occupied slot. We continue to do this until we find either the number we're looking for (success), an empty slot or we arrive back where we started. In this case, the very next slot is occupied by 43, so we remove it and mark the slot as deleted. (We'll see why in a moment.)

**SEARCH:**

Now we will search for 53. We check the first slot indicated, 3. It's occupied, but not by 53, so we move on. Now we come slot 4, which is empty. So we should stop, right? If we hit an empty slot, we know 53 can't be beyond this slot. This is why we marked slot 4 "deleted". We know there was something here, so 53 can be beyond this point. We move on to the next slot, and we have located 53.

**Probing and the Probe Sequence**

Probing is simply another word for the sequence we used for our search and insert routines. We "probe" the table until we find an available slot. Due to the (usually) circular nature of the table
we have to limit the number of slots the algorithm will examine. Since there are only m slots in the table, we limit the algorithm to m comparisons.

The probe sequence is the permutation of 0,1,...,m-1 used for searching for an available slot when inserting an element. For a key k the sequence is denoted by:

\[ h(k, 0), h(k, 1), ..., h(k, m-1) \]

Thus, h(k, 0) is what we used to call h(k). The probe sequence used in the simple introductory example is simply

\[ h(k), h(k) + 1, ..., h(k) + m - 1 \]

all mod m (to force the circular inspection of the array). Clearly, this is a permutation, but not a very good one.

**Linear Probing**

In linear probing, the slots in the has table are probed in a linear fashion, in order (ie. check slot 4, then slot 5, then 6, then 7,...etc). We express linear probing as

\[ h(k,i) = [h(k) + c*i] \mod m \]

where c is an integer. While the slots are probed in a linear fashion, they are not necessarily probed in exact order. Note that we use mod m to account for the (usually) circular nature of the table. One of the problems with linear probing is called primary clustering. In this, "blocks" of occupied slots tend to accumulate; the larger a block is, the faster it grows, by the principles of probability. This increases the average search time in the table.

**Random Probing**

Random probing makes use of a pseudo-random function to "jump around" in the hash table. The random probe sequence is

\[ h(k,i) = (h[k] + d(i)) \mod m \]

Where d(0), ..., d(m-1) is a permutation of 0, ..., m - 1. Here, d(i) is an integer, but it is generated uniquely by a recursive function for each i. The recursive definition of d is

\[ d(0) = 0 \]

\[ d(i+1) = [a*d(i) + 1] \mod m \]

where a is a carefully picked integer. The way a is chosen is to insure each number [0,...,m-1] eventually appears as d(i). It is known that such a recursively defined sequence is a permutation of 0, 1, ..., m - 1 if and only if a - 1 is a multiple of every prime divisor of m, where 4 is considered as prime.

**Example: m = 10, a = 3.** We get the sequence for d = 0 1 4 3 0 1 4 3 0 1 4 3 0... This probe sequence is not a permutation. The prime divisors of 10 are 2 and 5, so (a-1) must be a multiple
of \(2 \times 5 = 10\). If we choose \(a = 11\), then, we get the sequence \(0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 0 \ 1 \ 2 \ldots\) so \(a = 11\) is a reasonable choice.

**Example: \(m = 100\).** Note that \(m\) has prime divisors 2, 4 and 5. To have a permutation, we must pick \(a - 1\) as a multiple of \(4 \times 5 = 20\). Thus, \(a\) could be 1, 21, 41, 61, 81, 101, and so fourth.

Note: The name "random probing" is a bit of a misnomer, since the sequence is a deterministic one. But to an untrained eye, the sequence appears random, so the name stuck.

**Double Hashing**

Double hashing makes use of two functions to probe the table. This is usually the best method, since it has many of the characteristics of a random permutation. We express double hashing as

\[
h(k,i) = [h(k) + i \cdot h2(k)] \mod m
\]

where \(h2\) is an auxiliary hash function. Here we must make sure that \(h2\) and \(m\) are relatively prime (i.e. \(\gcd = 1\)) or the probe will only examine part of the table (for the same reasons as random probing, above).

3 **Sorting Algorithms**

A **sorting algorithm** is an algorithm that puts elements of a list into order. Efficient sorting is important to optimizing the use of other algorithms (such as search algorithms and merge algorithms) that require sorted lists to work correctly.

Many common sort algorithms are used in computer science. They are often classified by:

- **Computational complexity** (worst, average and best-case behaviour) in terms of the size of the list \(n\). Typically, good behaviour is \(O(n \log n)\) and bad behaviour is \(O(n^2)\). Sort algorithms which only use an abstract key comparison operation always need at least \(O(n \log n)\) comparisons on average; sort algorithms which exploit the structure of the key space cannot sort faster than \(O(kn)\) where \(k\) is the average key length.

- **Memory usage** (and use of other computer resources)

- **Stability**: a sort algorithm is **stable** if, whenever there are two records \(R\) and \(S\) with the same key and with \(R\) appearing before \(S\) in the original list, \(R\) will appear before \(S\) in the sorted list. (Unstable sort algorithms can usually be made artificially stable by adding an extra number to the key defining the position in the original list.)

Some sorting algorithms follow (\(n\) is the number of objects to be sorted, \(k\) the size of the key space):
<table>
<thead>
<tr>
<th>Name</th>
<th>Worst case complexity</th>
<th>Average case complexity</th>
<th>Best case complexity</th>
<th>Average case memory usage</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$ - already-sorted data</td>
<td>works in-place</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>works in-place, but loses stability or gains complexity</td>
<td>Yes</td>
</tr>
<tr>
<td>Straight insertion sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$ - already-sorted data</td>
<td>works in-place</td>
<td>Yes</td>
</tr>
<tr>
<td>Bucket sort</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$ extra space</td>
<td>Yes</td>
</tr>
<tr>
<td>Counting sort</td>
<td>$O(n+k)$</td>
<td>$O(n+k)$</td>
<td>$O(n+k)$</td>
<td>$O(n+k)$ extra space</td>
<td>Yes</td>
</tr>
<tr>
<td>Heap sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$ extra space</td>
<td>No</td>
</tr>
<tr>
<td>Merge sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$ extra space</td>
<td>Yes</td>
</tr>
<tr>
<td>Quick sort</td>
<td>$O(n^2)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>works in-place, needs $O(\log n)$ auxiliary storage</td>
<td>No</td>
</tr>
<tr>
<td>Binary tree sort</td>
<td>$O(n^2)$ -- already sorted data</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$, typically several pointers per input item</td>
<td>Yes</td>
</tr>
<tr>
<td>Radix sort</td>
<td>$O(n \log k)$</td>
<td>$O(n \log k)$</td>
<td>$O(n \log k)$</td>
<td>$O(n)$ extra space</td>
<td>Yes</td>
</tr>
<tr>
<td>Shell sort (decreasing gap insertion sort)</td>
<td>$O(n^{1.5})$</td>
<td>$O(n^{1.25})$</td>
<td>$O(n \log n)$ - already-sorted data</td>
<td>works in-place</td>
<td>No</td>
</tr>
</tbody>
</table>

### 3.1 Bubble sort

**Bubble sort** is a simple sorting algorithm. It takes a lot of passes through the list to be sorted, comparing two items at a time, swapping these two items in the correct order if necessary. Bubble sort gets its name from the fact that the items that belong at the top of the list gradually "float" up there.

Bubble sort needs $\Theta(n^2)$ comparisons to sort $n$ items and can sort in place. It is one of the simplest sorting algorithms to understand but is generally too inefficient for serious work sorting large numbers of elements.

It is essentially equivalent to insertion sort --- it compares and swaps the same pairs of elements, just in a different order. Naive implementations of bubble sort (like those below) usually perform badly on already-sorted lists ($\Theta(n^2)$), while insertion sort needs only $\Theta(n)$ operations in this case. It is possible to reduce the best case complexity to $\Theta(n)$ if a flag is used to denote whether any swaps were necessary during the first run of the inner loop. In this case, no swaps would indicate an already sorted list.

The bubble sort algorithm is:

1. Compare adjacent elements. If the first is greater than the second, swap them.
2. Do this for each pair of adjacent elements, starting with the first two and ending with the last two. At this point the last element should be the greatest.
3. Repeat the steps for all elements except the last one.
4. Keep repeating for one fewer elements each time, until you have no more pairs to compare.

A simple implementation of bubble sort in C programming language:
void bubbleSort(int *array, int length)
{
    int i, j;
    for(i = length - 1; i > 0; i--)
        for(j = 0; j < i; j++)
            if(array[j] > array[j+1]) /* compare neighbouring elements */
            {
                int temp;
                temp = array[j];    /* swap array[j] and array[j+1] */
                array[j] = array[j+1];
                array[j+1] = temp;
            }
}

3.2 Selection Sort

The selection sort is one of the easiest ways to sort data. Rather than swapping neighbours continuously, this algorithm finds the smallest element of the array and interchanges it with the element in the first position of the array. After that it re-examines the remaining elements in the array to find the second smallest element. The element, which is found, is interchanged with the element in the second position of the array. This process continues until all elements are placed in their proper order. The order would be defined by the user (i.e. descending or ascending).

Selection Sort

For each position in the array:

1. Scan the unsorted part of the data.
2. Select the smallest value.
3. Switch the smallest value with the first value in the unsorted part of the data.

Selection sort is \(O(n^2)\) in the best, worst, and expected cases. The selection sort is easy to understand and this makes it easy to implement the algorithm correctly and thus makes the sorting procedure easy to write. However, there are some reasons why programmers refrain from using this algorithm:

1. This program's performance of \(O(n^2)\) would become quite slow on large amounts of data.
2. At present many application databases have sorted lists of data and these lists are updated on a regular basis. Most times these databases are, for the most part, in order. An ideal method would be able to recognize this fact and would work only on the unsorted items. The selection sort is unable to do this.

Here is an example (the | divides the sorted region from the unsorted):

| 3 5 2 6 4   | ( 2 is the smallest, swap with 3) |
| 2 3 5 6 4   | ( 3 is the smallest, swap with 5) |
| 2 3 4 5 6   | ( 4 is the smallest, swap with 5) |
| 2 3 4 5 6   | ( 5 is the smallest, swap with 6) |
| 2 3 4 5 6   | ( 6 is the only value left, done) |

3.3 Insertion sort

Insertion sort is a simple sort algorithm where the result is built up one entry at a time.
In abstract terms, each iteration of an insertion sort removes an element from the input data, inserting it at the correct position in the already sorted list, until no elements are left in the input. The choice of which element to remove from the input is arbitrary.

Sorting is done in-place. The result array after \( n \) iterations contains the first \( n \) entries of the input array and is sorted. In each step, the first remaining entry of the input is removed, inserted into the result at the right position, thus extending the result:

```
+------ result ------+------ input ------+
|   <= x   |   > x   | x |      ...      |
+--------------------+-------------------+
```

becomes:

```
+-------- result --------+---- input ----+
|   <= x   | x |   > x   |      ...      |
+------------------------+---------------+
```

with each element \( > x \) copied to the right as it is compared against \( x \).

The algorithm can be described as:

1. Start with the result being the first element of the input.
2. Loop over the input until it is empty, "removing" the first remaining (leftmost) element.
3. Compare the removed element against the current result, starting from the highest (rightmost) element, and working left towards the lowest element.
4. If the removed input element is lower than the current result element, copy that value into the following element to make room for the new element below, and repeat with the next lowest result element.
5. Otherwise, the new element is in the correct location; save it in the cell left by copying the last examined result up, and start again from (2) with the next input element.

Insertion sort is very similar to bubble sort. In bubble sort, after \( N \) passes through the array, the \( N \) largest elements have bubbled to the top. (Or the \( N \) smallest elements have bubbled to the bottom, depending on which way you do it.) In insertion sort, after \( N \) passes through the array, you have a run of \( N \) sorted elements at the bottom of the array. Each pass inserts another element into the sorted run. So with bubble sort, each pass takes less time than the previous one, but with insertion sort, each pass may take more time than the previous one.

In the best case of an already sorted array, this implementation of insertion sort takes \( O(n) \) time: in each iteration, the first remaining element of the input is only compared with the last element of the result. It takes \( O(n^2) \) time in the average and worst cases, which makes it impractical for sorting large numbers of elements. However, insertion sort's inner loop is very fast, which often makes it one of the fastest algorithms for sorting small numbers of elements, typically less than 10 or so.

### 3.4 Bucket Sort

- Assumption: input is \( n \) reals from \([0, 1)\), i.e. \( a_i \) is \( U [0, 1) \)
- Basic idea:
  - Create \( n \) linked lists (buckets) to divide interval \([0,1]\) into subintervals of size \( 1/n \)
  - Add each input element to appropriate bucket and sort buckets with insertion sort
- Uniform input distribution? \( O(1) \) bucket size
  - Therefore the expected total time is \( O(n) \)
Bucket-Sort (A)
1. \( n \leftarrow \text{length}[A] \)
2. for \( i \leftarrow 1 \) to \( n \)
3. do insert \( A[i] \) into list \( B[\lceil n \times A[i] \rceil] \)
4. for \( i \leftarrow 0 \) to \( n - 1 \)
5. do sort list \( B[i] \) with insertion sort
6. concatenate the lists \( B[0], B[1] \ldots B[n-1] \) together in order

**Figure** The operation of **BUCKET-SORT**. (a) The input array \( A[1..10] \). (b) The array \( B[0..9] \) of sorted lists (buckets) after line 5 of the algorithm. Bucket \( i \) holds values in the interval \( [i/10, (i + 1)/10] \). The sorted output consists of a concatenation in order of the lists \( B[0], B[1], \ldots, B[9] \).

Note that \( \lceil nA[i] \rceil \rightarrow 0, 1, \ldots, n-1 \) when \( 0 < A[i] < 1 \). Also note that this is just a "hash sort" algorithm with \( \lceil nA[i] \rceil \) as the hash function.

**Running time:**

Everything except the little insertion sorts are \( O(n) \). Assume that \( B[i] \) has \( n_i \) elements, then the worst-case of insertion sort is \( O(n_i^2) \). Expected time to sort is \( O(\sum E[n_i^2]) \), and so all together:

\[
\sum_{i=0}^{n-1} O(E[n_i^2]) = O\left[ \sum_{i=0}^{n-1} E[n_i^2] \right] = O(n)
\]

What is : \( i^{th} \) bucket is chosen with \( p_i = 1/n \). Do this \( n \) times. \( n_i \) is a binomial with \( p = 1/n \) and \( E[n_i] = np = n \cdot 1/n = 1 \), \( \text{var}[n_i] = np(1 - p) = 1 - 1/n \). \( E[n_i^2] = \text{var}[n_i] + E^2[n_i] = 1 - 1/n + 1^2 = 2 - 1/n = O(1) \)
3.5 Counting Sort

Assumptions
- Each of the $n$ input elements is an integer in the range $[1..r]$
- (i.e., $1 \leq A[i] \leq r$, $i \leq n$)
- $r = O(n)$ ($r \leq cn$)
  (e.g., if $n = 100$, then $r$ can be equal to 100, 200 but not 1002)

Basic idea
- For each input element $x$, find the number of elements $\leq x$
- Place $x$ directly in the correct order

- Finding the number of elements $\leq A[i]$, for $1 \leq i \leq n$
  (1) Find the number of times $A[i]$ appears in $A$, $1 \leq i \leq n$
  (1a) Allocate $C[1..r]$ (histogram)
  (1b) For $1 \leq i \leq n$, $+ + C[A[i]]$;

```
input array  A:  3  6  4  2  5  8  10
   x=5, number of elements <= 5 = 4  (3,4,2,5)
input array  B:   

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

put 5 here !!!
```

```
input array  A:  3  6  4  1  3  4  1  4
allocate C

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

i=1, A[1]=3

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>


<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>


<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

i=8, A[8]=4

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

$C[i] = \text{number of times element } i \text{ appears in } A$
(2) Find the number of elements \( \leq A[i] \), \( 1 \leq i \leq n \)

(2a) Compute cumulative sums

---

**Algorithm**

**COUNTING-SORT** \((A,B,r)\)

```
for i ← 1 to r do
    C[i] ← 0
for j ← 1 to length[A] do
    if C[A[j]] < C[A[j]] + 1
    for i ← 2 to r do
        C[i] ← C[i] + C[i-1]
        for j ← length[A] downto 1 do
            if B[C[A[j]]] < A[j]
                B[C[A[j]]] ← A[j]
                C[A[j]] ← C[A[j]] - 1
print B
```

Total time: \( \Theta(r) + \Theta(n) = \Theta(r + n) = \Theta(n) \) if \( r = O(n) \)

**Comments**

- Counting sort is not an in place sort
- Counting sort is stable (elements with the same value appear in the output array in the same order they do in the input array)
3.6 Heapsort

Heaps
- A binary tree which is filled at all levels (complete binary tree) except possibly at the lowest level (filled from left to right).
- *(Heap property)* The value stored in a node is greater than or equal to the values stored at its children nodes.

![Heap Tree Example]

Largest element
- It is always stored at the root!

Height of a Heap
• Proposition: A heap $H$ storing $n$ keys has height $h = \lceil \log(n+1) \rceil$
• Justification: Due to $H$ being complete, we know:
  - # $i$ of nodes is at least:
    $$1 + 2 + 4 + \ldots + 2^{h-1} + 1 = 2^h - 1 + 1 = 2^h - 1$$
  - # $i$ of nodes is at most:
    $$1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$$
  - Therefore:
    $$2^{h-1} \leq n \leq 2^h - 1$$
  - Which implies that:
    $$\log(n + 1) \leq h \leq \log n + 1$$
  - Which in turn implies:
    $$h = \lceil \log(n+1) \rceil$$

Removing/Adding a node
- New nodes are inserted at the bottom level (left to right)
- Nodes are removed from the bottom level (right to left)

Implementing heaps using arrays
- Since there are no nodes at level $l$ unless level $l - 1$ is completely filled, a heap can be stored in an array level by level (beginning with the root), left to right within each level.
The root is always stored at A[1]

\[ PARENT(i) = \lfloor i/2 \rfloor \]

\[ LEFT\_CHILD(i) = 2i \]

\[ RIGHT\_CHILD(i) = 2i + 1 \]

Restoring the heap property
- **Heapify(A, i)** rearranges the keys so that the heap property is valid

```plaintext
Heapify(A,i)  
l <-- LEFT(i)  
r <-- RIGHT(i)  
    then largest <-- l  
    else largest <-- i  
    then largest <-- r  
if largest != i  
    then exchange A[largest] --> A[i]  
    Heapify(A,largest)
```

- Running time of **Heapify()**

  It traces a path from the root to a leaf (longest path length: \(d\))
  At each level, it makes exactly 2 comparisons
  Total number of comparisons is \(2d\)
  Running time is \(O(d)\) or \(O(lgn)\)

### Building a heap

1. Build the heap structure randomly
2. Apply **Heapify()** (alternative name **Fixheap()**) in a bottom up fashion to restore the heap property

- **Algorithm**

  ```plaintext
  Build-Heap(A)  
  heap-size[A] <-- length[A]  
  for i <-- \([length[A]/2]\) downto 1  
      do Heapify(A,i)
  ```

- An upper bound for **Build-Heap()**’s running time

  \(n/2 \text{ (running time of Heapify())} = n/2 \text{ } O(lgn) = O(nlgn)\)

  (not a tight bound !!!)

- A tight bound for **Build-Heap()**’s running time

  Running time depends on the depth of the subtree **Heapify()** operates on
  Different number of nodes exist at different depths !!!

  \[
  \sum_{i=0}^{d-1} (number \ of \ nodes \ at \ level \ i)(running \ time \ of \ FixHeap()) = \\
  \sum_{i=0}^{d-1} 2^i 2(d-1) = 2^{d+2} - 2d - 4 = 4n - 2lgn - 4 = \Theta(n)
  \]

- Bounding running time using recurrences

  Assume that the binary tree is complete (the proof is slightly more complicated if the binary tree is not complete)
The heapsort algorithm
1. Build the heap
2. Remove the element from the root (max element) and put it in the last position of the array
3. Move the next largest element to the root by calling Heapify() to restore the Heap property

Heapsort(A)
Build-Heap(A)
for i <-- length[A] downto 2
heap-size[A] <-- heap-size[A] - 1
Heapify(A, 1)
- Running time

Building a heap takes $\Theta(n)$ time
The heap strategy takes $O(n \log n)$
The total running time is $O(n \log n)$

- The above bound is actually tight ($\Theta(n \log n)$) !!
The depth of the heap decreases as we remove nodes from the heap. *Heapify*’s running time will depend on the current depth of the heap

\[
\text{Running time: } 2 \sum_{k=1}^{n-1} \lfloor \log k \rfloor
\]

**Upper bound:**

\[
\sum_{k=1}^{n-1} \lfloor \log k \rfloor \leq \sum_{k=1}^{n-1} \log k = \log n + \log 2 + \ldots + \log n \leq \log n + \log n + \ldots + \log n = n \log n
\]

\(O(n \log n)\)

**Lower bound:**

\[
\sum_{k=1}^{n-1} \lfloor \log k \rfloor = \sum_{k=1}^{n/2-1} \lfloor \log k \rfloor + \sum_{k=n/2}^{n-1} \lfloor \log k \rfloor \geq \sum_{k=n/2}^{n-1} \lfloor \log k \rfloor \geq \sum_{k=n/2}^{n-1} \lfloor \log n/2 \rfloor = \\
(n/2) \lfloor \log n/2 \rfloor \geq (n/2)(\log n/2 - 1) = n/2(\log n - \log 2 - 1) = (n/2)\log n - n
\]

\(\Omega(n \log n)\)

Comments

- Heapsort is an in place sort
- Heapsort’s worst case and average case are \(\Theta(n \log n)\)

### 3.7 Mergesort

The sorting algorithm Mergesort produces a sorted sequence by sorting its two halves and merging them. With a time complexity of \(O(n \log(n))\) Mergesort is optimal.

**Idea**

Similar to Quicksort, the Mergesort algorithm is based on a divide and conquer strategy. First, the sequence to be sorted is decomposed into two halves (Divide). Each half is sorted independently (Conquer). Then the two sorted halves are merged to a sorted sequence (Combine) (Figure 1).

![Figure 1: Mergesort(n)](image)

The following procedure *mergesort* sorts a sequence \(a\) from index \(lo\) to index \(hi\).
void mergesort(int[] a, int lo, int hi) {
    if (lo < hi) {
        m = (lo + hi) / 2;
        mergesort (a, lo, m);
        mergesort (a, m+1, hi);
        merge (a, lo, hi);
    }
}

First, index \( m \) in the middle between \( lo \) and \( hi \) is determined. Then the lower part of the sequence (from \( lo \) to \( m \)) and the upper part (from \( m+1 \) to \( hi \)) are sorted by recursive calls of \( \text{mergesort} \). Recursion ends when \( lo = hi \), i.e. when a subsequence consists of only one element. Then the two sorted halves are merged by procedure \( \text{merge} \).

Procedure \( \text{merge} \) is usually implemented in the following way: The two halves are first copied into an intermediate array \( b \). Then the two halves are scanned by pointers \( i \) and \( j \) and the respective next-greatest element at each time is copied back to array \( a \) (Figure 2).

![Figure 2: Merging two sorted halves](image1)

Though conceptually simple, this step is a little bit complicated to program. For it is necessary to watch that the bounds of the halves are not crossed by \( i \) and \( j \). And when scanning of one half is completed, the rest of the other half has still to be processed.

These difficulties are elegantly circumvented by copying the second half of the sequence in opposite order into the intermediate array \( b \). Now the two halves are scanned by \( i \) and \( j \) in opposite directions. Again, the respective next-greatest element at each time is copied back to array \( a \). Copying is completed when \( i \) and \( j \) cross, i.e. when \( i > j \) (Figure 3). Observe that it is not necessary for the pointers \( i \) and \( j \) to stay in their halves.

![Figure 3: Merging of two halves sorted in opposite order](image2)
In the following, the corresponding implementation of procedure \textit{merge} in Java is given.

```java
void merge(int[] a, int lo, int hi)
{
    int i, j, k, m, n=hi-lo+1;
    int[] b=new int[n];       // temporary array

    k=0;
    m=(lo+hi)/2;
    // copy lower half to array b
    for (i=lo; i<=m; i++)
        b[k++]=a[i];
    // copy upper half to array b in opposite order
    for (j=hi; j>=m+1; j--)
        b[k++]=a[j];

    i=0; j=n-1; k=lo;
    // copy back next-greatest element at each time
    // until i and j cross
    while (i<=j)
        if (b[i]<=b[j])
            a[k++]=b[i++];
        else
            a[k++]=b[j--];
}
```

In Java and C++ the short form \( k++ \) stands for \( k=k+1 \); the statement \( a[k++]=b[j--] \) is equivalent to the sequence of statements \( a[k]=b[j]; \ k=k+1; \ j=j-1 \).

\textbf{Analysis}

Procedure \textit{merge} requires \( 2n \) steps (\( n \) steps for copying the sequence to the intermediate array \( b \), another \( n \) steps for copying it back to array \( a \)). The time complexity of \textit{mergesort} is therefore

\[
T(n) = 2n + 2 T(n/2) \quad \text{and} \quad T(1) = 0
\]

The solution of this recursion yields

\[
T(n) = 2n \log(n) \in O(n \log(n))
\]

Thus, the Mergesort algorithm is optimal, since the lower bound for the sorting problem of \( \Omega(n \log(n)) \) is attained.

A drawback is that Mergesort needs an additional space of \( \Theta(n) \) for the temporary array \( b \).

\textbf{3.8 Quicksort}

Quicksort is one of the fastest and simplest sorting algorithms. It works recursively by a divide-and-conquer strategy.

The \textit{divide-and-conquer-strategy} for solving a problem consists of three steps:

1) \textbf{Divide} the problem is decomposed into subproblems
2) **Conquer** the subproblems are solved

3) **Combine** the solutions of the subproblems are recombined to the solution of the original problem

**Idea**

First, the sequence to be sorted \( a \) is partitioned into two parts, such that all elements of the first part \( b \) are less than or equal to all elements of the second part \( c \) (divide). Then the two parts are sorted separately by recursive application of the same procedure (conquer). Recombination of the two parts yields the sorted sequence (combine). Figure illustrates this approach.

The first step of the partition procedure is choosing a comparison element (pivot) \( x \). All elements of the sequence that are less than \( x \) are placed in the first part, all elements greater than \( x \) are placed in the second part. For elements equal to \( x \) it does not matter into which part they come. In the following algorithm it may also happen that an element equal to \( x \) remains between the two parts.

**Algorithm Partition**

**Input:** sequence \( a_0, ..., a_{n-1} \) with \( n \) elements

**Output:** permutation of the sequence such that all elements \( a_0, ..., a_j \) are less than or equal to all elements \( a_i, ..., a_{n-1} \) \( (i > j) \)

**Method:**

choose the element in the middle of the sequence as comparison element \( x \)

let \( i = 0 \) and \( j = n-1 \)

while \( i \leq j \)

search the first element \( a_i \) which is greater than or equal to \( x \)

search the last element \( a_j \) which is less than or equal to \( x \)

if \( i \leq j \)

exchange \( a_i \) and \( a_j \)

let \( i = i+1 \) and \( j = j-1 \)
After partitioning the sequence, Quicksort treats the two parts recursively by the same procedure. The recursion ends whenever a part consists of one element only.

```c
quicksort( void *a, int low, int high )
{
int pivot;
/* Termination condition! */
if ( high > low )
{
pivot = partition( a, low, high );
quicksort( a, low, pivot-1 );
quicksort( a, pivot+1, high );
}
}
```

**Analysis**

The best-case behaviour of the Quicksort algorithm occurs when in each recursion step the partitioning produces two parts of equal length. In order to sort \( n \) elements, in this case the running time is in \( \Theta(n \log(n)) \). This is because the recursion depth is \( \log(n) \) and on each level there are \( n \) elements to be treated (Figure 2 a).

The worst case occurs when in each recursion step an unbalanced partitioning is produced, namely that one part consists of only one element and the other part consists of the rest of the elements (Figure 2 c). Then the recursion depth is \( n-1 \) and Quicksort runs in time \( \Theta(n^2) \).

In the average case a partitioning as shown in Figure is to be expected.

![Recursion depth of Quicksort: a) best case, b) average case, c) worst case](image)

The choice of the comparison element \( x \) determines which partition is achieved. Suppose that the first element of the sequence is chosen as comparison element. This would lead to the worst
case behaviour of the algorithm when the sequence is initially sorted. Therefore, it is better to choose the element in the middle of the sequence as comparison element.

Even better would it be to take the \( n/2 \)-th greatest element of the sequence (the median). Then the optimal partition is achieved. Actually, it is possible to compute the median in linear time. This variant of Quicksort would run in time \( O(n \log(n)) \) even in the worst case.

However, the beauty of Quicksort lies in its simplicity. And it turns out that even in its simple form Quicksort runs in \( O(n \log(n)) \) on the average. Moreover, the constant hidden in the \( O \)-notation is small. Therefore, we trade this for the (rare) worst case behaviour of \( \Theta(n^2) \).

**Proposition:** The time complexity of Quicksort is in 
\[
\Theta(n \log(n)) \quad \text{in the average case} \quad \text{and} \quad \Theta(n^2) \quad \text{in the worst case}
\]

**Conclusions**

Quicksort turns out to be the fastest sorting algorithm in practice. It has a time complexity of \( \Theta(n \log(n)) \) on the average. However, in the (very rare) worst case Quicksort is as slow as Bubblesort, namely in \( \Theta(n^2) \). There are sorting algorithms with a time complexity of \( O(n \log(n)) \) even in the worst case, e.g. Heapsort and Mergesort. But on the average, these algorithms are by a constant factor slower than Quicksort.

### 3.9 Comparison Sorts

A Comparison Sort is a sorting algorithm where the final order is determined only by comparisons between the input elements.

- Insertion Sort, the proper position to insert the current element is found by comparing the current element with the elements in the sorted sub-array.
- In heap sort, the Heapify procedure determines where to place items based on their comparisons with adjacent elements (parent-child) of the tree.
- In merge sort, the merge procedure chooses an item from one of two arrays after comparing the top items from both arrays.
- In Quicksort, the Partition procedure compares each item of the subarray, one by one, to the pivot element to determine whether or not to swap it with another element.

**Summary for Comparison Sorting Algorithms**

<table>
<thead>
<tr>
<th>Sorting Methods</th>
<th>Worst Case</th>
<th>Best Case</th>
<th>Average Case</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>InsertionSort</td>
<td>( n^2 )</td>
<td>( n )</td>
<td>( n^2 )</td>
<td>very fast when ( n &lt; 50 )</td>
</tr>
<tr>
<td>MergeSort</td>
<td>( n \log n )</td>
<td>( n \log n )</td>
<td>( n \log n )</td>
<td>Need extra space; good for external sort</td>
</tr>
<tr>
<td>HeapSort</td>
<td>( n \log n )</td>
<td>( n \log n )</td>
<td>( n \log n )</td>
<td>Good for real-time appl.</td>
</tr>
<tr>
<td>QuickSort</td>
<td>( n^2 )</td>
<td>( n \log n )</td>
<td>( n \log n )</td>
<td>Practical and fast</td>
</tr>
</tbody>
</table>

**Decision Tree Model**

- Each comparison sort algorithm can be viewed abstractly in terms of a decision tree.
- It is a rooted binary tree where internal nodes represent a comparison between two keys and leaves represent an output.
A comparison Sorting algorithm + an input size $n$ ↔ a decision tree.

**The decision tree corresponding to Insertion Sort ($n = 3$)**

![Decision Tree Diagram](image)

**Lower Bound for Comparison-based Sorting**

The decision trees of comparison-based sorting algorithms:

- Each internal node contains a comparison.
- Each leaf contains a permutation. All the leaf nodes produce the same result: a correctly sorted sequence.
- Algorithm execution = a path from the root to a leaf.
- Worst-case number of comparisons = height of tree.
- **Idea**: If we find a lower bound on the height of the decision tree, we will have a lower bound on the running time of any comparison-based sorting algorithm.

**The necessary condition for a Correct Comparison Sorting Algorithm**

Given an input size $n$, there are $n!$ exactly different inputs, so the corresponding decision tree has at least $n!$ leaf nodes to produce correct outputs.

**Height of Decision Tree**

Theorem: Any decision tree that sorts $n$ elements has height $\Omega(n \log n)$

- Suppose the decision tree has height $h$.
- At level $h$, there are at most $2^{h-1}$ nodes.
- The tree must have at least $n!$ leaf nodes
  \[ 2^{h-1} \geq l \geq n! \Rightarrow h \geq \log(n!) + 1. \]
- **Claim**: $\log(n!) = \Theta(n \log n)$.
- $h \geq \Theta(n \log n) \Rightarrow h = \Omega(n \log n)$. 
3.10 Radix Sort

Assumptions
- Elements consist of $d$ digits, represented in radix $k$ (e.g., if $k = 2$ each number is represented in binary form)
  - $d = \Theta(1)$
  - $k = O(n)$
  - Both $d$ and $k$ are parameters to the algorithm and they affect running time

Radix sort belongs to the class of bucket sort algorithms
- Distribution: elements are distributed over a number of buckets
  The number of buckets is equal to the radix $k$
  Elements are distributed by examining a particular field from each element
  This step should take $\Theta(n)$
- Sort buckets: the elements in each bucket are sorted
  Suppose there are $n_i$ elements in bucket $i$
  Sorting $n_i$ numbers will take $S(n_i) = \Theta(n_i lg n_i)$ comparisons (e.g., using Quicksort)
  Assuming $k$ buckets, it takes $\sum S(n_i)$ comparisons
  \[\sum_{i=1}^{k} S(n_i)\]
- Combine buckets:
  This step should take $O(n)$

Running time
- Assume that the elements are distributed uniformly (i.e., $n/k$ elements in each bucket)
- The second step takes
  \[\sum_{i=1}^{k} c(n/k) lg(n/k) = k(c(n/k) lg(n/k)) = cn lg(n/k)\]
- All three steps take
  $\Theta(n) + \Theta(n lg(n/k)) + O(n) = \Theta(n lg(n/k))$
- These three steps are repeated $d$ times (e.g., number of digits)
- The total running time is
  $\Theta(d n lg(n/k)) = \Theta(n lg(n/k))$ since $d = \Theta(1)$
- If $k = O(n)$ (e.g., $k = n/10$), running time is $\Theta(n lg 10) = \Theta(n)$

Radix sort algorithm
1. Allocate $k$ buckets; Set $p = \text{rightmost digit}$
2. Distribute the elements from the array into the buckets by considering the $p$ digit
3. Combine the elements from the buckets into the array, with the elements in the bucket 0, preceding the elements in the bucket 1, preceding the elements in the bucket 2, and so on
4. $p = p - 1$; if $p \geq 1$, go to step 2.
Note: If \( p \) = leftmost, then step 3 cannot be performed before we distribute the elements with respect to all the digits (space demanding)

Very important: The relative order of two elements placed in the same bucket is not changed!

Implementation using a sorting algorithm

1. Sort the numbers with respect to the least significant digit
2. Sort the numbers with respect to the second least significant digit
   
   k. Sort the numbers with respect to the most significant digit

Algorithm

\[
\text{Radix-Sort}(A, d) \\
\text{for } i \leftarrow 1 \text{ to } d \text{ do} \\
\quad \text{use a stable sort to sort array } A \text{ on digit } i \\
\]

Idea: Use counting sort!

Running time

- For each digit, it takes \( \Theta(n + r) \) time where \( 1 \leq r \leq k - 1 \)
- For \( d \) digits, it takes \( \Theta(d(n + r)) \) time
- Since \( r = O(n) \) (\( k = O(n) \)) and \( d = \Theta(1) \), \( \Theta(d(n + r)) = \Theta(n) \)

(if \( n = 10 \) and the largest element is 10⁹ (d=10), the running time is \( \Theta(n^2) \))

How does the choice of radix \( k \) affect running time?

- As \( k \) increases, \( d \) decreases

Consider that \( n = 106, k=2, \) and \( d=64 \)
We need 2 buckets and 64 passes!! (less space, more work)

Consider that \( n = 106, k=16, \) and \( d=16 \)
We need 16 buckets and 4 passes!! (more space, less work)

- As \( k \) increases, the range of each digit increases as well

<table>
<thead>
<tr>
<th>( k )</th>
<th>( \text{digits} )</th>
<th>( \text{range} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0, 1</td>
<td>[0,1]</td>
</tr>
<tr>
<td>3</td>
<td>0, 1, 2</td>
<td>[0,2]</td>
</tr>
<tr>
<td>4</td>
<td>0, 1, 2, 3</td>
<td>[0,3]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10</td>
<td>0, 1, 2, ..., 9</td>
<td>[0,9]</td>
</tr>
</tbody>
</table>

- Remember: Counting sort’s performance depends on the range of digits!!

Comments

- Radix-sort is not in place
- More difficult to write a general purpose version of radix sort (i.e., integers, reals, characters, etc.)
3.11 Shell Sort

Shell sort is named after Donald Shell, who first published an account of this sorting technique. Shell sort is also called the diminishing increment sort, and is a variation on the basic insertion sort algorithm. In insertion sort, comparison is always done between adjacent elements \((a_i, a_{i+1})\), \(i < n\), and a swap occurs if \(a_{i+1} < a_i\). So, at most 1 inversion is eliminated for each comparison done with insertion sort.

The variation used in shell sort, is to avoid comparing adjacent elements until the last step of the algorithm. So, the last step of shell sort is effectively the insertion sort algorithm, but by the time we reach the last step, the list is either already sorted, or mostly sorted, so insertion sort runs in almost linear time (we already saw that the insertion sort algorithm is \(O(n)\) if a list of \(n\) elements is already sorted in ascending order).

Shell sort runs in subquadratic time and works well in practice for lists of several thousand entries (up to 250,000). The algorithm is also very simple to implement, so shell sort is commonly used in practice for sorting basic lists of keys (e.g., integers).

The running time of shell sort is highly dependent on the selection of an increment sequence \(h_t, h_{t-1}, ..., h_1\) that effectively partition the original list into a set of sublists, which are sorted after one pass through the entire list. The next increment is used to partition the list into a new set of sublists, which are again sorted by a single pass through the entire list, and so on until increment \(h_1\), which is equal to 1. So, the final increment means we have just one large list, and so for increment \(h_1\), an insertion sort is done on the list.

Shell sort works better than insertion sort in practice because it “knocks out” at least 1 inversion per comparison, and very often several inversions. When all the inversions are eliminated, the list is sorted, so the real job of a sorting algorithm is to eliminate inversions.

Shell Sort Increments

An exact analysis of the average running time of shell sort has not been completely achieved. However, in practice the algorithm has been shown to run in \(O(n^{3/2})\) or \(O(n(\log n)^2)\), depending on how the increment sequence is defined. A poor choice for the increment sequence will yield \(O(n^2)\) behaviour in the worst case.

The definition of the increment sequence is critical to achieving subquadratic running time. The increment sequence that seems to work well in practice is: 1093, 364, 121, 40, 13, 4, 1 but no one has yet given a strong mathematical argument for why this sequence works better than other sequences. The sequence is defined by \(h_i = (3^i - 1) / 2\) for \(1 \leq i \leq t\).

This starting element of the sequence, \(h_t\), is easy to compute using a simple loop:

```c
int h = 1;
while (h <= n)
    h = 3 * h + 1;
```

At the end of this loop, \(h\) has the highest increment in the sequence, \(h_t\), which is less than or equal to the size of the list \(n\). We use this increment to make the first partition of the list into \(h\) sublists, each with approximately \(n/h\) entries, and sort each sublist. We then need to compute \(h_{t-1}\), generate a new partition of the list, and sort the resulting sublists, and so on until \(h_1\), which means \(h = 1\).
It is not necessary to store the entire increment sequence at the start of the program, as the increments can be computed in reverse, using: \( h = (h-1) / 2; \)

The Shell Sort Algorithm in C++

We use a template function that takes an array of any type T, containing n elements.

```cpp
template<class T>
void shellSort (T a[], int n)
{
    int h = 1;
    while ( h <= n)
    h = 3 * h + 1; // compute first increment past the starting increment
    do {
        h = (h - 1) / 3; // compute the diminishing increment
        for (int i = h; i < n; i++) {
            T v = a[i];
            for (int j = i; a[j-h] > v && j >= h; j -= h)
                a[j] = a[j-h];
            a[j] = v;
        }
    } while (h != 1);
}
```

Note that when \( h == 1 \), the algorithm makes a pass over the entire list, comparing adjacent elements, but doing very few element exchanges. For \( h == 1 \), shell sort works just like insertion sort, except the number of inversions that have to be eliminated is greatly reduced by the previous steps of the algorithm with \( h > 1 \).

4 Search algorithms

Search algorithms are algorithms that find one element of a set by some key (perhaps containing other information related to the key). As this is a common problem in computer science, the computational complexity of searching algorithms has been well studied.

The simplest search algorithm is linear search. It has O(n) running time, but can operate on a list containing the elements of the set.

A more sophisticated search algorithm is binary search. It runs in O(log(n)). This is significantly better than linear search for large lists of data, but it requires that the list be sorted before searching (see sort algorithm) and also be random access.

There is a family of tree search algorithms that compare ordered keys with one another to see if they are greater or less; the simplest one uses a binary search tree;

4.1 Linear search

In the field of computer science, **linear search** is a search algorithm, also known as **sequential search** that is suitable for searching a set of data for a particular value.

It operates by checking every element of a list until a match is found. Linear search runs in O(N). If the data are distributed randomly, on average N/2 comparisons will be needed. The best case is that the value is equal to the first element tested, in which case only 1 comparison is needed. The worst case is that the value is not in the list, in which case N comparisons are needed.
4.2 Binary search

Suppose we have an array of integers, previously sorted in increasing numerical order. We want to know if some given integer is present in the array. The method shown in Figure

```java
public static boolean contains(int item, int[] array) {
    int low = 0;
    int high = array.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (item < array[mid]) {
            high = mid - 1;
        } else if (item > array[mid]) {
            low = mid + 1;
        } else {
            return true;
        }
    }
    return false;
}
```

Figure: A binary search method for integers.

searches for an item in an array using binary search. The idea follows the way you might look up a word in a dictionary. Naively, you might begin at the beginning and work through each word in turn, comparing it with your word. But if your word is not present, and there are \( N \) words in the dictionary, it will take \( N \) comparisons (you have to compare that word with your word \( N \) times). If it is present, you can expect it to take some time proportional to \( N \), depending on the assumptions made about the probable location. In any case, whether it is present or not, the expected number of comparisons will normally double as you double the size of the dictionary. In other words, simple sequential search is of \( O(N) \) complexity.

In practice you are more likely to begin by looking at a word somewhere near the middle. If that is your word, you are done. Otherwise, if your word is earlier than that word, you can discard the second half of the dictionary. If it is later, you can discard the first half. Either way, you now have to search a dictionary only half the previous size, and you have managed that with just one comparison! Putting it the other way round, if you doubled the size of the dictionary you would only have to make one extra comparison. Doing it by sequential search, you would have to double the number of comparisons. Doing it by binary search, you just add one. This is the difference between \( O(\log N) \) and \( O(N) \) complexity. Notice that this improvement was only possible because the words were assumed to be in their correct alphabetical order in the dictionary.

Concerning the actual code, you should satisfy yourself that it really does terminate correctly. The tricky parts are the condition

```
low <= high
```

and the assignment

```
mid = (low + high) / 2.
```

It won't work if you use \(<\) rather than \(\leq\) and you should worry about the value of \(\text{mid}\) in the cases where \(\text{low} + \text{high}\) is odd or even. The method will return from within the \while loop
if the item is present; otherwise it will exit the while loop with low > high and return false.
1 Graphs

A mathematical object composed of points known as graph vertices or nodes and lines connecting some (possibly empty) subset of them, known as graph edges. Formally, a graph is a binary relation on a set of vertices. If this relation is symmetric, the graph is said to be undirected; otherwise, the graph is said to be directed. Graphs in which at most one edge connects any two nodes are said to be simple graphs. Vertices are usually not allowed to be self-connected, but this restriction is sometimes relaxed to allow such "loops." The edges of a graph may be assigned specific values or labels, in which case the graph is called a labeled graph.

2 Greedy Algorithms

Greedy algorithms work in phases. In each phase, a decision is made that appears to be good, without regard for future consequences. Generally, this means that some local optimum is chosen. This 'take what you can get now' strategy is the source of the name for this class of algorithms. When the algorithm terminates, we hope that the local optimum is equal to the global optimum. If this is the case, then the algorithm is correct; otherwise, the algorithm has produced a suboptimal solution. If the best answer is not required, then simple greedy algorithms are sometimes used to generate approximate answers, rather than using the more complicated algorithms generally required to generate an exact answer.
2.1 Spanning Tree

A spanning tree of a graph is one that reaches all nodes of a graph without introducing any cycles. A minimum-spanning tree means that the cost of traversing all the edges of the tree is a minimum.

A graph often contains redundancy in that there can be multiple paths between two vertices. This redundancy may be desirable, for example to offer alternative routes in the case of breakdown or overloading of an edge (road, connection, phone line) in a network. However, we often require the cheapest sub-network that connects the vertices of a given graph. This must in fact be an unrooted tree, because there is only one path between any two vertices in a tree; if there is a cycle then at least one edge can be removed. The total cost or weight of a tree is the sum of the weights of the edges in the tree. We assume that the weight of every edge is greater than zero. Given a connected, undirected graph $G=\langle V,E \rangle$, the minimum spanning tree problem is to find a tree $T=\langle V,E' \rangle$ such that $E'$ subset of $E$ and the cost of $T$ is minimal.

Note that a minimum spanning tree is not necessarily unique. Recall that a tree over $|V|$ vertices contains $|V|-1$ edges. A tree can be represented by an array of this many edges.

Kruskal's algorithm is a method for deriving a minimum-spanning tree.

2.2 Kruskal's Algorithm

- Select the n-1 edges of a weighted, undirected graph one at a time.
- Start by selecting the least cost edge. Add that to the minimum-spanning tree.
- From the remaining edges, select the next least cost edge that does not result in a cycle, and add that to the tree. Continue until all the vertices are in the tree.

The greedy criterion in Kruskal's algorithm is: From the remaining edges, select a least-cost edge that does not result in a cycle when added to the set of already selected edges.

```
KruskalAlgorithm(graph)
  tree = null;
  sort edges in ascending order of weights;
  for(i=1; i<|E| and |tree| < |V| -1 ; i++)
    if (edges[i] does not form cycle with edges currently in tree)
      add edges[i] in tree
```

Finding cycle

- Each connected part can be considered as a tree.
- New edge does not form cycle only if it is connecting two different trees.
- Each node stores information about its ‘father’.
- Root of a tree can be traced by traversing these ‘father’ links from a node.
- Cycle if both trees connected by the new edge have same root.
- Combining two trees involves setting ‘father’ of one root to the other root.

Complexity

- Initial sorting of edges will take $O(|E|\log(|E|))$.
- Detecting cycle is of $O(|V|)$.
- The loop executed $O(|V| -1)$ times.
- Overall complexity is $O(|E|\log(|E|)) + ((|V| -1)|(|V|))$.
- For a graph $|V| << |E|$ this gives complexity of Kruskal’s algorithm as $O(|E|\log(|E|))$. 


**Example**

Weighted, Undirected Graph

Select the Minimum Edge

Tree is now

Select the Next Lowest Cost Edge

Tree is now

Select the Next Lowest Cost Edge

Tree is now

Select the Next Lowest Cost Edge

Not good, This edge would introduce a CYCLE into the graph. We would no longer have a tree!

Select the Next Lowest Cost Edge

Tree is now

Select the Next Lowest Cost Edge

Not good, This edge would introduce a CYCLE into the graph. We would no longer have a tree!
Select the Next Lowest Cost Edge  
Tree is now a minimum-cost spanning tree

\[ \text{Select the Next Lowest Cost Edge} \]

Tree is now a minimum-cost spanning tree

2.3 Dijkstra's algorithm

Dijkstra's algorithm, named after its inventor the Dutch computer scientist Edsger Dijkstra, solves a shortest path problem for a directed and connected graph \( G(V,E) \) which has nonnegative (\( \geq 0 \)) edge weights. The set \( V \) is the set of all vertices in the graph \( G \). The set \( E \) is the set of ordered pairs which represent connected vertexes in the graph (if \( (u,v) \) belongs to \( E \) then vertexes \( u \) and \( v \) are connected).

Assume that the function \( w : V \times V \rightarrow [0, \infty] \) describes the cost \( w(x,y) \) of moving from vertex \( x \) to vertex \( y \) (non-negative cost). (We can define the cost to be infinite for pairs of vertices that are not connected by an edge.) For a given vertex \( s \) in \( V \), the algorithm computes the shortest path (lowest total value of \( w \)) from \( s \) to any other vertex \( t \).

The algorithm works by constructing a subgraph \( S \) of such that the distance of any vertex \( v' \) (in \( S \)) from \( s \) is known to be a minimum within \( G \). Initially \( S \) is simply the single vertex \( s \), and the distance of \( s \) from itself is known to be zero. Edges are added to \( S \) at each stage by

(a) identifying all the edges \( e_i = (v_{i1}, v_{i2}) \) in \( G - S \) such that \( v_{i1} \) is in \( S \) and \( v_{i2} \) is in \( G \), and then

(b) choosing the edge \( e_j = (v_{j1}, v_{j2}) \) in \( G - S \) which gives the minimum distance of its vertex \( v_{j2} \) (in \( G \)) from \( s \) from all edges \( e_i \). The algorithm terminates either when \( S \) becomes a spanning tree of \( G \), or when all the vertices of interest are within \( S \).

The procedure for adding an edge \( e_j \) to \( S \) maintains the property that the distances of all the vertices within \( S \) from \( s \) are known to be minimum.

A few subroutines for use with Dijkstra's algorithm

Initialize-Single-Source(G,s)

\[
\text{for each vertex } v \text{ in } V[G] \\
\text{do } d[v] := \text{infinite} \\
\text{previous}[v] := 0 \\
\text{d}[s] := 0
\]

Relax(u,v,w)

\[
\text{if } d[v] > d[u] + w(u,v) \\
\text{then } d[v] := d[u] + w(u,v) \\
\text{previous}[v] := u
\]

4

\[ v = \text{Extract-Min}(Q) \text{ searches for the vertex } v \text{ in the vertex set } Q \text{ that has the least } d[v] \text{ value. That vertex is removed from the set } Q \text{ and then returned.} \]
The algorithm:

Dijkstra(G,w,s)
Initialize-Single-Source(G,s)
S := empty set
Q := set of all vertexes
while Q is not an empty set
do u := Extract-Min(Q)
S := S union {u}
for each vertex v which is a neighbour of u
do Relax(u,v,w)

Complexity

The running time of Dijkstra's algorithm is \( O(V^2 + E) \), where \( V \) is the number of vertices, and \( E \) the number of edges. The intuition is that, in a complete graph, when we visit each vertex, we will be considering its distance from \( V - 1 \) other vertices. Therefore, \( O(V^2) \). Moreover, all edges will be 'touched' once, and there are \( E \) edges. So \( O(V^2 + E) \). Returning to our original notational, where \( n \) is the number of vertices, and in a complete graph there are \( O(n^2) \) edges, Dijkstra’s algorithm is \( O(n^2) \). Some implementations of the Dijkstra's algorithm can run in much less time.

Dijkstra's algorithm can be implemented efficiently by storing the graph in the form of adjacency lists and using a heap as priority queue to implement the Extract-Min function. If the graph has \( m \) edges and \( n \) vertices, then the algorithm's time requirements are \( \Theta(m + n \log n) \), assuming that comparisons of edge weights take constant time.

Example

Suppose we want to find the shortest path from \( s \) to \( v \).

The numbers in the boxes will indicate the current shortest path from \( p \) to that node. Let \( d(node) \) denote this distance. First, we initialize \( d(s) = 0 \) because we travel zero distance to go from \( s \) to \( s \), and initialize all other \( d(i) = \infty \) because no paths have been determined yet. We now define a cut (denoted by a dotted line) to be a line dividing the nodes which shortest distances \( d(i) \) have already been determined, and we colour such nodes black. Moreover, we colour as grey the node which shortest distance from \( s \) we are currently determining.

![Diagram](image-url)
3 The Maximum Flow Problem

3.1 Flow Networks

A flow network \( G = (V, E) \) is a directed graph where each edge \((u,v) \in E\) has capacity \( c(u,v) \geq 0 \). If \((u,v) \notin E\), then \( c(u,v) = 0 \).

Distinguish source \( s \in V \) and sink \( t \in V \) as the vertices between which the flow is to be maximized.

- in-degree\((s) = 0\)
- out-degree\((t) = 0\)

For every \( v \in V \), \( v \) is on a path from \( s \) to \( t \). \( S \leadsto V \leadsto t \), therefore the graph is connected.

\(|E| \geq |V| - 1\)

A flow \( f: V \times V \rightarrow \mathbb{R} \) is a real-valued function representing the rate of flow between \( u \) and \( v \), constrained by the following:

- Capacity constraint: \( f(u,v) \leq c(u,v) \)
- Skew symmetry: \( f(u,v) = -f(v,u) \)
- Flow conservation: \( \sum_{u \in \delta(V \setminus \{s,t\})} f(u,v) = 0 \).

The total flow into and out of a vertex \( u \) is 0.

The value of \( f(u,v) \) is the net flow from vertex \( u \) to vertex \( v \).

Note that if \((u,v)\) and \((v,u)\) \( \notin E \), then \( f(u,v) = f(v,u) = 0 \).

The value of a flow \( f \) is \( |f| = \sum_{u \in V} f(u,v) \).

Maximum-Flow Problem

Given flow network \( G \), with source \( s \) and sink \( t \), find flow with maximum value from \( s \) to \( t \).

3.2 Ford-Fulkerson Method

Look for paths from \( s \) to \( t \) that can accept more flow (augmenting path) and increase flow along this path.

\[
\text{Ford-Fulkerson-Method}(G, s, t) \\
\text{flow } f = 0 \\
\text{while an augmenting path } p \text{ exists} \\
\quad \text{augment flow } f \text{ along } p \\
\text{return } f
\]

An augmenting path is a path from \( s \) to \( t \) in a residual network which consists of edges that can admit more flow.

Residual Capacity

Given flow network \( G = (V, E) \) with flow \( f \) and two vertices \( u, v \in V \), define the residual capacity \( c_f(u,v) \) as the additional net flow possible from \( u \) to \( v \), not exceeding \( c(u,v) \). Thus

\[
c_f(u,v) = c(u,v) - f(u,v).
\]
Example

Residual Network
Given flow network $G = (V, E)$ and flow $f$, the residual network of $G$ induced by $f$ is $G_f = (V, E_f)$, where

$$E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$$

$(u, v) \in E_f$ is a residual edge; i.e., any edge that can admit more flow.

Augmenting Paths
Given flow network $G$ and flow $f$, an augmenting path $p$ is a simple path from $s$ to $t$ in the residual network $G_f$.

The minimum net flow along path $p$ in $G_f$ indicates the amount flow can increase along this path in $G$.

Thus, define the residual capacity of path $p$ as

$$c_f(p) = \min \{ c_f(u, v) \mid (u, v) \text{ is on } p \}$$

Define flow $f_p$ in $G_f$ as

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ on } p \\ -c_f(p) & \text{if } (v, u) \text{ on } p \\ 0 & \text{otherwise} \end{cases}$$

$$| f_p | = c_f(p) > 0$$

Define flow sum $f_1 + f_2$ as

$$(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v)$$
**Corollary**  
If flow network $G$ has flow $f$, augmenting path $p$ in $G_f$, and $f' = f + f_p$, then $f'$ is a flow in $G$ with value $|f'| = |f| + |f_p| > |f|$.

So, keep adding augmenting paths until there are no more.

**Theorem**  
f is a maximum flow in $G$ if and only if residual network $G_f$ has no augmenting paths.

**Example**

![An augmenting path of residual capacity 2](image)

**Ford-Fulkerson Algorithm**

\[ G = (V, E) \]

\[ \text{Ford-Fulkerson}(G, s, t) ; \]

\[ \text{foreach edge } (u,v) \text{ in } E \]

\[ f(u,v) = f(v,u) = 0 \]

\[ \text{while exists path } p \text{ from } s \text{ to } t \text{ in residual network } G_f \]

\[ c_f(p) = \min\{c_f(u,v) | (u, v) \text{ on } p\} \]

\[ \text{foreach edge } (u,v) \text{ on } p \]

\[ f(u,v) = f(u,v) + c_f(p) \]

\[ f(v,u) = -f(u,v) \]

**Analysis**

Depends on method for finding augmenting path.  
Use breadth-first search (Edmonds-Karp Algorithm). Thus, the augmenting path is shortest from $s$ to $t$.  
The total number of augmentations is $O(VE)$, $O(E)$ per augmentation. Thus the algorithm has run time $O(VE^2)$.
Example