Chapter 1

Overview of the course

Slide FCS1 - Overview:

Course: Fundamental Computer Science I

Course Number: 320201

Time: Tue, 8.15-9.30, Fri: 11.15-12.30

Place: Research III Lecture Hall

Instructor: Dr. Holger Kenn, Tel: 3112,
E-mail: h.kenn@iu-bremen.de

Web page: http://www.faculty.iu-bremen.de/course/FundCS1/

Slide Overview of the course:

- Datastructures
  - Lists, Heaps, Graphs, Trees
- Algorithms
  - Sorting and Searching
  - Algorithm Design and Analysis
  - Graph Algorithms, Dynamic Programming, Number-Theoretical Algorithms, String Matching, Computational Geometry,. . . (if we find the time)

This script is based on the textbook to the course:
Cormen Leiserson Rivest Stoll: Introduction to algorithms Second Edition, McGraw-Hill/MIT Press

Other sources are lecture notes of Torben Hagerup's Lecture "Datenstrukturen und Algorithmen" of the Fall semester '96 at the Universität des Saarlandes and the lecture notes of the lecture "Algorithmen und Komplexität" by Christine Rüb and Stefan Schirra of the Fall semester '92 also at the Universität des Saarlandes.

1.1 Introduction

Computer science consists of several fields. Each field tries to answer specific questions such as:

Computer Architecture: How to build computers?

Operating Systems: How to share the ressources of a single computer for multiple tasks?

Computer Networks: How to connect computers?

Theory of Complexity: What can be computed? And what cannot?

Artificial Intelligence: How to solve problems that we do not really understand, e.g. understanding language, programming robots for an unknown environment ...

Software Engineering: How to cope with the complexity of large-scale software systems?

"Social" computer science: What is the impact of computer use on individuals and society?

In this lecture, we will try to find some answers to this question:

Datastructures and Algorithms: How to use computers efficiently?

There are many interdependencies between the different fields of computer science, e.g. Datastructures and Algorithms uses the model of a computer derrived from Computer Architecture, Operating Systems, Computer Networks and Artificial Intelligence use Algorithms ...

For this lecture, we will use some simple answers for the other questions in computer science:

Computer Architecture: We assume a simple computer architecture.

Operating Systems: We assume that our computer is only running our program, we assume that input and output to our program is somehow taken care of.

Computer Networks: We do not use network connections.

Artificial Intelligence: We only cover probles that we think are well-understood.

Software Engineering: We only write simple programs.

"Social" computer science: Computers are good!

1.2 Fundamentals

A simple example for a problem that can be solved by a computer is sorting. Let's say we have the following numbers:

5, 8, 3, 2, 7, 3, 1, 9

These numbers are to be sorted so that they form a sequence so that the next number is not smaller than the last one. Intuitively, you will probably see that the corresponding sorted sequence is the following:

1, 2, 3, 3, 5, 7, 8, 9

But unfortunately, such an intuitive approach does not work for a computer and your intuition won't work for 100000 numbers either. Therefore, we have to come up with a more structured approach. That should take into account that we do not always want to sort these eight numbers, maybe not eight numbers, maybe not even numbers but we want to sort. We have to differenciate between the *sorting problem* and a specific *instance* of the sorting problem. An instance is defined by the specific input, e.g. (5, 8, 3, 2, 7, 3, 1, 9). So we had to deal with an instance of the sorting problem in our first example. In this lecture, we may use the problem name both for the problem itself and for its instances, but the meaning will usually be obvious. With an instance, we usually can derive some simple numbers that can tell us wether it is an "easy" or a "hard" instance. One example is the size of the instance, for the sorting problem, this is the size of the input , in our case eight.

We can define the sorting problem more formally in the following way:

Definition 1 *The* sorting problem *of instance size n is the problem defines as follows:*

Input: *A sequence of objects* $< a_1, a_2, \ldots, a_n >$

Output: *A sequence* $< a_{\sigma(1)}, a_{\sigma(2)}, \ldots, a_{\sigma(n)} >$ *of the input sequence such that* $a_{\sigma(1)} \leq a_{\sigma(2)} \leq \cdots \leq a_{\sigma(n)} >$ *and* σ *being a permutation of* $1, \ldots, n$

Sometimes, we're not really interested in the output sequence but only in the permutation σ.

Now after we have defined the problem, what is the way to solve it with a computer?

Independent of computers, the theory of algorithms is old. The term originates from the 9th century in the arabic world. A famous mathematician of this century was Abu Abdallah Muhammad Ibn Musa al Khwarizmi (780-850, the times of Caliph Harun al-Rasid) who wote several books. The most famous one is the Kitab al-muhatasar fi hisab al-gabr wa al-muqabala which was the first algebra book, it introduced the use of variables, equations and it used the indian base-10 numeral system with zero. The book also contains a collection of indian calculation methods.

This book was brought to europe and translated various times in the 12th century. It is the source of the common believe that the base-10 numeral system was an arabic invention although it was in fact invented in india. One of the latin translations by Gherardo di Cremona (1114-1187) was called "Dixit Algorismi".

Figure 1.1: Abu Abdallah Muhammad Ibn Musa al Khwarizmi (780-850)

What is an *algorithm* ?

By using our problem definition, we can say that an algorithm is a computational procedure that transforms the problem input into the problem output. An algorithm that creates the right solution for every instance of a problem is said to be *correct*. We say that a correct algorithm *solves* a computational problem.

An incorrect algorithm may not halt at all and produce no output or produce an incorrect solution. Contrary to common intuition, incorrect algorithms can be used for many applications as long as their error rate can be controlled.

In this lecture, we will use the *pseudocode*. If you are familiar with any procedural programming language such as Pascal or C, you will probably have no problem understanding the algorithms.

An example for an algorithm solving the sorting problem is shown as Algorithm 1.

Algorithm 1 Insertion-Sort

```
1:  for  j ← 2 to length[A] do
2:      key ← A[j]
3:      { Insert A[j] into the sorted sequence A[1 . . . j − 1].}
4:      i ← j - 1
5:      while  i > 0 and A[i] > key  do
6:          A[i+1] ← A[i]
7:          i ← i-1
8:      end while
9:      A[i+1] ← key
10: end for
```

1.3 Analyzing algorithms

Analyzing an algorithm means predicting its ressource usage. Ressources can be memory requirements, communication bandwith or the use of special devices such as graphic render hardware, but in most cases, we're interested in the computational time that it takes for the algorithm to produce its output.

Before we can analyze an algorithm, we have to have some assumptions on the computational hardware the algorithm will be executed on. In this lecture, we will generally assume a simple one-processor Van-Neumann-like architecure called *random access machine* (*RAM*). In the RAM, instructions are executed sequencially without any concurrency or parallelism, we assume no cacheing of data or instructions and a uniform main memory access time. Occasionally, we will look at different architectures such as special-purpose hardware or parallel architectures.

Even analyzing simple algorithms can be quite challenging involving discrete combinatorics, probability theory, algebraic and analytic dexterity.

1.3.1 Analysis of Insertion Sort

To find out computational time the Algorithm 1 uses, we first assume that each line of the algorithm takes a constant time to be executed on the RAM, the time being $c_1 \dots c_{10}$. For the lines 3,8 and 10, we will assume $c_3 = c_8 = c_{10} = 0$.

As we can easily see, the runtime of this algorithm depends on its input size $n = length[A]$. For each $j = 2, \dots, n$, we define t_j to be the number of times the test of the while loop in line 5 is executed for that value of j.

To understand the algorithm, it is helpful to define a *loop invariant*:

At the start of each iteration of the `for` loop, of lines 1-10, the subarray $A[1 \dots j-1]$ consists of the elements originally in $A[1 \dots j-1]$, but in sorted order.

InsertionSort(A)	cost	times
1: **for** j ← 2 to length[A] **do**	c_1	n
2: key ← A[j]	c_2	$n-1$
3: { Insert $A[j]$ into the sorted sequence $A[1 \dots j-1]$.}	$c_3 = 0$	$n-2$
4: i ← j - 1	c_4	$n-1$
5: **while** i > 0 and A[i] > key **do**	c_5	$\sum_{j=2}^{n} t_j$
6: A[i+1] ← A[i]	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7: i ← i-1	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8: **end while**	$c_8 = 0$	$n-1$
9: A[i+1] ← key	c_9	$n-1$
10: **end for**	$c_{10} = 0$	n

The total runtime for an input size n is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + \\ & \quad c_6 \sum_{j=2}^{n} t_(j-1) + c_7 \sum_{j=2}^{n} t_(j-1) + c_9(n-1) \end{aligned}$$

This equation describes the runtime, but it is not very handy. We can now analyze different types of input and predict the runtime. Let's first find the *best-case* running time. If the sequence is already sorted, the equation simplifies to

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_9(n-1)$$

since lines 6 and 7 are never executed. This equation can be expressed in the form of a linear equation in the form of $an + b$ for some constants a and b.

What about the *worst case*? This occurs if the array is sorted in reverse order.

With

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

and

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

we get the worst case running time to be

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\Big(\frac{n(n-1)}{2} - 1\Big) + \\ & \quad c_6\Big(\frac{n(n-1)}{2}\Big) + c_7\Big(\frac{n(n-1)}{2}\Big) + c_9(n-1) \\ &= \Big(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\Big)n^2 + \Big(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_9\Big)n \\ & \quad -(c_2 + c_4 + c_5 + c_9) \end{aligned}$$

and this is a quadratic function of the form $an^2 + bn + c$

There is a third case that is sometimes analyzed, the so-called *average case* runtime. It is often as bad as the average case. For example, if we assume that for each subarray sequence on average about half of the sequence is greater than $A[j]$, the while loop has to search through about half the sequence, so $T_j = j/2$. By analyzing this case the same way we analyzed the other cases, we will find that it also leads to a quadratic function. In the average case analysis, the problem is to precisely define the average case, i.e. the likeliness that a certain instance of the problem occurs.

1.4 Algorithm Design Techniques

What we used in the insertion sort algorithm was the so-called *incremental* approach. Having sorted one subarray of length $j-1$ we inserted another element to it, thus creating a sorted subarray of length j. There are other approaches that will be sketched here, we will see applications of these techniques later on.

1.4.1 divide-and-conquer

By *recursivly* dividing the input into several sub-problems, these can be reduced to trivial cases that can be solved in a single step. Then, the results of the sub-problems are combined to form a solution. One example of this approach is the *merge sort* algorithm: It divides the input sequence in halves, reapplies itself to the halves and combines the two sorted halves into a single sorted sequence. Since the sub-sequences are sorted, mergesort only has to compare the first element of each sequence and pull out the lower one of the two. Mergesort is an old algorithm that dates back to the time of punchcard stack. Its greatest advantage is that it can be used even with minimal memory.

The main task of mergesort is done by the Merge procedure that looks as follows:

MERGE (A,p,q,r)

1: $n_1 \leftarrow q - p + 1$
2: $n_2 \leftarrow r - q$
3: create Arrays $L[1 \ldots n_1 + 1]$ and $R[1 \ldots n_2 + 1]$
4: **for** $i \leftarrow 1$ to n_1 **do**
5: $\quad L[i] \leftarrow A[p + i - 1]$
6: **end for**
7: **for** $j \leftarrow 1$ to n_2 **do**
8: $\quad R[j] \leftarrow A[q + j]$
9: **end for**
10: $L[n_1 + 1] \leftarrow \infty$
11: $R[n_2 + 1] \leftarrow \infty$
12: $i \leftarrow 1$
13: $j \leftarrow 1$
14: **for** $k \leftarrow p$ to r **do**
15: $\quad$ **if** $L[i] \leq R[j]$ **then**
16: $\quad\quad A[K] \leftarrow L[i]$
17: $\quad\quad i \leftarrow i + 1$
18: $\quad$ **else**
19: $\quad\quad A[K] \leftarrow R[j]$
20: $\quad\quad j \leftarrow j + 1$
21: $\quad$ **end if**
22: **end for**

Mergesort itself is then quite simple:

MERGESORT (A,p,r)

1: **if** $p < r$ **then**
2: $\quad q \leftarrow \lfloor (p + r)/2 \rfloor$
3: $\quad$ MERGESORT(A, p, q)
4: $\quad$ MERGESORT$(A, q + 1, r)$
5: $\quad$ MERGE(A, p, q, r)
6: **end if**

1.4.2 Asymptotic notation

Definition 2 *For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions*

$$\begin{aligned}\Theta(g(n)) &= \{f(n) : \exists c1 > 0, c2 > 0 \text{ and } n_0 > 0 \text{ such that} \\ &\quad 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0\}.\end{aligned}$$

We say that $g(n)$ is an asymptotically tight bound *for $f(n)$.*

Definition 3 *For a given function $g(n)$, we denote by $O(g(n))$ the set of functions*

$$\begin{aligned}O(g(n)) &= \{f(n) : \exists c > 0 \text{ and } n_0 > 0 \text{ such that} \\ &\quad 0 \leq f(n) \leq cg(n) \forall n \geq n_0\}.\end{aligned}$$

We say that $g(n)$ is an asymptotically upper bound *for $f(n)$.*

Definition 4 *For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions*

$$\begin{aligned}\Omega(g(n)) &= \{f(n) : \exists c > 0 \text{ and } n_0 > 0 \text{ such that} \\ &\quad 0 \leq cg(n) \leq f(n) \forall n \geq n_0\}.\end{aligned}$$

We say that $g(n)$ is an asymptotically lower bound *for $f(n)$.*

Other interesting properties of asymptotic notation:

Transitivity:

$$\begin{aligned}f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) &\Rightarrow f(n) = \Theta(h(n)) \\ f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) &\Rightarrow f(n) = O(h(n)) \\ f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) &\Rightarrow f(n) = \Omega(h(n))\end{aligned}$$

Reflexivity:

$$\begin{aligned}f(n) &= \Theta(f(n)) \\ f(n) &= O(f(n)) \\ f(n) &= \Omega(f(n))\end{aligned}$$

Symmetry:

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

Transpose symmetry:

$$\begin{aligned}f(n) = O(g(n)) &\Leftrightarrow g(n) = \Omega(f(n)) \\ f(n) = \Omega(g(n)) &\Leftrightarrow g(n) = O(f(n))\end{aligned}$$

1.4.3 Other useful mathematical notations

Floor and *Ceiling*:

$$\text{For } x \in \mathbb{R} : x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

Logarithms: Like $\ln$ denotes the natural logarithm $\log_e$, we introduce $\lg = \log_2$.

Modular arithmetic:

$$\text{For } a \in \mathbb{Z} \text{ and } n \in N : a \bmod n = a - \lfloor a/n \rfloor n$$

a is called *remainder* or *residue* of the quotient a/n.

If $(a \bmod n) = (b \bmod n)$ we write $a \equiv b \pmod{n}$.

Functional iteration:

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0 \\ f(f^{(i-1)}(n)) & \text{if } i > 0 \end{cases}$$

Iterated logarithm:

$$\lg^*(n) = \min \left\{ i \geq 0 : \lg^{(i)}(n) \leq 1 \right\}$$

Fibonacci numbers:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-1} + Fi - 2 \text{ for } i \geq 2 \end{aligned}$$

Fibonacci Numbers are related to the *golden ratio* ϕ and its conjugate $\hat{\phi}$.

$$\begin{aligned} \phi &= \frac{1 + \sqrt{(}5)}{2} \\ \hat{\phi} &= \frac{1 - \sqrt{(}5)}{2} \\ F_i &= \frac{\phi^i - \hat{\phi}^i}{\sqrt{(}5)} \end{aligned}$$

1.4.4 Runtime of mergesort

The MERGE procedure runs in $\Theta(n)$ time, where $n = r - p + 1$. The lines 1-3 and 10-13 take constant time. The **for** loop of lines 4-9 take $\Theta(n_1 + n_2) = \Theta(n)$ time and there are n iterations of the for loop of lines 14-22 of which each one takes constant time.

MERGESORT itself runs in $\Theta(n \lg n)$ time. In order to see that, we have to solve the recurrence

$$T(n) = \begin{cases} c & \text{if } n = 0 \\ 2T(n/2) + cn & \text{if } n > 0 \end{cases}$$

It can be shown that $T(n) = \Theta(n \lg n)$ but here, only an intuitive idea is sketched. Let's assume for simplicity that n is a power of two. MERGESORT recursively divides the problems into n subproblems of size 1. The depth of recursion is $\lg n$. At each recursion level, the algorithm needs $\Theta(n)$ time and this happens for $\lg n$ recursion levels.

1.5 Heapsort

Heapsort is a sorting algorithm that sorts in $O(n \lg n)$ time, but sorts in place, i.e. does not make copies of the data. Heapsort makes use of a datastructure called *heap*.

The binary heap datastructure is an array object that implements some form of a binary tree. For that array, we have to remember not only $length[A]$ but also *heap-size*$[A]$ which may not be equal, i.e. no element past $A[$*heap-size*$[A]]$ is part of the heap although it may be a valid array element.

Given the index i of a node, PARENT(i) is the parent node, LEFT(i) is the left child, RIGHT(i) is the right child.

The implementation of these functions is as follows:

PARENT(i)
return $\lfloor i/2 \rfloor$

LEFT(i)
return $2i$

RIGHT(i)
return $2i + 1$

Heaps can be organized as *min-heaps* and as *max-heaps*.

For a max-heap:

$$A[\text{PARENT}(i)] \geq A[i]$$

For a min-heap:

$$A[\text{PARENT}(i)] \leq A[i]$$

Heapsort consists of the following procedures:

MAX-HEAPIFY (A,i)
1: $l \leftarrow$ LEFT$[i]$
2: $r \leftarrow$ RIGHT$[i]$
3: **if** $l \leq$ *heap-size*$[A]$ and $A[l] > A[i]$ **then**
4: $largest \leftarrow l$
5: **else**
6: $largest \leftarrow i$
7: **end if**
8: **if** $r \leq$ *heap-size*$[A]$ and $A[r] > A[largest]$ **then**

```
    largest ← r
end if
if largest ≠ i then
    exchange A[i] ↔ A[largest]
    MAX-HEAPIFY(A, largest)
end if
```

BUILD-MAX-HEAP (A,i)

```
HEAP-SIZE[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1 do
    MAX-HEAPIFY(A, i)
end for
```

HEAPSORT (A)

```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
    exchange A[1] ↔ A[i]
    HEAP-SIZE[A] ← HEAP-SIZE[A] − 1
    MAX-HEAPIFY(A, 1)
end for
```

What are those three procedures doing?

MAX-HEAPIFY creates a heap from two existing heaps rooted at LEFT(i) and RIGHT(i) and an additional element $A[i]$. The additional element is inserted into the root and sinks down the heap until it reaches a final position and the heap property is re-established.

The running time of MAX-HEAPIFY on a subtree of size n rooted at given node i is the $\Theta(n)$ time to rearrange the three top elements plus the runtime of MAX-HEAPIFY of the changed subtree. The changed subtree has at most size $2n/3$. The runtime of MAX-HEAPIFY is then

$$T(n) \quad \leq \quad T(2n/3) + \Theta(1)$$

To solve this recurrence, we can apply the *master theorem*

Theorem 1 (Master Theorem) *Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function and let $T(n)$ be defined on the nonnegative integers by the recurrence*

$$T(n) \quad = \quad aT(n/b) + f(n)$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. *If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$ then $T(n) = \Theta\left(n^{\log_b a}\right)$.*
2. *If $f(n) = \Theta\left(n^{\log_b a}\right)$ then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.*
3. *If $f(n) = \Omega\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$ for some constant $c < i$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

1.5.1 Other useful applications of Heapsoprt: Priority Queues

Definition 5 (Priority queue) *A* priority queue *is a data structure for maintaining a set S of elements, each associated with a value called* key. *A* max-priority-queue *supports the following operations:*

1. INSERT(S, x) *inserts the element x into the set S. The operation could be written as $S \leftarrow S \cup \{x\}$.*
2. MAXIMUM(S) *returns the element of S with the largest key.*
3. EXTRACTMAX(S) *removes and returns the element of S with the largest key.*
4. INCREASEKEY(S, x, k) *increases the value of element x's key to thge new value k, which is assumed to be at least as large as x's current key value.*

A min-priority-queue *supports the following operations:*

1. INSERT(S, x) *inserts the element x into the set S. The operation could be written as $S \leftarrow S \cup \{x\}$.*
2. MINIMUM(S) *returns the element of S with the smallest key.*
3. EXTRACTMIN(S) *removes and returns the element of S with the smallest key.*
4. DECREASEKEY(S, x, k) *decreases the value of element x's key to thge new value k, which is assumed to be at least as small as x's current key value.*

The following procedures implement a max-priority-queue by unsing procedures from HEAPSORT.

HEAPMAXIMUM (A)
1: **return** A[1]

HEAPEXTRACTMAX (A)
1: **if** HEAP-SIZE$[A] < 1$ **then**
2: **error** "heap undeflow"
3: **end if**
4: $max \leftarrow A[1]$
5: $A[1] \leftarrow A[$HEAP-SIZE$[A]]$
6: HEAP-SIZE$[A] \leftarrow$ HEAP-SIZE$[A] - 1$
7: MAX-HEAPIFY$(A, 1)$
8: **return** max

HEAPINCREASEKEY (A, i, key)
1: **if** $key < A[i]$ **then**
2: **error** "new key is smaller than current key"
3: **end if**
4: $A[i] \leftarrow key$
5: **while** $i > 1$ and $A[$PARENT$(i)] < A[i]$ **do**
6: EXCHANGE $A[i] \leftrightarrow A[$PARENT$(i)]$

7: $i \leftarrow$ PARENT(i)
8: **end while**

MAXHEAPINSERT (A, key)
1: HEAP-SIZE$[A] \leftarrow$ HEAP-SIZE$[A] + 1$
2: $A[$HEAP-SIZE$[A]] \leftarrow -\infty$
3: HEAPINCREASEKEY $(A,$ HEAP-SIZE$[A], key)$

The runtimes of the functions are $\Theta(1)$ for HEAPMAXIMUM, $O(\lg n)$ for HEAPEXTRACT-MIN, HEAPINCREASEKEY and MAXHEAPINSERT.

1.6 Quicksort

Quicksort is based on divide and conquer like mergesort, but it uses a different scheme to divide the sequence and it rearranges the partial sequences in such a way that combining them is even easier than in Mergesort.

The divide step separates the initial sequence into two sub-sequences that are chosen based on a *pivot element*. Every element smaller (ore equal) than the pivot element belongs to the left sequence, every element larger than the pivot element belongs to the right sequence, the elements are thus exchanged accordingly. This can be done in $O(n)$ time with n being the length of the sequence.

Then, Quicksort is called again for the sub-sequences.

QUICKSORT (A, p, r)
1: **if** $p < r$ **then**
2: $q \leftarrow$ PARTITION(A, p, r)
3: QUICKSORT$(A, p, q-1)$
4: QUICKSORT$(A, q+1, r)$
5: **end if**

1.6.1 Analysis of Quicksort

Worst Case analysis

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

Guessing that $T(n) \le cn^2$ for some c.

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n) \end{aligned}$$

$q^2 - (n-q-1)^2$ achives a maximum over $0 \le q \le n-1$ at the endpoints. This gives us $\max_{0 \le q \le n-1}(q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1$.

$$\begin{aligned} T(n) &\leq cn^2 - c(2n-1) + \Theta(n) \\ &\leq cn^2 \end{aligned}$$

1.6.2 randomized quicksort

RANDOMIZEDQUICKSORT (A, p, r)

```
1: if p < r then
2:    q ← RANDOMIZEDPARTITION(A, p, r)
3:    RANDOMIZEDQUICKSORT(A, p, q − 1)
4:    RANDOMIZEDQUICKSORT(A, q + 1, r)
5: end if
```

RANDOMIZEDPARTITION (A, p, r)

```
1: i ← Random(p, r)
2: exchange A[r] ↔ A[i]
3: return PARTITION(A, p, r)
```

PARTITION (A, p, r)

```
1: x ← A[r]
2: i ← p − 1
3: for j ← p to r − 1 do
4:    if A[j] ≤ x then
5:       i ← i + 1
6:       exchange A[i] ↔ A[j]
7:    end if
8: end for
9: exchange A[i + 1] ↔ A[r]
10: return i + 1
```

Lemma 1 *Let X be the number of comparisions performed in line 4 of Partition over the entire execution of* QUICKSORT *on an n-element array. Then the running time of* QUICKSORT *is $O(n + X)$.*

Let $z_1, z_2, \ldots, z_n$ be the elements of array A so that Z_i is the ith smallest element. $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ is the set of elements between z_i and z_j, inclusive.

We define

$$X_{ij} = I\{z_i \text{ is compared to } z_j\}$$

on the complete run of the algorithm. Since each pair is compared at most once, we can calclulate the total number of comparisions

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

In expectations:

$$
\begin{aligned}
E[X] &= E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E\left[X_{ij}\right] \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}
\end{aligned}
$$

Now we just have to calculate $\Pr\{z_i \text{ is compared to } z_j\}$. Two elements are only compared if one of the two is chosen as a pivot element:

$$
\begin{aligned}
\Pr\{z_i \text{ is compared to } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is first chosen as a pivot element from } Z_{ij}\} \\
&= \Pr\{z_i \text{ is first chosen as a pivot element from } Z_{ij}\} + \\
&\quad \Pr\{z_j \text{ is first chosen as a pivot element from } Z_{ij}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1}
\end{aligned}
$$

$$
\begin{aligned}
E[X] &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&\quad \text{with } k = j-i \\
E[X] &= \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1}\sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n)
\end{aligned}
$$

Theorem 2 *Any comparision-based sort algorithm requires $\Omega(n \lg n)$ comparisions in the worst case.*

Corollary 1 *Heapsort and Mergesort are asymptotically optimal comparision sorts.*

1.7 Sorting without comparing

1.7.1 Counting Sort

Counting sort assumes that each of the n input elements is an integer in the range 0 to k for some integer k. When k = O(n), the sort runs in $\Theta(n)$ time.

COUNTINGSORT(A,B,k)

1: **for** $i \leftarrow 0$ **to** k **do**
2: $C[i] \leftarrow 0$
3: **end for**
4: **for** $j \leftarrow 1$ **to** $length[A]$ **do**
5: $C[A[j]] \leftarrow C[A[j]] + 1$
6: **end for**
7: **for** $i \leftarrow 1$ **to** k **do**
8: $C[i] \leftarrow C[i] + C[i-1]$
9: **end for**
10: **for** $j \leftarrow length[A]$ **downto** 1 **do**
11: $B[C[A[j]]] \leftarrow A[j]$
12: $C[A[j]] \leftarrow C[A[j]] - 1$
13: **end for**

Counting sort runs in $\Theta(k+n)$ time. Usually, we have $k = O(n)$, then Counting sort runs in $\Theta(n)$ time.

Definition 6 *A* stable *sorting algorithm preserves the input order of equal input vlaues in its output.*

1.7.2 Radix Sort

RADIXSORT(A,D)

1: **for** $i \leftarrow 1$ **to** d **do**
2: use a stable sort to sort array A on digit i.
3: **end for**

1.7.3 Bucket Sort

Bucket Sort assumes that the input values are created by a random process that uniformily distributes them over $[0, 1)$

BUCKETSORT(A)

1: $n \leftarrow length[A]$
2: **for** $i \leftarrow 1$ **to**n **do**
3: insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4: **end for**
5: **for** $i \leftarrow 0$ **to**$n-1$ **do**
6: sort list $B[i]$ with insertion sort
7: concatenate the lists $B[0], B[1], \ldots, B[n-1]$ together in order
8: **end for**

1.8 Medians and Order Statistics

Definition 7 *The ith* order statistic *of a set of n elements is the ith smallest element. The* minimum *is the first order statistic, the* maximum *is the nth order statistic.*

The median *for an odd number of elements in the set is the ith order statistic with $i = (n+1)/2$.*

The lower median *for an even number of elements in the set is the ith order statistic with $i = \lfloor (n+1)/2 \rfloor$.*

The upper median *for an even number of elements in the set is the ith order statistic with $i = \lceil (n+1)/2 \rceil$.*

The selection problem *is defined formally as:*

Input: A set A of n (distinct) numbers and a number i, with $1 \leq i \leq n$.

Output: The element $x \in A$ that is larger than exactly $i-1$ other elements of A.

If we use the term median of a set without knowing the number of elements, we use the lower median for sets with an even number of elements.

Obviously, the selection problem can be solved in $O(n \lg n)$ time by sorting the numbers and selecting the ith element in the sequence. But there are better algorithms.

1.8.1 Minimum and Maximum

MINIMUM(A)

```
1: min ← A[1]
2: for i ← 2 to length[A] do
3:    if min > A[i] then
4:       min ← A[i]
5:    end if
6: end for
7: return min
```

Both the upper and lower bound are $n-1$ comparisions.

It is sufficient to do $3\lfloor n/2 \rfloor$ comparisions instead of $2n-2$ do find both the minimum and the maximum. The trick is to work with pairs of elements. First compare two elements with eachother, then the larger one with the current maximum and the smaller one with the current minimum, so three comparisions for every two elements.

1.8.2 Selection in expected linear time

RANDOMIZEDSELECT(A, p, r, i)

```
1: if p = r then
2:    return A[p]
3: end if
```

```
4: q ←RANDOMIZEDPARTITION(A, p, r)
5: k ← q − p + 1
6: if i = k then
7:    return A[q]
8: else
9:    if i < k then
10:      return RANDOMIZEDSELECT(A, p, q − 1, i)
11:   else
12:      return RANDOMIZEDSELECT(A, q + 1, r, i − k)
13:   end if
14: end if
```

For RANDOMIZEDSELECT, $T(n) = O(n)$ on average. Therefore, any order statistic including the median can be determined on average in linear time.

1.8.3 Selection in worst-case linear time

SELECT

1. Divide the n elements of the input array into $\lfloor n/5 \rfloor$ groups of 5 elements each and at most one group made up of the remaining $n \mod 5$ elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups by sorting the elements of each group and then picking the median,i.e. element 3 from the sorted list.
3. Use SELECT recursively to find the median x of the $\lceil n/5 \rceil$ medians found in step 2.
4. Partition the input array around the median-of-medians x. Let k be one more than the elements of the low side of the partition so that x is the kth smallest element and there are $n - k$ elements on the high side of the partition.
5. if $i = k$ then return x. Otherwise, use SELECT recursively to find the ith smallest element in one of the partitions. For $i < k$, continue with the low side, for $i > k$ continue with the high side.

At least half of the medians found in step 2 are greater than the median-of-medians x. Thus, at least half of the $\lceil n/5 \rceil$ groups contribute three elements that are greater than x except two groups (the one containing x and the modulo group at the end, we will substract them).

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rceil - 2\right) \quad \geq \quad \frac{3n}{10} - 6$$

In the worst case, SELECT is called on at most $7n/10 + 6$ elements in step 5. Therefore, we can now develop a recurrence for the runtime of SELECT.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 140 \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n > 140 \end{cases}$$

For $n > 140$

$$\begin{aligned} T(n) &= T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) \\ &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + an \\ &\leq cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) \end{aligned}$$

This is $\leq cn$ if $(-cn/10 + 7c + an) \leq 0$ which is the case for $n > 140$ and $c \geq 20a$.

The worst case runming time of SELECT is therefore linear.

For a discussion of SELECT, see CLRS Chapter 9.3 and Mehlhorn, Datastructures And Efficient Algorithms, Vol1, Sorting and Searching, Chapter II.4

- Proofs about algorithms:
  - Correctness
    - Tool: Loop invariant
  - Runtime
    - best case
    - worst case
    - average case
- Types of algorithms
  - iterative: proof of loop invariants
  - recursive (divide-and-conquer): recurrences
  - recurrences: solved by substitution or master theorem
- Sorting Algorithms
  - Comparision-based sort

Algorithm	best case runtime	worst case runtime	average case runtime
INSERTIONSORT	$O(n)$	$O(n^2)$	$O(n^2)$
MERGESORT		$\Theta(n \lg n)$	
HEAPSORT		$O(n \lg n)$	
QUICKSORT	$O(n)$	$O(n^2)$	$O(n \lg n)$

  - Lower Bound for comparision-based sort:$\Omega(n \lg n)$
  - Non-comparision-based sort

Algorithm	worst case runtime	expected runtime
COUNTINGSORT	$\Theta(k+n)$	
RADIXSORT	$\Theta(d(n+k))$	
BUCKETSORT	$O(n^2)$	$\Theta(n)$

- Medians and Order Statistics
  - Minimum, Maximum run in $n-1$ comparisions.
  - Combined Minimum and Maximum runs in $3\lfloor n/2 \rfloor$ comparisions.
  - RANDOMIZEDSELECT runs in expected linear time.
  - SELECT runs in worst-case linear time.

Chapter 2

datastructures

Sets are as fundamental to computer science as they are to mathematics. Unlike in mathematics where sets are unchanging, algorithms can manipulate sets, they can create new sets, add elements, remove elements and so on. Therefore, we call such sets *dynamic* and we have to find a way for algorithms to handle sets in a computer.

A dynamic set that supports the operations *insert*, *delete* and *test membership* is called a *dictionary*. There are other dynamic sets that implement different operations such as the min-max-heap datastructure. Depending on the needed operations, dynamic sets are implemented differently.

Each element of a set is represented by an object in memory whose fields can be examined and manipulated if we have a pointer to this object. Some sets assume one identifying field of the object called *key*. If all keys are different, we can use the dynamic set as a set of key values. The object may contain *satellite data* in some of its fileds, that is application specific data that is carried around with the object but is not used in any way by the implementation of the dynamic set. It also has fields that are manipulated by the implementation, those fields contain pointers to other objects, array indeces or additional data. Some dynamic sets operate on key values from a totally ordered set such as real numbers or words under an alphabet and a lexical order. A total order allows us to speak of a "next element" by order or a "maximum element".

Typical operations on dynamic sets are:

SEARCH(S, k) Given a set S and a key k, this query returns a pointer x to an object in S so that $key[x] = k$ or NIL if no such object exists in S.

INSERT(S, x) This operation adds an element x to S, i.e. $S \leftarrow S \cup \{x\}$.

DELETE(S, x) This operation removes an element x to S, i.e. $S \leftarrow S \setminus \{x\}$.

MINIMUM(S) This operation returns a pointer to the element of a totaly ordered set S with the smallest key.

MAXIMUM(S) This operation returns a pointer to the element of a totaly ordered set S with the largest key.

SUCCESSOR(S, x) This operation returns the element of a totaly ordered set S with the next larger key than element x.

PREDECESSOR(S, x) This operation returns the element of a totaly ordered set S with the next smaller key than element x.

The SUCCESSOR and PREDECESSOR queries are often extendet for sets with nondistinct keys so that a call to MINIMUM with consecutive calls to SUCCESSOR enumerates the elements of the set. The execution time of the operations is usually given as a function of the size of the set that it is applied to.

2.1 Elementary datastructures

A *stack* is a dynamic set datastructure that implements a *last-in,first-out* (LIFO) policy. The only element that can be deleted from a stack is the one most recently inserted. A *queue* is a dynamic set datastructure that implements a *first-in,first-out* (FIFO) policy. The only element that can be deleted from the queue is the one that has been in the queue the longest time.

Arrays can be used to implement both queue and stack.

2.1.1 stacks

For a stack, the INSERT operation is often called PUSH, the DELETE operation is then called POP.

The stack is implemented by using an integer called *stack pointer*. In our example, we use $top[S]$ for the stack pointer of stack S.

The following code implements PUSH, POP and STACKEMPTY that returns TRUE if the stack does not contain any elements.

STACKEMPTY(S)
1: **if** $top[S] = 0$ **then**
2: **return** TRUE
3: **else**
4: **return** FALSE
5: **end if**

PUSH(S, x)
1: $top[S] \leftarrow top[S] + 1$
2: $S[top[S]] \leftarrow x$

POP(S)
1: **if** STACKEMPTY(S) **then**
2: **error** "Stack Underflow"
3: **else**
4: $top[S] \leftarrow top[S] - 1$
5: **return** $S[top[S] + 1]$
6: **end if**

Each of these operation takes $O(1)$ time.

2.1.2 queues

For a queue, the INSERT operation is often called ENQUEUE, the DELETE operation is then called DEQUEUE.

Unlike the stack, the queue needs two additional pointers, a *head* pointer and a *tail* pointer. The tail is where elements are enqueued, the head is where they are dequeued.

The following functions implement a queue with at most n-1 elements.

ENQUEUE(Q, x)
1: $Q[tail[Q]] \leftarrow x$
2: **if** $tail[Q] = length[Q]$ **then**
3: $\quad tail[Q] \leftarrow 1$
4: **else**
5: $\quad tail[Q] \leftarrow tail[Q] + 1$
6: **end if**

DEQUEUE(Q)
1: $x \leftarrow Q[head[Q]]$
2: **if** $head[Q] = length[Q]$ **then**
3: $\quad head[Q] \leftarrow 1$
4: **else**
5: $\quad head[Q] \leftarrow head[Q] + 1$
6: **end if**
7: **return** x

2.1.3 Linked lists

In a *linked list*, the elements are arranged in a linear order but unlike in the array, the order is not created by the linearity of the array index but by chaining the elements together, i.e. by following a pointer, we can go from one element to another element. A *doubly linked list* L has two pointers, $prev$ and $next$ for each element. For an element x, $prev[x]$ points to the predecessor of x and $next[x]$ points to the successor. For the first element (the head of the list which has no predecessor) we use the value NIL for $prev$. For the last element (the tail of the list which has no successor), we use the value NIL for $next$. $head[L]$ points to the first element of the list. A *singly linked list* is a list without the $prev$ ponter in the elements. If a list is *sorted*, the linear order of the list corresponds to the linear order of keys stored in the elements of the list, usually, the head of the list contains the minimum element and the tail of the list contains the maximum element. In a *circular list*, the $prev$ pointer of the head element points to the tail of the list and the $next$ pointer of the tail element points to the head of the list. Unless otherwise stated, we assume our lists to be doubly linked and unsorted.

We can search the list for a specific element:

LISTSEARCH(L, k)
1: $x \leftarrow head[L]$
2: **while** $x \neq$ NIL and $key[x] \neq k$ **do**
3: $\quad x \leftarrow next[x]$
4: **end while**
5: **return** x

We can insert an element into the list:

LISTINSERT(L, x)
1: $next[x] \leftarrow head[L]$
2: **if** $head[L] \neq$ NIL **then**
3: $\quad prev[head[L]] \leftarrow x$
4: **end if**
5: $head[L] \leftarrow x$
6: $prev[x] \leftarrow$ NIL

We can delete an element from the list:

LISTDELETE(L, x)
1: **if** $prev[x] \neq$ NIL **then**
2: $\quad next[prev[x]] \leftarrow next[x]$
3: **else**
4: $\quad head[L] \leftarrow next[x]$
5: **end if**
6: **if** $next[x] \neq$ NIL **then**
7: $\quad prev[next[x]] \leftarrow prev[x]$
8: **end if**

In order to make our life simpler (by avoiding all these if-statements that reflect the list boundary handling) we introduce a *sentinel* list element that we add to the endpoints of the list. This element is called $nil[L]$ like the pointer value but also has the fields $prev$ and $next$, so that $prev[nil]$ and $next[nil]$ have a well-defined meaning. This turns a regular doubly-linked list into a circular, doubly linked list with a sentinel in which the nil sentinel is placed before the head and after the tail element. Since $next[nil[L]] = head[L]$ we can eliminate the $head[L]$ pointer and use the nil element instead. An empty list consists only of the $nil[L]$ element pointing to itself with the $prev$ and $next$ pointers.

LISTSEARCH'(L, k)
1: $x \leftarrow next[nil[L]]$
2: **while** $x \neq nil[L]$ and $key[x] \neq k$ **do**
3: $\quad x \leftarrow next[x]$
4: **end while**
5: **return** x

LISTDELETE'(L, k)
1: $next[prev[x]] \leftarrow next[x]$
2: $prev[next[x]] \leftarrow prev[x]$

LISTINSERT'(L, x)
1: $next[x] \leftarrow next[nil[L]]$
2: $prev[next[nil[L]]] \leftarrow x$
3: $next[nil[L]] \leftarrow x$
4: $prev[x] \leftarrow nil[L]$

2.1.4 Memory management with lists

We can use lists to manage the memory of the computer.

ALLOCATEOBJECT()
  if $free = \text{NIL}$ **then**
    error "out of memory"
  else
    $x \leftarrow free$
    $free \leftarrow next[x]$
    return x
  end if

FREEOBJECT(x)
  $next[x] \leftarrow free$
  $free \leftarrow x$

2.2 Hashing and Hash Tables

2.2.1 Direct adressing

Assuming a dynamic set in which each element has a key k drawn from the universe $\{0, 1, \ldots, m-1\}$. Then we can use a *direct-address table* $T[0 \cdots -1]$ in which each *slot* corresponds to exactly one key $k \in U$. If we assume that no two elements in U have the same key k, we can simply store the elements (or pointers to the elements) of U in T. When no element of key k exists, we store NIL in $T[k]$.

DIRECTADRESSSEARCH(T,K)
  return $T[k]$

DIRECTADRESSINSERT(T,X)
  $T[key[x]] \leftarrow x$

DIRECTADRESSDELETE(T,K)
  $T[k] \leftarrow$NIL

2.2.2 Hashing as a technique for datastructures

Direct-adress tables habe a huge drawback: if $|U|$ is large, a large table has to be used. It can also be very inefficient: If $|U|$ is large but the set to be stored is small, then a lot of space is wasted since for each element *n*ot in U, a NIL value is stored. Hashing requires much less storage $\Theta(|K|)$ but on average only $O(1)$ time for all operations, if the hash function is easy to compute, i.e. in $O(1)$ time (and memory space).

Direct adressing guarantees $O(1)$ time in the worst case, but uses O($|U|$) memory space.

$h : U \rightarrow \{0, 1, \ldots, m-1\}$ is a *hash function* that maps the universe U into the hash table $T[0, 1, \ldots, m-1]$ of m slots.

If two elements of U hash to the same slot, this is a *collision*.

This is may happen if $|U| > m$ which is usually the case. (otherwise we could have stayed with direct adressing in the first place.) However, we try to avoid it so that if we store a set

with less or equal k elements, there should be few collisions. But we have to prepare for colisions. One solution is to store a list in each slot.

In order to resolve colisions, we can use *collision resolution by chaining*.

CHAINEDHASHSEARCH(T,K)
search for an element with key k in list $T[h(k)]$ and return it.

CHAINEDHASHINSERT(T,X)
Insert x in the head of list $T[h(key[x])]$.

CHAINEDHASHDELETE(T,X)
Delete x from list $T[h(key[x])]$

Analysis of hashing with chaining:

Given a hash table T with m slots and n elements stored, we define a *load factor* $\alpha = n/m$. It gives the average number of elements stored in a chain.

The worst-case behavior of hashing with chaining is terrible, but simple to analyse: Let's assume that all elements are in a single slot, so the delete and search operations take $\Theta(n)$ time.

We assume a uniform distribution of the hash values over the slots for the elements in the set, i.e. for the elements it is equally likely that they are hashed into every slot. This assumption is called *simple uniform hashing*.

The length of the list $T[j]$ is denoted by n_j so that $n = \sum_{j=0}^{m-1} n_j$.

Theorem 3 *In a hash table in which colisions are resolved by chaining, a (successful or unsuccessful) search takes expected time* $\Theta(1+\alpha)$*, under the assumption of simple uniform hashing.*

Proof in CLR, page 229

2.2.3 Hash functions

A good hash function satisfies the assumption of simple uniform hashing. Unfortunately, to check this property, the input (or the distribution of the input over U) must be known.

It is usually straight-forward to interpret any input as a numerical value, e.g. for strings, these can be interpreted as strings of 8-bit values and these can easily be transformed into a single value.

The division method uses a hash function of the type $h(k) = k \mod m$.

When using this method, some values of m are usually avoided. For example m should not be a power of two since the modulo operation just cuts off the most-significant bits of k. For words, that might lead to the situation that words are distributed according to their ending and some endings are much more likely than others. (Hint: list all the words that end on "xzy"...)

Usually, a prime not too close to a power of two is a good choice.

The multiplication method uses a hash function of the type $h(k) = \lfloor m((kA) \mod 1) \rfloor$ with $0 < A < 1$.

An advantage of the multiplication method is that the choice of m is not critical, so we can chose the size of the hash table independent of the hash function. We usually chose $m = 2^p$ since this simplifies the implementation.

We chose A to be a fraction of the form $s/2^w$ with s is an integer of the range $0 < s < 2^w$. We first multiply k with w-bit integer s and get a $2w$ bit value $r_1 2^w + r_0$ where r_1 is the high-order word of the value and r_2 is the low-order word of the value. The desired p-bit hash value consists of the p most-significant bits of r_0.

Knuth suggests that a good value for A is about $(\sqrt(5) - 1)/2$, so for $A = s/2^w$, s and w should be chosen accordingly.

There is an example in CLR (Page 232) that gives an example calculation.

2.2.4 Universal Hashing

For each hash function, we can usually chose a set of elements of U that give the same hash function. Therefore, if we use this set as an input, these elements will all be hashed into the same slot, leading to worst-case behavior.

Therefore, we chose a hash function (independent of the input values) randomly (but fixed for the runtime of our algorithm) in order to get a good performance on the average (input) case.

Let $\mathcal{H}$ be a finite collection of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m-1\}$. Such a collection is called *universal* if for each pair of distinct keys $k, l \in U$, the number of hash functions for which $h(k) = h(l)$ is at most $|\mathcal{H}|/m$.

If we chose a hash function from $\mathcal{H}$ randomly, the chance of a collision between two distinct keys k and l is at most $1/m$.

An example of a universal class of hash functions can be constructed as follows: Let the functions $h_{a,b}$ be defined as

$$h_{a,b} \quad = \quad ((ak + b) \mod p) \mod m \tag{2.1}$$

and p being a prime number and sufficiently large (so that for every possible value of k, $0 < k < p - 1$).

We assume that the universe of keys is larger than the number of slots (otherwiese we would not have to hash) so we can say that $p > m$.

For example: we have $p = 17$, $m = 6$, we have $h_{3,4}(8) = 5$

Then, the family of all such hash functions is

$$\mathcal{H}_{p,m} \quad = \quad \{h_{a,b} : a \in \mathbf{Z}_p^* \text{ and } b \in \mathbf{Z}_p \tag{2.2}$$

with $\mathbf{Z}_p = 0, 1, \ldots, p-1$ and $\mathbf{Z}_p^* = 1, \ldots, p-1$.

$|\mathcal{H}_{p,m}| = p(p-1)$ where m can be chosen freely.

Theorem 4 *The class $\mathcal{H}_{p,m}$ of has functions is universal.*

Proof in CLR, page 235

2.2.5 Other ways to resolve collisions

By using *open adressing*, all values are stored in the table. For that, we do need more than one slot where a key could be stored. To do this, we extend our hash function:

$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$

Now, if we insert something, we *probe* several slots for free space and insert the element into the first free one. Consequently, if we search for an element, we $probe$ each possible slot until we find a NIL value.

The *probe sequence* $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ is a permutation of $\langle 0, 1, \ldots, m-1\rangle$.

By this, every slot is probed exactly once and eventually, the complete hash table is filled. If no NIL value is found after m probes, the hash table is indeed full and we return an error.

HASHINSERT(T,k)
  $i \leftarrow 0$
  repeat
    $j \leftarrow h(k, i)$
    if $T[j] =$ NIL **then**
      $T[j] \leftarrow k$
      return j
    else
      $i \leftarrow i + 1$
    end if
  until $i = m$
  error "hash table overflow"

HASHSEARCH(T,k)
  $i \leftarrow 0$
  repeat
    $j \leftarrow h(k, i)$
    if $T[j] = k$ **then**
      return j
    else
      $i \leftarrow i + 1$
    end if
  until $T[j] =$NIL or $i = m$
  return NIL

Unfortunately, deleting is not so simple with open adressing since the saerch routine ends at the first NIL value encountered, but there may be elements stored "behind" that if this NIL has been written by a HASHDELETE operation. One solution is to use another special value DELETED and make sure that no regular key has that value.

For our analysis of open adressing, we are assuming that each key has equally likely one of the $m!$ possible probe sequences. This assumption is called *uniform hashing*.

Uniform hasing is a generalisation of simple uniform hasing. We're not only assuming

uniformity for the first hash value but for the sequence of hash values needed by the probe sequence. True uniform hasing is hard to implement, but there are good approximations.

In order to create the $h(k,i)$ functions, we can use a *auxiliary hash function* h'. This method is called *linear probing*

$$h(k,i) = (h'(k) + i) \mod m$$

The slots probed are $T[h'(k)], T[h'(k)+1], \ldots, T[m-1], T[0], \ldots, T[h'(k)-1]$.

This algorithm suffers from so-called *primary clustering*: Long runs of occupied slots build up, increasing the average search time. This happens since an empty slot preceded by i full slots gets filled in the next step with probability $(i+1)/m$.

Instead of running linearily, we can use *quadratic probing*

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$$

where c_1 and c_2 are auxiliary constants.

This works much better, but in order to make use of the full hash table (by makeing sure that the probe sequence is a permutation of $1 \ldots m-1$), we have to use special values for m, c_1 and c_2.

An even better approach is *double hashing* where we use two auxiliary hash functions.

$$h(k,i) = (h_1(k) + i h_2(k)) \mod m$$

This works much better, the step value for probing is determined by the original key value. Since two hash functions are used, the step value can be different for two keys $k_1 \neq k_2$ with $h_1(k_1) = h_1(k_2)$, since this does not imply $h_2(k_1) = h_2(k_2)$.

In order to cover all slots of the hash table, the value of h_2 has to be relatively prime to m. In order to make sure that this is the case, we can make m a power of two and chose h_2 so that it always produces odd numbers. Another approach is to chose m prime and to make h_2 always produce an integer less than m:

$$\begin{aligned} h_1(k) &= k \mod m \\ h_2(k) &= 1 + (k \mod m') \end{aligned}$$

where m' is chosen slightly less than m.

Theorem 5 *Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.*

Corollary 2 *Inserting an element into an open-address hash table with load factor α requires at most $1/(1-\alpha)$ probes on average, assuming uniform hashing.*

Theorem 6 *Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most*

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

Proofs in CLR, Page 241-244

2.3 Binary Search Trees

A binary search tree is a binary tree in which the keys are stored according to the *binary search tree property*:

Let x be a node in a binary search tree. If y is a node in the left subtree of x then $key[y] \leq key[x]$, if y is a node in the right subtree of x then $key[y] \geq key[x]$.

One tree node has the fields key, p for parent, $left$ and $right$ along with possible satellite data.

INORDERTREEWALK(x)
  if $x \neq$ NIL **then**
    INORDERTREEWALK($left[x]$)
    print $key[x]$
    INORDERTREEWALK($right[x]$)
  end if

Theorem 7 *If x is the root of an n-node subtree, then the call* INORDERTREEWALK(x) *takes $\Theta(n)$ time.*

Proof in CLR, Page 255

The TREESEARCH function looks for the element with key k in the subtree that is rooted by x.

TREESEARCH(x, k)
  if $x =$NIL or $k = key[x]$ **then**
    return x
  end if
  if $k < key[x]$ **then**
    return TREESEARCH($left[x], k$)
  else
    return TREESEARCH($right[x], k$)
  end if

Obviously, it is also possible to write an iterative version of the function.

ITERATIVETREESEARCH(x, k)
  while $x \neq$NIL and $k \neq key[x]$ **do**
    if $k < key[x]$ **then**
      $x \leftarrow left[x]$
    else
      $x \leftarrow right[x]$
    end if

end while
return x

By going always left, it is possible to find the minimum element of the tree.

TREEMINIMUM(x)
while $left[x] \neq NIL$ **do**
$x \leftarrow left[x]$
end while
return x

By going always right, it is possible to find the maximum element of the tree.

TREEMAXIMUM(x)
while $right[x] \neq NIL$ **do**
$x \leftarrow right[x]$
end while
return x

The Successor function identifies the element with the next bigger key. It can be used to iterate over the tree in increasing key values.

TREESUCCESSOR(x)
if $right[x] \neq$NIL **then**
return TREEMINIMUM$(right[x])$
end if
$y \leftarrow p[x]$
while $y \neq$NIL and $x = right[y]$ **do**
$x \leftarrow y$
$y \leftarrow p[y]$
end while
return y

Insertion is similar to search: first the algorithm does "as if" it would be searching for an element with a similar key, thereby finding the right spot for the insertion. Then it rearranges the pointers.

The insertion procedure receives a pointer to a pre-initialized node z with $key[z] = v$, $left[z] =$NIL, $right[z] =$NIL and additional satellite data.

TREEINSERT(T, z)
$y \leftarrow$NIL
$x \leftarrow root[T]$
while $x \neq$NIL **do**
$y \leftarrow x$
if $key[z] < key[x]$ **then**
$x \leftarrow left[x]$
else
$x \leftarrow right[x]$
end if
end while
$p[z] \leftarrow y$
if $y =$NIL **then**
$root[T] \leftarrow z$

```
  else
    if key[z] < key[y] then
      left[y] ← z
    else
      right[y] ← z
    end if
  end if
```

Deletion is purely rearranging the pointers. There are multiple cases to be handled, therefore the function looks rather complicated.

The procedure receives a pointer to the node z that is to be deleted.

It returns a pointer to the node removed so that this node can be reused by putting it into a free list.

```
TREEDELETE(T, z)
  if left[z] =NIL or right[z] =NIL then
    y ← z
  else
    y ←TREESUCCESSOR(z)
  end if
  if left[y] ≠NIL then
    x ← left[y]
  else
    x ← right[y]
  end if
  if x ≠NIL then
    p[x] ← p[y]
  end if
  if p[y] =NIL then
    root[T] ← x
  else
    if y = left[p[y]] then
      left[p[y]] ← x
    else
      right[p[y]] ← x
    end if
  end if
  if x ≠ y then
    key[z] ← key[y]
    copy y's satellite data into z
  end if
  return y
```

2.4 Red-Black Trees

The problem with binary search trees is that their depth is only limited by the amount of nodes in the tree. Since the depth is the determining factor for the runtime of most operations, we want to keep it as small as possible, i.e. logarithmic in the number of elements stored. One way to assure this is to check the length of every path from a node to the root and making sure that this path never grows too much, i.e. if a path becomes too long, we have to re-arrange the elements in the tree so that the length of the path is limited.

A binary search tree is a red-black tree if it satisfies the following *red-black properties*:

1. Every node is either red or black
2. The root is black
3. Every leaf (NIL) is black
4. if a node is red, then both children are black
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

We call the number of black nodes on any path from a node x down to the leaf the *black-height* of the node, denoted $bh(x)$.

Lemma 2 *A red-black tree with n internal nodes has height at most $2(lg(n+1))$.*

Running TREEINSERT and TREEDELETE take $O(\lg n)$ time, but unfortunately, the tree produced by them may not be a red-black tree anymore. In order to maintain the red-black properties, we may have to change some pointers and change the color of some nodes.

A *rotation* changes the pointer structure of a tree so that the red-black properties are met.

When doing a left rotation on a node x, we assume that its right child is not $nil[T]$.

The following procedure assumes that $right[x] \neq nil[T]$ and that the root's parent is $nil[t]$.

LEFTROTATE(T, x)
  $y \leftarrow right[x]$
  $right[x] \leftarrow left[y]$
  $p[left[y]] \leftarrow x$
  $p[y] \leftarrow p[x]$
  if $p[x] = nil[T]$ **then**
    $root[T] \leftarrow y$
  else
    if $x = left[p[x]]$ **then**
      $left[p[x]] \leftarrow y$
    else
      $right[p[x]] \leftarrow y$
    end if
  end if
  $left[y] \leftarrow x$
  $p[x] \leftarrow y$

```
RBINSERT(T, z)
  y ← nil[T]
  x ← root[T]
  while x ≠ nil[T] do
    y ← x
    if key[z] < key[x] then
      x ← left[x]
    else
      x ← right[x]
    end if
  end while
  p[z] ← y
  if y = nil[T] then
    root[T] ← z
  else
    if key[z] < key[y] then
      left[y] ← z
    else
      right[y] ← z
    end if
  end if
  left[z] ← nil[T]
  right[z] ← nil[T]
  color[z] ←RED
  RBINSERTFIXUP(T, z)
```

```
RBINSERTFIXUP(T, z)
  while color[p[z]] =RED do
    if p[z]=left[p[p[z]]] then
      y ← right[p[p[z]]]
      if color[y] =RED then
        color[p[z]] ←BLACK
        color[y] ←BLACK
        color[p[p[z]]] ←RED
        z ← p[p[z]]
      else
        if z = right[p[z]] then
          z ← p[z]
          LEFTROTATE(T, z)
        end if
        color[p[z]] ←BLACK
        color[p[p[z]]] ←RED
        RIGHTROTATE(T, p[p[z]])
      end if
    else
      (same as then clause with "right" and "left" exchanged)
    end if
  end while
  color[root[T]] ←BLACK
```

```
RBDELETE(T, z)
  if left[z] = nil[T] or right[z] = nil[t] then
    y ← z
  else
    y ←TREESUCCESSOR(z)
  end if
  if left[y] ≠ nil[T] then
    x ← left[y]
  else
    x ← right[y]
  end if
  p[x] ← p[y]
  if p[y] = nil[T] then
    root[T] ← x
  else
    if y = left[p[y]] then
      left[p[y]] ← x
    else
      right[p[y]] ← x
    end if
  end if
  if x ≠ y then
    key[z] ← key[y]
    copy y's satellite data into z
  end if
  if color[y] =BLACK then
    RBDELETEFIXUP(T, x)
  end if
  return y
```

```
RBDELETEFIXUP(T, z)
  while x ≠ root[T] and color[x] =BLACK do
    if x=left[p[x]] then
      w ← right[p[x]]
      if color[w] =RED then
        color[w] ←BLACK
        color[p[x]] ←RED
        LEFTROTATE(T, p[x])
        w ← right[p[x]]
      end if
      if color[left[w]] =BLACK and color[right[w]] = BLACK then
        color[w] ←RED
        x ← p[x]
      else
        if color[right[w]] =BLACK then
          color[left[w]] ←BLACK
          color[w] ←RED
          RIGHTROTATE(T, w)
          w ← right[p[x]]
        end if
        color[w] ← color[p[x]]
        color[p[x]] ←BLACK
        color[right[w]] ←BLACK
        LEFTROTATE(T, p[x])
        x ← root[T]
      end if
    else
      (same as then clause with "right" and "left" exchanged)
    end if
  end while
  color[x] ←BLACK
```

Chapter 3

Advanced Design and Analysis Techniques

- Dynamic Programming
- Greedy Algorithms
- Amortized Analysis

3.1 Dynamic Programming(DP)

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from the computed information.

DP Example: Assembly-Line Scheduling:

- A factory with two assembly lines $S1, S2$, each one having n stations.
- Stations $(S_{1,n})$ and $(S_{2,n})$ do the same thing but possibly need different time. Additional entry and exit times for the two lines are e_1, e_2, x_1, x_2.
- Station (S_i, n) needs $a_{i,j}$ time for assembly.
- Transition time $(S_{i,n}) \rightarrow (S_{i,n+1})$ can be neglected
- Transition time $(S_{1,n}) \rightarrow (S_{2,n+1}) = t_1, n$ and Transition time $(S_{2,n}) \rightarrow (S_{1,n+1}) = t_2, n$ cannot be neglected.
- Problem: Find the sequence of stations for a "rush order", i.e. an assembly that takes minimal time.

A first idea:

- It takes $O(n)$ time to compute the time for one given path.
- There are 2^n possible ways, so compute the time for all of them and then chose the fastest one.
- Bad idea if n is large.

A Better idea...DP (Step 1)

- An optimal path trough Station $(S_{1,j})$ for $j = 1$ is easy to compute.
- An optimal path trough Station $(S_{1,j})$ for $j > 1$ can be created in two ways: Either through station $(S_{1,j-1})$ or through $(S_{2,j-1})$.
- If the path was through $(S_{1,j-1})$, then the path through $(S_{1,j})$ is the time of the path through $(S_{1,j-1}) + a_{1,j}$.
- If the path was through $(S_{2,j-1})$, then the path through $(S_{1,j})$ is the time of the path through $(S_{2,j-1}) + t_2, j - 1 + a_{1,j}$.
- Idea: An optimal solution consists of optimal sub-solutions.

DP (Step 2):

- Idea: An optimal solution consists of optimal sub-solutions.
- Define the optimal solution recursively.
- $f_i[j]$ is the fastest time to get thorugh Station $S_{i,j}$
- $f* = \min(f_1[n] + x_1, f_2[n] + x_2)$
- $f_1[1] = e_1 + a_{1,1}$
- $f_2[1] = e_2 + a_{2,1}$
- $f_1[j] = \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a1, j$ for $j > 1$
- $f_2[j] = \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a2, j$ for $j > 1$
- Now we could write a recursive function to compute $f*$.
- Problem: Still exponential runtime!

FASTESTWAY(a, t, e, x, n) (1)

$f_1[1] \leftarrow e_1 + a_{1,1}$
$f_2[1] \leftarrow e_2 + a_{2,1}$
for $j \leftarrow 2$ **to** n **do**
  if $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$ **then**
    $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$
    $l_1[j] \leftarrow 1$
  else
    $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$
    $l_1[j] \leftarrow 2$
  end if

if $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$ **then**
$f_2[j] \leftarrow f_2[j-1] + a_{2,j}$
$l_2[j] \leftarrow 2$
else
$f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$
$l_2[j] \leftarrow 1$
end if
end for

FASTESTWAY(a, t, e, x, n) (2)

if $f_1[n] + x_1 \leq f_2[n] + x_2$ **then**
$f* \leftarrow f_1[n] + x_1$
$l* \leftarrow 1$
else
$f* \leftarrow f_2[n] + x_2$
$l* \leftarrow 2$
end if

PRINTSTATIONS(l, n):

$i \leftarrow l*$
print "line " i ", station" n
for $j \leftarrow n$ **downto** 2 **do**
$i \leftarrow l_i[j]$
print "line " i ", station" j-1
end for

Other applications of DP:

- Matrix-Chain-Multiplication
- Longest Common Subsequence

3.2 Greedy algorithms

- Determine the optimal substructure of the problem.
- Develop a recursive solution.
- Prove that at any stage of the recursion, one of the optimal choices is the greedy choice. Thus, it is alway safe to make the greedy choice.
- Show that all but one of the subproblems created by making the greedy choice are empty.
- Develop a recursive algorithm that implements the greedy strategy.
- Convert the recursive algorithm into an iterative algorithm.

The Activity Selection Problem:

- Given: A number of n activities and for each activity its start time s_i and end time f_i.
- Problem: Find the maximal subset of activities that are compatible with eachother, i.e. that do not take place in parallel.

Definitions:

- Def: $S_{ij} = \{a_k \in S : f_i \leq S_k \leq f_k \leq s_j\}$
- f_0 additional activity that ends before any other starts.
- f_{n+1} additional activity that starts after any other ends.
- $f_0 \leq f_1 \leq \cdots \leq f_n \leq f_{n+1}$, i.e. sorted according to the finishing times.
- A_{ij}: Optimal solution for S_{ij}

Recursive solution:

- $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ if the optimal solution contains a_k.
- $c[i,j] = c[i,k] + c[k,j] + 1$

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\limits_{i<k<j} \{c[i,k] + c[[k,j] + 1 & \text{if } S_{ij} \neq \emptyset \end{cases}$$

RECURSIVEACTIVITYSELECTOR(s, f, i, j)

```
m ← i + 1
while m < j and s_m < f_i do
    m ← m + 1
end while
if m < j then
    return {a_m} ∪ RECURSIVEACTIVITYSELECTOR(s, f, m, j)
else
    return ∅
end if
```

GREEDYACTIVITYSELECTOR(s, f, i, j)

```
n ← length[s]
A ← {a_1}
i ← 1
for m ← 2 to n do
    if s_m ≥ f_i then
        A ← A ∪ a_m
        i ← m
    end if
end for
return A
```

Other applications of greedy:

- fractional Knapsack
- not 0-1-Knapsack: Greedy choice can lead to suboptimal results.

3.3 Amortized Analysis

- Idea: Sum up the time needed for a sequence of operations and calculate the individual operation time by averaging over the sequence.
- Rationale: In some algorithms more time is needed for a single operation, but the following operations will benefit.
- three techniques: Aggregate Analysis, Accounting Method and Potential Method.
- two simple examples: a stack and a binary counter

Stack:

- Normal PUSH and POP operations that take $O(1)$ time
- New MULTIPOP operation:
  MULTIPOP(S, k)
  while not STACKEMPTYS and $k \neq 0$ **do**
  POP(S)
  $k \leftarrow k - 1$
  end while

INCREMENT(A)

$i \leftarrow 0$
while $i < length[A]$ and $A[i] = 1$ **do**
$A[i] \leftarrow 0$
$i \leftarrow i + 1$
end while
if $i < length[A]$ **then**
$A[i] \leftarrow 1$
end if

Aggregate Analysis:

- Stack: The worst-case runtime of the MULTIPOP operation is $O(n)$, so a sequence of n stack operations have worst-case runtime of $O(n^2)$
- Aggregate Analysis shows that any sequence of n operations can at most take $O(n)$ time.
- Binary Counter: A naive analysis shows that an increment operation can take $O(k)$ worst-case on a k-bit binary counter, thus n operations can take up to $O(kn)$ time.
- Aggregate Analysis shows that n increment operations at most take $O(n)$ time.

Accounting and Potential Methods:

- Accounting Method: Keep track of the “expenses” of the operations and charge more to certain operations. Then use the charged ammount on other operations
- Potential Method: Define a potential function for the datastructure. The cost of an operation is then its runtime plus the change in potential.Operations that lower the potential therefore have a lower cost, operations that raise the potential have higher cost.

Chapter 4

graph algorithms

4.1 definitions

Definition 8 *A* directed graph *or* digraph G *is a pair* (V, E) *where* V *is a finite set and* E *is a binary relation on* V. V *is called* vertex set *of* G *and its elements are called* vertices. *The set* E *is called* edge set *of* G *an its elements are called* edges.

In a directed graph we say that an edge (u, v) is incident from or leaves v and is incident to or enters u. It is possible to have self-loops, i.e. (v, v).

Definition 9 *In an* undirected graph $G = (V, E)$*, the edge set consists of unordered pairs, i.e. an edge is a set* $\{u, v\}$ *where* $u, v \in V$ *and* $u \neq v$.

In an undirected graph we say that (u, v) is incident on u and v.

If there is an edge (u, v) in G, we say that v is adjacent to u, in undirected graphs, this adjacency relation is symmetric.

The binary relation E can be represented as a collection (i.e. a list) of pairs or as a function from V to its power set $P(V)$ (i.e. as a matrix).

The degree of a vertex in an undirected graph is the number of edges incident on it. A vertex whose degree is 0 is called isolated. In a directed graph, the out-degree of a vertex is the number of edges leaving the vertex and the in-degree is the number of edges entering the vertex.

A path of length k from a vertex u to a vertex u' in a Graph $G = (V, E)$ is a sequence $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ of vertices such that $u = v_0$ and $u' = v_k$ and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$. The path contains the vertices v_i and the edges (v_{i-1}, v_i).

A subpath of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a consecutive sub-sequence of vertices, i.e. for any $0 \leq i \leq j \leq k$ the path $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ is a subpath of p.

In a directed graph, a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ forms a cycle if $v_0 = v_k$. The cycle is simple of $v_1, \ldots, v_k$ are distinct. A directed graph with no self-cycles is called simple. A graph with no cycles is called acyclic.

An undirected graph is connected if evey pair of vertices is connected by a path. The connected components of a graph are the equivalence classes of vertices under the relation "is reachable from". An undirected graph is connected if and only if it has only one connected component.

A directed graph is strongly connected if every two vertices are reachable from eachother. The strongly connected components of a directed graph are the equivalence classes under the relation "are mutually reachable".

Two graphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic if there exists a bijection $f : V \leftarrow V'$ such that $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$. A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V \subseteq V'$ and $E \subseteq E'$. Given a set $V' \subseteq V$ the subgraph of G induced by V' is the graph $G' = (V', E')$ where $E' = \{(u, v) \in E : u, v \in V'\}$.

Special graphs:

1. A complete graph is an undirected graph in which every pair for vertices is adjacent.
2. An undirected bipartite graph is an undirected graph $G = (V, E)$ in which V can be partitioned into two sets V_1 and V_2 such that $(u, v) \in E$ implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$.
3. A directed bipartite graph is a directed graph $G = (V, E)$ in which V can be partitioned into two sets V_1 and V_2 such that $(u, v) \in E$ implies $u \in V_1$ and $v \in V_2$.
4. An acyclic undirected graph is a forrest
5. A connected acyclic undirected graph is a tree
6. A directed acyclic graph is a dag.

4.2 Breadth First Search

BFS(G, x)
  for each vertex $u \in V[G] - \{s\}$ **do**
    $color[u] \leftarrow$ WHITE
    $d[u] \leftarrow \infty$
    $\pi[u] \leftarrow$NIL
  end for
  $color[s] =$GRAY
  $d[s] \leftarrow 0$
  $\pi[s] \leftarrow$NIL
  $Q \leftarrow \emptyset$
  ENQUEUE(Q, s)
  while $Q \neq \emptyset$ **do**
    $u \leftarrow$DEQUEUE(Q)
    for each $v \in Adj[u]$ **do**
      if $color[v] =$WHITE **then**
        $color[v] =$GRAY
        $d[v] \leftarrow d[u] + 1$
        $\pi[v] \leftarrow u$
        ENQUEUE(Q, v)
      end if

end for
$color[v]$ =BLACK
end while

Definition 10 *Let $G = (V, E)$ be a graph and $s, v \in V$ two vertices. The* shortest path distance $\delta(x, v)$ *from s to v is the minimum number of edges in a path from s to v. If there is no path from s to v, then $\delta(s, v) = \infty$. A path of length $\delta(s, v)$ is a* shortest path *from s to v.*

Theorem 8 (Correctness of breadth-first search) *Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $d[v] = \delta(s, v)$ for all $v \in V$. Moreover, for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $\pi(v)$ followed by an edge $(\pi[v], v)$.*

Proof in CLR, page 535 ff.

The procedure BFS produces a breath-first tree as it searches the graph, this tree is represented by $\pi[v]$ in each vertex.

For a graph $G = (V, E)$ with $s \in V$, the *predecessor subgraph* of G is $G_\pi = (V_\pi, E_\pi)$ with $V_\pi = \{v \in V : \pi[v] \neq$NIL$\} \cup \{s\}$ and $E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}$.

The predecessor subgraph G_π is a *breadth-first tree* if V_π consists of the vertices reachable from s and, for all $v \in V$, there is a unique simple path from s to v in G_π that is also a shortest path from s to v in G.

Theorem 9 (Properties for free trees) *Let $G = (V, E)$ be an undirected graph. The following statements are equivalent:*

1. *G is a free tree.*
2. *Any two vertices in G are connected by a unique simple path.*
3. *G is connected, but if any $E \in E$ is removed, the resulting graph is not connected.*
4. *G is connected and $|E| = |V| - 1$.*
5. *G is acyclic and $|E| = |V| - 1$.*
6. *G is acyclic, but if one edge is added to E, the graph contains at least one cycle.*

A *rooted tree* is a free tree in which one of the vertices is distinguished from the others. It's called the root.

PRINTPATH(G, s, v)
if $v = s$ **then**
print s
else
if $\pi[v]$ =NIL **then**
print No path from s to v exists

```
    else
      PRINTPATH(G, s, π[v])
      print v
    end if
  end if
```

4.3 Depth First Search and applications

```
DFS(G)
  for each vertex u ∈ V[G] do
    color[u] ← WHITE
    π[u] ←NIL
  end for
  time ← 0
  for each u ∈ V[G] do
    if color[u] =WHITE then
      DFSVISIT(u)
    end if
  end for
```

```
DFSVISIT(u)
  color[u] =GRAY
  time ← time + 1
  d[u] ← time
  for each v ∈ Adj[u] do
    if color[v] =WHITE then
      π[v] ← u
      DFSVISIT(v)
    end if
  end for
  color[u] =BLACK
  time ← time + 1
  f[u] ← time
```

Definition 11 *A* topological sort *of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if $(u, v) \in E$, then u appears in the ordering before v.*

```
TOPSORT(G)
  call DFS(G) to compute the finishing times f[v] for each vertex v.
  As each vertex is finished, insert it into the front of a linked list.
  return the linked list of vertices
```

Note that TOPSORT reverses the finish list by always inserting in front.

Definition 12 *The* **transpose** *of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. Thus, G^T is G with all edges reversed.*

Read $u \rightsquigarrow v$ as " there exists a path from u to v ".

A SCC of a Graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices $u, v \in C : u \rightsquigarrow v$ and $v \rightsquigarrow u$.

STRONGLYCONNECTEDCOMPONENTS(G)
DFS(G) to compute $f[u]$ for each $u \in V$.
compute G^T
call DFS(G^T) but in the main loop of DFS, consider the vertices on order of decreasing $f[u]$.
output the vertices of each tree in the resulting depth-first forrest as a separate SCC.

The algorithm computes the *component graph* $G^{SCC} = (V^{SCC}, G^{SCC})$. Suppose that the graph G has the SCCs $C_1, \ldots, C_k$. $V^{SCC} = \{v_1, \ldots, v_k\}$, so one vertex for each SCC. There is an edge $(u, v) \in E^{SCC}$ if G contains a directed edge (x, y) from some $x \in C_i$ and $y \in C_j$. That graph is a dag, otherwise at least two components would be merged until it is a dag.

4.4 Minimum Spanning Trees

- Input: A connected graph $G = (V, E)$ and an edge weight function $w(u, v)$.
- Output: An acyclic subset $T \subseteq E$ that connects all $v \in V$ and whose total weight $w(T) = \sum_{(u,v)\in T} w(u, v)$ is minimal.

Growing a MST

- Loop invariant: Prior to each iteration, A is a subset of some minimum spanning tree.
- At each step, determine (u, v) that can be added to A without violating the loop invariant. (u, v) is called a *safe edge*.

GENERIC-MST
$A \leftarrow \emptyset$
while A does not form a spanning tree **do**
find an edge (u, v) that is safe for A
$A \leftarrow A \cup \{(u, v)\}$
end while
return A

Correctness of GENERIC-MST

- Initialization: After line 1, the set A trivially satisfies the loopminvariant.
- Maintenance: The loop in lines 2-5 maintains the invariant by adding only safe edges
- Termination: All edges that are added to A are in a minimum spanning tree subset, so the first tree that forms a spanning tree must be a minimum spanning tree.

Idea of KRUSKAL-MST

- Sort all edges by weight
- Start with the least-weight-edge
- Add the edge to the MST if it connects two trees in the forrest and not two nodes in the same tree.
- Use a Disjoint-Set datastructure to represent the trees.

Disjoint-Set Datastructures

- Operations MAKE-SET(x), UNION(x, y), FINDSET(x)
- Implementation: using lists or disjoint-set forrests
- Application example: identify connected components of a graph

DJSCONNECTEDCOMPONENTS(G)
for each vertex $v \in V[G]$ **do**
  MAKESET(v)
end for
for each edge $(u, v) \in E[G]$ **do**
  if FINDSET$(u) \neq$ FINDSET(v) **then**
    UNION(u, v)
  end if
end for

DJSSAMECOMPONENT(u, v)
if FINDSET$(u) =$ FINDSET(v) **then**
  return true
else
  return false
end if

Idea of KRUSKAL-MST

- Sort all edges by weight
- Start with the least-weight-edge
- Add the edge to the MST if it connects two trees in the forrest and not two nodes in the same tree.
- Use a Disjoint-Set datastructure to represent the trees.

KRUSKAL-MST
$A \leftarrow \emptyset$
for each vertex $v \in V[G]$ **do**
  MAKESET(v)
end for
sort the edges of E by w, smallest first
for each edge $(u, v) \in E[G]$ sorted by w **do**
  if FINDSET$(u) \neq$ FINDSET(v) **then**
    $A \leftarrow A \cup \{(u, v)\}$

UNION(u, v)
end if
end for
return A

Idea of PRIM-MST

- Start with a single node
- Always add a single node to the tree
- Keep the remaining nodes in a min priority queue Q indexed by the least-weight edge connecting that node to the tree.
- Maintain A implicitly as a predecessor subgraph table $\pi[v]$.

PRIM-MST(1)
for each vertex $u \in V[G]$ **do**
$key[u] \leftarrow \infty$
$\pi[u] \leftarrow$NIL
end for
$key[r] \leftarrow 0$
$Q \leftarrow V[G]$

PRIM-MST(2)
while $Q \neq \emptyset$ **do**
$u \leftarrow$EXTRACTMIN(Q)
for each vertex $v \in Adj[u]$ **do**
if $v \in Q$ and $w(u, v) < key[v]$ **then**
$\pi[u] \leftarrow u$
$key[u] \leftarrow w(u, v)$
end if
end for
end while

4.5 single source shortest path

Given a weighted, directed graph $G = (V, E)$ with weight function $e : E \rightarrow \mathbb{R}$ mapping edges to real-valued weights.

The weight of path $p = \langle v_0, \ldots, v_k \rangle$ is the sum of the weights of the edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

The *shortest path weight* from u to v is defined as

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \rightsquigarrow^p v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

A *shortest path* from vertex u to vertex v is then defined as any path p with $w(p) = \delta(u, v)$.

Problems:

- Single source shortest path
- Single destination shortest path
- Single pair shortest path
- All pairs shortest path

Lemma 3 *Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \rightsquigarrow \mathbb{R}$, let $p = \langle v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_1 to vertex v_k and for any i and j such that $1 \leq i \leq j \leq k$, let $p_{ij} = \langle v_i, \ldots, v_j \rangle$ be the subpath of p from v_i to v_j. Then p_{ij} is a shortest path from v_i to v_j.*

To represent the shortest paths we use the $\pi[v]$ like in BFS, again forming a predecessor subgraph.

The technique used by this algorithm is called *relaxation*. $d[v]$ is called *shortest-path estimate*, it is a current upper bound of the shortest-path weight.

For a graph $G = (V, E)$ with $s \in V$, the *predecessor subgraph* of G is $G_\pi = (V_\pi, E_\pi)$ with $V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$ and $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$.

INITIALIZESINGLESOURCE(G, s)
  for each vertex $v \in V[G]$ **do**
    $d[v] \leftarrow \infty$
    $\pi[v] \leftarrow$ NIL
  end for
  $d[s] \leftarrow 0$

RELAX(u, v, w)
  if $d[v] > d[u] + w(u, v)$ **then**
    $d[v] \leftarrow d[u] + w(u, v)$
    $\pi[v] \leftarrow u$
  end if

BELLMANFORD(G, w, s)
  INITIALIZESINGLESOURCE(G, s)
  for $i = 1$ **to** $|V[G]| - 1$ **do**
    for each edge $(u, v) \in E$ **do**
      RELAX(u, v, w)
    end for
  end for
  for each edge $(u, v) \in E$ **do**
    if $d[v] > d[u] + w(u, v)$ **then**
      return FALSE
    end if
  end for
  return TRUE

BELLMANFORD runs in $O(VE)$.

DAGSHORTESTPATH(G, w, s)
  topologically sort the vertices of G
  INITIALIZESINGLESOURCE(G, s)
  for each vertex u, taken in topologically sorted order **do**
    for each vertex $v \in Adj[u]$ **do**
      RELAX(u, v, w)
    end for
  end for

The runtime of DAGSHORTESTPATH is $\Theta(V + E)$ since TOPSORT takes $\Theta(V + E)$, there are V iterations of the for loop and all the iterations treat $|E|$ edges.

If G is a dag, its predecessor subgraph formed by $\pi[v]$ is a shortest-path tree.

If we only allow non-negative edge weights, we can use Dijkstra's Algorithm.

DIJKSTRA(G, w, s)
  INITIALIZESINGLESOURCE(G, s)
  $S \leftarrow \emptyset$
  $Q \leftarrow V[G]$
  while $Q \neq \emptyset$ **do**
    $u \leftarrow$EXTRACTMIN(Q)
    $S \leftarrow S \cup \{u\}$
    for each vertex $v \in Adj[u]$ **do**
      RELAX(u, v, w)
    end for
  end while

The runtime of Dijkstra's algorithm depends on the implementation of the Priority Queue.

4.6 All-Pairs Shortest Path

We can solve this problem by running a single-source shortest path algorithm $|V|$ times. If there are no negative edge weights, we can use Djikstra. If there are, we have to use Bellman Ford.

We represent the graph in a adjacency matrix representation of the weight function: $W = (w_{ij})$ where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{the weight of directed edge } (i,j) & \text{if } i \neq j \text{ and } (i,j \in E) \\ \infty & \text{if } i \neq j \text{ and } (i,j \notin E) \end{cases}$$

EXTENDEDSHORTESTPATHS(L, W)
  $n \leftarrow rows[L]$
  let $L' = (l'_{ij})$ be an $n \times n$ matrix.
  for $i \leftarrow 1$ **to** n **do**

```
    for j ← 1 to n do
      l'_ij ← ∞
      for k ← 1 to n do
        l'_ij ← min(l'_ij, l_ik + w_kj
      end for
    end for
  end for
  return L'
```

```
MATRIXMULTIPLY(A, B)
  n ← rows[L]
  let C be an n × n matrix.
  for i ← 1 to n do
    for j ← 1 to n do
      c_ij ← 0
      for k ← 1 to n do
        c_ij ← c_ij + a_ik · b_kj
      end for
    end for
  end for
  return C
```

```
SLOWALLPAIRSSHORTESTPATHS(W)
  n ← rows[W]
  L^(1) ← W
  for m ← 2 to n − 1 do
    L(m) ←EXTENDEDSHORTESTPATHS(L^(m−1), W)
  end for
  return L^(n−1)
```

```
FASTERALLPAIRSSHORTESTPATHS(W)
  n ← rows[W]
  L^(1) ← W
  m ← 1
  while m < n − 1 do
    L(2m) ←EXTENDEDSHORTESTPATHS(L^(m), L^(m))
    m ← 2m
  end while
  return L^(m)
```

If we consider the operation executed by EXTENDSHORTESTPATHS as some form of matrix multiplication, our algorithm computes:

$$\begin{aligned} L^{(1)} &= L^{(0)} \cdot W = W \\ L^{(2)} &= L^{(1)} \cdot W = W^2 \\ L^{(3)} &= L^{(2)} \cdot W = W^3 \\ L^{(4)} &= L^{(3)} \cdot W = W^4 \\ &\vdots \\ L^{(n-1)} &= L^{(n-2)} \cdot W = W^{n-1} \end{aligned}$$

One operation takes $\Theta(n^3)$ time, by using the algorithm SLOWALLPAIRSSHORTESTPATHS, the total runtime is $\Theta(n^4)$.

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

FLOYDWARSHALL(W)
  $n \leftarrow rows[W]$
  $D^{(0)} \leftarrow W$
  for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
      for $j \leftarrow 1$ **to** n **do**
        $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
      end for
    end for
  end for
  return $D^{(n)}$

TRANSITIVECLOSURE(W)
  $n \leftarrow |V[G]|$
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      if $i = j$ or $(i, j) \in E[G]$ **then**
        $t_{ij}^{(0)} \leftarrow 1$
      else
        $t_{ij}^{(0)} \leftarrow 0$
      end if
    end for
  end for
  for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
      for $j \leftarrow 1$ **to** n **do**
        $t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \wedge \left(t_{ik}^{(k-1)} \vee t_{kj}^{(k-1)}\right)$
      end for
    end for
  end for
  return $T^{(n)}$

JOHNSON(G)
  compute G', where $V[G'] = V[G] \cup \{s\}$,
    $E[G'] = E[G] \cup \{(s, v) : v \in V[G]\}$ and
    $w(s, v) = 0$ for all $v \in V[G]$.
  if BELLMANFORD(G', w, s) =FALSE **then**
    print "The input graph contains negative cycles!"
  else
    for each vertex $v \in V[G']$ **do**
      set $h(v)$ to the value of $\delta(s, v)$ computed by BELLMANFORD.
    end for
    for each edge $(u, v) \in E[G']$ **do**
      $\hat{w}(u, v) \leftarrow w(u, v) + h(u) - h(v)$
    end for
    for each vertex $u \in V[G]$ **do**
      run DIJKSTRA$(G, \hat{w}, u)$ to compute $\hat{\delta}(u, v)$ for all $v \in V[G]$

for each vertex $v \in V[G]$ **do**
$d_{uv} \leftarrow \hat{\delta}(u, v) + h(v) - h(u)$
end for
end for
end if

4.7 Summary

Slide FCS1 Recap:

- Algorithms: iterative, recursive
- Correctness: Loop invariants, recurrrences
- Runtime: Worst-case, average-case, best-case, amortized analysis
- Design Techniques: iterative, divide-and-conquer, dynamic programming, greedy

---

Slide FCS1 Recap: datastructures:

- Simple: Lists, queues, stacks
- Simple with key: Heaps
- Dynamic Set Datastructures:
  - Heaps: Chaining, Open addressing, Universal hashing
  - Binary search trees: Search, Insert, Delete
  - Balanced Trees: Red-Black-Trees, Treaps

Slide FCS1 Recap: datastructures:

- Augmenting datastructures
- Disjoint Sets
- Graph Datastructures: Adjacency lists, Matrix
- Minimum Spanning Trees

Slide FCS1 Recap: algorithms:

- Sorting: Insertion Sort, Merge Sort, Quicksort, Heapsort, Counting Sort, Radix Sort, Bucket Sort
- Order Statistics: Min/Max, expected linear, worst-case linear
- Binary Search Trees: Search, Insert, Delete, Rotate

Slide FCS1 Recap: algorithms:

- Dynamic Programming: Assembly Line Scheduling, Matrix Chain Multiplication, Longest Common Subsequence
- Greedy Algorithms: Activity Selection
- Graph Algorithms: DFS, BFS, TopSort, SCC, Kruskal, Prim, Bellman-Ford, Djikstra, Floyd-Warshall, Johnson

---